

HALCON Version 6.1.4



HALCON/HDevelop

Reference Manual

This manual describes the operators of the image analysis system HALCON, version 6.1.4, in HDevelop syntax. It was generated on February 1, 2005.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without prior written permission of the publisher.

Copyright © 1997-2005 by MVTec Software GmbH, München, Germany



Contents

1	Classification	1
	<i>clear_sampset</i>	1
	<i>close_all_class_box</i>	1
	<i>close_class_box</i>	2
	<i>create_class_box</i>	2
	<i>descript_class_box</i>	2
	<i>enquire_class_box</i>	3
	<i>enquire_reject_class_box</i>	4
	<i>get_class_box_param</i>	4
	<i>learn_class_box</i>	5
	<i>learn_sampset_box</i>	6
	<i>read_class_box</i>	7
	<i>read_sampset</i>	7
	<i>set_class_box_param</i>	8
	<i>test_sampset_box</i>	9
	<i>write_class_box</i>	9
2	Control	11
	<i>assign</i>	11
	<i>break</i>	12
	<i>comment</i>	12
	<i>exit</i>	13
	<i>for</i>	13
	<i>if</i>	15
	<i>ifelse</i>	15
	<i>insert</i>	16
	<i>stop</i>	17
	<i>while</i>	17
3	Develop	19
	<i>dev_clear_obj</i>	19
	<i>dev_clear_window</i>	19
	<i>dev_close_inspect_ctrl</i>	20
	<i>dev_close_window</i>	20
	<i>dev_display</i>	21
	<i>dev_error_var</i>	22
	<i>dev_inspect_ctrl</i>	23
	<i>dev_map_par</i>	23
	<i>dev_map_prog</i>	24
	<i>dev_map_var</i>	24
	<i>dev_open_window</i>	24
	<i>dev_set_check</i>	27
	<i>dev_set_color</i>	27
	<i>dev_set_colored</i>	28
	<i>dev_set_draw</i>	29
	<i>dev_set_line_width</i>	30
	<i>dev_set_lut</i>	30

	<i>dev_set_paint</i>	31
	<i>dev_set_part</i>	32
	<i>dev_set_shape</i>	33
	<i>dev_set_window</i>	34
	<i>dev_set_window_extents</i>	34
	<i>dev_unmap_par</i>	36
	<i>dev_unmap_prog</i>	36
	<i>dev_unmap_var</i>	36
	<i>dev_update_pc</i>	37
	<i>dev_update_time</i>	37
	<i>dev_update_var</i>	38
	<i>dev_update_window</i>	38
4	File	41
4.1	Images	41
	<i>read_image</i>	41
	<i>read_sequence</i>	42
	<i>write_image</i>	44
4.2	Misc	45
	<i>file_exists</i>	45
	<i>read_world_file</i>	46
4.3	Region	46
	<i>read_region</i>	46
	<i>write_region</i>	47
4.4	Text	48
	<i>close_all_files</i>	48
	<i>close_file</i>	48
	<i>fnew_line</i>	49
	<i>fread_char</i>	49
	<i>fread_string</i>	50
	<i>fwrite_string</i>	50
	<i>open_file</i>	51
4.5	Tuple	52
	<i>read_tuple</i>	52
	<i>write_tuple</i>	52
4.6	XLD	53
	<i>read_contour_xld_arc_info</i>	53
	<i>read_polygon_xld_arc_info</i>	54
	<i>write_contour_xld_arc_info</i>	54
	<i>write_polygon_xld_arc_info</i>	55
5	Filter	57
5.1	Affine-Transformations	57
	<i>affine_trans_image</i>	57
	<i>affine_trans_image_size</i>	59
	<i>mirror_image</i>	60
	<i>polar_trans_image</i>	61
	<i>rotate_image</i>	62
	<i>zoom_image_factor</i>	62
	<i>zoom_image_size</i>	63
5.2	Arithmetic	64
	<i>abs_image</i>	64
	<i>add_image</i>	65
	<i>div_image</i>	66
	<i>invert_image</i>	67
	<i>max_image</i>	68
	<i>min_image</i>	68
	<i>mult_image</i>	69

	<i>scale_image</i>	70
	<i>sub_image</i>	71
5.3	Bit	72
	<i>bit_and</i>	72
	<i>bit_lshift</i>	73
	<i>bit_mask</i>	74
	<i>bit_not</i>	75
	<i>bit_or</i>	75
	<i>bit_rshift</i>	76
	<i>bit_slice</i>	77
	<i>bit_xor</i>	78
5.4	Color	79
	<i>cfa_to_rgb</i>	79
	<i>rgb1_to_gray</i>	80
	<i>rgb3_to_gray</i>	80
	<i>trans_from_rgb</i>	81
	<i>trans_to_rgb</i>	84
5.5	Edges	87
	<i>close_edges</i>	87
	<i>close_edges_length</i>	88
	<i>derivate_gauss</i>	89
	<i>diff_of_gauss</i>	92
	<i>edges_image</i>	93
	<i>edges_sub_pix</i>	96
	<i>frei_amp</i>	98
	<i>frei_dir</i>	99
	<i>highpass_image</i>	100
	<i>info_edges</i>	101
	<i>kirsch_amp</i>	102
	<i>kirsch_dir</i>	103
	<i>laplace</i>	105
	<i>laplace_of_gauss</i>	106
	<i>prewitt_amp</i>	107
	<i>prewitt_dir</i>	108
	<i>roberts</i>	109
	<i>robinson_amp</i>	110
	<i>robinson_dir</i>	111
	<i>sobel_amp</i>	112
	<i>sobel_dir</i>	113
5.6	Enhancement	115
	<i>emphasize</i>	115
	<i>equ_histo_image</i>	116
	<i>illuminate</i>	117
	<i>scale_image_max</i>	118
5.7	FFT	118
	<i>convol_fft</i>	118
	<i>convol_gabor</i>	119
	<i>energy_gabor</i>	120
	<i>fft_generic</i>	121
	<i>fft_image</i>	122
	<i>fft_image_inv</i>	123
	<i>gen_bandfilter</i>	124
	<i>gen_bandpass</i>	125
	<i>gen_filter_mask</i>	126
	<i>gen_gabor</i>	127
	<i>gen_highpass</i>	129
	<i>gen_lowpass</i>	130
	<i>gen_sin_bandpass</i>	131

	<i>gen_std_bandpass</i>	131
	<i>phase_deg</i>	132
	<i>phase_rad</i>	133
	<i>power_byte</i>	134
	<i>power_ln</i>	134
	<i>power_real</i>	135
5.8	Lines	136
	<i>bandpass_image</i>	136
	<i>lines_facet</i>	137
	<i>lines_gauss</i>	139
5.9	Match	141
	<i>corner_response</i>	141
	<i>exhaustive_match</i>	142
	<i>exhaustive_match_mg</i>	143
	<i>fill_dvf</i>	145
	<i>gen_gauss_pyramid</i>	146
	<i>monotony</i>	147
	<i>optical_flow_match</i>	147
5.10	Misc	149
	<i>convol_image</i>	149
	<i>expand_domain_gray</i>	150
	<i>gray_inside</i>	151
	<i>gray_skeleton</i>	152
	<i>lut_trans</i>	153
	<i>map_image</i>	153
	<i>principal_comp</i>	154
	<i>symmetry</i>	155
	<i>topographic_sketch</i>	156
5.11	Noise	157
	<i>add_noise_distribution</i>	157
	<i>add_noise_white</i>	158
	<i>gauss_distribution</i>	158
	<i>noise_distribution_mean</i>	159
	<i>sp_distribution</i>	160
5.12	Smoothing	161
	<i>anisotrope_diff</i>	161
	<i>eliminate_min_max</i>	162
	<i>eliminate_sp</i>	164
	<i>fill_interlace</i>	165
	<i>gauss_image</i>	166
	<i>info_smooth</i>	167
	<i>mean_image</i>	167
	<i>mean_n</i>	168
	<i>mean_sp</i>	169
	<i>median_image</i>	170
	<i>median_separate</i>	172
	<i>median_weighted</i>	173
	<i>midrange_image</i>	174
	<i>rank_image</i>	175
	<i>sigma_image</i>	176
	<i>smooth_image</i>	177
	<i>trimmed_mean</i>	179
5.13	Texture	180
	<i>deviation_image</i>	180
	<i>entropy_image</i>	181
	<i>texture_laws</i>	181
5.14	Wiener-Filter	183
	<i>gen_psf_defocus</i>	183

	<i>gen_psf_motion</i>	184
	<i>simulate_defocus</i>	186
	<i>simulate_motion</i>	187
	<i>wiener_filter</i>	188
	<i>wiener_filter_ni</i>	190
6	Graphics	193
6.1	Drawing	193
	<i>drag_region1</i>	193
	<i>drag_region2</i>	194
	<i>drag_region3</i>	195
	<i>draw_circle</i>	196
	<i>draw_circle_mod</i>	196
	<i>draw_ellipse</i>	197
	<i>draw_ellipse_mod</i>	198
	<i>draw_line</i>	200
	<i>draw_line_mod</i>	200
	<i>draw_point</i>	202
	<i>draw_point_mod</i>	202
	<i>draw_polygon</i>	203
	<i>draw_rectangle1</i>	204
	<i>draw_rectangle1_mod</i>	205
	<i>draw_rectangle2</i>	206
	<i>draw_rectangle2_mod</i>	207
	<i>draw_region</i>	208
6.2	Gnuplot	209
	<i>gnuplot_close</i>	209
	<i>gnuplot_open_file</i>	209
	<i>gnuplot_open_pipe</i>	210
	<i>gnuplot_plot_ctrl</i>	211
	<i>gnuplot_plot_funct_1d</i>	211
	<i>gnuplot_plot_image</i>	212
6.3	LUT	213
	<i>disp_lut</i>	213
	<i>draw_lut</i>	214
	<i>get_fixed_lut</i>	215
	<i>get_lut</i>	215
	<i>get_lut_style</i>	215
	<i>query_lut</i>	216
	<i>set_fixed_lut</i>	216
	<i>set_lut</i>	217
	<i>set_lut_style</i>	219
	<i>write_lut</i>	220
6.4	Mouse	221
	<i>get_mbutton</i>	221
	<i>get_mposition</i>	222
	<i>get_mshape</i>	222
	<i>query_mshape</i>	223
	<i>set_mshape</i>	223
6.5	Output	224
	<i>disp_arc</i>	224
	<i>disp_arrow</i>	225
	<i>disp_channel</i>	227
	<i>disp_circle</i>	227
	<i>disp_color</i>	228
	<i>disp_distribution</i>	229
	<i>disp_ellipse</i>	230
	<i>disp_image</i>	232

	<i>disp_line</i>	233
	<i>disp_obj</i>	234
	<i>disp_polygon</i>	235
	<i>disp_rectangle1</i>	236
	<i>disp_rectangle2</i>	237
	<i>disp_region</i>	238
	<i>disp_xld</i>	239
6.6	Parameters	240
	<i>get_comprise</i>	240
	<i>get_draw</i>	240
	<i>get_fix</i>	241
	<i>get_hsi</i>	241
	<i>get_icon</i>	242
	<i>get_insert</i>	242
	<i>get_line_approx</i>	243
	<i>get_line_style</i>	243
	<i>get_line_width</i>	244
	<i>get_paint</i>	244
	<i>get_part</i>	245
	<i>get_part_style</i>	245
	<i>get_pixel</i>	246
	<i>get_rgb</i>	247
	<i>get_shape</i>	247
	<i>query_all_colors</i>	248
	<i>query_color</i>	248
	<i>query_colored</i>	249
	<i>query_gray</i>	250
	<i>query_insert</i>	250
	<i>query_line_width</i>	251
	<i>query_paint</i>	251
	<i>query_shape</i>	251
	<i>set_color</i>	252
	<i>set_colored</i>	253
	<i>set_comprise</i>	253
	<i>set_draw</i>	254
	<i>set_fix</i>	255
	<i>set_gray</i>	256
	<i>set_hsi</i>	257
	<i>set_icon</i>	258
	<i>set_insert</i>	258
	<i>set_line_approx</i>	259
	<i>set_line_style</i>	260
	<i>set_line_width</i>	261
	<i>set_paint</i>	261
	<i>set_part</i>	265
	<i>set_part_style</i>	266
	<i>set_pixel</i>	267
	<i>set_rgb</i>	267
	<i>set_shape</i>	268
6.7	Text	269
	<i>get_font</i>	269
	<i>get_string_extents</i>	270
	<i>get_tposition</i>	271
	<i>get_tshape</i>	271
	<i>new_line</i>	272
	<i>query_font</i>	272
	<i>query_tshape</i>	273
	<i>read_char</i>	273
	<i>read_string</i>	274

	<i>set_font</i>	275
	<i>set_tposition</i>	276
	<i>set_tshape</i>	277
	<i>write_string</i>	277
6.8	Window	278
	<i>clear_rectangle</i>	278
	<i>clear_window</i>	279
	<i>close_window</i>	280
	<i>copy_rectangle</i>	280
	<i>dump_window</i>	282
	<i>get_os_window_handle</i>	283
	<i>get_window_attr</i>	285
	<i>get_window_extents</i>	285
	<i>get_window_pointer3</i>	286
	<i>get_window_type</i>	287
	<i>move_rectangle</i>	288
	<i>new_extern_window</i>	289
	<i>open_textwindow</i>	292
	<i>open_window</i>	295
	<i>query_window_type</i>	298
	<i>set_window_attr</i>	299
	<i>set_window_dc</i>	299
	<i>set_window_extents</i>	300
	<i>set_window_type</i>	301
	<i>slide_image</i>	302
7	Image	305
7.1	Access	305
	<i>get_grayval</i>	305
	<i>get_image_pointer1</i>	306
	<i>get_image_pointer1_rect</i>	307
	<i>get_image_pointer3</i>	308
	<i>get_image_time</i>	309
7.2	Channel	309
	<i>access_channel</i>	309
	<i>append_channel</i>	310
	<i>channels_to_image</i>	311
	<i>compose2</i>	311
	<i>compose3</i>	312
	<i>compose4</i>	312
	<i>compose5</i>	313
	<i>compose6</i>	314
	<i>compose7</i>	314
	<i>count_channels</i>	315
	<i>decompose2</i>	316
	<i>decompose3</i>	316
	<i>decompose4</i>	317
	<i>decompose5</i>	318
	<i>decompose6</i>	319
	<i>decompose7</i>	319
	<i>image_to_channels</i>	320
7.3	Creation	321
	<i>copy_image</i>	321
	<i>gen_image1</i>	321
	<i>gen_image1_extern</i>	322
	<i>gen_image1_rect</i>	324
	<i>gen_image3</i>	325
	<i>gen_image_const</i>	327

	gen_image_gray_ramp	328
	gen_image_proto	329
	gen_image_surface_first_order	330
	gen_image_surface_second_order	331
	region_to_bin	333
	region_to_label	334
	region_to_mean	335
7.4	Domain	336
	add_channels	336
	change_domain	337
	full_domain	337
	get_domain	338
	rectangle1_domain	338
	reduce_domain	339
7.5	Features	340
	area_center_gray	340
	cooc_feature_image	341
	cooc_feature_matrix	341
	elliptic_axis_gray	343
	entropy_gray	343
	fit_surface_first_order	344
	fit_surface_second_order	345
	fuzzy_entropy	347
	fuzzy_perimeter	348
	gen_cooc_matrix	349
	gray_histo	350
	gray_histo_abs	351
	gray_projections	351
	histo_2dim	352
	intensity	353
	min_max_gray	354
	moments_gray_plane	355
	plane_deviation	356
	select_gray	357
	shape_histo_all	358
	shape_histo_point	360
7.6	Format	361
	change_format	361
	crop_domain	361
	crop_domain_rel	362
	crop_part	363
	crop_rectangle1	363
	tile_channels	364
	tile_images	365
	tile_images_offset	366
7.7	Framegrabber	368
	close_all_framegrabbers	368
	close_framegrabber	368
	get_framegrabber_lut	369
	get_framegrabber_param	369
	grab_image	370
	grab_image_async	371
	grab_image_start	372
	grab_region	373
	grab_region_async	374
	info_framegrabber	375
	open_framegrabber	376
	set_framegrabber_lut	378

	<i>set_framegrabber_param</i>	379
7.8	Manipulation	379
	<i>paint_gray</i>	379
	<i>paint_region</i>	380
	<i>paint_xld</i>	381
	<i>set_grayval</i>	382
7.9	Type-Conversion	383
	<i>complex_to_real</i>	383
	<i>convert_image_type</i>	384
	<i>dvf_to_int1</i>	384
	<i>int1_to_dvf</i>	385
	<i>real_to_complex</i>	385
8	Lines	387
8.1	Access	387
	<i>approx_chain</i>	387
	<i>approx_chain_simple</i>	390
8.2	Features	392
	<i>line_orientation</i>	392
	<i>line_position</i>	392
	<i>partition_lines</i>	393
	<i>select_lines</i>	395
	<i>select_lines_longest</i>	396
9	Matching	399
9.1	Gray-Value-Based	399
	<i>adapt_template</i>	399
	<i>best_match</i>	399
	<i>best_match_mg</i>	400
	<i>best_match_pre_mg</i>	402
	<i>best_match_rot</i>	403
	<i>best_match_rot_mg</i>	404
	<i>clear_all_templates</i>	405
	<i>clear_template</i>	406
	<i>create_template</i>	406
	<i>create_template_rot</i>	408
	<i>fast_match</i>	409
	<i>fast_match_mg</i>	410
	<i>read_template</i>	411
	<i>set_offset_template</i>	411
	<i>set_reference_template</i>	412
	<i>write_template</i>	412
9.2	Shape-Based	413
	<i>clear_all_shape_models</i>	413
	<i>clear_shape_model</i>	413
	<i>create_scaled_shape_model</i>	414
	<i>create_shape_model</i>	417
	<i>find_scaled_shape_model</i>	419
	<i>find_scaled_shape_models</i>	422
	<i>find_shape_model</i>	426
	<i>find_shape_models</i>	428
	<i>get_shape_model_origin</i>	432
	<i>get_shape_model_params</i>	432
	<i>inspect_shape_model</i>	433
	<i>read_shape_model</i>	434
	<i>set_shape_model_origin</i>	435
	<i>write_shape_model</i>	435

10 Morphology	437
10.1 Gray-Values	437
<i>dual_rank</i>	437
<i>gen_disc_se</i>	438
<i>gray_bothat</i>	439
<i>gray_closing</i>	440
<i>gray_closing_shape</i>	441
<i>gray_dilation</i>	442
<i>gray_dilation_rect</i>	442
<i>gray_dilation_shape</i>	443
<i>gray_erosion</i>	444
<i>gray_erosion_rect</i>	444
<i>gray_erosion_shape</i>	445
<i>gray_opening</i>	446
<i>gray_opening_shape</i>	447
<i>gray_range_rect</i>	448
<i>gray_tophat</i>	448
<i>read_gray_se</i>	449
10.2 Region	450
<i>bottom_hat</i>	450
<i>boundary</i>	451
<i>closing</i>	452
<i>closing_circle</i>	453
<i>closing_golay</i>	455
<i>closing_rectangle1</i>	456
<i>dilation1</i>	457
<i>dilation2</i>	458
<i>dilation_circle</i>	459
<i>dilation_golay</i>	460
<i>dilation_rectangle1</i>	462
<i>dilation_seq</i>	463
<i>erosion1</i>	464
<i>erosion2</i>	465
<i>erosion_circle</i>	467
<i>erosion_golay</i>	468
<i>erosion_rectangle1</i>	469
<i>erosion_seq</i>	470
<i>fitting</i>	471
<i>gen_struct_elements</i>	472
<i>golay_elements</i>	473
<i>hit_or_miss</i>	476
<i>hit_or_miss_golay</i>	477
<i>hit_or_miss_seq</i>	479
<i>minkowski_add1</i>	479
<i>minkowski_add2</i>	481
<i>minkowski_sub1</i>	482
<i>minkowski_sub2</i>	483
<i>morph_hat</i>	485
<i>morph_skeleton</i>	486
<i>morph_skiz</i>	487
<i>opening</i>	488
<i>opening_circle</i>	489
<i>opening_golay</i>	490
<i>opening_rectangle1</i>	491
<i>opening_seg</i>	492
<i>pruning</i>	493
<i>thickening</i>	494
<i>thickening_golay</i>	495
<i>thickening_seq</i>	496

	<i>thinning</i>	497
	<i>thinning_golay</i>	499
	<i>thinning_seq</i>	499
	<i>top_hat</i>	501
11	Object	503
11.1	Information	503
	<i>count_obj</i>	503
	<i>get_channel_info</i>	503
	<i>get_obj_class</i>	504
	<i>test_equal_obj</i>	505
	<i>test_equal_region</i>	505
	<i>test_obj_def</i>	506
11.2	Manipulation	507
	<i>clear_obj</i>	507
	<i>concat_obj</i>	508
	<i>copy_obj</i>	508
	<i>gen_empty_obj</i>	509
	<i>integer_to_obj</i>	510
	<i>obj_to_integer</i>	510
	<i>select_obj</i>	511
12	Regions	513
12.1	Access	513
	<i>get_region_chain</i>	513
	<i>get_region_contour</i>	514
	<i>get_region_convex</i>	514
	<i>get_region_points</i>	515
	<i>get_region_polygon</i>	516
	<i>get_region_runs</i>	516
12.2	Affine-Transformations	517
	<i>affine_trans_region</i>	517
	<i>mirror_region</i>	518
	<i>move_region</i>	519
	<i>transpose_region</i>	520
	<i>zoom_region</i>	521
12.3	Creation	522
	<i>gen_checker_region</i>	522
	<i>gen_circle</i>	523
	<i>gen_ellipse</i>	524
	<i>gen_empty_region</i>	526
	<i>gen_grid_region</i>	526
	<i>gen_random_region</i>	528
	<i>gen_random_regions</i>	528
	<i>gen_rectangle1</i>	530
	<i>gen_rectangle2</i>	531
	<i>gen_region_contour_xld</i>	533
	<i>gen_region_histo</i>	533
	<i>gen_region_hline</i>	534
	<i>gen_region_line</i>	535
	<i>gen_region_points</i>	536
	<i>gen_region_polygon</i>	537
	<i>gen_region_polygon_filled</i>	538
	<i>gen_region_polygon_xld</i>	538
	<i>gen_region_runs</i>	539
	<i>label_to_region</i>	540
12.4	Features	541
	<i>area_center</i>	541

	<i>circularity</i>	542
	<i>compactness</i>	543
	<i>connect_and_holes</i>	544
	<i>contlength</i>	545
	<i>convexity</i>	546
	<i>diameter_region</i>	547
	<i>eccentricity</i>	547
	<i>elliptic_axis</i>	548
	<i>euler_number</i>	549
	<i>find_neighbors</i>	550
	<i>get_region_index</i>	551
	<i>get_region_thickness</i>	552
	<i>hamming_distance</i>	552
	<i>hamming_distance_norm</i>	553
	<i>inner_circle</i>	554
	<i>moments_region_2nd</i>	555
	<i>moments_region_2nd_invar</i>	556
	<i>moments_region_2nd_rel_invar</i>	557
	<i>moments_region_3rd</i>	558
	<i>moments_region_3rd_invar</i>	559
	<i>moments_region_central</i>	560
	<i>moments_region_central_invar</i>	561
	<i>orientation_region</i>	562
	<i>roundness</i>	562
	<i>runlength_distribution</i>	563
	<i>runlength_features</i>	564
	<i>select_region_point</i>	565
	<i>select_region_spatial</i>	566
	<i>select_shape</i>	567
	<i>select_shape_proto</i>	570
	<i>select_shape_std</i>	572
	<i>smallest_circle</i>	573
	<i>smallest_rectangle1</i>	574
	<i>smallest_rectangle2</i>	574
	<i>spatial_relation</i>	576
	<i>test_region_point</i>	577
12.5	Sets	578
	<i>complement</i>	578
	<i>difference</i>	578
	<i>intersection</i>	579
	<i>union1</i>	580
	<i>union2</i>	581
12.6	Transformation	581
	<i>background_seg</i>	581
	<i>clip_region</i>	582
	<i>clip_region_rel</i>	583
	<i>connection</i>	584
	<i>distance_transform</i>	585
	<i>eliminate_runs</i>	587
	<i>expand_region</i>	588
	<i>fill_up</i>	589
	<i>fill_up_shape</i>	589
	<i>hamming_change_region</i>	590
	<i>interjacent</i>	591
	<i>junctions_skeleton</i>	593
	<i>merge_regions_line_scan</i>	593
	<i>partition_dynamic</i>	594
	<i>partition_rectangle</i>	595
	<i>rank_region</i>	596

	<i>remove_noise_region</i>	597
	<i>shape_trans</i>	598
	<i>skeleton</i>	599
	<i>sort_region</i>	599
	<i>split_skeleton_lines</i>	600
	<i>split_skeleton_region</i>	601
13	Segmentation	603
	<i>auto_threshold</i>	603
	<i>bin_threshold</i>	604
	<i>char_threshold</i>	605
	<i>check_difference</i>	607
	<i>class_2dim_sup</i>	608
	<i>class_2dim_unsup</i>	610
	<i>class_ndim_box</i>	612
	<i>class_ndim_norm</i>	613
	<i>detect_edge_segments</i>	615
	<i>dual_threshold</i>	616
	<i>dyn_threshold</i>	618
	<i>expand_gray</i>	619
	<i>expand_gray_ref</i>	621
	<i>expand_line</i>	623
	<i>fast_threshold</i>	624
	<i>histo_to_thresh</i>	625
	<i>hysteresis_threshold</i>	627
	<i>learn_ndim_box</i>	628
	<i>learn_ndim_norm</i>	629
	<i>local_max</i>	630
	<i>nonmax_suppression_amp</i>	631
	<i>nonmax_suppression_dir</i>	632
	<i>plateaus</i>	633
	<i>plateaus_center</i>	634
	<i>pouring</i>	635
	<i>regiongrowing</i>	637
	<i>regiongrowing_mean</i>	638
	<i>regiongrowing_n</i>	639
	<i>threshold</i>	644
	<i>threshold_sub_pix</i>	645
	<i>watersheds</i>	646
	<i>zero_crossing</i>	647
	<i>zero_crossing_sub_pix</i>	647
14	System	649
14.1	Database	649
	<i>count_relation</i>	649
	<i>get_modules</i>	650
	<i>reset_obj_db</i>	651
14.2	Error-Handling	652
	<i>get_check</i>	652
	<i>get_error_text</i>	652
	<i>get_spy</i>	653
	<i>query_spy</i>	654
	<i>set_check</i>	654
	<i>set_spy</i>	655
14.3	Information	657
	<i>disp_info</i>	657
	<i>get_chapter_info</i>	658
	<i>get_keywords</i>	659
	<i>get_operator_info</i>	659

	<i>get_operator_name</i>	660
	<i>get_param_info</i>	661
	<i>get_param_names</i>	663
	<i>get_param_num</i>	663
	<i>get_param_types</i>	664
	<i>query_operator_info</i>	665
	<i>query_param_info</i>	665
	<i>search_operator</i>	666
14.4	Operating-System	666
	<i>count_seconds</i>	666
	<i>system_call</i>	667
	<i>wait_seconds</i>	667
14.5	Parallelization	668
	<i>check_par_hw_potential</i>	668
	<i>load_par_knowledge</i>	669
	<i>store_par_knowledge</i>	670
14.6	Parameters	670
	<i>get_system</i>	670
	<i>set_system</i>	673
14.7	Serial	678
	<i>clear_serial</i>	678
	<i>close_all_serials</i>	679
	<i>close_serial</i>	679
	<i>get_serial_param</i>	680
	<i>open_serial</i>	680
	<i>read_serial</i>	681
	<i>set_serial_param</i>	681
	<i>write_serial</i>	683
14.8	Sockets	683
	<i>close_socket</i>	683
	<i>get_next_socket_data_type</i>	684
	<i>get_socket_timeout</i>	684
	<i>open_socket_accept</i>	685
	<i>open_socket_connect</i>	686
	<i>receive_image</i>	686
	<i>receive_region</i>	687
	<i>receive_tuple</i>	687
	<i>receive_xld</i>	688
	<i>send_image</i>	688
	<i>send_region</i>	689
	<i>send_tuple</i>	689
	<i>send_xld</i>	689
	<i>set_socket_timeout</i>	690
	<i>socket_accept_connect</i>	691
15	Tools	693
15.1	Affine-Transformations	693
	<i>affine_trans_point_2d</i>	693
	<i>affine_trans_point_3d</i>	694
	<i>dvf_to_hom_mat2d</i>	695
	<i>hom_mat2d_compose</i>	695
	<i>hom_mat2d_identity</i>	696
	<i>hom_mat2d_invert</i>	697
	<i>hom_mat2d_rotate</i>	697
	<i>hom_mat2d_scale</i>	698
	<i>hom_mat2d_slant</i>	699
	<i>hom_mat2d_to_affine_par</i>	701
	<i>hom_mat2d_translate</i>	702

	<i>hom_mat3d_compose</i>	703
	<i>hom_mat3d_identity</i>	703
	<i>hom_mat3d_invert</i>	704
	<i>hom_mat3d_rotate</i>	704
	<i>hom_mat3d_scale</i>	706
	<i>hom_mat3d_translate</i>	708
	<i>vector_angle_to_rigid</i>	709
	<i>vector_to_hom_mat2d</i>	710
	<i>vector_to_rigid</i>	711
	<i>vector_to_similarity</i>	712
15.2	Background-Estimator	713
	<i>close_all_bg_esti</i>	713
	<i>close_bg_esti</i>	713
	<i>create_bg_esti</i>	714
	<i>get_bg_esti_params</i>	716
	<i>give_bg_esti</i>	718
	<i>run_bg_esti</i>	718
	<i>set_bg_esti_params</i>	720
	<i>update_bg_esti</i>	722
15.3	Barcode	723
	<i>decode_1d_bar_code</i>	723
	<i>decode_2d_bar_code</i>	724
	<i>discrete_1d_bar_code</i>	725
	<i>find_1d_bar_code</i>	726
	<i>find_1d_bar_code_region</i>	729
	<i>find_2d_bar_code</i>	731
	<i>gen_1d_bar_code_descr</i>	734
	<i>gen_1d_bar_code_descr_gen</i>	735
	<i>gen_2d_bar_code_descr</i>	736
	<i>get_1d_bar_code</i>	738
	<i>get_2d_bar_code</i>	739
	<i>get_2d_bar_code_pos</i>	743
15.4	Calibration	745
	<i>caltab_points</i>	745
	<i>camera_calibration</i>	746
	<i>change_radial_distortion_cam_par</i>	751
	<i>change_radial_distortion_contours_xld</i>	752
	<i>change_radial_distortion_image</i>	753
	<i>contour_to_world_plane_xld</i>	753
	<i>convert_pose_type</i>	754
	<i>create_caltab</i>	755
	<i>create_pose</i>	758
	<i>disp_caltab</i>	761
	<i>find_caltab</i>	763
	<i>find_marks_and_pose</i>	764
	<i>gen_image_to_world_plane_map</i>	766
	<i>gen_radial_distortion_map</i>	768
	<i>get_line_of_sight</i>	769
	<i>get_pose_type</i>	770
	<i>hand_eye_calibration</i>	771
	<i>hom_mat3d_to_pose</i>	778
	<i>image_points_to_world_plane</i>	779
	<i>image_to_world_plane</i>	781
	<i>pose_to_hom_mat3d</i>	783
	<i>project_3d_point</i>	784
	<i>read_cam_par</i>	785
	<i>read_pose</i>	786
	<i>set_origin_pose</i>	787

	<i>sim_caltab</i>	788
	<i>write_cam_par</i>	789
	<i>write_pose</i>	791
15.5	Fourier-Descriptor	793
	<i>abs_invar_fourier_coeff</i>	793
	<i>fourier_1dim</i>	794
	<i>fourier_1dim_inv</i>	795
	<i>invar_fourier_coeff</i>	795
	<i>match_fourier_coeff</i>	796
	<i>move_contour_orig</i>	797
	<i>prep_contour_fourier</i>	798
15.6	Function	799
	<i>abs_funct_1d</i>	799
	<i>create_funct_1d_array</i>	799
	<i>create_funct_1d_pairs</i>	800
	<i>distance_funct_1d</i>	800
	<i>funct_1d_to_pairs</i>	801
	<i>get_pair_funct_1d</i>	801
	<i>integrate_funct_1d</i>	801
	<i>match_funct_1d_trans</i>	802
	<i>negate_funct_1d</i>	803
	<i>num_points_funct_1d</i>	803
	<i>read_funct_1d</i>	804
	<i>sample_funct_1d</i>	804
	<i>scale_y_funct_1d</i>	805
	<i>smooth_funct_1d_gauss</i>	805
	<i>smooth_funct_1d_mean</i>	806
	<i>transform_funct_1d</i>	807
	<i>write_funct_1d</i>	807
	<i>x_range_funct_1d</i>	808
	<i>y_range_funct_1d</i>	808
15.7	Geometry	808
	<i>angle_ll</i>	808
	<i>angle_lx</i>	809
	<i>distance_lr</i>	810
	<i>distance_pl</i>	811
	<i>distance_pp</i>	813
	<i>distance_pr</i>	814
	<i>distance_ps</i>	814
	<i>distance_rr_min</i>	816
	<i>distance_rr_min_dil</i>	816
	<i>distance_sl</i>	817
	<i>distance_sr</i>	818
	<i>distance_ss</i>	819
	<i>get_points_ellipse</i>	821
	<i>intersection_ll</i>	821
	<i>projection_pl</i>	823
15.8	Hough	824
	<i>hough_circle_trans</i>	824
	<i>hough_circles</i>	824
	<i>hough_line_trans</i>	825
	<i>hough_line_trans_dir</i>	826
	<i>hough_lines</i>	827
	<i>hough_lines_dir</i>	828
	<i>select_matching_lines</i>	830
15.9	Image-Comparison	831
	<i>clear_all_variation_models</i>	831
	<i>clear_variation_model</i>	831

	<i>compare_variation_model</i>	831
	<i>create_variation_model</i>	832
	<i>get_variation_model</i>	834
	<i>prepare_variation_model</i>	834
	<i>read_variation_model</i>	835
	<i>train_variation_model</i>	835
	<i>write_variation_model</i>	837
15.10	Kalman-Filter	837
	<i>filter_kalman</i>	837
	<i>read_kalman</i>	841
	<i>sensor_kalman</i>	843
	<i>update_kalman</i>	844
15.11	Measure	846
	<i>close_all_measures</i>	846
	<i>close_measure</i>	847
	<i>fuzzy_measure_pairing</i>	847
	<i>fuzzy_measure_pairs</i>	849
	<i>fuzzy_measure_pos</i>	851
	<i>gen_measure_arc</i>	853
	<i>gen_measure_rectangle2</i>	855
	<i>measure_pairs</i>	857
	<i>measure_pos</i>	859
	<i>measure_projection</i>	860
	<i>measure_thresh</i>	861
	<i>reset_fuzzy_measure</i>	862
	<i>set_fuzzy_measure</i>	863
	<i>set_fuzzy_measure_norm_pair</i>	865
	<i>translate_measure</i>	866
15.12	OCR	867
	<i>append_ocr_trainf</i>	867
	<i>close_all_ocrs</i>	868
	<i>close_ocr</i>	869
	<i>concat_ocr_trainf</i>	869
	<i>create_ocr_class_box</i>	870
	<i>do_ocr_multi</i>	872
	<i>do_ocr_single</i>	873
	<i>info_ocr_class_box</i>	873
	<i>ocr_change_char</i>	874
	<i>ocr_get_features</i>	874
	<i>read_ocr</i>	875
	<i>read_ocr_trainf</i>	876
	<i>read_ocr_trainf_names</i>	876
	<i>read_ocr_trainf_select</i>	877
	<i>testd_ocr_class_box</i>	877
	<i>traind_ocr_class_box</i>	878
	<i>trainf_ocr_class_box</i>	879
	<i>write_ocr</i>	879
	<i>write_ocr_trainf</i>	880
	<i>write_ocr_trainf_image</i>	880
15.13	OCV	881
	<i>close_all_ocvs</i>	881
	<i>close_ocv</i>	882
	<i>create_ocv_proj</i>	882
	<i>do_ocv_simple</i>	883
	<i>read_ocv</i>	884
	<i>traind_ocv_proj</i>	885
	<i>write_ocv</i>	886
15.14	Shape-from	887

	<i>depth_from_focus</i>	887
	<i>estimate_al_am</i>	888
	<i>estimate_sl_al_lr</i>	888
	<i>estimate_sl_al_zc</i>	889
	<i>estimate_tilt_lr</i>	889
	<i>estimate_tilt_zc</i>	889
	<i>phot_stereo</i>	890
	<i>select_grayvalues_from_channels</i>	891
	<i>sfs_mod_lr</i>	891
	<i>sfs_orig_lr</i>	892
	<i>sfs_pentland</i>	894
	<i>shade_height_field</i>	895
16	Tuple	897
16.1	Arithmetic	897
	<i>tuple_abs</i>	897
	<i>tuple_acos</i>	897
	<i>tuple_add</i>	898
	<i>tuple_asin</i>	898
	<i>tuple_atan</i>	899
	<i>tuple_atan2</i>	899
	<i>tuple_ceil</i>	899
	<i>tuple_cos</i>	900
	<i>tuple_cosh</i>	900
	<i>tuple_div</i>	901
	<i>tuple_exp</i>	901
	<i>tuple_fabs</i>	901
	<i>tuple_floor</i>	902
	<i>tuple_fmod</i>	902
	<i>tuple_ldexp</i>	903
	<i>tuple_log</i>	903
	<i>tuple_log10</i>	903
	<i>tuple_mult</i>	904
	<i>tuple_neg</i>	904
	<i>tuple_pow</i>	905
	<i>tuple_rad</i>	905
	<i>tuple_sin</i>	905
	<i>tuple_sinh</i>	906
	<i>tuple_sqrt</i>	906
	<i>tuple_sub</i>	907
	<i>tuple_tan</i>	907
	<i>tuple_tanh</i>	907
16.2	Bit-Operations	908
	<i>tuple_band</i>	908
	<i>tuple_bnot</i>	908
	<i>tuple_bor</i>	909
	<i>tuple_bxor</i>	909
	<i>tuple_lsh</i>	910
	<i>tuple_rsh</i>	910
16.3	Comparison	911
	<i>tuple_equal</i>	911
	<i>tuple_greater</i>	911
	<i>tuple_greater_equal</i>	911
	<i>tuple_less</i>	912
	<i>tuple_less_equal</i>	912
	<i>tuple_not_equal</i>	913
16.4	Conversion	913
	<i>tuple_chr</i>	913

	<i>tuple_chrt</i>	914
	<i>tuple_deg</i>	914
	<i>tuple_is_number</i>	915
	<i>tuple_number</i>	915
	<i>tuple_ord</i>	916
	<i>tuple_ords</i>	916
	<i>tuple_real</i>	916
	<i>tuple_round</i>	917
	<i>tuple_string</i>	917
16.5	Creation	919
	<i>tuple_concat</i>	919
	<i>tuple_gen_const</i>	919
16.6	Element-Order	919
	<i>tuple_inverse</i>	919
	<i>tuple_sort</i>	920
	<i>tuple_sort_index</i>	920
16.7	Features	921
	<i>tuple_deviation</i>	921
	<i>tuple_length</i>	921
	<i>tuple_max</i>	921
	<i>tuple_mean</i>	922
	<i>tuple_min</i>	922
	<i>tuple_sum</i>	923
16.8	Logical-Operations	923
	<i>tuple_and</i>	923
	<i>tuple_not</i>	923
	<i>tuple_or</i>	924
	<i>tuple_xor</i>	924
16.9	Selection	925
	<i>tuple_first_n</i>	925
	<i>tuple_last_n</i>	925
	<i>tuple_select</i>	926
	<i>tuple_select_range</i>	926
	<i>tuple_str_bit_select</i>	927
16.10	String-Operators	927
	<i>tuple_environment</i>	927
	<i>tuple_split</i>	928
	<i>tuple_str_first_n</i>	928
	<i>tuple_str_last_n</i>	929
	<i>tuple_strchr</i>	929
	<i>tuple_strlen</i>	930
	<i>tuple_strchr</i>	930
	<i>tuple_strstr</i>	931
	<i>tuple_strstr</i>	931
17	XLD	933
17.1	Access	933
	<i>get_contour_xld</i>	933
	<i>get_lines_xld</i>	933
	<i>get_parallels_xld</i>	934
	<i>get_polygon_xld</i>	935
17.2	Creation	935
	<i>gen_contour_polygon_rounded_xld</i>	935
	<i>gen_contour_polygon_xld</i>	936
	<i>gen_contour_region_xld</i>	937
	<i>gen_contours_skeleton_xld</i>	938
	<i>gen_ellipse_contour_xld</i>	939
	<i>gen_parallels_xld</i>	940

	<i>gen_polygons_xld</i>	941
	<i>mod_parallels_xld</i>	942
17.3	Features	943
	<i>area_center_xld</i>	943
	<i>circularity_xld</i>	944
	<i>compactness_xld</i>	945
	<i>contour_point_num_xld</i>	945
	<i>convexity_xld</i>	946
	<i>diameter_xld</i>	946
	<i>dist_ellipse_contour_xld</i>	947
	<i>eccentricity_xld</i>	949
	<i>elliptic_axis_xld</i>	950
	<i>fit_circle_contour_xld</i>	951
	<i>fit_ellipse_contour_xld</i>	952
	<i>fit_line_contour_xld</i>	955
	<i>get_contour_angle_xld</i>	957
	<i>get_contour_attrib_xld</i>	957
	<i>get_contour_global_attrib_xld</i>	958
	<i>get_regress_params_xld</i>	959
	<i>info_parallels_xld</i>	960
	<i>length_xld</i>	960
	<i>local_max_contours_xld</i>	961
	<i>max_parallels_xld</i>	962
	<i>moments_any_xld</i>	962
	<i>moments_xld</i>	963
	<i>orientation_xld</i>	964
	<i>query_contour_attribs_xld</i>	965
	<i>query_contour_global_attribs_xld</i>	965
	<i>select_contours_xld</i>	966
	<i>select_shape_xld</i>	967
	<i>smallest_circle_xld</i>	969
	<i>smallest_rectangle1_xld</i>	969
	<i>smallest_rectangle2_xld</i>	970
17.4	Transformation	971
	<i>add_noise_white_contour_xld</i>	971
	<i>affine_trans_contour_xld</i>	972
	<i>affine_trans_polygon_xld</i>	973
	<i>clip_contours_xld</i>	973
	<i>combine_roads_xld</i>	974
	<i>gen_parallel_contour_xld</i>	975
	<i>merge_cont_line_scan_xld</i>	976
	<i>regress_contours_xld</i>	977
	<i>segment_contours_xld</i>	978
	<i>shape_trans_xld</i>	980
	<i>smooth_contours_xld</i>	981
	<i>split_contours_xld</i>	981
	<i>union_straight_contours_histo_xld</i>	982
	<i>union_straight_contours_xld</i>	983

Chapter 1

Classification

clear_sampset (: : SampKey :)

Free memory of a data set.

clear_sampset frees the memory which was used for training data set having read by **read_sampset**. This memory is only reusable in combination with **read_sampset**.

Parameter

- ▷ **SampKey** (input_control) feature_set $\leadsto$ integer
Number of the data set.

Result

clear_sampset returns 2 (H_MSG_TRUE). An exception handling is raised if the key **SampKey** does not exist.

Parallelization Information

clear_sampset is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

create_class_box, **enquire_class_box**, **learn_class_box**, **write_class_box**

See also

test_sampset_box, **learn_sampset_box**, **read_sampset**

Module

Tools

close_all_class_box (: : :)

Destroy all classifiers.

close_all_class_box deletes all classifiers and frees the used memory space. All the trained data will be lost.

Attention

Since all classifiers are closed by **close_all_class_box** all handles become invalid.

Result

If it is possible to close the classifiers the operator **close_all_class_box** returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

close_all_class_box is *local* and processed *completely exclusively* without parallelization.

Alternatives

close_class_box

Module

Tools

close_class_box (: : ClassifHandle :)

Destroy the classifier.

[close_class_box](#) deletes the classifier and frees the used memory space. All the trained data will be lost. For saving this trained data you should use [write_class_box](#) before.

Parameter

- ▷ **ClassifHandle** (input_control) class_box $\rightsquigarrow$ integer
Classifier's handle number.

Result

[close_class_box](#) returns 2 (H_MSG_TRUE).

Parallelization Information

[close_class_box](#) is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[create_class_box](#), [enquire_class_box](#), [learn_class_box](#), [write_class_box](#)

See also

[create_class_box](#), [enquire_class_box](#), [learn_class_box](#)

Module

Tools

create_class_box (: : : ClassifHandle)

Create a new classifier.

[create_class_box](#) creates a new adaptive classifier. All procedures which are explained in chapter classification refer to such a initialized classifier (of type 2). See [learn_class_box](#) for more details about the functionality of the classifier.

Parameter

- ▷ **ClassifHandle** (output_control) class_box $\rightsquigarrow$ integer
Classifier's handle number.

Result

[create_class_box](#) returns 2 (H_MSG_TRUE) if the parameter is correct. An exception handling is raised if a classifier with this name already exists or there is not enough memory.

Parallelization Information

[create_class_box](#) is *local* and processed *completely exclusively* without parallelization.

Possible Successors

[learn_class_box](#), [enquire_class_box](#), [write_class_box](#), [close_class_box](#),
[clear_sampset](#)

See also

[learn_class_box](#), [enquire_class_box](#), [close_class_box](#)

Module

Tools

descript_class_box (: : ClassifHandle, Dimensions :)

Description of the classifier.

A classifier uses a set of hyper cuboids for every class. With these hyper cuboids it is attempted to get the array attributes inside the class. [descript_class_box](#) returns for every class the expansion of every appropriate cuboid from dimension 1 up to [Dimensions](#) (to 'standard_output').

Parameter
▷ ClassifHandle (input_control) class_box $\leadsto$ integer Classifier's handle number. ▷ Dimensions (input_control) integer $\leadsto$ integer Highest dimension for output. Default Value : 3
Result
descript_class_box returns 2 (H_MSG_TRUE).
Parallelization Information
descript_class_box is <i>local</i> and processed <i>completely exclusively</i> without parallelization.
Possible Predecessors
create_class_box , learn_class_box , set_class_box_param
Possible Successors
enquire_class_box , learn_class_box , write_class_box , close_class_box
See also
create_class_box , enquire_class_box , learn_class_box , read_class_box , write_class_box
Module
Tools

enquire_class_box (: : ClassifHandle, FeatureList : Class)

Classify a tuple of attributes.

FeatureList is a tuple of any floating point- or integer numbers (attributes) which has to be assigned to a class with assistance of a previous trained ([learn_class_box](#)) classifier. It is possible to specify attributes as unknown by indicating the symbol '*' instead of a number. If you specify n values, then all following values, i.e. the attributes n+1 until 'max', are automatically supposed to be undefined.

See [learn_class_box](#) for more details about the functionality of the classifier.

You may call the procedures [learn_class_box](#) and [enquire_class_box](#) alternately, so that it is possible to classify already in the phase of learning. This means you could see when a satisfying behavior had been reached.

Parameter
▷ ClassifHandle (input_control) class_box $\leadsto$ integer Classifier's handle number. ▷ FeatureList (input_control) real-array $\leadsto$ real / integer / string Array of attributes which has to be classified. Default Value : 1.0 ▷ Class (output_control) integer $\leadsto$ integer Number of the class to which the array of attributes had been assigned.
Result
enquire_class_box returns 2 (H_MSG_TRUE).
Parallelization Information
enquire_class_box is <i>local</i> and processed <i>completely exclusively</i> without parallelization.
Possible Predecessors
create_class_box , learn_class_box , set_class_box_param
Possible Successors
learn_class_box , write_class_box , close_class_box
Alternatives
enquire_reject_class_box
See also
test_sampset_box , learn_class_box , learn_sampset_box
Module
Tools

enquire_reject_class_box (: : ClassifHandle, FeatureList : Class)

Classify a tuple of attributes with rejection class.

FeatureList is a tuple of any floating point- or integer numbers (attributes) which has to be assigned to a class with assistance of a previous trained ([learn_class_box](#)) classifier. It is possible to specify attributes as unknown by indicating the symbol '*' instead of a number. If you specify n values, then all following values, i.e. the attributes n+1 until 'max', are automatically supposed to be undefined.

If the array of attributes cannot be assigned to a class, i.e. the array does not reside inside of one of the hyper boxes, then in contrary to [enquire_class_box](#) not the next class is going to be returned, but the rejection class -1 as a result is going to be passed.

See [learn_class_box](#) for more details about the functionality of the classifier.

You may call the procedures [learn_class_box](#) and [enquire_class_box](#) alternately, so that it is possible to classify already in the phase of learning. By this means you could see when a satisfying behavior had been reached.

Parameter

- ▷ **ClassifHandle** (input_control) class_box $\leadsto$ integer
Classifier's handle number.
- ▷ **FeatureList** (input_control) real-array $\leadsto$ real / integer / string
Array of attributes which has to be classified.
Default Value : 1.0
- ▷ **Class** (output_control) integer $\leadsto$ integer
Number of the class, to which the array of attributes had been assigned or -1 for the rejection class.

Result

[enquire_reject_class_box](#) returns 2 (H_MSG_TRUE).

Parallelization Information

[enquire_reject_class_box](#) is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[create_class_box](#), [learn_class_box](#), [set_class_box_param](#)

Possible Successors

[learn_class_box](#), [write_class_box](#), [close_class_box](#)

Alternatives

[enquire_class_box](#)

See also

[test_sampset_box](#), [learn_class_box](#), [learn_sampset_box](#)

Module

Tools

get_class_box_param (: : ClassifHandle, Flag : Value)

Get information about the current parameter.

[get_class_box_param](#) gets the parameter of the classifier. The meaning of the parameter is explained in [set_class_box_param](#).

Default values:

'min_samples_for_split' = 80,

'split_error' = 0.1,

'prop_constant' = 0.25

Parameter

- ▷ **ClassifHandle** (input_control) class_box $\leadsto$ integer
Classifier's handle number.
- ▷ **Flag** (input_control) string $\leadsto$ string
Name of the system parameter.
Default Value : 'split_error'
List of values : Flag $\in$ {'split_error', 'prop_constant', 'used_memory', 'min_samples_for_split'}
- ▷ **Value** (output_control) number $\leadsto$ real / integer
Value of the system parameter.

Result

[get_class_box_param](#) returns 2 (H_MSG_TRUE). An exception handling is raised if [Flag](#) has been set with wrong values.

Parallelization Information

[get_class_box_param](#) is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[create_class_box](#), [enquire_class_box](#), [learn_class_box](#), [write_class_box](#)

Possible Successors

[set_class_box_param](#), [learn_class_box](#), [enquire_class_box](#), [write_class_box](#),
[close_class_box](#), [clear_sampset](#)

See also

[create_class_box](#), [set_class_box_param](#)

Module

Tools

learn_class_box (: : ClassifHandle, Features, Class :)

Train the classifier.

[Features](#) is a tuple of any floating point numbers or integers (attributes) which has to be assigned to the class [Class](#). This class is specified by an integer. You may use procedure [enquire_class_box](#) later to find the most probable class for any array (=tuple). The algorithm tries to describe the set of arrays of one class by hyper cuboids in the feature space. On demand you may even create several cuboids per class. Hence it is possible to learn disjunct concepts, too. I.e such concepts which split in several "cluster" of points in the feature space. The data structure is hidden to the user and only accessible with such procedures which are described in this chapter.

It is possible to specify attributes as unknown by indicating the symbol '*' instead of a number. If you specify n values, then all following values, i.e. the attributes n+1 until 'max', are automatically supposed to be undefined.

You may call the procedures [learn_class_box](#) and [enquire_class_box](#) alternately, so that it is possible to classify already in the phase of learning. By this means you could see when a satisfying behavior had been reached.

The classifier is going to be bigger using further training. This means, that it is not advisable to continue training after reaching a satisfactory behavior.

Parameter

- ▷ **ClassifHandle** (input_control) class_box $\leadsto$ integer
Classifier's handle number.
- ▷ **Features** (input_control) number-array $\leadsto$ real / integer / string
Array of attributes to learn.
Default Value : '[1.0,1.5,2.0]'
- ▷ **Class** (input_control) integer $\leadsto$ integer
Class to which the array has to be assigned.
Default Value : 1

Result

[learn_class_box](#) returns 2 (H_MSG_TRUE) for a normal case. An exception handling is raised if there are

memory allocation problems. The number of classes is constrained. If this limit is passed, an exception handling is raised, too.

Parallelization Information

[learn_class_box](#) is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[create_class_box](#), [enquire_class_box](#)

Possible Successors

[test_sampset_box](#), [learn_class_box](#), [enquire_class_box](#), [write_class_box](#),
[close_class_box](#), [clear_sampset](#)

See also

[test_sampset_box](#), [close_class_box](#), [create_class_box](#), [enquire_class_box](#),
[learn_sampset_box](#)

Module

Tools

learn_sampset_box (: : ClassifHandle, SampKey, Outfile, NSamples,
StopError, ErrorN :)

Train the classifier with one data set.

[learn_sampset_box](#) trains the classifier with data for the key [SampKey](#) (see [read_sampset](#)). The training sequence is terminated at least after [NSamples](#) examples. If [NSamples](#) is bigger than the number of examples in [SampKey](#), then a cyclic start at the beginning occurs. If the error underpasses the value [StopError](#), then the training sequence is prematurely terminated. [StopError](#) is calculated with N / ErrorN . Whereby N signifies the number of examples which were wrong classified during the last [ErrorN](#) training examples. Typically [ErrorN](#) is the number of examples in [SampKey](#) and [NSamples](#) is a multiple of it. If you want a data set with 100 examples to run 5 times at most and if you want it to terminate with an error lower than 5%, then the corresponding values are [NSamples](#) = 500, [ErrorN](#) = 100 and [StopError](#) = 0.05. A protocol of the training activity is going to be written in file [Outfile](#).

Parameter

- ▷ **ClassifHandle** (input_control) class_box $\leadsto$ *integer*
Classifier's handle number.
- ▷ **SampKey** (input_control) feature_set $\leadsto$ *integer*
Number of the data set to train.
- ▷ **Outfile** (input_control) filename $\leadsto$ *string*
Name of the protocol file.
Default Value : 'training_prot'
- ▷ **NSamples** (input_control) integer $\leadsto$ *integer*
Number of arrays of attributes to learn.
Default Value : 500
- ▷ **StopError** (input_control) real $\leadsto$ *real*
Classification error for termination.
Default Value : 0.05
- ▷ **ErrorN** (input_control) integer $\leadsto$ *integer*
Error during the assignment.
Default Value : 100

Result

[learn_sampset_box](#) returns 2 (H_MSG_TRUE). An exception handling is raised if key [SampKey](#) does not exist or there are problems while opening the file.

Parallelization Information

[learn_sampset_box](#) is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[create_class_box](#)

Possible Successors

[test_sampset_box](#), [enquire_class_box](#), [write_class_box](#), [close_class_box](#),
[clear_sampset](#)

See also

[test_sampset_box](#), [enquire_class_box](#), [learn_class_box](#), [read_sampset](#)

Module

Tools

read_class_box (: : ClassifHandle, FileName :)

Read the classifier from a file.

[read_class_box](#) reads the saved classifier from the file [FileName](#) (see [write_class_box](#)). The values of the current classifier are overwritten.

Attention

All values of the classifier are going to be overwritten.

Parameter

- ▷ **ClassifHandle** (input_control) class_box $\leadsto$ integer
Classifier's handle number.
- ▷ **FileName** (input_control) filename $\leadsto$ string
Filename of the classifiers.
Default Value : 'klassifikator1'

Result

[read_class_box](#) returns 2 (H_MSG_TRUE). An exception handling is raised if it was not possible to open file [FileName](#) or the file has the wrong format.

Parallelization Information

[read_class_box](#) is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[create_class_box](#)

Possible Successors

[test_sampset_box](#), [enquire_class_box](#), [write_class_box](#), [close_class_box](#),
[clear_sampset](#)

See also

[create_class_box](#), [write_class_box](#)

Module

Tools

read_sampset (: : FileName : SampKey)

Read a training data set from a file.

The training examples are accessible with the key [SampKey](#) by calling procedures [clear_sampset](#) and [learn_sampset_box](#). You may edit the file using an editor. Every row contains an array of attributes with corresponding class. An example for a format might be:

(1.0, 25.3, *, 17 | 3)

This row specifies an array of attributes which belong to class 3. In this array the third attribute is unknown. Attributes upwards 5 are supposed to be unknown, too. You may insert comments like /* .. */ in any place.

Parameter

- ▷ **FileName** (input_control) filename $\leadsto$ string
Filename of the data set to train.
Default Value : 'sampset1'

- ▷ **SampKey** (output_control) feature_set $\leadsto$ integer
Identification of the data set to train.

Result

[read_sampset](#) returns 2 (H_MSG_TRUE). An exception handling is raised if it is not possible to open the file or it contains syntax errors or there is not enough memory.

Parallelization Information

[read_sampset](#) is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[create_class_box](#)

Possible Successors

[test_sampset_box](#), [enquire_class_box](#), [write_class_box](#), [close_class_box](#),
[clear_sampset](#)

See also

[test_sampset_box](#), [clear_sampset](#), [learn_sampset_box](#)

Module

Tools

set_class_box_param (: : ClassifHandle, Flag, Value :)

Set system parameters for classification.

[set_class_box_param](#) modifies parameter which manipulate the training sequence while calling [learn_class_box](#). Only parameters of the classifier are modified, all other classifiers remain unmodified. 'min_samples_for_split' is the number of examples at least which have to train in one cuboid of this classifier, before the cuboid is allowed to divide itself. 'split_error' indicates the critical error. By its exceeding the cuboid divides itself, if there are more than 'min_samples_for_split' examples to train. 'prop_constant' manipulates the extension of the cuboids. It is proportional to the average distance of the training examples in this cuboid to the center of the cuboid. More detailed:

$$\text{extension} \times \text{prop} = \text{average distance of the expectation value.}$$

This relation is valid in every dimension. Hence inside a cuboid the dimensions of the feature space are supposed to be independent.

The parameters are set with problem independent default values, which must not modified without any reason. Parameters are only important during a learning sequence. They do not influence on the behavior of [enquire_class_box](#).

Default setting:

'min_samples_for_split' = 80,

'split_error' = 0.1,

'prop_constant' = 0.25

Parameter

- ▷ **ClassifHandle** (input_control) class_box $\leadsto$ integer
Classifier's handle number.
- ▷ **Flag** (input_control) string $\leadsto$ string
Name of the wanted parameter.
Default Value : 'split_error'
Suggested values : Flag $\in$ {'min_samples_for_split', 'split_error', 'prop_constant'}
- ▷ **Value** (input_control) number $\leadsto$ real / integer
Value of the parameter.
Default Value : 0.1

Result

[read_sampset](#) returns 2 (H_MSG_TRUE).

Parallelization Information

[set_class_box_param](#) is *local* and processed *completely exclusively* without parallelization.

<i>Possible Predecessors</i>
create_class_box , enquire_class_box
<i>Possible Successors</i>
learn_class_box , test_sampset_box , write_class_box , close_class_box , clear_sampset
<i>See also</i>
enquire_class_box , get_class_box_param , learn_class_box
<i>Module</i>
Tools

test_sampset_box (: : ClassifHandle, SampKey : Error)

Classify a set of arrays.

In contrast to [learn_sampset_box](#) there is not a learning here. Typically you use [test_sampset_box](#) to classify independent test data. [Error](#) gives you information about the applicability of the learned training set on new examples.

<i>Parameter</i>
▷ ClassifHandle (input_control) class_box $\leadsto$ integer Classifier's handle number. ▷ SampKey (input_control) feature_set $\leadsto$ integer Key of the test data. ▷ Error (output_control) real $\leadsto$ real Error during the assignment.

<i>Result</i>
test_sampset_box returns 2 (H_MSG_TRUE). An exception handling is raised, if if key SampKey does not exist or problems occur while opening the file.

<i>Parallelization Information</i>
test_sampset_box is <i>local</i> and processed <i>completely exclusively</i> without parallelization.
<i>Possible Predecessors</i>
create_class_box , learn_class_box , set_class_box_param
<i>Possible Successors</i>
enquire_class_box , learn_class_box , write_class_box , close_class_box , clear_sampset
<i>See also</i>
enquire_class_box , learn_class_box , learn_sampset_box , read_sampset
<i>Module</i>
Tools

write_class_box (: : ClassifHandle, FileName :)

Save the classifier in a file.

[write_class_box](#) saves the classifier in a file. You may read the data by calling [read_class_box](#).

<i>Attention</i>
If a file with this name exists, it is overwritten without a warning. The file can not be edited.
<i>Parameter</i>
▷ ClassifHandle (input_control) class_box $\leadsto$ integer Classifier's handle number. ▷ FileName (input_control) filename $\leadsto$ string Name of the file which contains the written data. Default Value : 'klassifikator1'

Result

`write_class_box` returns 2 (H_MSG_TRUE). An exception handling is raised if it was not possible to open file `FileName`.

Parallelization Information

`write_class_box` is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

`create_class_box`, `enquire_class_box`, `learn_class_box`, `test_sampset_box`, `write_class_box`

Possible Successors

`close_class_box`, `clear_sampset`

See also

`create_class_box`, `read_class_box`

Module

Tools

Chapter 2

Control

assign (: : Input : Result)

Assign a new value to a control variable.

assign assigns a new value to a variable. In HDevelop an assignment is treated like an operator. To use an assignment you have to select the operator **assign(Input, Result)**. This operator has the following semantics: It evaluates **Input** (right side of assignment) and stores it in **Result** (left side of assignment). However, in the program text the assignment is represented by the usual syntax of the assignment operator: `' := '`. The following example outlines the difference between an assignment in C syntax and its transformed version in HDevelop:

The assignment in C syntax

```
u = sin(x) + cos(y);
```

is defined in HDevelop using the assignment operator as

```
assign(sin(x) + cos(y), u)
```

which is displayed in the program window as:

```
u := sin(x) + cos(y)
```

Parameter

- ▷ **Input** (input_control) real(-array) $\rightsquigarrow$ real / integer / string
New value.
Default Value : 1
- ▷ **Result** (output_control) real(-array) $\rightsquigarrow$ real / integer / string
Variable that has to be changed.

Example

```
Tuple1 := [1,0,3,4,5,6,7,8,9]
Val := sin(1.2) + cos(1.2)
Tuple1[1] := 2
Tuple2 := []
for i := 0 to 10 by 1
  Tuple2[i] := i
endfor
```

Result

assign returns 2 (H.MSG.TRUE) if the evaluation of the expression yields no error.

Parallelization Information

assign is *reentrant*, *local*, and processed *without* parallelization.

Alternatives

insert

Module

Basic operators

break (: : :)*Terminate loop execution.*

break terminates the smallest enclosing **for** or **while** loop. Program execution is continued at the next program line after the end of the loop or at the next line after the **break** statement in case no enclosing loop exists.

Example

```

read_image (Image, 'monkey')
threshold (Image, Region, 160, 180)
connection (Region, Regions)
Number := |Regions|
AllRegionsValid := 1
* check if for all regions area <=30
for i := 1 to Number by 1
    ObjectSelected := Regions[i]
    area_center (ObjectSelected, Area, Row, Column)
    if (Area > 30)
        AllRegionsValid := 0
        break ()
    endif
endfor

```

Result

break always returns 2 (H_MSG_TRUE)

Parallelization Information

break is *reentrant*, *local*, and processed *without* parallelization.

See also

for, **while**

Module

Basic operators

comment (: : Comment :)*Add a comment of one line to the program.*

comment allows to add a comment of one line to the program. As parameter value, i.e., as comment, all characters are allowed. This operator has no effect on the program execution.

Parameter

- ▷ **Comment** (input_control) string $\leadsto$ *string*
 Arbitrary sequence of characters.

Example

```

* This is a program width comments
* 'this is a string as comment'
* here are numbers: 4711, 0.815
stop ()

```

Result

comment always returns 2 (H_MSG_TRUE).

Parallelization Information

comment is *reentrant*, *local*, and processed *without* parallelization.

Module

Basic operators

exit (: : :)

Terminate HDevelop.

exit terminates HDevelop. The operator is equivalent to the menu **File ▸ Quit**. Internally and for exported C++ code the C-function call **exit(0)** is used.

Example

```
read_image (Image, 'fabrik')
intensity (Image, Image, Mean, Deviation)
open_file ('intensity.txt', 'output', FileHandle)
fwrite_string (FileHandle, Mean + ' ' + Deviation)
close_file (FileHandle)
exit ()
```

Result

exit returns 0 (o.k.) to the calling environment of HDevelop = operating system.

Parallelization Information

exit is *reentrant*, *local*, and processed *without* parallelization.

See also

stop

Module

Basic operators

for (: : Start, End, Step : Variable)

Execute the body for a fixed number.

The **for** loop is controlled by a start and termination value and an incrementation value that determines the number of loop steps. These values may also be expressions which are evaluated immediately before the loop is entered. The expressions may be of type **integer** or of type **real**. If all input values are of type **integer** the loop variable will also be of type **integer**. In all other cases the loop variable will be of type **real**. If the start value is less or equal to the termination value, the loop index is assigned with the starting value and the body of the loop is entered. If the increment is less than zero the loop is entered if the start value is larger or equal to the end value. Each time the body is executed, the loop index is incremented by the incrementation value. If the loop index is equal to the termination value, the body of the loop is performed for the last time. If the loop index is larger than the termination value the body will not be executed any longer. For negative increment values the loop is terminated if the loop index is less than the termination value.

Please note that it is not necessary, that the loop index has to be equal to the termination value before terminating the loop. The loop index is set to the termination value when the loop is being left. Please note, that the expressions for start and termination value are evaluated only once when entering the loop. A modification of a variable that appears within these expressions has no influence on the termination of the loop. The same applies to the modifications of the loop index. It also has no influence on the termination. The loop value is assigned to the correct value each time the **for** operator is executed.

If the **for** loop is left too early (e.g., if you press **Stop** and set the PC) and the loop is entered again, the expressions will be evaluated, as if the loop were entered for the first time.

Attention

For exported C++ please note the different semantics of the for loop.

Parameter

- ▷ **Start** (input_control) number $\leadsto$ *integer* / real
Start value for the loop variable.
Default Value : 1
- ▷ **End** (input_control) number $\leadsto$ *integer* / real
End value for the loop variable.
Default Value : 5
- ▷ **Step** (input_control) number $\leadsto$ *integer* / real
Increment value for the loop variable.
Default Value : 1
- ▷ **Variable** (output_control) number $\leadsto$ *integer* / real
Loop variable.

Example

```

dev_update_window ('off')
dev_close_window ()
dev_open_window (0, 0, 728, 512, 'black', WindowID)
read_image (Bond, 'die3')
dev_display (Bond)
stop ()
threshold (Bond, Bright, 100, 255)
shape_trans (Bright, Die, 'rectangle2')
dev_set_color ('green')
dev_set_line_width (3)
dev_set_draw ('margin')
dev_display (Die)
stop ()
reduce_domain (Bond, Die, DieGrey)
threshold (DieGrey, Wires, 0, 50)
fill_up_shape (Wires, WiresFilled, 'area', 1, 100)
dev_display (Bond)
dev_set_draw ('fill')
dev_set_color ('red')
dev_display (WiresFilled)
stop ()
opening_circle (WiresFilled, Balls, 15.5)
dev_set_color ('green')
dev_display (Balls)
stop ()
connection (Balls, SingleBalls)
select_shape (SingleBalls, IntermediateBalls, 'circularity', 'and', 0.85, 1.0)
sort_region (IntermediateBalls, FinalBalls, 'first_point', 'true', 'column')
dev_display (Bond)
dev_set_colored (12)
dev_display (FinalBalls)
stop ()
smallest_circle (FinalBalls, Row, Column, Radius)
NumBalls := |Radius|
Diameter := 2*Radius
meanDiameter := sum(Diameter)/NumBalls
minDiameter := min(Diameter)
dev_display (Bond)
disp_circle (WindowID, Row, Column, Radius)
dev_set_color ('white')
set_font (WindowID, 'system26')

```



```

for i := 1 to NumBalls by 1
    if (fmod(i,2)=1)
        set_tposition (WindowID, Row[i-1]-1.5*Radius[i-1], Column[i-1]-60)
    else
        set_tposition (WindowID, Row[i-1]+2.5*Radius[i-1], Column[i-1]-60)
    endif
    write_string (WindowID, 'Diam: '+Diameter[i-1])
endfor
dev_set_color ('green')
dev_update_window ('on')

```

Result

for returns 2 (H_MSG_TRUE) if the evaluation of the expression yields no error. **endfor** (as operator) always returns 2 (H_MSG_TRUE)

Parallelization Information

for is *reentrant*, *local*, and processed *without* parallelization.

Alternatives

while

See also

if, **ifelse**

Module

Basic operators

if (: : Condition :)

Conditional statement.

if is a conditional statement. The condition contains a boolean expression. If the condition is true, the body is executed. Otherwise the execution is continued at the first expression or operator call that follows the corresponding **endif**.

Parameter

- ▷ **Condition** (input_control) integer $\leadsto$ integer
 Condition for the if statement.
Default Value : 1

Result

if returns 2 (H_MSG_TRUE) if the evaluation of the expression yields no error. **endif** (as operators) always returns 2 (H_MSG_TRUE)

Parallelization Information

if is *reentrant*, *local*, and processed *without* parallelization.

Alternatives

ifelse

See also

for, **while**

Module

Basic operators

ifelse (: : Condition :)

Condition with alternative.

ifelse is a condition with an alternative. If the condition is true (i.e., not 0), all expressions and calls between the head and operator **endif** are performed. If the condition is false (i.e., 0) the part between **else** and **endif** is executed. Note that the operator is called **ifelse** and it is displayed as **if** in the program text area.

Parameter
▷ Condition (input_control) integer $\leadsto$ integer Condition for the if statement. Default Value : 1
Result
ifelse returns 2 (H_MSG_TRUE) if the evaluation of the expression yields no error. else and endif (as operators) always return 2 (H_MSG_TRUE)
Parallelization Information
ifelse is <i>reentrant</i>, <i>local</i>, and processed <i>without</i> parallelization.
Alternatives
if
See also
for, while
Module
Basic operators

insert (: : Input, Value, Index : Result)

Assignment of a value into a tuple.

insert assigns a single value into an tuple. If the first input parameter and the first output parameter are identical, the call:

```
insert (Areas, Area, Radius-1, Areas)
```

is not presented in the program text as an operator call, but in the more intuitive form as:

```
Areas[Radius-1] := Area
```

.

Parameter
▷ Input (input_control) real(-array) $\leadsto$ real / integer / string Tuple, where the new value has to be inserted. Default Value : '[]'
▷ Value (input_control) real $\leadsto$ real / integer / string Value that has to be inserted. Default Value : 1 Typical range of values : $0 \leq \text{Value} \leq 1000000$
▷ Index (input_control) integer $\leadsto$ integer Index position for new value. Default Value : 0 Suggested values : $\text{Index} \in \{0, 1, 2, 3, 4, 5, 6\}$ Minimum Increment : 1
▷ Result (output_control) real(-array) $\leadsto$ real / integer / string Result tuple with inserted values.
Result
insert returns 2 (H_MSG_TRUE) if the evaluation of the expression yields no error.
Parallelization Information
insert is <i>reentrant</i>, <i>local</i>, and processed <i>without</i> parallelization.
Alternatives
assign
Module
Basic operators

stop (: : :)

Stop program execution.

stop stops the program execution of HDevelop. The PC is then placed at the program line behind **stop**. The operator is equivalent the pressing the **stop** button in the menu bar.

Attention

stop is not available in C++.

Example

```
read_image (Image, 'fabrik')
regiongrowing (Image, Regions, 3, 3, 6, 100)
Number := |Regions|
dev_update_window ('off')
for i := 1 to Number by 1
    RegionSelected := Regions[i]
    dev_clear_window ()
    dev_display (RegionSelected)
    stop ()
endfor
```

Result

stop always returns 2 (H_MSG_TRUE)

Parallelization Information

stop is *reentrant*, *local*, and processed *without* parallelization.

See also

exit

Module

Basic operators

while (: : Condition :)

Continue to execute the body as long as the condition is true.

while executes the body as long as the condition is true. The while loop has a boolean expression as the conditional part. As long as it is **true** (i.e., not equal 0), the body of the loop is performed. In order to enter the loop, the condition has to be true in the first place.

Parameter

- ▷ **Condition** (input_control) integer $\leadsto$ integer
Condition for loop.

Example

```
dev_update_window ('off')
dev_close_window ()
dev_open_window (0, 0, 512, 512, 'black', WindowID)
read_image (Image, 'particle')
dev_display (Image)
stop ()
threshold (Image, Large, 110, 255)
dilation_circle (Large, LargeDilation, 7.5)
dev_display (Image)
dev_set_draw ('margin')
dev_set_line_width (3)
dev_set_color ('green')
```

```

dev_display (LargeDilation)
dev_set_draw ('fill')
stop ()
complement (LargeDilation, NotLarge)
reduce_domain (Image, NotLarge, ParticlesRed)
mean_image (ParticlesRed, Mean, 31, 31)
dyn_threshold (ParticlesRed, Mean, SmallRaw, 3, 'light')
opening_circle (SmallRaw, Small, 2.5)
connection (Small, SmallConnection)
dev_display (Image)
dev_set_colored (12)
dev_display (SmallConnection)
stop ()
dev_set_color ('green')
dev_display (Image)
dev_display (SmallConnection)
Button := 1
while (Button = 1)
    dev_set_color ('green')
    get_mbutton (WindowID, Row, Column, Button)
    dev_display (Image)
    dev_display (SmallConnection)
    dev_set_color ('red')
    select_region_point (SmallConnection, SmallSingle, Row, Column)
    dev_display (SmallSingle)
    NumSingle := |SmallSingle|
    if (NumSingle=1)
        intensity (SmallSingle, Image, MeanGray, DeviationGray)
        area_center (SmallSingle, Area, Row, Column)
        dev_set_color ('yellow')
        set_tposition (WindowID, Row, Column)
        write_string (WindowID, 'Area='+Area+', Int='+MeanGray)
    endif
endwhile
dev_set_line_width (1)
dev_update_window ('on')

```

Result

while returns 2 (H_MSG_TRUE) if the evaluation of the expression yields no error. **endwhile** (as operator) always returns 2 (H_MSG_TRUE)

Parallelization Information

while is *reentrant*, *local*, and processed *without* parallelization.

Alternatives

for

See also

if, **ifelse**

Module

Basic operators

Chapter 3

Develop

dev_clear_obj (Objects : : :)

Delete an iconic object from the HALCON database.

dev_clear_obj deletes iconic objects, which are no longer needed, from the HALCON database. It should be noted that **dev_clear_obj** cannot be exported to C++ due to the automatic memory management in C++.

Attention

Never use **clear_obj** to clear objects in HDevelop. The operator **dev_clear_obj** has to be used instead.

Parameter

▷ **Objects** (input_object) object(-array) $\leadsto$ *Hobject*
Objects to be deleted.

Result

dev_clear_obj returns 2 (H_MSG_TRUE) if the variable is instantiated. If necessary, an exception is raised.

Parallelization Information

dev_clear_obj is *local* and processed *completely exclusively* without parallelization.

See also

clear_obj, **test_obj_def**, **dev_set_check**, **reset_obj_db**

Module

Basic operators

dev_clear_window (: : :)

Clear the active graphics window.

dev_clear_window clears the graphics window content and the history of the active window. Parameters assigned to this window (e.g., with **dev_set_color**, **dev_set_draw**, etc.) remain unmodified. The operator is equivalent to pressing the **Clear** button of the active graphics window. A graphics window can be activated by calling **dev_set_window**.

Attention

If **dev_clear_window** should be used for exported Code (C++), please note the description of **clear_window** due to the different semantics in C++.

Example

```
read_image (Image, 'fabrik')
regiongrowing (Image, Regions, 3, 3, 6, 100)
Number := |Regions|
dev_update_window ('off')
for i := 1 to Number by 1
```

```

    RegionSelected := Regions[i]
    dev_clear_window ()
    dev_display (RegionSelected)
    * stop ()
endfor

```

Result

`dev_clear_window` always returns 2 (H_MSG_TRUE).

Parallelization Information

`dev_clear_window` is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

`dev_set_window`, `dev_open_window`, `dev_display`

Possible Successors

`dev_display`

See also

`clear_window`

Module

System

dev_close_inspect_ctrl (: : Variable :)

Close an inspect window of a control variable.

`dev_close_inspect_ctrl` is the opposite operator to `dev_inspect_ctrl`, and closes the inspect window corresponding to `Variable`. The window can also be closed by pressing the **Close**-button of the dialog.

Attention

This operator is not supported for exported C++ code.

Parameter

- ▷ **Variable** (input_control) real(-array) $\leadsto$ real / integer / string
 Name of the variable which inspect window has to be closed.

Example

```

Var := 1
dev_inspect_ctrl (Var)
Var := [1,2,3,9,5,6,7,8]
Var[3] := 4
stop
dev_close_inspect_ctrl (Var)

```

Result

If an inspect window associated with `Variable` is open `dev_close_inspect_ctrl` returns 2 (H_MSG_TRUE).

Parallelization Information

`dev_close_inspect_ctrl` is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

`dev_inspect_ctrl`

Module

Basic operators

dev_close_window (: : :)

Close the active graphics window.

[dev_close_window](#) closes the active graphics window which has been opened by [dev_open_window](#) or by HDevelop (default window). The operator is equivalent to pressing the **Close** button of the active window. A graphics window can be activated by calling [dev_set_window](#).

Attention

If [dev_close_window](#) should be used for exported Code (C++), please note the description of [close_window](#) due to the different semantics in C++.

Example

```
* close all windows
for i := 1 to 10 by 1
  dev_close_window ()
endfor
read_image (For5, 'for5')
get_image_pointer1 (For5, Pointer, Type, Width, Height)
dev_open_window (0, 0, Width, Height, 'black', WindowHandle)
dev_display (For5)
```

Result

[dev_close_window](#) always returns 2 (H_MSG_TRUE).

Parallelization Information

[dev_close_window](#) is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[dev_set_window](#), [dev_open_window](#)

Possible Successors

[dev_open_window](#)

See also

[close_window](#)

Module

System

dev_display (Object : : :)

Displays image objects in the current graphics window.

[dev_display](#) displays an image object (image, region, or XLD) in the active graphics window. This is equivalent to a double click on an icon variable inside the variable window.

Attention

If [dev_display](#) should be used for exported Code (C++), please note the description of [disp_obj](#) due to the different semantics in C++.

Parameter

- ▷ **Object** (input_object) object(-array) $\leadsto$ *Hobject*
 Image objects to be displayed.

Example

```
read_image (Image, 'fabrik')
regiongrowing (Image, Regions, 3, 3, 6, 100)
dev_clear_window ()
dev_display (Image)
dev_set_colored (12)
dev_set_draw ('margin')
dev_display (Regions)
```

Result

[dev_display](#) always returns 2 (H_MSG_TRUE)

	<i>Parallelization Information</i>	
<code>dev_display</code> is <i>local</i> and processed <i>completely exclusively</i> without parallelization.		
	<i>Alternatives</i>	
<code>disp_obj</code> , <code>disp_image</code> , <code>disp_region</code> , <code>disp_xld</code>		
	<i>See also</i>	
<code>dev_set_color</code> , <code>dev_set_colored</code> , <code>dev_set_draw</code> , <code>dev_set_line_width</code>		
	<i>Module</i>	
System		

dev_error_var (: : ErrorVar, Mode :)

Define or undefine an error variable.

`dev_error_var` defines an error variable, i.e., a variable which contains the status of the last call of an operator. `ErrorVar` will be `H_MSG_TRUE` (2) if no error had occurred. The parameter `Mode` specifies if the error variable should be used (1) or not (0). If an error variable is active it will be updated each an operator execution is finished. Thus a value is only valid until the next call of an operator. The value can be saved by assigning it to another variable (see example) or by calling `dev_error_var(ErrorVar, 0)`.

Attention
If `dev_error_var` should be used for exported Code (C++), please note the different handling of return values in C++.

	<i>Parameter</i>	
▷ ErrorVar (input_control)	integer(-array) $\leadsto$ integer	
Name of the variable which shall contain the error status.		
▷ Mode (input_control)	integer $\leadsto$ integer	
Switch the error variable on or off.		
Default Value : 1		
List of values : Mode $\in$ {0, 1}		

Example

```
dev_close_window ()
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
dev_error_var (Error, 1)
dev_set_check ('~give_error')
FileName := 'wrong_name'
read_image (Image, FileName)
ReadError := Error
if (ReadError # H_MSG_TRUE)
    write_string (WindowHandle, 'wrong file name: '+FileName)
endif
```

Result

`dev_error_var` always returns 2 (`H_MSG_TRUE`)

	<i>Parallelization Information</i>	
<code>dev_error_var</code> is <i>local</i> and processed <i>completely exclusively</i> without parallelization.		
	<i>Possible Predecessors</i>	
<code>dev_set_check</code>		
	<i>Possible Successors</i>	
<code>dev_set_check</code> , <code>if</code> , <code>ifelse</code> , <code>assign</code>		
	<i>See also</i>	
<code>set_check</code>		
	<i>Module</i>	
Basic operators		

dev_inspect_ctrl (: : Variable :)

Open a window to inspect a control variable.

dev_inspect_ctrl opens a dialog to check the contents of a control variable. This dialog has a scrolled list with all the values of the variable. In the case of an frame grabber handle a specific dialog is opened which displays the most important frame grabber parameters and can be used to switch the frame grabber only interactively. The contents of the dialog will be updated whenever the value(s) of variable changes. The update mode can be influenced by the operator **dev_update_var**. The dialog can be closed by pressing the Close-button or by calling **dev_close_inspect_ctrl**.

Attention

This operator is not supported for exported C++ code.

Parameter

- ▷ **Variable** (input_control) integer(-array) $\leadsto$ integer / real / string
Name of the variable to be checked.

Example

```
read_image (Image, 'fabrik')
regiongrowing (Image, Regions, 3, 3, 6, 100)
area_center (Regions, Area, Row, Column)
dev_inspect_ctrl (Area)
```

Result

dev_inspect_ctrl always returns 2 (H_MSG_TRUE)

Parallelization Information

dev_inspect_ctrl is *local* and processed *completely exclusively* without parallelization.

See also

dev_update_var

Module

Basic operators

dev_map_par (: : :)

Open the dialog to specify the display parameters.

dev_map_par opens the dialog which can also be accessed from the menu **Visualization** ▷ **Set Parameters** The dialog is used to configure the modes to display data like images, regions, or polygons.

Attention

This operator is not supported for exported C++ code.

Example

```
read_image (Image, 'fabrik')
threshold (Image, Region, 128, 255)
dev_map_par ( )
```

Result

dev_map_par always returns 2 (H_MSG_TRUE)

Parallelization Information

dev_map_par is *local* and processed *completely exclusively* without parallelization.

Possible Successors

dev_unmap_par

Module

Basic operators

dev_map_prog (: : :)

Make the main window of HDevelop visible.

[dev_map_prog](#) is used to map the main window of HDevelop after it has been unmapped by [dev_unmap_prog](#).

Attention

This operator is not supported for exported C++ code.

Depending on the operating system or the window manager the execution of [dev_map_prog](#) will result only in a visible icon of the window. In this case it has to be opened by the user with mouse interaction.

Result

[dev_map_prog](#) always returns 2 (H_MSG_TRUE)

Parallelization Information

[dev_map_prog](#) is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[dev_unmap_prog](#)

Possible Successors

[dev_unmap_prog](#)

See also

[dev_map_par](#), [dev_map_var](#)

Module

Basic operators

dev_map_var (: : :)

Map the variable window on the screen.

[dev_map_var](#) maps the variable window on the screen (i.e., makes it visible) that has been unmapped using [dev_unmap_var](#).

Attention

This operator is not supported for exported C++ code.

Result

[dev_map_var](#) always returns 2 (H_MSG_TRUE)

Parallelization Information

[dev_map_var](#) is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[dev_unmap_var](#)

Possible Successors

[dev_unmap_var](#)

See also

[dev_map_par](#), [dev_map_prog](#)

Module

Basic operators

dev_open_window (: : Row, Column, Width, Height, Background : WindowHandle)

Open a graphics window.

[dev_open_window](#) opens a new graphics window, which can be used to perform output of gray value data, regions, and graphics as well as to perform textual output. This new window automatically becomes active, which means that all output ([dev_display](#) and automatical display of operator results) is redirected to this window. This is shown by the green dot in the **Active** button.

In the case of the standard display operators (like `disp_image`, `disp_region`, `disp_line`, etc.) instead of `dev_display` the logical window number `WindowHandle` has to be used.

The background of the created window is set to the color specified in `Background`.

Pressing the `Clear` button clears the graphics window contents and the history of the window. This can also be achieved by using the operator `dev_clear_window`. You close a graphics window using the `Close` button of the window frame or by calling `dev_close_window`.

The origin of the graphics window is the upper left corner with the coordinates (0,0). The x values (column) increase from left to right, the y values increase from top to bottom. Normally, the coordinate system of the graphics window corresponds to the most recently displayed image, which is automatically zoomed so that every pixel of the image is visible. The coordinate system can be changed interactively using the menu `Visualization>Set Parameters>Zoom` or with the operator `dev_set_part`. Every time an image with a different size is displayed, the coordinate system will be adapted automatically.

Each window has a history which contains all

- objects and
- display parameters

which have been displayed or changed since the most recent clear action or display of a full image. This history is used for redrawing the contents of the window. Other output like text or general graphics like `disp_line` or `disp_circle` or iconic data that is displayed using HALCON operators like `disp_image` or `disp_region` are **not** part of the history, and are **not** redrawn. Only the object classes image, region, and XLD that are displayed with the HDevelop operator `dev_display` or by double clicking on an icon are part of the history.

You may change the size of the graphics window interactively by “gripping” the window border with the mouse. Then you can resize the window by dragging the mouse pointer. After this size modification the window content is redisplayed. Now you see the same part of the window with changed zoom.

If the mouse cursor is inside the window its look-up-table is reactivated. This is necessary if other programs use their own look-up table. Thus if there is a “strange” graphics window presentation, you may load the proper look-up table by placing the mouse inside the window.

Opening a window causes the assignment of a default font. It is used in connection with procedures like `write_string` and you may overwrite it by performing `set_font` after calling `dev_open_window`. On the other hand you have the possibility to specify a default font by calling `set_system('default_font', <Fontname>)` before opening a window (and all following windows; see also `query_font`).

If you want to specify display parameters for a window you may select the menu item `Visualization` in the menu bar. Here you can set the appropriate parameters by clicking the desired item. Parameters which you have set in this way are used for **all** windows (in contrast to standard windows opened with `open_window`). The effects of the new parameters will be applied directly to the **last** object of the window history and alter its parameters only.

Attention

Never use `close_window` to close an HDevelop graphics window. The operator `dev_close_window` has to be used instead.

If `dev_open_window` should be used for exported Code (C++), please note the description of `open_window` due to the different semantics in C++.

Parameter

- ▷ **Row** (input_control) rectangle.origin.y $\rightsquigarrow$ *integer*
 Row index of upper left corner.
Default Value : 0
Typical range of values : $0 \leq \text{Row}$
Minimum Increment : 1
Recommended Increment : 1
Restriction : $\text{Row} \geq 0$

- ▷ **Column** (input_control) rectangle.origin.x $\leadsto$ *integer*
Column index of upper left corner.
Default Value : 0
Typical range of values : $0 \leq \text{Column}$
Minimum Increment : 1
Recommended Increment : 1
Restriction : $\text{Column} \geq 0$
- ▷ **Width** (input_control) rectangle.extent.x $\leadsto$ *integer*
Width of the window.
Default Value : 256
Typical range of values : $0 \leq \text{Width}$
Minimum Increment : 1
Recommended Increment : 1
Restriction : $(\text{Width} > 0) \vee (\text{Width} = -1)$
- ▷ **Height** (input_control) rectangle.extent.y $\leadsto$ *integer*
Height of the window.
Default Value : 256
Typical range of values : $0 \leq \text{Height}$
Minimum Increment : 1
Recommended Increment : 1
Restriction : $(\text{Height} > 0) \vee (\text{Height} = -1)$
- ▷ **Background** (input_control) integer $\leadsto$ *integer* / *string*
Color of the background of the new window.
Default Value : "black"
- ▷ **WindowHandle** (output_control) window $\leadsto$ *integer*
Window identifier.

Example

```

dev_close_window ()
read_image (For5, 'for5')
get_image_pointer1 (For5, Pointer, Type, Width, Height)
dev_open_window (0, 0, Width, Height, 'black', WindowHandle)
dev_display (For5)
dev_set_lut ('rainbow')
dev_display (For5)
stop ()
dev_set_lut ('default')
dev_display (For5)
stop ()
dev_set_part (100, 100, 300, 300)
dev_display (For5)

```

Result

If the values of the specified parameters are correct [dev_open_window](#) returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Parallelization Information

[dev_open_window](#) is *local* and processed *completely exclusively* without parallelization.

Possible Successors

[dev_display](#), [dev_set_lut](#), [dev_set_color](#), [dev_set_draw](#), [dev_set_part](#)

Alternatives

[open_window](#)

See also

[query_color](#)

Module

System

dev_set_check (: : Mode :)

Specify the error handling.

dev_set_check specifies how HDevelop should react if an error occurs. If **Mode** has the value 'give_error' – which is the system default – HDevelop stops the program execution if an exception occurs and displays an error message. If you use '~give_error' the exception will be ignored and the program continues. **dev_set_check** is intended to be used in connection with **dev_error_var**, which allows to check for the result state of an operator.

Attention

If **dev_set_check** should be used for exported Code (C++), please note the description of **set_check** due to the different semantics in C++.

Parameter

▷ **Mode** (input_control) string ~> string
 Mode of error handling.
Default Value : 'give_error'

Example

```
dev_close_window ()
dev_open_window (0, 0, 512, 512, 'black', WindowHandle)
dev_error_var (Error, 1)
dev_set_check ('~give_error')
FileName := 'wrong_name'
read_image (Image, FileName)
dev_set_check ('give_error')
ReadError := Error
if (ReadError # H_MSG_TRUE)
  write_string (WindowHandle, 'wrong file name: '+FileName)
endif
* Now the program will stop with an exception
read_image (Image, FileName)
```

Result

dev_set_check always returns 2 (H_MSG_TRUE)

Parallelization Information

dev_set_check is *local* and processed *completely exclusively* without parallelization.

Possible Successors

dev_error_var

See also

set_check

Module

System

dev_set_color (: : ColorName :)

Set output color.

dev_set_color defines the color for region and line oriented output in the graphics windows. The available colors can be queried with the operator **query_color**. The "colors" 'black' and 'white' are available for all screens. If colors are used that are not displayable on the screen, HALCON can choose a similar, displayable color of the output. For this, **set_check('~color')** must be called.

The defined color is used until **dev_set_color** or **dev_set_colored** is called.

Colors are defined for all graphics windows in contrast to the operator **set_color**.

Attention

If **dev_set_color** should be used for exported Code (C++), please note the description of **set_color** due to the different semantics in C++.

Parameter
▷ ColorName (input_control) string(-array) $\leadsto$ string Output color names. Default Value : 'white' Suggested values : ColorName $\in$ {'white', 'black', 'gray', 'red', 'green', 'blue'}
Example
<pre>read_image(Image, 'mreut') dev_set_draw('fill') dev_set_color('red') threshold(Image, Region, 180, 255) dev_set_color('green') threshold(Image, Region, 0, 179)</pre>
Result
<code>dev_set_color</code> always returns 2 (H_MSG_TRUE)
Parallelization Information
<code>dev_set_color</code> is <i>local</i> and processed <i>completely exclusively</i> without parallelization.
Possible Predecessors
<code>dev_open_window</code>, <code>query_color</code>, <code>query_all_colors</code>
Possible Successors
<code>dev_display</code>
Alternatives
<code>dev_set_colored</code>
See also
<code>dev_set_draw</code>, <code>dev_set_line_width</code>, <code>set_color</code>
Module
System

dev_set_colored (: : NumColors :)

Set multiple output colors.

`dev_set_colored` allows the user to display a tuple of regions in different colors. `NumColors` defines the number of colors that are used. Valid values for `NumColors` can be queried with `query_colored`.

Attention
If `dev_set_colored` should be used for exported Code (C++), please note the description of `set_colored` due to the different semantics in C++.

Parameter
▷ NumColors (input_control) integer $\leadsto$ integer Number of output colors. Default Value : 6 List of values : NumColors $\in$ {3, 6, 12}
Example

```
read_image(Image, 'monkey')
threshold(Image, Region, 128, 255)
dev_set_colored(6)
connection(Region, Regions)
```

Result
<code>dev_set_colored</code> always returns 2 (H_MSG_TRUE)

Parallelization Information
dev_set_colored is <i>local</i> and processed <i>completely exclusively</i> without parallelization.
Possible Predecessors
dev_open_window
Possible Successors
dev_display
Alternatives
dev_set_color
See also
dev_set_draw , dev_set_line_width , set_colored
Module
System

dev_set_draw (: : DrawMode :)

Define the region fill mode.

[dev_set_draw](#) defines the region fill mode. If [DrawMode](#) is set to 'fill', output regions are filled, if set to 'margin', only contours are displayed. It is used by region output like [dev_display](#), [disp_region](#), [disp_circle](#), [disp_rectangle1](#), [disp_rectangle2](#), [disp_arrow](#), etc. If the mode is 'margin', the contour can be affected by [dev_set_line_width](#), [set_line_approx](#) and [set_line_style](#).

[Attention](#)
If [dev_set_draw](#) should be used for exported Code (C++), please note the description of [set_draw](#) due to the different semantics in C++.

Parameter
▷ DrawMode (input_control) string $\leadsto$ string Fill mode for region output. Default Value : 'fill' List of values : DrawMode $\in$ {'fill', 'margin'}
Example

```
read_image(Image, 'monkey')
threshold(Image, Region, 128, 255)
dev_clear_window
dev_set_color('red')
dev_set_draw('fill')
dev_display(Region)
dev_set_color('white')
dev_set_draw('margin')
dev_display(Region)
```

[Result](#)
[dev_set_draw](#) always returns 2 (H_MSG_TRUE)

Parallelization Information
dev_set_draw is <i>local</i> and processed <i>completely exclusively</i> without parallelization.
Possible Successors
dev_set_line_width , dev_display
See also
set_draw
Module
System

dev_set_line_width (: : Linewidth :)

Define the line width for region contour output.

dev_set_line_width defines the line width (in pixel) in which a region contour or lines are displayed (e.g., with **dev_display**, **disp_region**, **disp_line**, **disp_polygon**, etc.).

Attention

If **dev_set_line_width** should be used for exported Code (C++), please note the description of **set_line_width** due to the different semantics in C++.

Parameter

▷ **Linewidth** (input_control) integer $\leadsto$ integer
Line width for region output in contour mode.

Default Value : 1

Restriction : $\text{Linewidth} \geq 1$

Example

```
read_image(Image, 'monkey')
threshold(Image, Region, 128, 255)
dev_set_draw('margin')
dev_set_line_width(5)
dev_clear_window
dev_display(Region)
```

Result

dev_set_line_width always returns 2 (H_MSG_TRUE)

Parallelization Information

dev_set_line_width is *local* and processed *completely exclusively* without parallelization.

Possible Successors

dev_display

See also

set_line_width, **query_line_width**

Module

System

dev_set_lut (: : LutName :)

Set “look-up-table” (lut).

dev_set_lut sets look-up-table of the the output window. A look-up-table defines the transformation of a “gray value” within an image into a gray value or color on the screen. It describes the screen gray value/color as a combination of red, green and blue for any image gray value (0..255) (so it is a ‘table’ to ‘look up’ the screen gray value/color for each image gray value: look-up-table). Transformation into screen-colors is performed in real-time at every time the screen is displayed new (typically this happens about 60 - 70 times per second). So it is possible to change the look-up-table to get a new look of images or regions. Please remind that not all machines support changing the look-up-table (e.g., monochrome resp. truecolor).

For common monitors only one look-up-table can be loaded per screen. Whereas **dev_set_lut** can be activated separately for each window. There is the following solution for this problem: It will always be activated the look-up-table that is assigned to the “active window” (a window is set into the state “active” by placing the mouse inside the window).

look-up-tables can also be used with truecolor displays. In this case the look-up-table will be simulated in software. This means, that the look-up-table will be used each time an image is displayed.

query_lut lists the names of all look-up-tables.

Attention

If `dev_set_lut` should be used for exported Code (C++), please note the description of `set_lut` due to the different semantics in C++.

Parameter

▷ **LutName** (input_control) string $\leadsto$ *string*
 Name of look-up-table, values of look-up-table (RGB) or file name.
Default Value : 'default'
Suggested values : LutName $\in$ {'default', 'linear', 'inverse', 'sqr', 'inv_sqr', 'cube', 'inv_cube', 'sqrt', 'inv_sqrt', 'cubic_root', 'inv_cubic_root', 'color1', 'color2', 'color3', 'color4', 'three', 'six', 'twelve', 'twenty_four', 'rainbow', 'temperature', 'cyclic_gray', 'cyclic_temperature', 'hsi', 'change1', 'change2', 'change3'}

Example

```
read_image(Image, 'mreut')
dev_set_lut('inverse')
* For true color only:
dev_display(Image)
```

Result

`dev_set_lut` always returns 2 (H_MSG_TRUE)

Parallelization Information

`dev_set_lut` is *local* and processed *completely exclusively* without parallelization.

Possible Successors

`dev_display`

See also

`set_lut`

Module

System

dev_set_paint (: : Mode :)

Define the grayvalue output mode.

`dev_set_paint` defines the output mode for grayvalue display in the graphics window. The mode is used by `dev_display`.

This page describes the different modes, that can be used for grayvalue output. It should be noted, that the mode 'default' is the most suitable.

A different way to display grayvalues is the histogram (mode: 'histogram'). This mode has three additional parameter values: Row (second value) and column (third value). They denote row and column of the histogram center for positioning on the screen. The scale factor (fourth value) determines the histogram size: a scale factor of 1 distinguishes 256 grayvalues, 2 distinguishes 128 grevalues, and so on. The four values are passed as a tuple, e.g., ['histogram', 256, 256, 1]. If only the first value is passed ('histogram'), the other values are set to defaults or the last values, respectively. For histogram computation see [gray_histo](#).

The modes 'row' and 'column' allow to display gray values along rows or columns, respectively. The position (row and column index) is passed with the second parameter value. The third parameter value is the scale factor in percent (100 means 1 pixel per grayvalue, 50 means one pixel per two grayvalues).

Gray images can also be interpreted as 3d data, depending on the grayvalue. To view these 3d plots, select the modes 'contourline', '3D-plot' or '3D-plot.hidden'.

Parameters for modes that need more than one parameter can be passed the following ways:

- Only the name of the mode is passed: the defaults or the last values are used, respectively. Example: `dev_set_paint('contourline')`
- All values are passed: all output characteristics can be set. Example: `dev_set_paint(['contourline', 10, 1])`

- Only the first *n* values are passed: only the passed values are changed. Example: `dev_set_paint(['contourline',10])`

Attention

If `dev_set_paint` should be used for exported Code (C++), please note the description of `set_paint` due to the different semantics in C++.

Parameter

- ▷ **Mode** (input_control) string-array $\leadsto$ string / integer
 Grevalue output name. Additional parameters possible.
Default Value : 'default'
List of values : Mode $\in$ {'default', 'histogram', 'row', 'column', 'contourline', '3D-plot', '3D-plot_hidden', '3D-plot_point'}

Example

```
read_image(Image, 'fabrik')
dev_set_paint('3D-plot')
dev_display(Image)
```

Parallelization Information

`dev_set_paint` is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

`dev_open_window`

Possible Successors

`dev_set_color`, `dev_display`

See also

`set_paint`

Module

System

dev_set_part (: : Row1, Column1, Row2, Column2 :)

Modify the displayed image part.

`dev_set_part` modifies the image part that is displayed in the graphics window. (`Row1,Column1`) denotes the upper left corner and (`Row2,Column2`) the lower right corner of the image part to display.

If `Row1` is larger than `Row2` the zooming will be reset. That means that the last displayed image will be completely visible. Please note that this is not possible with the operator `set_part` outside HDevelop.

Attention

If `dev_set_part` should be used for exported Code (C++), please note the description of `set_part` due to the different semantics in C++.

Parameter

- ▷ **Row1** (input_control) rectangle.origin.y $\leadsto$ integer
 Row of the upper left corner of the chosen image part.
Default Value : 0
- ▷ **Column1** (input_control) rectangle.origin.x $\leadsto$ integer
 Column of the upper left corner of the chosen image part.
Default Value : 0
- ▷ **Row2** (input_control) rectangle.corner.y $\leadsto$ integer
 Row of the lower right corner of the chosen image part.
Default Value : 128
- ▷ **Column2** (input_control) rectangle.corner.x $\leadsto$ integer
 Column of the lower right corner of the chosen image part.
Default Value : 128

Example

```
read_image (Image, 'fabrik')
for i := 1 to 240 by 10
    dev_set_part (i, i, 511-i, 511-i)
    dev_display (Image)
endfor
dev_set_part (1, 1, -1, -1)
dev_display (Image)
```

Result

`dev_set_part` always returns 2 (H_MSG_TRUE)

Parallelization Information

`dev_set_part` is *local* and processed *completely exclusively* without parallelization.

Possible Successors

`dev_display`

See also

`set_part`

Module

System

dev_set_shape (: : Shape :)

Define the region output shape.

`dev_set_shape` defines the shape for region output. The output shape is used by `dev_display` for regions. The available shapes can be queried with `query_shape`.

Available modes:

'original': The shape is displayed unchanged. Nevertheless modifications via parameters like `dev_set_line_width` can take place. This is also true for all other modes.

'outer_circle': Each region is displayed by the smallest surrounding circle. (See `smallest_circle`.)

'inner_circle': Each region is displayed by the largest included circle. (See `inner_circle`.)

'ellipse': Each region is displayed by an ellipse with the same moments and orientation (See `elliptic_axis`.)

'rectangle1': Each region is displayed by the smallest surrounding rectangle parallel to the coordinate axes. (See `smallest_rectangle1`.)

'rectangle2': Each region is displayed by the smallest surrounding rectangle. (See `smallest_rectangle2`.)

'convex': Each region is displayed by its convex hull (See `shape_trans`.)

'icon' Each region is displayed by the icon set with `set_icon` in the center of gravity.

Attention

If `dev_set_shape` should be used for exported Code (C++), please note the description of `set_shape` due to the different semantics in C++.

Parameter

- ▷ **Shape** (input_control) string $\rightsquigarrow$ string
 Region output mode.
Default Value : 'original'
List of values : Shape $\in$ {'original', 'convex', 'outer_circle', 'inner_circle', 'rectangle1', 'rectangle2', 'ellipse', 'icon'}

Example

```
read_image(Image, 'monkey')
threshold(Image, Region, 128, 255)
connection(Region, Regions)
dev_set_shape('rectangle1')
dev_set_draw('margin')
dev_display(Regions)
```

Parallelization Information

`dev_set_shape` is *local* and processed *completely exclusively* without parallelization.

Possible Successors

`dev_display`, `dev_set_color`

See also

`set_shape`, `dev_set_line_width`

Module

System

dev_set_window (: : WindowID :)

Activate a graphics window.

`dev_set_window` activates a graphics window. This is equivalent to pressing the **Active** button of the graphics window.

Attention

If `dev_set_window` should be used for exported Code (C++), please note the different handling of windows in C++.

`dev_set_window` is not supported for C++.

Parameter

▷ **WindowID** (input_control) window $\leadsto$ integer
Window_id.

Example

```
dev_open_window (1, 1, 200, 200, 'black', WindowID1)
dev_open_window (1, 220, 200, 200, 'black', WindowID2)
read_image(Image, 'monkey')
dev_set_window(WindowID1)
dev_display(Image)
dev_set_window(WindowID2)
dev_display(Image)
```

Parallelization Information

`dev_set_window` is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

`dev_open_window`

Possible Successors

`dev_display`

Module

Basic operators

dev_set_window_extents (: : Row, Column, Width, Height :)

Change position and size of a graphics window.

`dev_set_window_extents` changes the position and/or the size of the currently active graphics window.

The parameters `Row` and `Column` specify the new position (upper left corner) of the window. If one of both values is negative, the position will remain unchanged. The parameters `Width` and `Height` specify the new size of the window. This is the size of the inner part that actually displays the data. If one of the two values is negative, the size will remain unchanged.

Attention

Never use `set_window_extents` to change the size and position of an HDevelop graphics window. The operator `dev_set_window_extents` has to be used instead.

Parameter

- ▷ **Row** (input_control) rectangle.origin.y $\leadsto$ *integer*
 Row index of upper left corner.
Default Value : 0
Typical range of values : $0 \leq \text{Row}$
Minimum Increment : 1
Recommended Increment : 1
Restriction : $(\text{Row} \geq 0) \vee (\text{Row} = -1)$
- ▷ **Column** (input_control) rectangle.origin.x $\leadsto$ *integer*
 Column index of upper left corner.
Default Value : 0
Typical range of values : $0 \leq \text{Column}$
Minimum Increment : 1
Recommended Increment : 1
Restriction : $(\text{Column} \geq 0) \vee (\text{Column} = -1)$
- ▷ **Width** (input_control) rectangle.extent.x $\leadsto$ *integer*
 Width of the window.
Default Value : 256
Typical range of values : $0 \leq \text{Width}$
Minimum Increment : 1
Recommended Increment : 1
Restriction : $(\text{Width} > 0) \vee (\text{Width} = -1)$
- ▷ **Height** (input_control) rectangle.extent.y $\leadsto$ *integer*
 Height of the window.
Default Value : 256
Typical range of values : $0 \leq \text{Height}$
Minimum Increment : 1
Recommended Increment : 1
Restriction : $(\text{Height} > 0) \vee (\text{Height} = -1)$

Example

```
dev_close_window ()
read_image (For5, 'for5')
get_image_pointer1 (For5, Pointer, Type, Width, Height)
dev_open_window (0, 0, Width, Height, 'black', WindowHandle)
dev_display (For5)
stop ()
dev_set_window_extents (-1, -1, Width/2, Height/2)
dev_display (For5)
stop ()
dev_set_window_extents (200, 200, -1, -1)
```

Result

If the values of the specified parameters are correct `dev_set_window_extents` returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Parallelization Information

`dev_set_window_extents` is *local* and processed *completely exclusively* without parallelization.

Possible Successors

`dev_display`, `dev_set_lut`, `dev_set_color`, `dev_set_draw`, `dev_set_part`

[set_window_extents](#) *See also*

Module

System

dev_unmap_par (: : :)

Hide the window for the graphic parameters.

[dev_unmap_par](#) hides the window for the graphic parameters so that it is no longer visible. It can be mapped again using the operator [dev_map_par](#).

Attention

This operator is not supported for exported C++ code.

Result

[dev_unmap_par](#) always returns 2 (H_MSG_TRUE)

Parallelization Information

[dev_unmap_par](#) is *local* and processed *completely exclusively* without parallelization.

Possible Successors

[dev_map_prog](#)

See also

[dev_map_par](#), [dev_map_prog](#), [dev_map_var](#)

Module

Basic operators

dev_unmap_prog (: : :)

Hide the main window.

[dev_unmap_prog](#) hides the main window so that it is no longer visible. It can be mapped again using the operator [dev_map_prog](#).

Attention

This operator is not supported for exported C++ code.

Result

[dev_unmap_prog](#) always returns 2 (H_MSG_TRUE)

Parallelization Information

[dev_unmap_prog](#) is *local* and processed *completely exclusively* without parallelization.

Possible Successors

[dev_map_prog](#), [stop](#)

See also

[dev_map_par](#), [dev_map_prog](#), [dev_map_var](#)

Module

Basic operators

dev_unmap_var (: : :)

Hide the variable window.

[dev_unmap_var](#) hides the variable window so that it is no longer visible. It can be mapped again using the operator [dev_map_var](#).

Attention

This operator is not supported for exported C++ code.

Result

[dev_unmap_var](#) always returns 2 (H_MSG_TRUE)

Parallelization Information

[dev_unmap_var](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Successors

[dev_map_var](#)

See also

[dev_map_par](#), [dev_map_prog](#)

Module

Basic operators

dev_update_pc (: : DisplayMode :)

Specify the behaviour of the PC during program execution.

[dev_update_pc](#) specifies the behaviour of the PC during program execution. In the mode 'on' (default) the PC is always displayed in front of the current operator. In addition the program text is scrolled – if necessary – so that the current operator is visible. In the mode 'off' the PC is not visible during program execution and the program text will not be scrolled automatically.

This option can also be controled by the dialog

File ▷ Options ▷ Update PC.

Attention

This operator is not supported for exported C++ code.

Parameter

- ▷ **DisplayMode** (input_control) string $\leadsto$ *string*
 Mode for runtime behaviour.
Default Value : 'off'
List of values : DisplayMode $\in$ {'on', 'off'}

Result

[dev_update_pc](#) always returns 2 (H_MSG_TRUE)

Parallelization Information

[dev_update_pc](#) is *reentrant*, *local*, and processed *without* parallelization.

See also

[dev_update_time](#), [dev_update_window](#), [dev_update_var](#)

Module

Basic operators

dev_update_time (: : DisplayMode :)

Switch time measurement for operators on or off.

[dev_update_time](#) controls if the execution time of an operator has to be measured.

This option can also be controled by the dialog

File ▷ Options ▷ Show Processing Time.

Attention

This operator is not supported for exported C++ code.

Parameter
▷ DisplayMode (input_control) string $\leadsto$ string Mode for graphic output. Default Value : 'off' List of values : DisplayMode $\in$ {'on', 'off'}
Result
dev_update_time always returns 2 (H_MSG_TRUE)
Parallelization Information
dev_update_time is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.
See also
dev_update_pc , dev_update_window , dev_update_var
Module
Basic operators

dev_update_var (: : DisplayMode :)

Specify the behaviour of the variable window during program execution.

dev_update_var specifies the behaviour of the variable window during program execution. Using the mode 'on' (default) the contents of the variable window (iconic and control variables) is updated each time a variable is modified by the program. In the mode 'off' the variables are updated only when the execution is finished. Please not that update in this contents only means the graphical representation of the internal values in the variable window.

This option can also be controlled by the dialog

File ▷ Options ▷ Update Variables.

Attention
This operator is not supported for exported C++ code.
Parameter
▷ DisplayMode (input_control) string $\leadsto$ string Mode for graphic output. Default Value : 'off' List of values : DisplayMode $\in$ {'on', 'off'}
Result
dev_update_var always returns 2 (H_MSG_TRUE)
Parallelization Information
dev_update_var is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.
See also
dev_update_pc , dev_update_window , dev_update_time
Module
Basic operators

dev_update_window (: : DisplayMode :)

Specify the output behaviour during program execution.

dev_update_window specifies the output behaviour during program execution. By default every object (image, region, or XLD) is displayed in the active graphics window. This can be changed by using the value 'off' for **DisplayMode**. In this case objects are only displayed in single step mode. Here one would use the operator **dev_display** to output objects.

This option can also be controlled by the dialog

File ▷ Options ▷ Update Window.

<i>Attention</i>	
This operator is not supported for exported C++ code.	
<i>Parameter</i>	
▷ DisplayMode (input_control)	string $\leadsto$ <i>string</i>
Mode for graphic output.	
Default Value : 'off'	
List of values : DisplayMode $\in$ {'on', 'off'}	
<i>Result</i>	
dev_update_window always returns 2 (H_MSG_TRUE)	
<i>Parallelization Information</i>	
dev_update_window is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.	
<i>Possible Successors</i>	
dev_display	
<i>See also</i>	
dev_update_pc , dev_update_var , dev_update_time	
<i>Module</i>	
Basic operators	

Chapter 4

File

4.1 Images

read_image (: Image : FileName :)

Read an image with different file formats.

The operator **read_image** reads the indicated image files from the background storage and generates the image. One or more files can be indicated. The region of the generated image object (= all pixels of the matrix) is maximal chosen.

All images files written by the operator **write_image** (format 'ima') have the extension '.ima'. A description file can be available for every image in HALCON format (same file name with extension '.exp'). The type of the pixel data (*byte*, *int4*, *real*) can also be taken from the description file. If this information is not available the type *byte* is used as presetting.

Besides the HALCON format TIFF, GIF, BMP, JPEG, PNG, PCX, SUN-Raster, PGM, PPM, PBM and XWD files can also be read. The gray values of PBM images are set at the values 0 and 255. The file formats are either recognized by the extension (if indicated) or because of the internal structure of the files. If the extension is indicated the image can be found faster. In case of PGM, PPM and PBM the corresponding extension (e.g. 'pgm') or the general value 'pnm' can be used. In case of TIFF 'tiff' and 'tif' are accepted. In case of colored images an image with three color channels (matrices) is created, the red channel being stored in the first, the blue channel in the second and the green channel in the third component (channel number).

Image files are searched in the current directory (determined by the environment variable) and in the image directory of HALCON. The image directory of HALCON is preset at '.' and '/usr/local/halcon/images' in a UNIX environment and can be set via the operator **set_system**. More than one image directory can be indicated. This is done by separating the individual directories by a colon.

Furthermore the search path can be set via the environment variable HALCONIMAGES (same structure as 'image_dir'). Example:

```
setenv HALCONIMAGES "/usr/images:/usr/local/halcon/images"\\*
```

HALCON also searches images in the subdirectory "'images'" (Images for the program examples). The environment variable HALCONROOT is used for the HALCON directory.

Attention

If CMYK or YCCK JPEG files are read, HALCON assumes that these files follow the Adobe Photoshop convention that the CMYK channels are stored inverted, i.e., 0 represents 100% ink coverage, rather than 0% ink as one would expect. The images are converted to RGB images using this convention. If the JPEG file does not follow this convention, but stores the CMYK channels in the usual fashion, **invert_image** must be called after reading the image.

If PNG images that contain an alpha channel are read, the alpha channel is returned as the second or fourth channel of the output image, unless the alpha channel contains exactly two different gray values, in which case a one or

three channel image with a reduced domain is returned, in which the points in the domain correspond to the points with the higher gray value in the alpha channel.

Parameter

- ▷ **Image** (output_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Read image.
- ▷ **FileName** (input_control) filename(-array) $\leadsto$ *string*
Name of the image to be read.
Default Value : 'fabrik'
Suggested values : **FileName** $\in$ {'monkey', 'fabrik', 'mreut'}

Example

```
/* Reading an image: */
read_image(Image, 'monkey').

/* Reading 3 images into an image object: */
read_image(Bildobjekt, ['house_red', 'house_green', 'house_blue']).

/* Setting of search path for images on '/mnt/images' and '/home/images':
*/
set_system('image_dir', '/mnt/images:/home/images').
```

Result

If the parameters are correct the operator **read_image** returns the value 2 (H_MSG_TRUE). If the indicated files cannot be found **read_image** returns the value 5 (H_MSG_FAIL). Otherwise an exception handling is raised.

Parallelization Information

read_image is processed under *mutual exclusion* against itself and without parallelization.

Possible Successors

disp_image, **threshold**, **regiongrowing**, **count_channels**, **decompose3**,
class_ndim_norm, **gauss_image**, **fill_interlace**, **zoom_image_size**, **zoom_image_factor**,
crop_part, **write_image**, **rgb1_to_gray**

Alternatives

read_sequence

See also

set_system, **write_image**

Module

Image / region / XLD management

read_sequence (: Image : HeaderSize, SourceWidth, SourceHeight,
StartRow, StartColumn, DestWidth, DestHeight, PixelType, BitOrder,
ByteOrder, Pad, Index, FileName :)

Read images.

The operator **read_sequence** reads unformatted image data, from a file and returns a “suitable” image. The image data must be filled consecutively pixel by pixel and line by line.

Any file headers (with the length **HeaderSize** bytes) are skipped. The parameters **SourceWidth** and **SourceHeight** indicate the size of the filled image. **DestWidth** and **DestHeight** indicate the size of the image. In the simplest case these parameters are the same. However, areas can also be read. The upper left corner of the required image area can be determined via **StartRow** and **StartColumn**.

The pixel types 'bit', 'byte', 'short' (16 bits, unsigned), 'signed_short' (16 bits, signed), 'long' (32 bits, signed), 'swapped_long' (32 bits, with swapped segments), and 'real' (32 bit floating point numbers) are supported. Furthermore, the operator **read_sequence** enables the extraction of components of a RGB image, if a triple of three bytes (in the sequence “red”, “green”, “blue”) was filed in the image file. For the red component the pixel type 'r_byte' must be chosen, and correspondingly for the green and blue components 'g_byte' or 'b_byte', respectively.

'*MSBFirst*' (most significant bit first) or the inversion thereof ('*LSBFirst*') can be chosen for the bit order (*BitOrder*). The byte orders (*ByteOrder*) '*MSBFirst*' (most significant byte first) or '*LSBFirst*', respectively, are processed analogously. Finally an alignment (*Pad*) can be set at the end of the line: '*byte*', '*short*' or '*long*'. If a whole image sequence is stored in the file a single image (beginning at Index 1) can be chosen via the parameter *Index*.

Image files are searched in the current directory (determined by the environment variable) and in the image directory of HALCON. The image directory of HALCON is preset at '.' and '/usr/local/halcon/images' in a UNIX environment and can be set via the operator *set_system*. More than one image directory can be indicated. This is done by separating the individual directories by a colon.

Furthermore the search path can be set via the environment variable HALCONIMAGES (same structure as '*image_dir*'). Example:

```
setenv HALCONIMAGES "/usr/images:/usr/local/halcon/images"\\*
```

HALCON also searches images in the subdirectory "'images'" (Images for the program examples). The environment variable HALCONROOT is used for the HALCON directory.

Attention

If files of pixel type '*real*' are read and the byte order is chosen incorrectly (i.e., differently from the byte order in which the data is stored in the file) program error and even crashes because of floating point exceptions may result.

Parameter

- ▷ **Image** (output_object) image $\leadsto$ *Hobject* : byte / int2 / uint2 / int4
Image read.
- ▷ **HeaderSize** (input_control) integer $\leadsto$ *integer*
Number of bytes for file header.
Default Value : 0
Typical range of values : $0 \leq \text{HeaderSize}$
- ▷ **SourceWidth** (input_control) extent.x $\leadsto$ *integer*
Number of image columns of the filed image.
Default Value : 512
Typical range of values : $1 \leq \text{SourceWidth}$
- ▷ **SourceHeight** (input_control) extent.y $\leadsto$ *integer*
Number of image lines of the filed image.
Default Value : 512
Typical range of values : $1 \leq \text{SourceHeight}$
- ▷ **StartRow** (input_control) point.y $\leadsto$ *integer*
Starting point of image area (line).
Default Value : 0
Typical range of values : $0 \leq \text{StartRow}$
Restriction : $\text{StartRow} < \text{SourceHeight}$
- ▷ **StartColumn** (input_control) point.x $\leadsto$ *integer*
Starting point of image area (column).
Default Value : 0
Typical range of values : $0 \leq \text{StartColumn}$
Restriction : $\text{StartColumn} < \text{SourceWidth}$
- ▷ **DestWidth** (input_control) extent.x $\leadsto$ *integer*
Number of image columns of output image.
Default Value : 512
Typical range of values : $1 \leq \text{DestWidth}$
Restriction : $\text{DestWidth} \leq \text{SourceWidth}$
- ▷ **DestHeight** (input_control) extent.y $\leadsto$ *integer*
Number of image lines of output image.
Default Value : 512
Typical range of values : $1 \leq \text{DestHeight}$
Restriction : $\text{DestHeight} \leq \text{SourceHeight}$

- ▷ **PixelFormat** (input_control) string $\leadsto$ string
Type of pixel values.
Default Value : 'byte'
List of values : `PixelFormat` $\in$ {'bit', 'byte', 'r_byte', 'g_byte', 'b_byte', 'short', 'signed_short', 'long', 'swapped_long'}
- ▷ **BitOrder** (input_control) string $\leadsto$ string
Sequence of bits within one byte.
Default Value : 'MSBFirst'
List of values : `BitOrder` $\in$ {'MSBFirst', 'LSBFirst'}
- ▷ **ByteOrder** (input_control) string $\leadsto$ string
Sequence of bytes within one 'short' unit.
Default Value : 'MSBFirst'
List of values : `ByteOrder` $\in$ {'MSBFirst', 'LSBFirst'}
- ▷ **Pad** (input_control) string $\leadsto$ string
Data units within one image line (alignment).
Default Value : 'byte'
List of values : `Pad` $\in$ {'byte', 'short', 'long'}
- ▷ **Index** (input_control) integer $\leadsto$ integer
Number of images in the file.
Default Value : 1
Typical range of values : $1 \leq \text{Index}(\text{lin})$
- ▷ **FileName** (input_control) filename $\leadsto$ string
Name of input file.

Result

If the parameter values are correct the operator `read_sequence` returns the value 2 (`H_MSG_TRUE`). If the file cannot be opened 5 (`H_MSG_FAIL`) is returned. Otherwise an exception handling is raised.

Parallelization Information

`read_sequence` is processed under *mutual exclusion* against itself and without parallelization.

Possible Successors

`disp_image`, `count_channels`, `decompose3`, `write_image`, `rgb1_to_gray`

Alternatives

`read_image`

See also

`read_image`

Module

Image / region / XLD management

write_image (Image : : Format, FillColor, FileName :)

Write images in graphic formats.

The operator `write_image` returns the indicated image (`Image`) in different image formats in files. Pixels outside the region receive the color defined by `FillColor`. For gray value images a value between 0 (black) and 255 (white) must be passed, with RGB color images the RGB values can be passed directly as a hexadecimal value: e.g., 0xffff00 for a yellow background (red=255, green=255, blue=0).

The following formats are currently supported:

- 'tiff' TIFF format, 3-channel-images (RGB): 3 samples per pixel; other images (grayvalue images): 1 sample per pixel, 8 bits per sample, uncompressed, 72 dpi; file extension: *.tif
- 'bmp' Windows-BMP format, 3-channel-images (RGB): 3 bytes per pixel; other images (gray value image): 1 byte per pixel; file extension: *.bmp
- 'jpeg' JPEG format, with lost of information; together with the format string the quality value determining the compression rate can be provided: e.g., 'jpeg 30'. Attention: images stored for being processed later should not be compressed with the jpeg format according to the lost of information.

'png' PNG format (lossless compression); together with the format string, a compression level between 0 and 9 can be specified, where 0 corresponds to no compression and 9 to the best possible compression. Alternatively, the compression can be selected with the following strings: 'best', 'fastest', and 'none'. Hence, examples for correct parameters are 'png', 'png 7', and 'png none'. Images of type byte and uint2 can be stored in PNG files. If an image with a reduced domain is written, the region is stored as the alpha channel, where the points within the domain are stored as the maximum gray value of the image type and the points outside the domain are stored as the gray value 0. If an image with a full domain is written, no alpha channel is stored.

'ima' The data is written binary line by line (without header or carriage return). The size of the image and the pixel type are stored in the description file `""FileName.exp""`. *byte*, *int2*, *uint2*, *int4* and *real* images can be written. The file extension is: '.ima'

Parameter
▷ Image (input_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real Output image(s). ▷ Format (input_control) string $\leadsto$ <i>string</i> Graphic format. Default Value : 'tiff' List of values : Format $\in$ {'tiff', 'bmp', 'jpeg', 'ima', "jpeg 100", "jpeg 80", "jpeg 60", "jpeg 40", "jpeg 20", "png", "png best", "png fastest", "png none"} ▷ FillColor (input_control) integer $\leadsto$ <i>integer</i> Fill gray value for pixels not belonging to the image region. Default Value : 0 Suggested values : FillColor $\in$ {-1, 0, 255, '0xff0000', '0xff00'} ▷ FileName (input_control) filename(-array) $\leadsto$ <i>string</i> Name of graphic file.

Result
If the parameter values are correct the operator `write_image` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised. If the file cannot be opened `write_image` returns 5 (H_MSG_FAIL).

Parallelization Information
`write_image` is processed under *mutual exclusion* against itself and without parallelization.

Possible Predecessors
`open_window`, `read_image`

Module
Image / region / XLD management

4.2 Misc

file_exists (: : FileName :)

Check whether file exists.

The operator `file_exists` checks whether the indicated file already exists. If this is the case, the operator `file_exists` returns TRUE, otherwise FALSE.

Parameter
▷ FileName (input_control) filename $\leadsto$ <i>string</i> Name of file to be checked. Default Value : '/bin/cc'

Result
The operator `file_exists` returns the value 2 (H_MSG_TRUE) (file exists) or 3 (H_MSG_FALSE) (file does not exist).

Parallelization Information
`file_exists` is *reentrant* and processed *without* parallelization.

Possible Successors
`open_file`

Alternatives

[open_file](#)

Module

Basic operators

read_world_file (: : FileName : WorldTransformation)

Read the geo coding from an ARC/INFO world file.

[read_world_file](#) reads a geocoding from an ARC/INFO world file with the file name [FileName](#) and returns it as a homogeneous 2D transformation matrix in [WorldTransformation](#). To find the file [FileName](#), all directories contained in the HALCON system variable 'image_dir' (usually this is the content of the environment variable HALCONIMAGES) are searched (see [read_image](#)). This transformation matrix can be used to transform XLD contours to the world coordinate system before writing them with [write_contour_xld_arc_info](#). If the matrix [WorldTransformation](#) is inverted by calling [hom_mat2d_invert](#), the resulting matrix can be used to transform contours that have been read with [read_contour_xld_arc_info](#) to the image coordinate system.

Parameter

- ▷ **FileName** (input_control) filename $\leadsto$ *string*
Name of the ARC/INFO world file.
- ▷ **WorldTransformation** (output_control) affine2d-array $\leadsto$ *real*
Transformation matrix from image to world coordinates.
Number of elements : 6

Result

If the parameters are correct and the world file could be read, the operator [read_world_file](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Parallelization Information

[read_world_file](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[hom_mat2d_invert](#), [affine_trans_contour_xld](#), [affine_trans_polygon_xld](#)

See also

[write_contour_xld_arc_info](#), [read_contour_xld_arc_info](#), [write_polygon_xld_arc_info](#), [read_polygon_xld_arc_info](#)

Module

Sub-pixel operators

4.3 Region

read_region (: Region : FileName :)

Read binary images or HALCON regions.

The operator [read_region](#) reads regions from a binary file. The data is stored in packed form.

Tiff: Binary Tiff images with extension 'tiff' or 'tif'. The result is always *one* region. The color black is used as foreground.

BMP: Binary Windows bitmap images with extension 'bmp'. The result is always *one* region. The color black is used as foreground.

HALCON regions: File format of HALCON for regions. Several images can be stored (in one file) or read simultaneously via the operators [write_region](#) and [read_region](#). All region files have the extension '.reg', which is not indicated when reading or writing the file.

A search path ('image_dir') can be defined analogous to the operator [read_image](#).

Parameter

- ▷ **Region** (output_object) region(-array) $\leadsto$ *Hobject*
Read region.
- ▷ **FileName** (input_control) filename $\leadsto$ *string*
Name of the region to be read.

Example

```
/* Reading of regions and giving them gray values. */
read_image(Img, 'bild_test5')
read_region(Regs, 'reg_test5')
reduce_domain(Img, Regs, Res)
```

Result

If the parameter values are correct the operator [read_region](#) returns the value 2 (H_MSG_TRUE). If the file cannot be opened 5 (H_MSG_FAIL) is returned. Otherwise an exception handling is raised.

Parallelization Information

[read_region](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[read_image](#)

Possible Successors

[reduce_domain](#), [disp_region](#)

See also

[write_region](#), [read_image](#)

Module

Image / region / XLD management

write_region (Region : : FileName :)

Write regions on file.

The operator [write_region](#) writes the regions of the input images (in runlength coding) to a binary file. The data is stored in packed form. The output data can be read via the operator [read_region](#). If no extension has been specified in [FileName](#), the extension '.reg' is appended to [FileName](#).

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Region of the images which are returned.
- ▷ **FileName** (input_control) filename $\leadsto$ *string*
Name of region file.
Default Value : 'region.reg'

Example

```
regiongrowing(Img, Segmente, 3, 3, 5, 10)
write_region(Segmente, 'result1').
```

Result

If the parameter values are correct the operator [write_region](#) returns the value 2 (H_MSG_TRUE). If the file cannot be opened 5 (H_MSG_FAIL) is returned. Otherwise an exception handling is raised.

Parallelization Information

[write_region](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[open_window](#), [read_image](#), [read_region](#), [threshold](#), [regiongrowing](#)

	See also
read_region	
	Module
Image / region / XLD management	

4.4 Text

close_all_files (: : :)

Close all open files.

[close_all_files](#) closes all open files.

	Attention
Since all files are closed by close_all_files all handles become invalid.	
	Result
If it is possible to close the files the operator close_all_files returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.	

	Parallelization Information
close_all_files is <i>local</i> and processed <i>completely exclusively</i> without parallelization.	
	Alternatives
close_file	
	Module
Basic operators	

close_file (: : FileHandle :)

Closing a text file.

The operator [close_file](#) closes a file which was opened via the operator [open_file](#).

	Parameter
▷ FileHandle (input_control) file $\leadsto$ integer File handle.	
	Example

```
open_file('/tmp/data.txt', 'input', FileHandle)
// ....
close_file(FileHandle).
```

	Result
If the file handle is correct close_file returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.	
	Parallelization Information
close_file is <i>local</i> and processed <i>completely exclusively</i> without parallelization.	
	Possible Predecessors
open_file	
	See also
open_file	
	Module
Basic operators	

fnew_line (: : FileHandle :)

Create a line feed.

The operator **fnew_line** puts out a line feed into the output file. At the same time the output buffer is cleaned.

Parameter

- ▷ **FileHandle** (input_control) file $\leadsto$ integer
File handle.

Example

```
fwrite_string(FileHandle, 'Good Morning')
fnew_line(FileHandle)
```

Result

If an output file is open and it can be written to the file the operator **fnew_line** returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

fnew_line is *reentrant* and processed *without* parallelization.

Possible Predecessors

fwrite_string

See also

fwrite_string

Module

Basic operators

fread_char (: : FileHandle : Char)

Read a character from a text file.

The operator **fread_char** reads a character from the current input file. **fread_char** returns the character sequence 'eof'. At the end of a line the value 'nl' is returned.

Parameter

- ▷ **FileHandle** (input_control) file $\leadsto$ integer
File handle.
- ▷ **Char** (output_control) string $\leadsto$ string
Read character or control string ('nl','eof').

Example

```
repeat >
  fread_char(FileHandle:Char)
  (if(Char = 'nl') > fnew_line(FileHandle)) |
  (if(Char != 'nl') > fwrite_string(FileHandle, Char))
until(Char = 'eof').
```

Result

If an input file is open the operator **fread_char** returns 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

fread_char is *reentrant* and processed *without* parallelization.

Possible Predecessors

open_file

	Possible Successors
<code>close_file</code>	
	Alternatives
<code>fread_string</code> , <code>read_string</code>	
	See also
<code>open_file</code> , <code>close_file</code> , <code>fread_string</code>	
	Module
Basic operators	

fread_string (:: FileHandle : OutString, IsEOF)

Read strings from a text file.

The operator `fread_string` reads a string from the current input file. A string begins with the first representable character: letters, numbers, additional characters (except blanks). A string ends when a blank or a line skip is reached. Several successive line skips are ignored. If the end of the file is reached `IsEOF` return the value 1, otherwise 0.

	Parameter
▷ FileHandle (input_control) file ~ integer	File handle.
▷ OutString (output_control) string ~ string	Read character sequence.
▷ IsEOF (output_control) integer ~ integer	Reached end of file.
	Example

```
fwrite_string(FileHandle, 'Please enter text and return: ..')
fread_string(FileHandle, String, IsEOF) >
fwrite_string(FileHandle, ['here it is again: ', String])
fnew_line(FileHandle).
```

	Result
If a file is open and a suitable string is read <code>fread_string</code> returns the value 2 (H.MSG_TRUE). Otherwise an exception handling is raised.	

	Parallelization Information
<code>fread_string</code> is <i>reentrant</i> and processed <i>without</i> parallelization.	
	Possible Predecessors
<code>open_file</code>	
	Possible Successors
<code>close_file</code>	
	Alternatives
<code>fread_char</code> , <code>read_string</code>	
	See also
<code>open_file</code> , <code>close_file</code> , <code>fread_char</code>	
	Module
Basic operators	

fwrite_string (:: FileHandle, String :)

Write values in a text file.

The operator `fwrite_string` puts out a string or numbers on the output file. The operator `open_file` opens a file. The call `set_system(::'flush_file', <boolean-value>:)` determines whether the output characters are put out directly on the output medium. If the value `'flush_file'` is set to `'false'` the characters (especially in case of screen output) show up only after the operator `fnewline` is called.

Strings as well as whole numbers and floating point numbers can be used as arguments. If more than one value serves as input the values are put out consecutively without blanks.

Parameter	
▷ FileHandle (input_control)	file $\leadsto$ integer File handle.
▷ String (input_control)	string(-array) $\leadsto$ string / integer / real Values to be put out on the text file. Default Value : 'hallo'

Example	
<pre>fwrite_string(FileHandle,['text with numbers:',5,' and ',1.0]). /* results in the following output: */ /* 'text with numbers:5 and 1.00000' */</pre>	

Result	
If the writing procedure was carried out successfully the operator <code>fwrite_string</code> returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.	

Parallelization Information	
<code>fwrite_string</code> is reentrant and processed without parallelization.	

Possible Predecessors	
<code>open_file</code>	

Possible Successors	
<code>close_file</code>	

Alternatives	
<code>write_string</code>	

See also	
<code>open_file</code> , <code>close_file</code> , <code>set_system</code>	

Module	
Basic operators	

open_file (: : FileName, FileType : FileHandle)

Open text file.

The operator `open_file` opens a file. `FileType` determines whether this file is an input ('input') or output file ('output' or 'append'). `open_file` creates files which can be accessed either by reading ('input') or by writing ('output' or 'append') are created. For terminal input and output the file names 'standard' ('input' and 'output') and 'error' (only 'output') are reserved.

Parameter	
▷ FileName (input_control)	filename $\leadsto$ string Name of file to be opened. Default Value : 'standard' Suggested values : FileName $\in$ {'standard', 'error', '/tmp/dat.dat'}
▷ FileType (input_control)	string $\leadsto$ string Type of file. Default Value : 'output' List of values : FileType $\in$ {'input', 'output', 'append'}
▷ FileHandle (output_control)	file $\leadsto$ integer File handle.

Example

```

/* Creating of an outputfile with the name '/tmp/log.txt' and writing */
/* of one string: */
open_file('/tmp/log.txt','output',FileHandle)
fwrite_string(FileHandle,'these are the first and last lines')
fnew_line(FileHandle)
close_file(FileHandle).

```

Result

If the parameters are correct the operator `open_file` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

`open_file` is *local* and processed *completely exclusively* without parallelization.

Possible Successors

`fwrite_string`, `fread_char`, `fread_string`, `close_file`

See also

`close_file`, `fwrite_string`, `fread_char`, `fread_string`

Module

Basic operators

4.5 Tuple

read_tuple (: : FileName : Tuple)

Read a tuple from a file.

The operator `read_tuple` reads the contents of `FileName` and converts it into `Tuple`. The file has to be generated by `write_tuple`.

Parameter

- ▷ **FileName** (input_control) filename $\leadsto$ *string*
Name of the file to be read.
- ▷ **Tuple** (output_control) number(-array) $\leadsto$ *real / integer / string*
Tuple with any kind of data.

Result

If the parameters are correct the operator `read_tuple` returns the value 2 (H_MSG_TRUE). If the file could not be opened `read_tuple` returns 5 (H_MSG_FAIL). Otherwise an exception handling is raised.

Parallelization Information

`read_tuple` is *reentrant* and processed *without* parallelization.

Alternatives

`fwrite_string`

See also

`write_tuple`, `gnuplot_plot_ctrl`, `write_image`, `write_region`, `open_file`

Module

Operators not requiring licensing

write_tuple (: : Tuple, FileName :)

Write a tuple to a file.

The operator `write_tuple` writes the contents of `Tuple` to a file. The data is written in an ASCII format. Therefore, the file can be exchanged between different architectures. There is no specific extension for this kind of file.

Parameter	
▷ Tuple (input_control)	number(-array) $\leadsto$ <i>real</i> / <i>integer</i> / <i>string</i> Tuple with any kind of data.
▷ FileName (input_control)	filename $\leadsto$ <i>string</i> Name of the file to be written.

	Result

If the parameters are correct the operator `write_tuple` returns the value 2 (H_MSG_TRUE). If the file could not be opened `write_tuple` returns 5 (H_MSG_FAIL). Otherwise an exception handling is raised.

	Parallelization Information

`write_tuple` is *reentrant* and processed *without* parallelization.

	Alternatives

`fwrite_string`

	See also

`read_tuple`, `write_image`, `write_region`, `open_file`

	Module

Operators not requiring licensing

4.6 XLD

read_contour_xld_arc_info (: Contours : FileName :)

Read XLD contours to a file in ARC/INFO generate format.

`read_contour_xld_arc_info` reads the lines stored in ARC/INFO generate format in the file `FileName`, and returns them as XLD contours in `Contours`. To find the file `FileName`, all directories contained in the HALCON system variable 'image_dir' (usually this is the content of the environment variable HALCONIMAGES) are searched (see `read_image`). The returned contours are in world coordinates. They can be transformed to the image coordinate system with the operator `affine_trans_contour_xld`. The necessary transformation matrix can be generated by using `read_world_file` to read the transformation matrix from image to world coordinates, and inverting this matrix by calling `hom_mat2d_invert`.

Parameter	
▷ Contours (output_object)	xld_cont(-array) $\leadsto$ <i>Hobject</i> Read XLD contours.
▷ FileName (input_control)	filename $\leadsto$ <i>string</i> Name of the ARC/INFO file.

	Example

```
/* Read the transformation and invert it */
read_world_file ('image.tfw', WorldTransformation)
hom_mat2d_invert (WorldTransformation, ImageTransformation)
/* Read the image */
read_image (Image, 'image.tif')
/* Read the line data */
read_contour_xld_arc_info (LinesWorld, 'lines.gen')
/* Transform the line data to image coordinates */
affine_trans_contour_xld (LinesWorld, Lines, ImageTransformation)
```

	Result

If the parameters are correct and the file could be read, the operator `read_contour_xld_arc_info` returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Parallelization Information

[read_contour_xld_arc_info](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[hom_mat2d_invert](#), [affine_trans_contour_xld](#)

See also

[read_world_file](#), [write_contour_xld_arc_info](#), [read_polygon_xld_arc_info](#)

Module

Sub-pixel operators

read_polygon_xld_arc_info (: Polygons : FileName :)

Read XLD polygons from a file in ARC/INFO generate format.

[read_polygon_xld_arc_info](#) reads the lines stored in ARC/INFO generate format in the file [FileName](#), and returns them as XLD polygons in [Polygons](#). To find the file [FileName](#), all directories contained in the HALCON system variable 'image_dir' (usually this is the content of the environment variable HALCONIMAGES) are searched (see [read_image](#)). The returned polygons are in world coordinates. They can be transformed to the image coordinate system with the operator [affine_trans_polygon_xld](#). The necessary transformation matrix can be generated by using [read_world_file](#) to read the transformation matrix from image to world coordinates, and inverting this matrix by calling [hom_mat2d_invert](#).

Parameter

- ▷ **Polygons** (output_object) xld_poly(-array) $\leadsto$ Hobject
Read XLD polygons.
- ▷ **FileName** (input_control) filename $\leadsto$ string
Name of the ARC/INFO file.

Example

```
/* Read the transformation and invert it */
read_world_file ('image.tfw', WorldTransformation)
hom_mat2d_invert (WorldTransformation, ImageTransformation)
/* Read the image */
read_image (Image, 'image.tif')
/* Read the line data */
read_polygon_xld_arc_info (LinesWorld, 'lines.gen')
/* Transform the line data to image coordinates */
affine_trans_polygon_xld (LinesWorld, Lines, ImageTransformation)
```

Result

If the parameters are correct and the file could be read, the operator [read_polygon_xld_arc_info](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Parallelization Information

[read_polygon_xld_arc_info](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[hom_mat2d_invert](#), [affine_trans_polygon_xld](#)

See also

[read_world_file](#), [write_polygon_xld_arc_info](#), [read_contour_xld_arc_info](#)

Module

Sub-pixel operators

write_contour_xld_arc_info (Contours : : FileName :)

Write XLD contours to a file in ARC/INFO generate format.

[write_contour_xld_arc_info](#) writes the XLD contours [Contours](#) to an ARC/INFO generate format file with name [FileName](#). If no absolute path is given in [FileName](#), the output file is created in the current directory of the HALCON process. The contours must have been transformed to the world coordinate system with [affine_trans_contour_xld](#) beforehand. The necessary transformation can be read from an ARC/INFO world file with [read_world_file](#).

Parameter

- ▷ **Contours** (input_object) xld_cont(-array) $\leadsto$ *Hobject*
XLD contours to be written.
- ▷ **FileName** (input_control) filename $\leadsto$ *string*
Name of the ARC/INFO file.

Example

```
/* Read transformation and image */
read_world_file ('image.tfw', WorldTransformation)
read_image (Image, 'image.tif')
/* Segment image */
...
/* Write result */
affine_trans_contour_xld (Contours, ContoursWorld, WorldTransformation)
write_contour_xld_arc_info (ContoursWorld, 'result.gen')
```

Result

If the parameters are correct and the file could be written, the operator [write_contour_xld_arc_info](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Parallelization Information

[write_contour_xld_arc_info](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[affine_trans_contour_xld](#)

See also

[read_world_file](#), [read_contour_xld_arc_info](#), [write_polygon_xld_arc_info](#)

Module

Sub-pixel operators

write_polygon_xld_arc_info (Polygons : : FileName :)

Write XLD polygons to a file in ARC/INFO generate format.

[write_polygon_xld_arc_info](#) writes the XLD polygons [Polygons](#) to an ARC/INFO generate format file with name [FileName](#). If no absolute path is given in [FileName](#), the output file is created in the current directory of the HALCON process. The polygons must have been transformed to the world coordinate system with [affine_trans_polygon_xld](#) beforehand. The necessary transformation can be read from an ARC/INFO world file with [read_world_file](#).

Attention

The XLD contours that are possibly referenced by [Polygons](#) are not stored in the ARC/INFO file, since this is not possible with the ARC/INFO generate file format. Therefore, when the polygons are read again using [read_polygon_xld_arc_info](#), this information is lost, and no references to contours are generated for the polygons. Hence, operators that access the contours associated with a polygon, e.g., [split_contours_xld](#) will not work correctly.

Parameter

- ▷ **Polygons** (input_object) xld_poly(-array) $\leadsto$ *Hobject*
XLD polygons to be written.
- ▷ **FileName** (input_control) filename $\leadsto$ *string*
Name of the ARC/INFO file.

Example

```
/* Read transformation and image */
read_world_file ('image.tfw', WorldTransformation)
read_image (Image, 'image.tif')
/* Segment image */
...
/* Write result */
affine_trans_polygon_xld (Polygons, PolygonsWorld, WorldTransformation)
write_polygon_xld_arc_info (PolygonsWorld, 'result.gen')
```

Result

If the parameters are correct and the file could be written, the operator [write_polygon_xld_arc_info](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Parallelization Information

[write_polygon_xld_arc_info](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[affine_trans_polygon_xld](#)

See also

[read_world_file](#), [read_polygon_xld_arc_info](#), [write_contour_xld_arc_info](#)

Module

Sub-pixel operators

Chapter 5

Filter

5.1 Affine-Transformations

affine_trans_image (Image : ImageAffinTrans : HomMat2D, Interpolation, AdaptImageSize :)

Apply an arbitrary affine 2D transformation to images.

affine_trans_image applies an arbitrary affine 2D transformation, i.e., scaling, rotation, translation, and slant (skewing), to the images given in **Image** and returns the transformed images in **ImageAffinTrans**. The affine transformation is described by the homogeneous transformation matrix given in **HomMat2D**, which can be created using the operators **hom_mat2d_identity**, **hom_mat2d_scale**, **hom_mat2d_rotate**, **hom_mat2d_translate**, etc., or be the result of operators like **vector_angle_to_rigid**.

The components of the homogeneous transformation matrix are interpreted as follows: The *row* coordinate of the image corresponds to the *x* coordinate of the matrix, while the *column* coordinate of the image corresponds to the *y* coordinate of the matrix. This is necessary to obtain a right-handed coordinate system for the image. In particular, this assures that rotations are performed in the correct direction. Note that the (x,y) order of the matrices quite naturally corresponds to the usual (row,column) order for coordinates in the image.

The region of the input image is ignored, i.e., assumed to be the full rectangle of the image. The region of the resulting image is set to the transformed rectangle of the input image. If necessary, the resulting image is filled with zero (black) outside of the region of the original image.

Generally, transformed points will lie between pixel coordinates. Therefore, an appropriate interpolation scheme has to be used. The interpolation can also be used to avoid aliasing effects for scaled images. The quality and speed of the interpolation can be set by the parameter **Interpolation**:

- none* No interpolation: The gray value is determined from the nearest pixel's gray value (possibly low quality, very fast).
- constant* Use equally weighted interpolation between adjacent pixels (medium quality and run time).
- weighted* Use Gaussian interpolation between adjacent pixels (best quality, slow).

In addition, the system parameter '*int_zooming*' (see **set_system**) affects the accuracy of the transformation. If '*int_zooming*' is set to '*true*', the transformation for byte, int2 and uint2 images is carried out internally using fixed point arithmetic, leading to much shorter execution times. However, the accuracy of the transformed gray values is smaller in this case. For byte images, the differences to the more accurate calculation (using '*int_zooming*' = '*false*') is typically less than two gray levels. Correspondingly, for int2 and uint2 images, the gray value differences are less than 1/128 times the dynamic gray value range of the image, i.e., they can be as large as 512 gray levels if the entire dynamic range of 16 bit is used. For real images, the parameter '*int_zooming*' does not affect the accuracy, since the internal calculations are always done using floating point arithmetic.

The size of the target image can be controlled by the parameter **AdaptImageSize**: With value '*true*' the size will be adapted so that no clipping occurs at the right or lower edge. With value '*false*' the target image has the same size as the input image. Note that, independent of **AdaptImageSize**, the image is always clipped at the left and upper edge, i.e., all image parts that have negative coordinates after the transformation are clipped.

Attention

The region of the input image is ignored.

Parameter
▷ Image (input_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / uint2 / real Input image. ▷ ImageAffinTrans (output_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / uint2 / real Transformed image. ▷ HomMat2D (input_control) affine2d-array $\leadsto$ <i>real</i> Input transformation matrix. Number of elements : 6 ▷ Interpolation (input_control) string $\leadsto$ <i>string</i> Type of interpolation. Default Value : 'constant' List of values : Interpolation $\in$ {'none', 'constant', 'weighted'} ▷ AdaptImageSize (input_control) string $\leadsto$ <i>string</i> Adaption of size of result image. Default Value : 'false' List of values : AdaptImageSize $\in$ {'true', 'false'}
Example

```
/* Reduction of an image (512 x 512 Pixels) by 50%, rotation */
/* by 180 degrees and translation to the upper-left corner: */
```

```
hom_mat2d_identity(Matrix1)
hom_mat2d_scale(Matrix1,0.5,0.5,256.0,256.0,Matrix2)
hom_mat2d_rotate(Matrix2,3.14,256.0,256.0,Matrix3)
hom_mat2d_translate(Matrix3,-128.0,-128.0,Matrix4,)
affine_trans_image(Image,TransImage,Matrix4,1).
```

```
/* Enlarging the part of an image in the interactively */
/* chosen rectangular window sector: */
```

```
draw_rectangle2(WindowHandle,L,C,Phi,L1,L2)
hom_mat2d_identity(Matrix1)
get_system(width,Width)
get_system(height,Height)
hom_mat2d_translate(Matrix1,Height/2.0-L,Width/2.0-C,Matrix2)
hom_mat2d_rotate(Matrix2,3.14-Phi,Height/2.0,Width/2.0,Matrix3)
hom_mat2d_scale(Matrix3,Height/(2.0*L2),Width/(2.0*L1),
                Height/2.0,Width/2.0,Matrix4)
affine_trans_image(Image,Matrix4,TransImage,1).
```

Result

[affine_trans_image](#) returns 2 (H_MSG_TRUE) if all parameter values are correct. If the input is empty the behaviour can be set via [set_system\(:,:, 'no_object_result', <Result>:\)](#). If necessary, an exception handling is raised.

Parallelization Information

[affine_trans_image](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*).

Possible Predecessors

[hom_mat2d.identity](#), [hom_mat2d.translate](#), [hom_mat2d.rotate](#), [hom_mat2d.scale](#)

Alternatives

[affine_trans_image_size](#), [zoom_image_size](#), [zoom_image_factor](#), [mirror_image](#), [rotate_image](#), [affine_trans_region](#)

See also

[set_part_style](#)

Module

Image filters

affine_trans_image_size (Image : ImageAffinTrans : HomMat2D,
Interpolation, Width, Height :)

Apply an arbitrary affine 2D transformation to an image and specify the output image size.

affine_trans_image_size applies an arbitrary affine 2D transformation, i.e., scaling, rotation, translation, and slant (skewing), to the images given in **Image** and returns the transformed images in **ImageAffinTrans**. The affine transformation is described by the homogeneous transformation matrix given in **HomMat2D**, which can be created using the operators **hom_mat2d_identity**, **hom_mat2d_scale**, **hom_mat2d_rotate**, **hom_mat2d_translate**, etc., or be the result of operators like **vector_angle_to_rigid**.

The components of the homogeneous transformation matrix are interpreted as follows: The *row* coordinate of the image corresponds to the *x* coordinate of the matrix, while the *column* coordinate of the image corresponds to the *y* coordinate of the matrix. This is necessary to obtain a right-handed coordinate system for the image. In particular, this assures that rotations are performed in the correct direction. Note that the (x,y) order of the matrices quite naturally corresponds to the usual (row,column) order for coordinates in the image.

The region of the input image is ignored, i.e., assumed to be the full rectangle of the image. The region of the resulting image is set to the transformed rectangle of the input image. If necessary, the resulting image is filled with zero (black) outside of the region of the original image.

Generally, transformed points will lie between pixel coordinates. Therefore, an appropriate interpolation scheme has to be used. The interpolation can also be used to avoid aliasing effects for scaled images. The quality and speed of the interpolation can be set by the parameter **Interpolation**:

- none* No interpolation: The gray value is determined from the nearest pixel's gray value (possibly low quality, very fast).
- constant* Use equally weighted interpolation between adjacent pixels (medium quality and run time).
- weighted* Use Gaussian interpolation between adjacent pixels (best quality, slow).

In addition, the system parameter '*int_zooming*' (see **set_system**) affects the accuracy of the transformation. If '*int_zooming*' is set to '*true*', the transformation for byte, int2 and uint2 images is carried out internally using fixed point arithmetic, leading to much shorter execution times. However, the accuracy of the transformed gray values is smaller in this case. For byte images, the differences to the more accurate calculation (using '*int_zooming*' = '*false*') is typically less than two gray levels. Correspondingly, for int2 and uint2 images, the gray value differences are less than 1/128 times the dynamic gray value range of the image, i.e., they can be as large as 512 gray levels if the entire dynamic range of 16 bit is used. For real images, the parameter '*int_zooming*' does not affect the accuracy, since the internal calculations are always done using floating point arithmetic.

The size of the target image is specified by the parameters **Width** and **Height**. Note that the image is always clipped at the left and upper edge, i.e., all image parts that have negative coordinates after the transformation are clipped. If the affine transformation (in particular, the translation) is chosen appropriately, a part of the image can be transformed as well as cropped in one call. This is useful, for example, when using the variation model (see **compare_variation_model**), because with this mechanism only the parts of the image that should be examined, are transformed.

Attention

The region of the input image is ignored.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / real
Input image.
- ▷ **ImageAffinTrans** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / real
Transformed image.
- ▷ **HomMat2D** (input_control) affine2d-array $\leadsto$ *real*
Input transformation matrix.
Number of elements : 6
- ▷ **Interpolation** (input_control) string $\leadsto$ *string*
Type of interpolation.
Default Value : 'constant'
List of values : Interpolation $\in$ {'none', 'constant', 'weighted'}

- ▷ **Width** (input_control) extent.x $\leadsto$ integer
Width of the output image.
Default Value : 640
Suggested values : Width $\in$ {128, 160, 192, 256, 320, 384, 512, 640, 768}
- ▷ **Height** (input_control) extent.y $\leadsto$ integer
Height of the output image.
Default Value : 480
Suggested values : Height $\in$ {120, 128, 144, 240, 256, 288, 480, 512, 576}

Result

`affine_trans_image_size` returns 2 (H_MSG_TRUE) if all parameter values are correct. If the input is empty the behaviour can be set via `set_system( : : 'no_object_result', <Result> : )`. If necessary, an exception handling is raised.

Parallelization Information

`affine_trans_image_size` is *reentrant* and automatically *parallelized* (on tuple level, channel level).

Possible Predecessors

`hom_mat2d_identity`, `hom_mat2d_translate`, `hom_mat2d_rotate`, `hom_mat2d_scale`

Alternatives

`affine_trans_image`, `zoom_image_size`, `zoom_image_factor`, `mirror_image`,
`rotate_image`, `affine_trans_region`

See also

`set_part_style`

Module

Image filters

mirror_image (Image : ImageMirror : Mode :)

Mirror an image.

`mirror_image` reflects an image `Image` about one of three possible axes. If `Mode` is set to 'row', it is reflected about the horizontal axis, if `Mode` is set to 'column', about the vertical axis, and if `Mode` is set to 'main', about the main diagonal $x = y$.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ Hobject : byte / int2 / uint2 / int4 / real
Input image.
- ▷ **ImageMirror** (output_object) (multichannel-)image(-array) $\leadsto$ Hobject : byte / int2 / uint2 / int4 / real
Reflected image.
- ▷ **Mode** (input_control) string $\leadsto$ string
Axis of reflection.
Default Value : 'row'
List of values : Mode $\in$ {'row', 'column', 'main'}

Example

```
read_image(Image, 'affe')
disp_image(Image, WindowHandle)
mirror_image(Image, MirImage, 'row').
disp_image(MirImage, WindowHandle)
```

Parallelization Information

`mirror_image` is *reentrant* and automatically *parallelized* (on tuple level, channel level).

Alternatives

`hom_mat2d_rotate`, `affine_trans_image`, `rotate_image`

See also

`rotate_image`, `hom_mat2d_rotate`

Module

Image filters

polar_trans_image (ImageXY : ImagePolar : Row, Column, Width, Height :)

Transform an image to polar coordinates

polar_trans_image transforms an image in cartesian coordinates to an image in polar coordinates. The size of the resulting image is selected with **Width** and **Height**. **Width** determines the angular resolution, while **Height** determines the resolution of the radius. **Row** and **Column** determine the center of the polar coordinate system in the original image **ImageXY**. This point is mapped to the upper row of **ImagePolar**.

A point (x',y') in the result image corresponds to the point (x,y) in the original image in the following manner:

$$x = y' \cos(2\pi(x'/resultwidth)) + Column, y = y' \sin(2\pi(x'/resultwidth)) + Row .$$

Parameter

- ▷ **ImageXY** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Input image in cartesian coordinates.
- ▷ **ImagePolar** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Result image in polar coordinates.
- ▷ **Row** (input_control) point.y $\leadsto$ *integer*
Row coordinate of the center of the coordinate system.
Default Value : 100
Suggested values : Row $\in$ {0, 10, 100, 200}
Typical range of values : $0 \leq Row \leq 512$
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Column** (input_control) point.x $\leadsto$ *integer*
Column coordinate of the center of the coordinate system.
Default Value : 100
Suggested values : Column $\in$ {0, 10, 100, 200}
Typical range of values : $0 \leq Column \leq 512$
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer*
Width of the result image.
Default Value : 314
Suggested values : Width $\in$ {100, 200, 157, 314, 512}
Typical range of values : $2 \leq Width \leq 512$
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer*
Height of the result image.
Default Value : 200
Suggested values : Height $\in$ {100, 128, 256, 512}
Typical range of values : $2 \leq Height \leq 512$
Minimum Increment : 1
Recommended Increment : 10

Example

```
read_image(Image, 'affe')
disp_image(Image, WindowHandle)
polar_trans_image(Image, PolarImage, 100, 100, 314, 200) .
disp_image(PolarImage, WindowHandle)
```

Parallelization Information

polar_trans_image is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*).

Alternatives

affine_trans_image

Module

Image filters

rotate_image (Image : ImageRotate : Phi, Interpolation :)

Rotate an image about its center.

rotate_image rotates the image **Image** counterclockwise by **Phi** degrees about its center. This operator is much faster if **Phi** is a multiple of 90 degrees than the general operator **affine_trans_image**. For rotations by 90, 180, and 270 degrees, the region is rotated accordingly. For all other rotations the region is set to the maximum region, i.e., to the extent of the resulting image. The effect of the parameter **Interpolation** is the same as in **affine_trans_image**. It is ignored for rotations by 90, 180, and 270 degrees. The size of the resulting image is the same as that of the input image, with the exception of rotations by 90 and 270 degrees, where the width and height will be exchanged.

Attention

The angle **Phi** is given in degrees, not in radians.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Input image.
- ▷ **ImageRotate** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Rotated image.
- ▷ **Phi** (input_control) angle.deg $\leadsto$ *real* / integer
Rotation angle.
Default Value : 90
Suggested values : $\Phi \in \{90, 180, 270\}$
Typical range of values : $0 \leq \Phi \leq 360$
Minimum Increment : 0.001
Recommended Increment : 0.2
- ▷ **Interpolation** (input_control) string $\leadsto$ *string*
Type of interpolation.
Default Value : 'constant'
List of values : $\text{Interpolation} \in \{\text{'none'}, \text{'constant'}, \text{'weighted'}\}$

Example

```
read_image(Image, 'affe')
disp_image(Image, WindowHandle)
rotate_image(ImageRotImage, 270).
disp_image(RotImage, WindowHandle)
```

Parallelization Information

rotate_image is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*).

Alternatives

hom_mat2d_rotate, **affine_trans_image**

See also

mirror_image

Module

Image filters

zoom_image_factor (Image : ImageZoomed : ScaleWidth, ScaleHeight, Interpolation :)

Zoom an image by a given factor.

zoom_image_factor scales the image **Image** by a factor of **ScaleWidth** in width and a factor **ScaleHeight** in height. The parameter **Interpolation** determines the type of interpolation used (see **affine_trans_image**).

Parameter	
▷ Image (input_object)	(multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / uint2 / real Input image.
▷ ImageZoomed (output_object)	(multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / uint2 / real Scaled image.
▷ ScaleWidth (input_control)	extent.x $\leadsto$ <i>real</i> Scale factor for the width of the image. Default Value : 0.5 Suggested values : $\text{ScaleWidth} \in \{0.25, 0.5, 1.5, 2.0\}$ Typical range of values : $0.001 \leq \text{ScaleWidth} \leq 10.0$ Minimum Increment : 0.001 Recommended Increment : 0.1
▷ ScaleHeight (input_control)	extent.y $\leadsto$ <i>real</i> Scale factor for the height of the image. Default Value : 0.5 Suggested values : $\text{ScaleHeight} \in \{0.25, 0.5, 1.5, 2.0\}$ Typical range of values : $0.001 \leq \text{ScaleHeight} \leq 10.0$ Minimum Increment : 0.001 Recommended Increment : 0.1
▷ Interpolation (input_control)	string $\leadsto$ <i>string</i> Type of interpolation. Default Value : 'constant' List of values : $\text{Interpolation} \in \{\text{'none'}, \text{'constant'}, \text{'weighted'}\}$
Example	

```
read_image(Image, 'affe')
disp_image(Image, WindowHandle)
zoom_image_factor(Image, ZooImage, 0, 0.5, 0.5).
disp_image(ZooImage, WindowHandle)
```

Parallelization Information	
zoom_image_factor is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i> , <i>channel level</i>).	
Alternatives	
zoom_image_factor , affine_trans_image , hom_mat2d_scale	
See also	
hom_mat2d_scale , affine_trans_image	
Module	
Image filters	

zoom_image_size (Image : ImageZoom : Width, Height, Interpolation :)

Zoom an image to a given size.

zoom_image_size scales the image **Image** to the size given by **Width** and **Height**. The parameter **Interpolation** determines the type of interpolation used (see **affine_trans_image**).

Parameter	
▷ Image (input_object)	(multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / uint2 / real Input image.
▷ ImageZoom (output_object)	(multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / uint2 / real Scaled image.

- ▷ **Width** (input_control) extent.x $\leadsto$ integer
Width of the resulting image.
Default Value : 512
Suggested values : Width $\in \{128, 256, 512\}$
Typical range of values : $2 \leq \text{Width} \leq 512$
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Height** (input_control) extent.y $\leadsto$ integer
Height of the resulting image.
Default Value : 512
Suggested values : Height $\in \{128, 256, 512\}$
Typical range of values : $2 \leq \text{Height} \leq 512$
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Interpolation** (input_control) string $\leadsto$ string
Type of interpolation.
Default Value : 'constant'
List of values : Interpolation $\in \{\text{'none'}, \text{'constant'}, \text{'weighted'}\}$

Example

```
read_image(Image, 'affe')
disp_image(Image, WindowHandle)
zoom_image_size(Image, ZooImage, 0, 200, 200).
disp_image(ZooImage, WindowHandle)
```

Parallelization Information

[zoom_image_size](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*).

Alternatives

[zoom_image_factor](#), [affine_trans_image](#), [hom_mat2d_scale](#)

See also

[hom_mat2d_scale](#), [affine_trans_image](#)

Module

Image filters

5.2 Arithmetic

abs_image (Image : ImageAbs : :)

Calculate the absolute value (modulus) of an image.

The operator [abs_image](#) calculates the absolute gray values of images of any type and stores the result in [ImageAbs](#). The power spectrum of complex images is calculated as a 'real' image. The operator [abs_image](#) generates a logical copy of unsigned images.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ Hobject : int1 / int2 / int4 / real / complex
Image(s) for which the absolute gray values are to be calculated.
- ▷ **ImageAbs** (output_object) (multichannel-)image(-array) $\leadsto$ Hobject : int1 / int2 / int4 / real
Result image(s).

Example

```
laws_int2(Image, &Texture, 0, 5);
abs_image(Texture, Abs);
```

Result

The operator `abs_image` returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator `set_system(::'no_object_result', <Result>:)`

Parallelization Information

`abs_image` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

See also

`convert_image_type`, `power_byte`

Module

Image filters

add_image (Image1, Image2 : ImageResult : Mult, Add :)

Add two images.

The operator `add_image` adds two images. The gray values (g_1, g_2) of the input images (`Image1` and `Image2`) are transformed as follows:

$$g' := (g_1 + g_2) * \text{Mult} + \text{Add}$$

If an overflow or an underflow occurs the values are clipped. This is not the case with int2 images if `Mult` is equal to 1 and `Add` is equal to 0. To reduce the runtime the underflow and overflow check is skipped. The resulting image is stored in `ImageResult`.

It is possible to add byte images with int2, uint2 or int4 images and to add int4 to int2 or uint2 images. In this case the result will be of type int2 or int4 respectively.

Several images can be processed in one call. In this case both input parameters contain the same number of images which are then processed in pairs. An output image is generated for every pair.

Please note that the runtime of the operator varies with different control parameters. For frequently used combinations special optimizations were used.

Parameter

▷ **Image1** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int1 / int2 / uint2 / int4 / real / direction / cyclic / complex

Image(s) 1.

▷ **Image2** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int1 / int2 / uint2 / int4 / real / direction / cyclic / complex

Image(s) 2.

▷ **ImageResult** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int1 / int2 / uint2 / int4 / real / direction / cyclic / complex

Result image(s) by the addition.

▷ **Mult** (input_control) number $\leadsto$ *real* / integer
Factor for gray value adaption.

Default Value : 0.5

Suggested values : `Mult` $\in$ {0.2, 0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 5.0}

Typical range of values : $-255.0 \leq \text{Mult} \leq 255.0$

Minimum Increment : 0.001

Recommended Increment : 0.1

▷ **Add** (input_control) number $\leadsto$ *real* / integer
Value for gray value range adaption.

Default Value : 0

Suggested values : `Add` $\in$ {0, 64, 128, 255, 512}

Typical range of values : $-512.0 \leq \text{Add} \leq 512.0$

Minimum Increment : 0.01

Recommended Increment : 1.0

Example

```

read_image(Image0,"fabrik")
disp_image(Image0,WindowHandle)
read_image(Image1,"Affe")
disp_image(Image1,WindowHandle)
add_image(Image0,Image1,Result,2.0,10.0)
disp_image(Result,WindowHandle)

```

Result

The operator `add_image` returns the value 2 (H_MSG_TRUE) if the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator `set_system (::'no_object_result', <Result>:)` If necessary an exception handling is raised.

Parallelization Information

`add_image` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Alternatives

`sub_image`, `mult_image`

See also

`sub_image`, `mult_image`

Module

Image filters

div_image (Image1, Image2 : ImageResult : Mult, Add :)

Divide two images.

The operator `div_image` divides two images. The gray values (g_1, g_2) of the input images (`Image1`) are transformed as follows:

$$g' := g_1 / g_2 * \text{Mult} + \text{Add}$$

If an overflow or an underflow occurs the values are clipped.

Several images can be processed in one call. In this case both input parameters contain the same number of images which are then processed in pairs. An output image is generated for every pair.

Parameter

- ▷ **Image1** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int1 / int2 / uint2 / int4 / real / complex
Image(s) 1.
- ▷ **Image2** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int1 / int2 / uint2 / int4 / real / complex
Image(s) 2.
- ▷ **ImageResult** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int1 / int2 / uint2 / int4 / real / complex
Result image(s) by the division.
- ▷ **Mult** (input_control) number $\leadsto$ *real* / integer
Factor for gray range adaption.
Default Value : 255
Suggested values : `Mult` $\in$ {0.1, 0.2, 0.5, 1.0, 2.0, 3.0, 10, 100, 500, 1000}
Typical range of values : $-1000 \leq \text{Mult} \leq 1000$
Minimum Increment : 0.001
Recommended Increment : 1
- ▷ **Add** (input_control) number $\leadsto$ *real* / integer
Value for gray range adaption.
Default Value : 0
Suggested values : `Add` $\in$ {0.0, 128.0, 256.0, 1025}
Typical range of values : $-1000 \leq \text{Add} \leq 1000$
Minimum Increment : 0.01
Recommended Increment : 1.0

Example

```

read_image(Image0,"fabrik")
disp_image(Image0,WindowHandle)
read_image(Image1,"Affe")
disp_image(Image1,WindowHandle)
div_image(Image0,Image1,Result,2.0,10.0)
disp_image(Result,WindowHandle)

```

Result

The operator `div_image` returns the value 2 (H_MSG_TRUE) if the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator `set_system (::'no_object_result', <Result>:)` If necessary an exception handling is raised.

Parallelization Information

`div_image` is *reentrant* and automatically *parallelized* (on tuple level, channel level, domain level).

Alternatives

`add_image`, `sub_image`, `mult_image`

See also

`add_image`, `sub_image`, `mult_image`

Module

Image filters

invert_image (Image : ImageInvert : :)

Invert an image.

The operator `invert_image` inverts the gray values of an image. For images of the 'byte' and 'cyclic' type the result is calculated as:

$$g' = 255 - g$$

Images of the 'direction' type are transformed by

$$g' = (g + 90) \text{ modulo } 180$$

In the case of signed types the values are negated. The resulting image has the same pixel type as the input image. Several images can be processed in one call. An output image is generated for every input image.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Input image(s).
- ▷ **ImageInvert** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Image(s) with inverted gray values.

Example

```

read_image(Orig,"fabrik")
invert_image(Orig,Invert)
disp_image(Invert,WindowHandle).

```

Parallelization Information

`invert_image` is *reentrant* and automatically *parallelized* (on tuple level, channel level, domain level).

Possible Successors

`watersheds`

	Alternatives
scale_image	
	See also
scale_image , add_image , sub_image	
	Module
Image filters	

max_image (Image1, Image2 : ImageMax : :)

Calculate the maximum of two images pixel by pixel.

max_image calculates the maximum of the images **Image1** and **Image2** (pixel by pixel). The result is stored in the image **ImageMax**. The resulting image has the same pixel type as the input image. If several (pairs of) images are processed in one call, every i-th image from **Image1** is compared to the i-th image from **Image2**. Thus the number of images in both input parameters must be the same. An output image is generated for every input pair.

	Attention
The two input images must be of the same type and size.	

	Parameter

- ▷ **Image1** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int1 / int2 / uint2 / int4 / real / direction / cyclic
Image(s) 1.
- ▷ **Image2** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int1 / int2 / uint2 / int4 / real / direction / cyclic
Image(s) 2.
- ▷ **ImageMax** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int1 / int2 / uint2 / int4 / real / direction / cyclic
Result image(s) by the maximization.

	Example

```
read_image(Bild1, "affe")
read_image(Bild2, "fabrik")
max_image(Bild1, Bild2, Max)
disp_image(Max, WindowHandle)
```

	Result

If the parameter values are correct the operator **max_image** returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator **set_system (::'no_object_result', <Result>:)**. If necessary an exception handling is raised.

	Parallelization Information

max_image is *reentrant* and automatically *parallelized* (on tuple level, channel level, domain level).

	Alternatives

[max_image](#)

	See also

[min_image](#)

	Module

Image filters

min_image (Image1, Image2 : ImageMin : :)

Calculate the minimum of two images pixel by pixel.

The operator **min_image** determines the minimum (pixel by pixel) of the images **Image1** and **Image2**. The result is stored in the image **ImageMin**. The resulting image has the same pixel type as the input image. If several

(pairs of) images are processed in one call, every i-th image from **Image1** is compared to the i-th image from **Image2**. Thus the number of images in both input parameters must be the same. An output image is generated for every input pair.

Parameter	
▷ Image1 (input_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int1 / int2 / uint2 / int4 / real / direction / cyclic	Image(s) 1.
▷ Image2 (input_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int1 / int2 / uint2 / int4 / real / direction / cyclic	Image(s) 2.
▷ ImageMin (output_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int1 / int2 / uint2 / int4 / real / direction / cyclic	Result image(s) by the minimization.

Result
If the parameter values are correct the operator **min_image** returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator **set_system** (`::'no_object_result', <Result>:`) If necessary an exception handling is raised.

Parallelization Information
min_image is *reentrant* and automatically *parallelized* (on tuple level, channel level, domain level).

Alternatives
gray_erosion

See also
max_image, min_image

Module
Image filters

mult_image (Image1, Image2 : ImageResult : Mult, Add :)

Multiply two images.

mult_image multiplies two images. The gray values (g_1, g_2) of the input images (**Image1**) are transformed as follows:

$$g' := g_1 * g_2 * \text{Mult} + \text{Add}$$

If an overflow or an underflow occurs the values are clipped.

Several images can be processed in one call. In this case both input parameters contain the same number of images which are then processed in pairs. An output image is generated for every pair.

Parameter	
▷ Image1 (input_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int1 / int2 / uint2 / int4 / real / direction / cyclic / complex	Image(s) 1.
▷ Image2 (input_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int1 / int2 / uint2 / int4 / real / direction / cyclic / complex	Image(s) 2.
▷ ImageResult (output_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int1 / int2 / uint2 / int4 / real / direction / cyclic / complex	Result image(s) by the product.

- ▷ **Mult** (input_control) number $\leadsto$ real / integer
 Factor for gray range adaption.
Default Value : 0.005
Suggested values : $\text{Mult} \in \{0.001, 0.01, 0.5, 1.0, 2.0, 3.0, 5.0, 10.0\}$
Typical range of values : $-255.0 \leq \text{Mult} \leq 255.0$
Minimum Increment : 0.001
Recommended Increment : 0.1
- ▷ **Add** (input_control) number $\leadsto$ real / integer
 Value for gray range adaption.
Default Value : 0
Suggested values : $\text{Add} \in \{0.0, 128.0, 256.0\}$
Typical range of values : $-512.0 \leq \text{Add} \leq 512.0$
Minimum Increment : 0.01
Recommended Increment : 1.0

Example

```
read_image(Image0,"fabrik")
disp_image(Image0,WindowHandle)
read_image(Image1,"Affe")
disp_image(Image1,WindowHandle)
mult_image(Image0,Image1,Result,2.0,10.0)
disp_image(Result,WindowHandle)
```

Result

The operator `mult_image` returns the value 2 (H_MSG_TRUE) if the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator `set_system (::'no_object_result', <Result>:)` If necessary an exception handling is raised.

Parallelization Information

`mult_image` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Alternatives

`add_image`, `sub_image`, `div_image`

See also

`add_image`, `sub_image`, `div_image`

Module

Image filters

scale_image (Image : ImageScaled : Mult, Add :)

Scale the gray values of an image.

The operator `scale_image` scales the input images (**Image**) by the following transformation:

$$g' := g * \text{Mult} + \text{Add}$$

If an overflow or an underflow occurs the values are clipped.

This operator can be applied, e.g., to map the gray values of an image, i.e., the interval [GMin,GMax], to the maximum range [0:255]. For this, the parameters are chosen as follows:

$$\text{Mult} = \frac{255}{\text{GMax} - \text{GMin}} \quad \text{Add} = -\text{Mult} * \text{GMin}$$

The values for GMin and GMax can be determined, e.g., with the operator `min_max_gray`.

	Parameter
▷ Image (input_object)	(multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int1 / int2 / uint2 / int4 / real / direction / cyclic / complex
	Image(s) whose gray values are to be scaled.
▷ ImageScaled (output_object)	(multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int1 / int2 / uint2 / int4 / real / direction / cyclic / complex
	Result image(s) by the scale.
▷ Mult (input_control)	number $\leadsto$ <i>real</i> / integer
	Scale factor.
	Default Value : 0.01
	Suggested values : $\text{Mult} \in \{0.001, 0.003, 0.005, 0.008, 0.01, 0.02, 0.03, 0.05, 0.08, 0.1, 0.5, 1.0\}$
	Typical range of values : $-255.0 \leq \text{Mult} \leq 255.0$
	Minimum Increment : 0.001
	Recommended Increment : 0.1
▷ Add (input_control)	number $\leadsto$ <i>real</i> / integer
	Offset.
	Default Value : 0
	Suggested values : $\text{Add} \in \{0, 10, 50, 100, 200, 500\}$
	Typical range of values : $-512.0 \leq \text{Add} \leq 512.0$
	Minimum Increment : 0.01
	Recommended Increment : 1.0

Example

```
/* Complement of the gray values: */
scale_image(Bild, Invert, -1.0, 255.0, ).
```

Result

The operator `scale_image` returns the value 2 (H_MSG_TRUE) if the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator `set_system (::'no_object_result', <Result>:)` Otherwise an exception treatment is carried out.

Parallelization Information

`scale_image` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Predecessors

`min_max_gray`

Alternatives

`mult_image`, `add_image`, `sub_image`

See also

`min_max_gray`

Module

Image filters

sub_image (ImageMinuend, ImageSubtrahend : ImageSub : Mult, Add :)

Subtract two images.

The operator `sub_image` subtracts two images. The gray values (g_1, g_2) of the input images (`ImageMinuend` and `ImageSubtrahend`) are transformed as follows:

$$g' := (g_1 - g_2) * \text{Mult} + \text{Add}$$

If an overflow or an underflow occurs the values are clipped.

Several images can be processed in one call. In this case both input parameters contain the same number of images which are then processed in pairs. An output image is generated for every pair.

Parameter	
▷ ImageMinuend (input_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int1 / int2 / uint2 / int4 / real / direction / cyclic / complex	Minuend(s).
▷ ImageSubtrahend (input_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int1 / int2 / uint2 / int4 / real / direction / cyclic / complex	Subtrahend(s).
▷ ImageSub (output_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int1 / int2 / uint2 / int4 / real / direction / cyclic / complex	Result image(s) by the subtraction.
▷ Mult (input_control) number $\leadsto$ <i>real</i> / integer	Correction factor. Default Value : 1.0 Suggested values : $\text{Mult} \in \{0.0, 1.0, 2.0, 3.0, 4.0\}$ Typical range of values : $-255.0 \leq \text{Mult} \leq 255.0$ Minimum Increment : 0.001 Recommended Increment : 0.1
▷ Add (input_control) number $\leadsto$ <i>real</i> / integer	Correction value. Default Value : 128.0 Suggested values : $\text{Add} \in \{0.0, 128.0, 256.0\}$ Typical range of values : $-512.0 \leq \text{Add} \leq 512.0$ Minimum Increment : 0.01 Recommended Increment : 1.0

Example

```

read_image(Image0,"fabrik")
disp_image(Image0,WindowHandle)
read_image(Image1,"Affe")
disp_image(Image1,WindowHandle)
sub_image(Image0,Image1,Result,2.0,10.0)
disp_image(Result,WindowHandle)

```

Result

The operator `sub_image` returns the value 2 (H_MSG_TRUE) if the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator `set_system (::'no_object_result', <Result>:)` If necessary an exception handling is raised.

Parallelization Information

`sub_image` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Successors

`dual_threshold`

Alternatives

`mult_image`, `add_image`, `sub_image`

See also

`add_image`, `mult_image`, `dyn_threshold`, `check_difference`

Module

Image filters

5.3 Bit

bit_and (Image1, Image2 : ImageAnd : :)

Bit-by-bit AND of all pixels of the input images.

The operator **bit_and** calculates the “and” of all pixels of the input images bit by bit. The semantics of the “and” operation corresponds to that of C for the respective types (signed char, unsigned char, short, unsigned short, int/long). The images must have the same size and pixel type. The pixels within the definition range of the image in the first parameter are processed.

Several images can be processed in one call. In this case both input parameters contain the same number of images which are then processed in pairs. An output image is generated for every pair.

Parameter

- ▷ **Image1** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4
Input image(s) 1.
- ▷ **Image2** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4
Input image(s) 2.
- ▷ **ImageAnd** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4
Result image(s) by AND-operation.

Example

```
read_image(Image0, 'affe')
disp_image(Image0, WindowHandle)
read_image(Image1, 'fabrik')
disp_image(Image1, WindowHandle)
bit_and(Image0, Image1, ImageBitA)
disp_image(ImageBitA, WindowHandle).
```

Result

If the images are correct (type and number) the operator **bit_and** returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator **set_system (::'no_object_result', <Result>:)** If necessary an exception handling is raised.

Parallelization Information

bit_and is *reentrant* and automatically *parallelized* (on tuple level, channel level, domain level).

Alternatives

bit_mask, **add_image**, **max_image**

See also

bit_mask, **add_image**, **max_image**

Module

Image filters

bit_lshift (Image : ImageLShift : Shift :)

Left shift of all pixels of the image.

The operator **bit_lshift** calculates a “left shift” of all pixels of the input image bit by bit. The semantics of the “left shift” operation corresponds to that of C (“<<”) for the respective types (signed char, unsigned char, short, unsigned short, int/long). If an overflow occurs the result is limited to the maximum value of the respective pixel type. Only the pixels within the definition range of the image are processed.

Several images can be processed in one call. An output image is generated for every input image.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4
Input image(s).
- ▷ **ImageLShift** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4
Result image(s) by shift operation.

	Alternatives
bit_slice	
	See also
bit_and , bit_lshift	
	Module
Image filters	

bit_not (Image : ImageNot : :)

Complement all bits of the pixels.

The operator [bit_not](#) calculates the “complement” of all pixels of the input image bit by bit. The semantics of the “complement” operation corresponds to that of C (“~”) for the respective types (signed char, unsigned char, short, unsigned short, int/long). Only the pixels within the definition range of the image are processed. Several images can be processed in one call. An output image is generated for every input image.

	Parameter
▷ Image (input_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4	
Input image(s).	
▷ ImageNot (output_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4	
Result image(s) by complement operation.	
	Example

```
read_image(Image0, 'affe')
disp_image(Image0, WindowHandle)
bit_not(Image0, ImageBitN)
disp_image(ImageBitN, WindowHandle).
```

	Result
If the images are correct (type) the operator bit_not returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator set_system (<code>::'no_object_result', <Result>:</code>) If necessary an exception handling is raised.	

	Parallelization Information
bit_not is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level, channel level, domain level).	

	Alternatives
bit_or , bit_and , add_image	
	See also
bit_slice , bit_mask	
	Module
Image filters	

bit_or (Image1, Image2 : ImageOr : :)

Bit-by-bit OR of all pixels of the input images.

The operator [bit_or](#) calculates the “or” of all pixels of the input images bit by bit. The semantics of the “or” operation corresponds to that of C for the respective types (signed char, unsigned char, short, unsigned short, int/long). The images must have the same size and pixel type. The pixels within the definition range of the image in the first parameter are processed.

Several images can be processed in one call. In this case both input parameters contain the same number of images which are then processed in pairs. An output image is generated for every pair.

Parameter	
▷ Image1 (input_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4	Input image(s) 1.
▷ Image2 (input_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4	Input image(s) 2.
▷ ImageOr (output_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4	Result image(s) by OR-operation.

Example

```
read_image(Image0, 'affe')
disp_image(Image0, WindowHandle)
read_image(Image1, 'fabrik')
disp_image(Image1, WindowHandle)
bit_or(Image0, Image1, ImageBit0)
disp_image(ImageBit0, WindowHandle).
```

Result

If the images are correct (type and number) the operator `bit_or` returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator `set_system (::'no_object_result', <Result>:)` If necessary an exception handling is raised.

Parallelization Information

`bit_or` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Alternatives

`bit_and`, `add_image`

See also

`bit_xor`, `bit_and`

Module

Image filters

bit_rshift (Image : ImageRShift : Shift :)

Right shift of all pixels of the image.

The operator `bit_rshift` calculates a “right shift” of all pixels of the input image bit by bit. The semantics of the “right shift” operation corresponds to that of C (“>>”) for the respective types (signed char, unsigned char, short, unsigned short, int/long). Only the pixels within the definition range of the image are processed.

Several images can be processed in one call. An output image is generated for every input image.

Parameter	
▷ Image (input_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4	Input image(s).
▷ ImageRShift (output_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4	Result image(s) by shift operation.
▷ Shift (input_control) integer $\leadsto$ <i>integer</i>	shift value
Default Value : 3	
Suggested values : Shift $\in$ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 24, 30, 31}	
Typical range of values : $0 \leq \text{Shift} \leq 31$	
Minimum Increment : 1	
Recommended Increment : 1	
Restriction : (Shift ≥ 1) $\wedge$ (Shift ≤ 31)	

Example

```
bit_rshift(Int2Image, &ReducedInt2Image, 8);
convert_image_type(ReducedInt2Image, &ByteImage, "byte");
```

Result

If the images are correct (type) and **Shift** has a valid value the operator **bit_rshift** returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator **set_system(::'no-object-result', <Result>:)** If necessary an exception handling is raised.

Parallelization Information

bit_rshift is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Alternatives

scale_image

See also

bit_lshift

Module

Image filters

bit_slice (Image : ImageSlice : Bit :)

Extract a bit from the pixels.

The operator **bit_slice** extracts a bit level from the input image. The semantics of the “and” operation corresponds to that of C for the respective types (signed char, unsigned char, short, unsigned short, int/long). Only the pixels within the definition range of the image are processed.

Several images can be processed in one call. An output image is generated for every input image.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4
Input image(s).
- ▷ **ImageSlice** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4
Result image(s) by extraction.
- ▷ **Bit** (input_control) integer $\leadsto$ *integer*
Bit to be selected.
Default Value : 8
Suggested values : Bit $\in$ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 20, 24, 30, 32}
Typical range of values : $1 \leq \text{Bit} \leq 32$
Minimum Increment : 1
Recommended Increment : 1
Restriction : $(\text{Bit} \geq 1) \wedge (\text{Bit} \leq 32)$

Example

```
read_image(&ByteImage, "fabrik");
for (bit=1; bit<=8; i++)
{
    bit_slice(ByteImage, &Slice, bit);
    threshold(Slice, &Region, 0, 255);
    disp_region(Region, WindowHandle);
    clear(bit_slice); clear(Slice); clear(Region);
}
```

Result

If the images are correct (type) and **Bit** has a valid value, the operator **bit_slice** returns the value 2

(H.MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator `set_system(::'no_object_result', <Result>:)` If necessary an exception handling is raised.

Parallelization Information

`bit_slice` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Successors

`threshold`, `bit_or`

Alternatives

`bit_mask`

See also

`bit_and`, `bit_lshift`

Module

Image filters

bit_xor (Image1, Image2 : ImageXor : :)

Bit-by-bit XOR of all pixels of the input images.

The operator `bit_xor` calculates the “xor” of all pixels of the input images bit by bit. The semantics of the “xor” operation corresponds to that of C for the respective types (signed char, unsigned char, short, unsigned short, int/long). The images must have the same size and pixel type. The pixels within the definition range of the image in the first parameter are processed.

Several images can be processed in one call. In this case both input parameters contain the same number of images which are then processed in pairs. An output image is generated for every pair.

Parameter

- ▷ **Image1** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4
Input image(s) 1.
- ▷ **Image2** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4
Input image(s) 2.
- ▷ **ImageXor** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4
Result image(s) by XOR-operation.

Example

```
read_image(Image0, 'affe')
disp_image(Image0, WindowHandle)
read_image(Image1, 'fabrik')
disp_image(Image1, WindowHandle)
bit_xor(Image0, Image1, ImageBitX)
disp_image(ImageBitX, WindowHandle).
```

Result

If the parameter values are correct the operator `bit_xor` returns the value 2 (H.MSG_TRUE). The behavior in case of empty input (no input images available) can be determined by the operator `set_system(::'no_object_result', <Result>:)` If necessary an exception handling is raised.

Parallelization Information

`bit_xor` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Alternatives

`bit_or`, `bit_and`, `add_image`

See also

`bit_or`, `bit_and`

Module

Image filters

5.4 Color

cfa_to_rgb (CFAImage : RGBImage : CFAType, Interpolation :)

Convert a single-channel color filter array image into an RGB image.

cfa_to_rgb converts a single-channel color filter array image **CFAImage** into an RGB image **RGBImage**. Color filter array images are typically generated by single-chip CCD cameras. The conversion from color filter array image to RGB image is typically done on the camera itself or is performed by the device driver of the frame grabber that is used to grab the image. In some cases, however, the device driver simply passes the color filter array image through unchanged. In this case, the corresponding HALCON frame grabber interface typically converts the image into an RGB image. Hence, the operator **cfa_to_rgb** is normally used if the images are not being grabbed using the HALCON frame grabber interface (**grab_image** or **grab_image_async**), but are grabbed using function calls from the frame grabber SDK, and are passed to HALCON using **gen_image1** or **gen_image1_extern**.

In single-chip CCD cameras, a color filter array in front of the sensor provides (subsampled) color information. The most frequently used filter is the so called Bayer filter. The color filter array has the following layout in this case:

G	B	G	B	G	B	...
R	G	R	G	R	G	...
G	B	G	B	G	B	...
R	G	R	G	R	G	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮

Each gray value of the input image **CFAImage** corresponds to the brightness of the pixel behind the corresponding color filter. Hence, in the above layout, the pixel (0,0) corresponds to a green color value, while the pixel (0,1) corresponds to a blue color value. The layout of the Bayer filter is completely determined by the first two elements of the first row of the image, and can be chosen with the parameter **CFAType**. In particular, this enables the correct conversion of color filter array images that have been cropped out of a larger image (e.g., using **crop_part** or **crop_rectangle1**). The algorithm that is used to interpolate the RGB values is determined by the parameter **Interpolation**. Currently, the only possible choice is 'bilinear'.

Parameter

- ▷ **CFAImage** (input_object) image $\leadsto$ Hobject : byte / uint2
Input image.
- ▷ **RGBImage** (output_object) image $\leadsto$ Hobject : byte / uint2
Output image.
- ▷ **CFAType** (input_control) string $\leadsto$ string
Color filter array type.
Default Value : 'bayer_gb'
List of values : CFAType $\in$ { 'bayer_gb', 'bayer_gr', 'bayer_bg', 'bayer_rg' }
- ▷ **Interpolation** (input_control) string $\leadsto$ string
Interpolation type.
Default Value : 'bilinear'
List of values : Interpolation $\in$ { 'bilinear' }

Result

cfa_to_rgb returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via **set_system (:: 'no_object_result', <Result> :)**. If necessary, an exception handling is raised.

Parallelization Information

cfa_to_rgb is *reentrant* and automatically *parallelized* (on tuple level, domain level).

Possible Predecessors

gen_image1_extern, **gen_image1**, **grab_image**

Possible Successors

decompose3

See also

[trans_from_rgb](#)

Module

Image filters

rgb1_to_gray (RGBImage : GrayImage : :)

Transform an RGB image into a gray scale image.

[rgb1_to_gray](#) transforms an RGB image into a gray scale image. The three channels of the RGB image are passed as the first three channels of the input image. The image is transformed according to the following formula:

$$k = 0.299r + 0.587g + 0.144b .$$

Parameter

- ▷ **RGBImage** (input_object) image(-array) $\rightsquigarrow$ Hobject : byte / int2
Three-channel RGB image.
- ▷ **GrayImage** (output_object) image(-array) $\rightsquigarrow$ Hobject : byte / int2
Gray scale image.

Example

```
/* Transformation from rgb to gray */
read_image(Image, 'patras')
disp_color(Image, WindowHandle)
rgb1_to_gray(Image, GrayImage)
disp_image(GrayImage, WindowHandle).
```

Parallelization Information

[rgb1_to_gray](#) is *reentrant* and automatically *parallelized* (on tuple level, domain level).

Possible Predecessors

[compose3](#)

Alternatives

[trans_from_rgb](#), [rgb3_to_gray](#)

Module

Image filters

rgb3_to_gray (ImageRed, ImageGreen, ImageBlue : ImageGray : :)

Transform an RGB image to a gray scale image.

[rgb3_to_gray](#) transforms an RGB image into a gray scale image. The three channels of the RGB image are passed as three separate images. The image is transformed according to the following formula:

$$k = 0.299r + 0.587g + 0.144b .$$

Parameter

- ▷ **ImageRed** (input_object) image(-array) $\rightsquigarrow$ Hobject : byte / int2
Input image (red channel).
- ▷ **ImageGreen** (input_object) image(-array) $\rightsquigarrow$ Hobject : byte / int2
Input image (green channel).
- ▷ **ImageBlue** (input_object) image(-array) $\rightsquigarrow$ Hobject : byte / int2
Input image (blue channel).
- ▷ **ImageGray** (output_object) image(-array) $\rightsquigarrow$ Hobject : byte / int2
Gray scale image.

Example

```

/* Transformation from rgb to gray */
read_image(Image, 'patras')
disp_color(Image, WindowHandle)
decompose3(Image, Rimage, Gimage, Bimage)
rgb3_to_gray(Rimage, Gimage, Bimage, GrayImage)
disp_image(GrayImage, WindowHandle).

```

Parallelization Information

[rgb3_to_gray](#) is *reentrant* and automatically *parallelized* (on tuple level, domain level).

Possible Predecessors

[decompose3](#)

Alternatives

[rgb1_to_gray](#), [trans_from_rgb](#)

Module

Image filters

trans_from_rgb (ImageRed, ImageGreen, ImageBlue : ImageResult1,
ImageResult2, ImageResult3 : ColorSpace :)

Transform an image from the RGB color space to an arbitrary color space.

[trans_from_rgb](#) transforms an image from the RGB color space to an arbitrary color space ([ColorSpace](#)). The three channels of the image are passed as three separate images on input and output.

The following transformations are supported:

'yiq'

$$\begin{pmatrix} Y \\ I \\ Q \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.144 \\ 0.595 & -0.276 & -0.333 \\ 0.209 & -0.522 & 0.287 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

'yuv'

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & 0.100 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

'argyb'

$$\begin{pmatrix} A \\ Rg \\ Yb \end{pmatrix} = \begin{pmatrix} 0.30 & 0.59 & 0.11 \\ 0.50 & -0.50 & 0.00 \\ 0.25 & 0.25 & -0.50 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

'ciexyz'

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 0.476 & 0.299 & 0.175 \\ 0.262 & 0.656 & 0.082 \\ 0.020 & 0.161 & 0.909 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

'hls'

```

min = min(R, G, B)
max = max(R, G, B)
L = (min + max) / 2
if (max == min)
    H = 0
    S = 0
else
    if (L > 0.5)

```

```

        S = (max - min) / (2 - max - min)
    else
        S = (max - min) / (max + min)
    fi
    if (R == max)
        H = ((G - B) / (max - min)) * 60
    elif (G == max)
        H = (2 + (B - R) / (max - min)) * 60
    elif (B == max)
        H = (4 + (R - G) / (max - min)) * 60
    fi
fi
'hsi'

```

$$\begin{pmatrix} M1 \\ M2 \\ I1 \end{pmatrix} = \begin{pmatrix} \frac{2}{\sqrt{6}} & \frac{-1}{\sqrt{6}} & \frac{-1}{\sqrt{6}} \\ 0 & \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \\ \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} & \frac{1}{\sqrt{3}} \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

$$\begin{pmatrix} H \\ S \\ I \end{pmatrix} = \begin{pmatrix} \arctan \frac{M2}{M1} \\ \sqrt{M1^2 + M2^2} \\ \frac{I1}{\sqrt{3}} \end{pmatrix}$$

```

'hsv'    min = min(R,G,B)
        max = max(R,G,B)
        V = max
        if (max == min)
            S = 0
            H = 0
        else
            S = (max - min) / max
            if (R == max)
                H = ((G - B) / (max - min)) * 60
            elif (G == max)
                H = (2 + (B - R) / (max - min)) * 60
            elif (B == max)
                H = (4 + (R - G) / (max - min)) * 60
            fi
        fi
fi
'ihv'    min = min(R,G,B)
        max = max(R,G,B)
        I = (R + G + B) / 3
        if (I == 0)
            H = 0
            S = 1
        else
            S = 1 - min / I
            if (S == 0)
                H = 0
            else
                A = (R + R - G - B) / 2
                B = (R - G) * (R - G) + (R - B) * (G - B)
                C = sqrt(B)
                if (C == 0)
                    H = 0
                else
                    H = acos(A / C)
                fi
                if (B > G)
                    H = 2 * pi - H
                fi
            fi
        fi
fi

```

fi
fi
fi
'cielab'

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 0.476 & 0.299 & 0.175 \\ 0.262 & 0.656 & 0.082 \\ 0.020 & 0.161 & 0.909 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

$$L = 116 * Y^{\frac{1}{3}} - 16$$

$$a = 500 * \left(\frac{X^{\frac{1}{3}}}{0.95} - Y^{\frac{1}{3}} \right)$$

$$b = 200 * \left(Y^{\frac{1}{3}} - \frac{Z^{\frac{1}{3}}}{1.09} \right)$$

'i1i2i3'

$$\begin{pmatrix} I1 \\ I2 \\ I3 \end{pmatrix} = \begin{pmatrix} 0.333 & 0.333 & 0.333 \\ 1.0 & 0.0 & -1.0 \\ -0.5 & 1.0 & -0.5 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

'ciexyz2'

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 0.620 & 0.170 & 0.180 \\ 0.310 & 0.590 & 0.110 \\ 0.000 & 0.066 & 1.020 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

'ciexyz3'

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} 0.618 & 0.177 & 0.205 \\ 0.299 & 0.587 & 0.114 \\ 0.000 & 0.056 & 0.944 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

If one of the following conditions is fulfilled, certain scalings are performed accordingly to the image type:

- Considering byte images, the domain of color space values is generally mapped to the full domain of [0..255]. The origin of signed values is shifted to 128.
- Hue values are represented by angles of [0..2π] and are coded for the particular image types differently:
 - *byte*-images map the angle domain on [0..255].
 - *int4*-images are coded in angle minutes [0..21600].
 - *real*-images are coded in radians [0..2π] .
- *int4*-images map a domain of [0..1] of transformed values on [0..10000].

Parameter

- ▷ **ImageRed** (input_object) image(-array) $\leadsto$ *Hobject* : byte / int4 / real
Input image (red channel).
- ▷ **ImageGreen** (input_object) image(-array) $\leadsto$ *Hobject* : byte / int4 / real
Input image (green channel).
- ▷ **ImageBlue** (input_object) image(-array) $\leadsto$ *Hobject* : byte / int4 / real
Input image (blue channel).
- ▷ **ImageResult1** (output_object) image(-array) $\leadsto$ *Hobject* : byte / int4 / real
Color-transformed output image (channel 1).
- ▷ **ImageResult2** (output_object) image(-array) $\leadsto$ *Hobject* : byte / int4 / real
Color-transformed output image (channel 1).
- ▷ **ImageResult3** (output_object) image(-array) $\leadsto$ *Hobject* : byte / int4 / real
Color-transformed output image (channel 1).
- ▷ **ColorSpace** (input_control) string $\leadsto$ *string*
Color space of the output image.
Default Value : 'hsv'
List of values : ColorSpace $\in$ {'cielab', 'hsv', 'hsi', 'yiq', 'yuv', 'argyb', 'ciexyz', 'ciexyz2', 'ciexyz3', 'hls', 'ihs', 'i1i2i3'}

Example

```

/* Tranformation from rgb to hsv and conversely */
read_image(Image,'patras')
disp_color(Image,WindowHandle)
decompose3(Image,Rimage,Gimage,Bimage)
trans_from_rgb(Rimage,Gimage,Bimage,Image1,Image2,Image3,'hsv')
trans_to_rgb(Image1,Image2,Image3,ImageRed,ImageGreen,ImageBlue,'hsv')
compose3(ImageRed,ImageGreen,ImageBlue,Multichannel)
disp_color(Multichannel,WindowHandle).

```

Result

[trans_from_rgb](#) returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via [set_system\(:,:, 'no_object_result', <Result>:\)](#). If necessary, an exception handling is raised.

Parallelization Information

[trans_from_rgb](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *domain level*).

Possible Predecessors

[decompose3](#)

Possible Successors

[compose3](#)

Alternatives

[rgb1_to_gray](#), [rgb3_to_gray](#)

See also

[trans_to_rgb](#)

Module

Image filters

trans_to_rgb (ImageInput1, ImageInput2, ImageInput3 : ImageRed,
ImageGreen, ImageBlue : ColorSpace :)

Transform an image from an arbitrary color space to the RGB color space.

[trans_to_rgb](#) transforms an image from an arbitrary color space ([ColorSpace](#)) to the RGB color space. The three channels of the image are passed as three separate images on input and output.

The following transformations are supported:

'yiq'

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 0.999 & 0.962 & 0.615 \\ 0.949 & -0.220 & -0.732 \\ 0.999 & -1.101 & 1.706 \end{pmatrix} \begin{pmatrix} Y \\ I \\ Q \end{pmatrix}$$

'yuv'

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.0 & 0.0 & 1.140 \\ 1.0 & -0.394 & -0.581 \\ 1.0 & 2.032 & 0.0 \end{pmatrix} \begin{pmatrix} Y \\ U \\ V \end{pmatrix}$$

'argyb'

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 1.00 & 1.29 & 0.22 \\ 1.00 & -0.71 & 0.22 \\ 1.00 & 0.29 & -1.78 \end{pmatrix} \begin{pmatrix} A \\ Rg \\ Yb \end{pmatrix}$$

'ciexyz'

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} 2.750 & -1.149 & -0.426 \\ -1.118 & 2.026 & 0.033 \\ 0.138 & -0.333 & 1.104 \end{pmatrix} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix}$$

```

'hls'    Hi = integer(H * 6)
         Hf = fraction(H * 6)
         if (L <= 0.5)
             max = L * (S + 1)
         else
             max = L + S - (L * S)
         fi
         min = 2 * L - max
         if (S == 0)
             R = L
             G = L
             B = L
         else
             if (Hi == 0)
                 R = max
                 G = min + Hf * (max - min)
                 B = min
             elif (Hi == 1)
                 R = min + (1 - Hf) * (max - min)
                 G = max
                 B = min
             elif (Hi == 2)
                 R = min
                 G = max
                 B = min + Hf * (max - min)
             elif (Hi == 3)
                 R = min
                 G = min + (1 - Hf) * (max - min)
                 B = max
             elif (Hi == 4)
                 R = min + Hf * (max - min)
                 G = min
                 B = max
             elif (Hi == 5)
                 R = max
                 G = min
                 B = min + (1 - Hf) * (max - min)
             fi
         fi
'hsi'

```

$$M1 = S * \sin H$$

$$M2 = S * \cos H$$

$$I1 = \frac{I}{\sqrt{3}}$$

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = \begin{pmatrix} \frac{2}{\sqrt{6}} & 0 & \frac{1}{\sqrt{3}} \\ \frac{-1}{\sqrt{6}} & \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{3}} \\ \frac{-1}{\sqrt{6}} & \frac{-1}{\sqrt{2}} & \frac{1}{\sqrt{3}} \end{pmatrix} \begin{pmatrix} M1 \\ M2 \\ I1 \end{pmatrix}$$

```

'hsv'    if (S == 0)
         R = V
         G = V
         B = V
         else
             Hi = integer(H)
             Hf = fraction(H)
             if (Hi == 0)
                 R = V

```

```

        G = V * (1 - (S * (1 - Hf)))
        B = V * (1 - S)
    elif (Hi == 1)
        R = V * (1 - (S * Hf))
        G = V
        B = V * (1 - S)
    elif (Hi == 2)
        R = V * (1 - S)
        G = V
        B = V * (1 - (S * (1 - Hf)))
    elif (Hi == 3)
        R = V * (1 - S)
        G = V * (1 - (S * Hf))
        B = V
    elif (Hi == 4)
        R = V * (1 - (S * (1 - Hf)))
        G = V * (1 - S)
        B = V
    elif (Hi == 5)
        R = V
        G = V * (1 - S)
        B = V * (1 - (S * Hf))
    fi
fi

```

If one of the following conditions is fulfilled, certain scalings are expected accordingly to the image type:

- Considering byte images, the domain of color space values is generally expected to be spread to the full domain of $[0..255]$. The origin of signed values must be shifted to 128.
- Hue values are represented by angles of $[0..2\pi]$ and are coded for the particular image types differently:
 - *byte*-images map the angle domain on $[0..255]$.
 - *int4*-images are coded in angle minutes $[0..21600]$.
 - *real*-images are coded in radians $[0..2\pi]$.
- For *int4*-images a domain of $[0..1]$ of transformed values must be spread to $[0..10000]$.

Parameter

- ▷ **ImageInput1** (input_object) image(-array) $\leadsto$ *Hobject*: byte / int4 / real
Input image (channel 1).
- ▷ **ImageInput2** (input_object) image(-array) $\leadsto$ *Hobject*: byte / int4 / real
Input image (channel 2).
- ▷ **ImageInput3** (input_object) image(-array) $\leadsto$ *Hobject*: byte / int4 / real
Input image (channel 3).
- ▷ **ImageRed** (output_object) image(-array) $\leadsto$ *Hobject*: byte / int4 / real
Red channel.
- ▷ **ImageGreen** (output_object) image(-array) $\leadsto$ *Hobject*: byte / int4 / real
Green channel.
- ▷ **ImageBlue** (output_object) image(-array) $\leadsto$ *Hobject*: byte / int4 / real
Blue channel.
- ▷ **ColorSpace** (input_control) string $\leadsto$ *string*
Color space of the input image.
Default Value: 'hsv'
List of values: ColorSpace $\in \{ 'hsi', 'yiq', 'yuv', 'argyb', 'ciexyz', 'hls', 'hsv' \}$

Example

```

/* Tranformation from rgb to hsv and conversely */
read_image(Image, 'patras')

```



```

disp_color(Image, WindowHandle)
decompose3(Image, Rimage, Gimage, Bimage)
trans_from_rgb(Rimage, Gimage, Bimage, Image1, Image2, Image3, 'hsv')
trans_to_rgb(Image1, Image2, Image3, ImageRed, ImageGreen, ImageBlue, 'hsv')
compose3(ImageRed, ImageGreen, ImageBlue, Multichannel)
disp_color(Multichannel, WindowHandle).

```

Result

trans_to_rgb returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via **set_system(, 'no_object_result', <Result>:)**. If necessary, an exception handling is raised.

Parallelization Information

trans_to_rgb is *reentrant* and automatically *parallelized* (on tuple level, domain level).

Possible Predecessors

decompose3

Possible Successors

compose3, disp_color

See also

decompose3

Module

Image filters

5.5 Edges

close_edges (Edges, EdgeImage : RegionResult : MinAmplitude :)

Close edge gaps using the edge amplitude image.

close_edges closes gaps in the output of an edge detector, and thus tries to produce complete object contours. This is done by examining the neighbors of each edge point to determine the point with maximum amplitude (i.e., maximum gradient), and adding the point to the edge if its amplitude is larger than the minimum amplitude passed in **MinAmplitude**. This operator expects as input the edges (**Edges**) and amplitude image (**EdgeImage**) returned by typical edge operators, such as **edges_image** or **sobel_amp**. **close_edges** does not take into account the edge directions that may be returned by an edge operator. Thus, in areas where the gradient is almost constant the edges may become rather “wiggly.”

Parameter

- ▷ **Edges** (input_object) region(-array) $\leadsto$ *Hobject*
Region containing one pixel thick edges.
- ▷ **EdgeImage** (input_object) image $\leadsto$ *Hobject* : byte / int4
Edge amplitude (gradient) image.
- ▷ **RegionResult** (output_object) region(-array) $\leadsto$ *Hobject*
Region containing closed edges.
- ▷ **MinAmplitude** (input_control) integer $\leadsto$ *integer*
Minimum edge amplitude.
Default Value : 16
Suggested values : MinAmplitude $\in \{5, 8, 10, 12, 16, 20, 25, 30, 40, 50\}$
Typical range of values : $1 \leq \text{MinAmplitude} \leq 255$
Minimum Increment : 1
Recommended Increment : 1
Restriction : MinAmplitude ≥ 0

Example

```

sobel_amp(Image, &EdgeAmp, "sum_abs", 5);
threshold(EdgeAmp, &EdgeRegion, 40.0, 255.0);

```

```
skeleton(EdgeRegion,&ThinEdge);
close_edges(ThinEdge,EdgeAmp,&CloseEdges,15);
skeleton(CloseEdges,&ThinCloseEdges);
```

Result

`close_edges` returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via `set_system('no_object_result',<Result>)`. If necessary, an exception handling is raised.

Parallelization Information

`close_edges` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`edges_image`, `sobel_amp`, `threshold`, `skeleton`

Possible Successors

`skeleton`

Alternatives

`close_edges_length`, `dilation1`, `closing`

See also

`gray_skeleton`

Module

Image filters

close_edges_length (Edges, Gradient : ClosedEdges : MinAmplitude, MaxGapLength :)

Close edge gaps using the edge amplitude image.

`close_edges_length` closes gaps in the output of an edge detector, and thus tries to produce complete object contours. This operator expects as input the edges (`Edges`) and amplitude image (`Gradient`) returned by typical edge operators, such as `edges_image` or `sobel_amp`.

Contours are closed in two steps: First, one pixel wide gaps in the input contours are closed, and isolated points are eliminated. After this, open contours are extended by up to `MaxGapLength` points by adding edge points until either the contour is closed or no more significant edge points can be found. A gradient is regarded as significant if it is larger than `MinAmplitude`. The neighboring points examined as possible new edge points are the point in the direction of the contour and its two adjacent points in an 8-neighborhood. For each of these points, the sum of its gradient and the maximum gradient of that points three possible neighbors is calculated (*look ahead* of length 1). The point with the maximum sum is then chosen as the new edge point.

Parameter

- ▷ **Edges** (input_object) region(-array) $\leadsto$ *Hobject*
Region containing one pixel thick edges.
- ▷ **Gradient** (input_object) image $\leadsto$ *Hobject* : byte
Edge amplitude (gradient) image.
- ▷ **ClosedEdges** (output_object) region(-array) $\leadsto$ *Hobject*
Region containing closed edges.
- ▷ **MinAmplitude** (input_control) integer $\leadsto$ *integer*
Minimum edge amplitude.
Default Value : 16
Suggested values : `MinAmplitude` $\in$ {5, 8, 10, 12, 16, 20, 25, 30, 40, 50}
Typical range of values : $1 \leq \text{MinAmplitude} \leq 255$
Minimum Increment : 1
Recommended Increment : 1
Restriction : `MinAmplitude` ≥ 0

- ▷ **MaxGapLength** (input_control) integer $\leadsto$ integer
 Maximal number of points by which edges are extended.
Default Value : 3
Suggested values : MaxGapLength $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 20, 30, 40, 50, 70, 100\}$
Typical range of values : $1 \leq \text{MaxGapLength} \leq 127$
Minimum Increment : 1
Recommended Increment : 1
Restriction : $(\text{MaxGapLength} > 0) \wedge (\text{MaxGapLength} \leq 127)$

Example

```
sobel_amp(Image, &EdgeAmp, "sum_abs", 5);
threshold(EdgeAmp, &EdgeRegion, 40.0, 255.0);
skeleton(EdgeRegion, &ThinEdge);
close_edges_length(ThinEdge, EdgeAmp, &CloseEdges, 15, 3);
```

Result

close_edges_length returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via **set_system('no_object_result', <Result>)**. If necessary, an exception handling is raised.

Parallelization Information

close_edges_length is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

edges_image, **sobel_amp**, **threshold**, **skeleton**

Alternatives

close_edges, **dilation1**, **closing**

References

M. Üsbeck: "Untersuchungen zur echtzeitfähigen Segmentierung"; Studienarbeit, Bayerisches Forschungszentrum für Wissensbasierte Systeme (FORWISS), Erlangen, 1993.

Module

Image filters

derivate_gauss (Image : DerivGauss : Sigma, Component :)

Convolve an image with derivatives of the Gaussian.

derivate_gauss convolves an image with the derivatives of a Gaussian and calculates various features derived thereof. Possible values for **Component** are:

'none' Smoothing only.

'x' First derivative along x.

$$g'(x, y) = \frac{\partial g(x, y)}{\partial x}$$

'y' First derivative along y.

$$g'(x, y) = \frac{\partial g(x, y)}{\partial y}$$

'gradient' Absolute value of the gradient.

$$g'(x, y) = \sqrt{\frac{\partial g(x, y)^2}{\partial x} \frac{\partial g(x, y)^2}{\partial y}}$$

'gradient_dir' Gradient direction in radians

$$\phi = \text{atan2}\left(\frac{\partial g(x, y)}{\partial y}, \frac{\partial g(x, y)}{\partial x}\right)$$

'xx' Second derivative along x.

$$g'(x, y) = \frac{\partial^2 g(x, y)}{\partial x^2}$$

'yy' Second derivative along y.

$$g'(x, y) = \frac{\partial^2 g(x, y)}{\partial y^2}$$

'xy' Second derivative along x and y.

$$g'(x, y) = \frac{\partial^2 g(x, y)}{\partial x \partial y}$$

'xxx' Third derivative along x.

$$g'(x, y) = \frac{\partial^3 g(x, y)}{\partial x^3}$$

'yyy' Third derivative along y.

$$g'(x, y) = \frac{\partial^3 g(x, y)}{\partial y^3}$$

'xxy' Third derivative along x, x and y.

$$g'(x, y) = \frac{\partial^3 g(x, y)}{\partial x^2 \partial y}$$

'xyy' Third derivative along x, y and y.

$$g'(x, y) = \frac{\partial^3 g(x, y)}{\partial x \partial y^2}$$

'det' Determinant of the Hessian matrix:

$$DET = \frac{\partial^2 g(x, y)}{\partial x^2} \frac{\partial^2 g(x, y)}{\partial y^2} - \left(\frac{\partial^2 g(x, y)}{\partial y \partial x} \right)^2$$

'laplace' Laplace operator (trace of the Hessian matrix):

$$TR = \frac{\partial^2 g(x, y)}{\partial x^2} + \frac{\partial^2 g(x, y)}{\partial y^2}$$

'mean_curvature' Mean curvature H

$$\begin{aligned} a &= \left(1 + \frac{\partial g(x, y)^2}{\partial x}\right) \frac{\partial^2 g(x, y)}{\partial y^2} \\ b &= 2 \frac{\partial g(x, y)}{\partial x} \frac{\partial g(x, y)}{\partial y} \frac{\partial^2 g(x, y)}{\partial y \partial x} \\ c &= \left(1 + \frac{\partial g(x, y)^2}{\partial y}\right) \frac{\partial^2 g(x, y)}{\partial x^2} \\ d &= \left(1 + \frac{\partial g(x, y)^2}{\partial x} + \frac{\partial g(x, y)^2}{\partial y}\right)^{\frac{3}{2}} \\ H &= \frac{a - b + c}{d} \\ H &= \frac{1}{2}(\kappa_{min} + \kappa_{max}) \end{aligned}$$

'gauss_curvature' Gaussian curvature K

$$K = \frac{DET}{\left(1 + \frac{\partial g(x, y)^2}{\partial x} + \frac{\partial g(x, y)^2}{\partial y}\right)^2}$$

'area' Differential Area A

$$\begin{aligned} A &= EG - F^2 \\ E &= 1 + \frac{\partial g(x, y)^2}{\partial x} \\ F &= \frac{\partial g(x, y)}{\partial x} \frac{\partial g(x, y)}{\partial y} \\ G &= 1 + \frac{\partial g(x, y)^2}{\partial y} \end{aligned}$$

'eigenvalue1' First eigenvalue

$$a = \frac{\frac{\partial^2 g(x,y)}{\partial x^2} + \frac{\partial^2 g(x,y)}{\partial y^2}}{2}$$

$$\lambda_1 = a + \sqrt{a^2 - \left(\frac{\partial^2 g(x,y)}{\partial x^2} \frac{\partial^2 g(x,y)}{\partial y^2} - \frac{\partial^2 g(x,y)^2}{\partial y \partial x} \right)}$$

'eigenvalue2' Second eigenvalue

$$a = \frac{\frac{\partial^2 g(x,y)}{\partial x^2} + \frac{\partial^2 g(x,y)}{\partial y^2}}{2}$$

$$\lambda_2 = a - \sqrt{a^2 - \left(\frac{\partial^2 g(x,y)}{\partial x^2} \frac{\partial^2 g(x,y)}{\partial y^2} - \frac{\partial^2 g(x,y)^2}{\partial y \partial x} \right)}$$

'eigenvec_dir' Direction of the eigenvector corresponding to the first eigenvalue in radians

'main1_curvature' First principal curvature

$$\kappa_{max} = H + \sqrt{H^2 - K}$$

'main2_curvature' Second principal curvature

$$\kappa_{min} = H - \sqrt{H^2 - K}$$

'kitchen_rosenfeld' Second derivative perpendicular to the gradient

$$k = \frac{\frac{\partial^2 g(x,y)}{\partial x^2} \frac{\partial g(x,y)^2}{\partial y} + \frac{\partial^2 g(x,y)}{\partial y^2} \frac{\partial g(x,y)^2}{\partial x} - 2 \frac{\partial^2 g(x,y)}{\partial y \partial x} \frac{\partial g(x,y)^2}{\partial x} \frac{\partial g(x,y)^2}{\partial y}}{\frac{\partial g(x,y)^2}{\partial x} + \frac{\partial g(x,y)^2}{\partial y}}$$

'zuniga_haralick' Normalized second derivative perpendicular to the gradient

$$k = \frac{\frac{\partial^2 g(x,y)}{\partial x^2} \frac{\partial g(x,y)^2}{\partial y} + \frac{\partial^2 g(x,y)}{\partial y^2} \frac{\partial g(x,y)^2}{\partial x} - 2 \frac{\partial^2 g(x,y)}{\partial y \partial x} \frac{\partial g(x,y)^2}{\partial x} \frac{\partial g(x,y)^2}{\partial y}}{\left(\frac{\partial g(x,y)^2}{\partial x} + \frac{\partial g(x,y)^2}{\partial y} \right)^{\frac{3}{2}}}$$

'2nd_ddg' Second derivative along the gradient

$$k = \frac{\frac{\partial^2 g(x,y)}{\partial x^2} \frac{\partial g(x,y)^2}{\partial x} + 2 \frac{\partial g(x,y)}{\partial x} \frac{\partial g(x,y)}{\partial y} \frac{\partial^2 g(x,y)}{\partial y \partial x} + \frac{\partial^2 g(x,y)}{\partial y^2} \frac{\partial g(x,y)^2}{\partial y}}{\frac{\partial g(x,y)^2}{\partial x} + \frac{\partial g(x,y)^2}{\partial y}}$$

'de_saint_venant' Second derivative along and perpendicular to the gradient

$$k = \frac{\frac{\partial g(x,y)}{\partial x} \frac{\partial g(x,y)}{\partial y} \left(\frac{\partial^2 g(x,y)}{\partial x^2} - \frac{\partial^2 g(x,y)}{\partial y^2} \right) - \left(\frac{\partial g(x,y)^2}{\partial x} - \frac{\partial g(x,y)^2}{\partial y} \right) \frac{\partial^2 g(x,y)}{\partial x \partial y}}{\frac{\partial g(x,y)^2}{\partial x} + \frac{\partial g(x,y)^2}{\partial y}}$$

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Input image.
- ▷ **DerivGauss** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : real
Filtered result image.
- ▷ **Sigma** (input_control) real $\leadsto$ *real*
Sigma of the Gaussian.
Default Value : 1.0
Suggested values : Sigma $\in$ {0.7, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0}
Typical range of values : 0.2 $\leq$ Sigma $\leq$ 50.0
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : Sigma > 0.0

- ▷ **Component** (input_control) string $\leadsto$ string
Derivative or feature to be calculated.
Default Value : 'x'
List of values : Component $\in \{ \text{'none', 'x', 'y', 'gradient', 'xx', 'yy', 'xy', 'xxx', 'yyy', 'xxy', 'xyy', 'det', 'mean_curvature', 'gauss_curvature', 'eigenvalue1', 'eigenvalue2', 'main1_curvature', 'main2_curvature', 'kitchen_rosenfeld', 'zuniga_haralick', '2nd_ddg', 'de_saint_venant', 'area', 'laplace', 'gradient_dir', 'eigenvec_dir'} \}$

Example

```
read_image(&Image, "mreut");
derivate_gauss(Image, &Gauss, 3.0, "x");
zero_crossing(Gauss, &ZeroCrossings);
```

Parallelization Information

`derivate_gauss` is *reentrant* and automatically *parallelized* (on tuple level, channel level, domain level).

Possible Successors

`zero_crossing`, `dual_threshold`

Alternatives

`laplace`, `laplace_of_gauss`, `gauss_image`, `smooth_image`

See also

`zero_crossing`, `dual_threshold`

Module

Image filters

diff_of_gauss (Image : DiffOfGauss : Sigma, SigFactor :)

Approximate the LoG operator (Laplace of Gaussian).

`diff_of_gauss` approximates the Laplace-of-Gauss operator by a difference of Gaussians. The standard deviations of these Gaussians can be calculated, according to Marr, from the Parameter `Sigma` of the LoG and the ratio of the two standard deviations (`SigFactor`) as:

$$\sigma_1 = \frac{\text{Sigma}}{\sqrt{-2 \frac{\log(\frac{1}{\text{SigFactor}})}{\text{SigFactor}^2 - 1}}}$$

$$\sigma_2 = \frac{\sigma_1}{\text{SigFactor}}$$

$$\text{DiffOfGauss} = (\text{Image} * \text{gauss}(\sigma_1)) - (\text{Image} * \text{gauss}(\sigma_2))$$

For a `SigFactor` = 1.6, according to Marr, an approximation to the Mexican-Hat-Operator results. The resulting image is stored in `DiffOfGauss`.

Parameter

- ▷ **Image** (input_object)(multichannel-)image(-array) $\leadsto$ *Hobject* : byte
Input image
- ▷ **DiffOfGauss** (output_object)(multichannel-)image(-array) $\leadsto$ *Hobject* : int2
LoG image.
- ▷ **Sigma** (input_control) real $\leadsto$ real
Smoothing parameter of the Laplace operator to approximate.
Default Value : 3.0
Suggested values : Sigma $\in \{2.0, 3.0, 4.0, 5.0\}$
Typical range of values : $0.2 \leq \text{Sigma} \leq 50.0$
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : Sigma > 0.0

- ▷ **SigFactor** (input_control) real $\leadsto$ real
 Ratio of the standard deviations used (Marr recommends 1.6).
Default Value : 1.6
Typical range of values : $0.1 \leq \text{SigFactor} \leq 10.0$
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : $\text{SigFactor} > 0.0$

Example

```
read_image(Image, 'fabrik')
diff_of_gauss(Image, Laplace, 2.0, 1.6)
zero_crossing(Laplace, ZeroCrossings).
```

Complexity

The execution time depends linearly on the number of pixels and the size of sigma.

Result

diff_of_gauss returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via **set_system('no_object_result', <Result>)**. If necessary, an exception handling is raised.

Parallelization Information

diff_of_gauss is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Successors

zero_crossing, **dual_threshold**

Alternatives

laplace, **derivate_gauss**

References

D. Marr: "Vision (A computational investigation into human representation and processing of visual information)"; New York, W.H. Freeman and Company; 1982.

Module

Image filters

edges_image (Image : ImaAmp, ImaDir : Filter, Alpha, NMS, Low, High :)

Extract edges using Deriche, Lanser, Shen, or Canny filters.

edges_image detects step edges using recursively implemented filters (according to Deriche, Lanser and Shen) or the conventionally implemented "derivative of Gaussian" filter (using filter masks) proposed by Canny. Thus, the following edge operators are available:

'deriche1', 'lanser1', 'deriche1_int4', 'deriche2', 'lanser2', 'deriche2_int4', 'shen', 'mshen', and 'canny' (parameter **Filter**).

The edge amplitudes (gradient magnitude) and directions are returned in **ImaAmp** and **ImaDir**, respectively. The edge operators 'deriche1' bzw. 'deriche2' are also available for int4-images, and return the signed filter response instead of its absolute value. This behavior can be obtained for byte-images as well by selecting 'deriche1_int4' bzw. 'deriche2_int4' as filter. This can be used to calculate the second derivative of an image by applying **edges_image** (with parameter 'lanser2') to the signed first derivative. Edge directions are stored in 2-degree steps, i.e., an edge direction of x degrees with respect to the horizontal axis is stored as $x/2$ in the edge direction image. Furthermore, the direction of the change of intensity is taken into account. Let $[E_x, E_y]$ denote the image gradient. Then the following edge directions are returned as $r/2$:

intensity increase	E_x/E_y	
edge direction r		
from bottom to top	0/+	0
from lower right to upper left	+/-	]0, 90[
from right to left	+ / 0	90
from upper right to lower left	+ / +	]90, 180[
from top to bottom	0 / +	180
from upper left to lower right	- / +	]180, 270[
from left to right	+ / 0	270
from lower left to upper right	- / -	]270, 360[

Points with edge amplitude 0 are assigned the edge direction 255 (undefined direction).

The “filter width” (i.e., the amount of smoothing) can be chosen arbitrarily, and can be estimated by calling `info_edges` for concrete values of the parameter `Alpha`. It decreases for increasing `Alpha` for the Deriche, Lanser and Shen filters and increases for the Canny filter, where it is the standard deviation of the Gaussian on which the Canny operator is based. “Wide” filters exhibit a larger invariance to noise, but also a decreased ability to detect small details. Non-recursive filters, such as the Canny filter, are realized using filter masks, and thus the execution time increases for increasing filter width. In contrast, the execution time for recursive filters does not depend on the filter width. Thus, arbitrary filter widths are possible using the Deriche, Lanser and Shen filters without increasing the run time of the operator. The resulting advantage in speed compared to the Canny operator naturally increases for larger filter widths. As border treatment, the recursive operators assume that the images to be zero outside of the image, while the Canny operator repeats the gray value at the image’s border. Comparable filter widths can be obtained by the following choices of `Alpha`:

```
Alpha('lanser1')  = Alpha('deriche1')
Alpha('deriche2') = Alpha('deriche1')/2
Alpha('lanser2')  = Alpha('deriche2')
Alpha('shen')     = Alpha('deriche1')/2
Alpha('mshen')    = Alpha('shen')
Alpha('canny')    = 1.77/Alpha('deriche1')
```

The originally proposed recursive filters ('deriche1', 'deriche2', 'shen') return a biased estimate of the amplitude of diagonal edges. This bias is removed in the corresponding modified version of the operators ('lanser1', 'lanser2' und 'mshen'), while maintaining the same execution speed.

For relatively small filter widths (11×11), i.e., for `Alpha('lanser2') = 0.5`), all filters yield similar results. Only for “wider” filters differences begin to appear: the Shen filters begin to yield qualitatively inferior results. However, they are the fastest of the implemented operators — closely followed by the Deriche operators.

`edges_image` optionally offers to apply a non-maximum-suppression (`NMS = 'nms'/'inms'/'hvnms'; 'none'` if not desired) and hysteresis threshold operation (`Low,High`; at least one negative if not desired) to the resulting edge image. Conceptually, this corresponds to the following calls:

```
nonmax_suppression_dir(..., NMS, ...)
hysteresis_threshold(..., Low, High, 999, ...)
```

However, this calling sequence would return appropriately modified regions, whereas `edges_image` marks suppressed edge points by 0 in the amplitude and 255 in the direction image, while leaving the region of the image unchanged.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / uint2 / int4
Input image.
- ▷ **ImaAmp** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / uint2 / int4 / real
Edge amplitude (gradient magnitude) image.
- ▷ **ImaDir** (output_object) image(-array) $\leadsto$ *Hobject* : direction
Edge direction image.

- ▷ **Filter** (input_control) string $\leadsto$ string
Edge operator to be applied.
Default Value : 'lanser2'
List of values : Filter $\in \{ \text{'deriche1', 'deriche1_int4', 'deriche2', 'deriche2_int4', 'lanser1', 'lanser2', 'shen', 'mshen', 'canny'} \}$
- ▷ **Alpha** (input_control) real $\leadsto$ real
Filter parameter: small values result in strong smoothing, and thus less detail (opposite for 'canny').
Default Value : 0.5
Suggested values : Alpha $\in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 0.9, 1.1\}$
Typical range of values : $0.2 \leq \text{Alpha} \leq 50.0$
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : Alpha > 0.0
- ▷ **NMS** (input_control) string $\leadsto$ string
Non-maximum suppression ('none', if not desired).
Default Value : 'nms'
List of values : NMS $\in \{ \text{'nms', 'inms', 'hvnms', 'none'} \}$
- ▷ **Low** (input_control) integer $\leadsto$ integer
Lower threshold for the hysteresis threshold operation (negative, if no thresholding is desired).
Default Value : 20
Suggested values : Low $\in \{5, 10, 15, 20, 25, 30, 40\}$
Typical range of values : $1 \leq \text{Low} \leq 255$
Minimum Increment : 1
Recommended Increment : 5
Restriction : $(\text{Low} > 1) \vee (\text{Low} < 0)$
- ▷ **High** (input_control) integer $\leadsto$ integer
Upper threshold for the hysteresis threshold operation (negative, if no thresholding is desired).
Default Value : 40
Suggested values : High $\in \{10, 15, 20, 25, 30, 40, 50, 60, 70\}$
Typical range of values : $1 \leq \text{High} \leq 255$
Minimum Increment : 1
Recommended Increment : 5
Restriction : $((\text{High} > 1) \vee (\text{High} < 0)) \wedge (\text{High} \geq \text{Low})$

Example

```
read_image(Image, 'fabrik')
edges_image(Image, Amp, Dir, 'lanser2', 0.5, 'none', -1, -1)
hysteresis_threshold(Amp, Margin, 20, 30, 30).
```

Result

[edges_image](#) returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during execution. If the input is empty the behaviour can be set via [set_system\('no_object_result', <Result>\)](#). If necessary, an exception handling is raised.

Parallelization Information

[edges_image](#) is *reentrant* and automatically *parallelized* (on tuple level, channel level).

Possible Predecessors

[info_edges](#)

Possible Successors

[threshold](#), [hysteresis_threshold](#), [close_edges_length](#)

Alternatives

[sobel_dir](#), [frei_dir](#), [kirsch_dir](#), [prewitt_dir](#), [robinson_dir](#)

See also

[info_edges](#), [nonmax_suppression_amp](#), [hysteresis_threshold](#), [bandpass_image](#)

References

S.Lanser, W.Eckstein: "Eine Modifikation des Deriche-Verfahrens zur Kantendetektion"; 13. DAGM-Symposium, München; Informatik Fachberichte 290; Seite 151 - 158; Springer-Verlag; 1991.

S.Lanser: “Detektion von Stufenkanten mittels rekursiver Filter nach Deriche”; Diplomarbeit; Technische Universität München, Institut für Informatik, Lehrstuhl Prof. Radig; 1991.

J.Canny: “Finding Edges and Lines in Images”; Report, AI-TR-720; M.I.T. Artificial Intelligence Lab., Cambridge; 1983.

J.Canny: “A Computational Approach to Edge Detection”; IEEE Transactions on Pattern Analysis and Machine Intelligence; PAMI-8, vol. 6; S. 679-698; 1986.

R.Deriche: “Using Canny’s Criteria to Derive a Recursively Implemented Optimal Edge Detector”; International Journal of Computer Vision; vol. 1, no. 2; S. 167-187; 1987.

R.Deriche: “Optimal Edge Detection Using Recursive Filtering”; Proc. of the First International Conference on Computer Vision, London; S. 501-505; 1987.

R.Deriche: “Fast Algorithms for Low-Level Vision”; IEEE Transactions on Pattern Analysis and Machine Intelligence; PAMI-12, no. 1; S. 78-87; 1990.

S.Castan, J.Zhao und J.Shen: “Optimal Filter for Edge Detection Methods and Results”; Proc. of the First European Conference on Computer Vision, Antibes; Lecture Notes on computer Science; no. 427; S. 12-17; Springer-Verlag; 1990.

Module

Image filters

edges_sub_pix (Image : Edges : Filter, Alpha, Low, High :)

Extract sub-pixel precise edges using Deriche, Lanser, Shen, or Canny filters.

edges_sub_pix detects step edges using recursively implemented filters (according to Deriche, Lanser and Shen) or the conventionally implemented “derivative of Gaussian” filter (using filter masks) proposed by Canny. Thus, the following edge operators are available:

'deriche1', 'lanser1', 'deriche2', 'lanser2', 'shen', 'mshen', 'canny' and 'sobel'

(parameter **Filter**).

The extracted edges are returned as sub-pixel precise XLD contours in **Edges**. For each edge point the following attributes are defined (see **get_contour_attrib_xld**):

'edge_direction'	Edge direction
'angle'	Direction of the normal vectors to the contour (oriented such that the normal vectors point to the right side of the contour as the contour is traversed from start to end point; the angles are given with respect to the row axis of the image.)
'response'	Edge amplitude (gradient magnitude)

The “filter width” (i.e., the amount of smoothing) can be chosen arbitrarily, and can be estimated by calling **info_edges** for concrete values of the parameter **Alpha**. It decreases for increasing **Alpha** for the Deriche, Lanser and Shen filters and increases for the Canny filter, where it is the standard deviation of the Gaussian on which the Canny operator is based. “Wide” filters exhibit a larger invariance to noise, but also a decreased ability to detect small details. Non-recursive filters, such as the Canny filter, are realized using filter masks, and thus the execution time increases for increasing filter width. In contrast, the execution time for recursive filters does not depend on the filter width. Thus, arbitrary filter widths are possible using the Deriche, Lanser and Shen filters without increasing the run time of the operator. The resulting advantage in speed compared to the Canny operator naturally increases for larger filter widths. As border treatment, the recursive operators assume that the images to be zero outside of the image, while the Canny operator repeats the gray value at the image’s border. Comparable filter widths can be obtained by the following choices of **Alpha**:

Alpha ('lanser1')	=	Alpha ('deriche1')
Alpha ('deriche2')	=	Alpha ('deriche1')/2
Alpha ('lanser2')	=	Alpha ('deriche2')
Alpha ('shen')	=	Alpha ('deriche1')/2
Alpha ('mshen')	=	Alpha ('shen')
Alpha ('canny')	=	1.77/ Alpha ('deriche1')

The originally proposed recursive filters ('deriche1', 'deriche2', 'shen') return a biased estimate of the amplitude of diagonal edges. This bias is removed in the corresponding modified version of the operators ('lanser1', 'lanser2' und 'mshen'), while maintaining the same execution speed.

For relatively small filter widths (11×11), i.e., for Alpha ('lanser2' = 0.5), all filters yield similar results. Only for "wider" filters differences begin to appear: the Shen filters begin to yield qualitatively inferior results. However, they are the fastest of the implemented operators — closely followed by the Deriche operators.

`edges_sub_pix` links the edge points into edges by using an algorithm similar to a hysteresis threshold operation, which is also used in `lines_gauss`. Points with an amplitude larger than `High` are immediately accepted as belonging to an edge, while points with an amplitude smaller than `Low` are rejected. All other points are accepted as edges if they are connected to accepted edge points (see also `lines_gauss` and `hysteresis_threshold`).

Because edge extractors are often unable to extract certain junctions, a mode that tries to extract these missing junctions by different means can be selected by appending '_junctions' to the values of `Filter` that are described above. This mode is analogous to the mode for completing junctions that is available in `lines_gauss`.

Parameter

- ▷ **Image** (input_object) image $\leadsto$ *Hobject*: byte / uint2
Input image.
- ▷ **Edges** (output_object) xld_cont-array $\leadsto$ *Hobject*
Extracted edges.
- ▷ **Filter** (input_control) string $\leadsto$ *string*
Edge operator to be applied.
Default Value : 'lanser2'
List of values : `Filter` $\in$ {'deriche1', 'lanser1', 'deriche2', 'lanser2', 'shen', 'mshen', 'canny', 'sobel', 'deriche1_junctions', 'lanser1_junctions', 'deriche2_junctions', 'lanser2_junctions', 'shen_junctions', 'mshen_junctions', 'canny_junctions', 'sobel_junctions'}
- ▷ **Alpha** (input_control) real $\leadsto$ *real*
Filter parameter: small values result in strong smoothing, and thus less detail (opposite for 'canny').
Default Value : 0.5
Suggested values : `Alpha` $\in$ {0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 0.9, 1.1}
Typical range of values : $0.2 \leq \text{Alpha} \leq 50.0$
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : `Alpha` > 0.0
- ▷ **Low** (input_control) integer $\leadsto$ *integer*
Lower threshold for the hysteresis threshold operation.
Default Value : 20
Suggested values : `Low` $\in$ {5, 10, 15, 20, 25, 30, 40}
Typical range of values : $1 \leq \text{Low} \leq 255$
Minimum Increment : 1
Recommended Increment : 5
Restriction : `Low` > 0
- ▷ **High** (input_control) integer $\leadsto$ *integer*
Upper threshold for the hysteresis threshold operation.
Default Value : 40
Suggested values : `High` $\in$ {10, 15, 20, 25, 30, 40, 50, 60, 70}
Typical range of values : $1 \leq \text{High} \leq 255$
Minimum Increment : 1
Recommended Increment : 5
Restriction : (`High` > 0) $\wedge$ (`High` $\geq$ `Low`)

Example

```
read_image(Image, 'fabrik')
edges_sub_pix(Image, Edges, 'lanser2', 0.5, 20, 40).
```

Complexity

Let *A* be the number of pixels in the domain of `Image`. Then the runtime complexity is $O(A * \text{Sigma})$ for the Canny filter and $O(A)$ for the recursive Lanser, Deriche, and Shen filters.

Let $S = \text{Width} * \text{Height}$ be the number of pixels of `Image`. Then `edges_sub_pix` requires at least $60 * S$ bytes of temporary memory during execution.

Result

`edges_sub_pix` returns 2 (`H_MSG_TRUE`) if all parameters are correct and no error occurs during execution. If the input is empty the behaviour can be set via `set_system('no-object-result', <Result>)`. If necessary, an exception handling is raised.

Parallelization Information

`edges_sub_pix` is *reentrant* and automatically *parallelized* (on *tuple level*).

Alternatives

`sobel_dir`, `frei_dir`, `kirsch_dir`, `prewitt_dir`, `robinson_dir`, `edges_image`

See also

`info_edges`, `hysteresis_threshold`, `bandpass_image`, `lines_gauss`, `lines_facet`

References

S.Lanser, W.Eckstein: "Eine Modifikation des Deriche-Verfahrens zur Kantendetektion"; 13. DAGM-Symposium, München; Informatik Fachberichte 290; Seite 151 - 158; Springer-Verlag; 1991.

S.Lanser: "Detektion von Stufenkanten mittels rekursiver Filter nach Deriche"; Diplomarbeit; Technische Universität München, Institut für Informatik, Lehrstuhl Prof. Radig; 1991.

J.Canny: "Finding Edges and Lines in Images"; Report, AI-TR-720; M.I.T. Artificial Intelligence Lab., Cambridge; 1983.

J.Canny: "A Computational Approach to Edge Detection"; IEEE Transactions on Pattern Analysis and Machine Intelligence; PAMI-8, vol. 6; S. 679-698; 1986.

R.Deriche: "Using Canny's Criteria to Derive a Recursively Implemented Optimal Edge Detector"; International Journal of Computer Vision; vol. 1, no. 2; S. 167-187; 1987.

R.Deriche: "Optimal Edge Detection Using Recursive Filtering"; Proc. of the First International Conference on Computer Vision, London; S. 501-505; 1987.

R.Deriche: "Fast Algorithms for Low-Level Vision"; IEEE Transactions on Pattern Analysis and Machine Intelligence; PAMI-12, no. 1; S. 78-87; 1990.

S.Castan, J.Zhao und J.Shen: "Optimal Filter for Edge Detection Methods and Results"; Proc. of the First European Conference on Computer Vision, Antibes; Lecture Notes on computer Science; no. 427; S. 12-17; Springer-Verlag; 1990.

Module

Sub-pixel operators

frei_amp (Image : ImageEdgeAmp : :)

Detect edges (amplitude) using the Frei-Chen operator.

`frei_amp` calculates an approximation of the first derivative of the image data and is used as an edge detector. The filter is based on the following filter masks:

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}$$

The result image contains the maximum response of the masks A and B .

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Input image.
- ▷ **ImageEdgeAmp** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Edge amplitude (gradient magnitude) image.

Example

```
read_image(Image, 'fabrik')
frei_amp(Image, Frei_amp)
threshold(Frei_amp, Edges, 128, 255).
```

Result

frei_amp always returns 2 (H_MSG_TRUE). If the input is empty the behaviour can be set via **set_system('no_object_result', <Result>)**. If necessary, an exception handling is raised.

Parallelization Information

frei_amp is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Predecessors

gauss_image, **sigma_image**, **median_image**, **smooth_image**

Alternatives

sobel_amp, **kirsch_amp**, **prewitt_amp**, **robinson_amp**, **roberts**

See also

bandpass_image, **laplace_of_gauss**

Module

Image filters

frei_dir (Image : ImageEdgeAmp, ImageEdgeDir : :)

Detect edges (amplitude and direction) using the Frei-Chen operator.

frei_dir calculates an approximation of the first derivative of the image data and is used as an edge detector. The filter is based on the following filter masks:

$$A = \begin{pmatrix} 1 & \sqrt{2} & 1 \\ 0 & 0 & 0 \\ -1 & -\sqrt{2} & -1 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 0 & -1 \\ \sqrt{2} & 0 & -\sqrt{2} \\ 1 & 0 & -1 \end{pmatrix}$$

The result image contains the maximum response of the masks *A* and *B*. The edge directions are returned in **ImageEdgeDir**, and are stored in 2-degree steps, i.e., an edge direction of *x* degrees with respect to the horizontal axis is stored as *x/2* in the edge direction image. Furthermore, the direction of the change of intensity is taken into account. Let $[E_x, E_y]$ denote the image gradient. Then the following edge directions are returned as *r/2*:

intensity increase	E_x/E_y
edge direction <i>r</i>	
from bottom to top	0/+ 0
from lower right to upper left	+/-]0, 90[
from right to left	+/0 90
from upper right to lower left	+/+]90, 180[
from top to bottom	0/+ 180
from upper left to lower right	-/+]180, 270[
from left to right	+/0 270
from lower left to upper right	-/-]270, 360[

Points with edge amplitude 0 are assigned the edge direction 255 (undefined direction).

Parameter
▷ Image (input_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / uint2 Input image. ▷ ImageEdgeAmp (output_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / uint2 Edge amplitude (gradient magnitude) image. ▷ ImageEdgeDir (output_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : direction Edge direction image.
Example
<pre>read_image(Image, 'fabrik') frei_dir(Image, Frei_dirA, Frei_dirD) threshold(Frei_dirA, Res, 128, 255).</pre>
Result
frei_dir always returns 2 (H_MSG_TRUE). If the input is empty the behaviour can be set via set_system('no_object_result', <Result>). If necessary, an exception handling is raised.
Parallelization Information
frei_dir is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>, <i>channel level</i>, <i>domain level</i>).
Possible Predecessors
gauss_image, sigma_image, median_image, smooth_image
Possible Successors
hysteresis_threshold, threshold, gray_skeleton, nonmax_suppression_dir, close_edges, close_edges_length
Alternatives
edges_image, sobel_dir, robinson_dir, prewitt_dir, kirsch_dir
See also
bandpass_image, laplace_of_gauss
Module
Image filters

highpass_image (Image : Highpass : Width, Height :)

Extract high frequency components from an image.

highpass_image extracts high frequency components in an image by applying a linear filter with the following matrix (in case of a 7×5 matrix):

1	1	1	1	1	1	1
1	1	1	1	1	1	1
1	1	1	-35	1	1	1
1	1	1	1	1	1	1
1	1	1	1	1	1	1

This corresponds to applying a mean operator (**mean_image**), and then subtracting the original gray value. A value of 128 is added to the result, i.e., zero crossings occur for 128.

This filter emphasizes high frequency components (edges and corners). The cutoff frequency is determined by the size (**Height** $\times$ **Width**) of the filter matrix: the larger the matrix, the smaller the cutoff frequency is.

At the image borders the pixels' gray values are mirrored. In case of over- or underflow the gray values are clipped (255 and 0, resp.).

Attention
If even values are passed for **Height** or **Width**, the operator uses the next larger odd value instead. Thus, the center of the filter mask is always uniquely determined.

Parameter

- ▷ **Image** (input_object)(multichannel-)image(-array) $\leadsto$ *Hobject* : byte
Input image.
- ▷ **Highpass** (output_object)(multichannel-)image(-array) $\leadsto$ *Hobject* : byte
High-pass-filtered result image.
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer*
Width of the filter mask.
Default Value : 9
Suggested values : $\text{Width} \in \{3, 5, 7, 9, 11, 13, 17, 21, 29, 41, 51, 73, 101\}$
Typical range of values : $3 \leq \text{Width} \leq 501$
Minimum Increment : 2
Recommended Increment : 2
Restriction : $(\text{Width} \geq 3) \wedge \text{odd}(\text{Width})$
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer*
Height of the filter mask.
Default Value : 9
Suggested values : $\text{Height} \in \{3, 5, 7, 9, 11, 13, 17, 21, 29, 41, 51, 73, 101\}$
Typical range of values : $3 \leq \text{Height} \leq 501$
Minimum Increment : 2
Recommended Increment : 2
Restriction : $(\text{Height} \geq 3) \wedge \text{odd}(\text{Height})$

Example

```
highpass_image(Image, &Highpass, 7, 5);
threshold(Highpass, &Region, 60.0, 255.0);
skeleton(Region, &Skeleton);
```

Result

[highpass_image](#) returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via [set_system\('no_object_result', <Result>\)](#). If necessary, an exception handling is raised.

Parallelization Information

[highpass_image](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Successors

[threshold](#), [skeleton](#)

Alternatives

[mean_image](#), [sub_image](#), [convol_image](#), [bandpass_image](#)

See also

[dyn_threshold](#)

Module

Image filters

info_edges (: : Filter, Mode, Alpha : Size, Coeffs)

Estimate the width of a filter in [edges_image](#).

[info_edges](#) returns an estimate of the width of any of the filters used in [edges_image](#). To do so, the corresponding continuous impulse responses of the filters are sampled until the first filter coefficient is smaller than five percent of the largest coefficient. **Alpha** is the filter parameter (see [edges_image](#)). Seven edge operators are supported (parameter **Filter**):

'deriche1', 'lanser1', 'deriche2', 'lanser2', 'shen', 'mshen' und 'canny'.

The parameter **Mode** ('edge'/'smooth') is used to determine whether the corresponding edge or smoothing operator is to be sampled. The Canny operator (which uses the Gaussian for smoothing) is implemented using conventional

filter masks, while all other filters are implemented recursively. Therefore, for the Canny filter the coefficients of the one-dimensional impulse responses $f(n)$ with $n \geq 0$ are returned in **Coefffs** in addition to the filter width.

Parameter

- ▷ **Filter** (input_control) string $\leadsto$ string
Name of the edge operator.
Default Value : 'lanser2'
List of values : Filter $\in \{\text{'deriche1', 'lanser1', 'deriche2', 'lanser2', 'shen', 'mshen', 'canny'}\}$
- ▷ **Mode** (input_control) string $\leadsto$ string
1D edge filter ('edge') or 1D smoothing filter ('smooth').
Default Value : 'edge'
List of values : Mode $\in \{\text{'edge', 'smooth'}\}$
- ▷ **Alpha** (input_control) real $\leadsto$ real
Filter parameter: small values result in strong smoothing, and thus less detail (opposite for 'canny').
Default Value : 0.5
Typical range of values : $0.2 \leq \text{Alpha} \leq 50.0$
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : Alpha > 0.0
- ▷ **Size** (output_control) integer $\leadsto$ integer
Filter width in pixels.
- ▷ **Coefffs** (output_control) integer-array $\leadsto$ integer
For Canny filters: Coefficients of the "positive" half of the 1D impulse response.

Example

```
read_image(Image, 'fabrik')
info_edges('lanser2', 'edge', 0.5, Size, Coefffs)
edges_image(Image, Amp, Dir, 'lanser2', 0.5, 'none', -1, -1)
hysteresis_threshold(Amp, Margin, 20, 30, 30).
```

Result

info_edges returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via **set_system('no_object_result', <Result>)**. If necessary, an exception handling is raised.

Parallelization Information

info_edges is *reentrant* and processed *without* parallelization.

Possible Successors

edges_image, **threshold**, **skeleton**

See also

edges_image

Module

Image filters

kirsch_amp (Image : ImageEdgeAmp : :)

Detect edges (amplitude) using the Kirsch operator.

kirsch_amp calculates an approximation of the first derivative of the image data and is used as an edge detector. The filter is based on the following filter masks:

$$\begin{array}{ccc} -3 & -3 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & 5 \\ \\ -3 & 5 & 5 \\ -3 & 0 & 5 \\ -3 & -3 & -3 \end{array}$$


```

      5   5   5
     -3   0  -3
     -3  -3  -3

      5   5  -3
      5   0  -3
     -3  -3  -3

      5  -3  -3
      5   0  -3
      5  -3  -3

     -3  -3  -3
      5   0  -3
      5   5  -3

     -3  -3  -3
     -3   0  -3
      5   5   5

     -3  -3  -3
     -3   0   5
     -3   5   5

```

The result image contains the maximum response of all masks.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Input image.
- ▷ **ImageEdgeAmp** (output_object) image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Edge amplitude (gradient magnitude) image.

Example

```

read_image(Image, 'fabrik')
kirsch_amp(Image, Kirsch_amp)
threshold(Kirsch_amp, Edges, 128, 255).

```

Result

kirsch_amp always returns 2 (H_MSG_TRUE). If the input is empty the behaviour can be set via **set_system('no_object_result', <Result>)**. If necessary, an exception handling is raised.

Parallelization Information

kirsch_amp is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Predecessors

gauss_image, **sigma_image**, **median_image**, **smooth_image**

Alternatives

sobel_amp, **frei_amp**, **prewitt_amp**, **robinson_amp**, **roberts**

See also

bandpass_image, **laplace_of_gauss**

Module

Image filters

kirsch_dir (Image : ImageEdgeAmp, ImageEdgeDir : :)

Detect edges (amplitude and direction) using the Kirsch operator.

kirsch_dir calculates an approximation of the first derivative of the image data and is used as an edge detector. The filter is based on the following filter masks:

```

-3  -3  5
-3   0  5
-3  -3  5

-3   5  5
-3   0  5
-3  -3 -3

 5   5  5
-3   0 -3
-3  -3 -3

 5   5 -3
 5   0 -3
-3  -3 -3

 5  -3 -3
 5   0 -3
 5  -3 -3

-3  -3 -3
 5   0 -3
 5   5 -3

-3  -3 -3
-3   0 -3
 5   5  5

-3  -3 -3
-3   0  5
-3   5  5

```

The result image contains the maximum response of all masks. The edge directions are returned in [ImageEdgeDir](#), and are stored as $x/2$. They correspond to the direction of the mask yielding the maximum response.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Input image.
- ▷ **ImageEdgeAmp** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Edge amplitude (gradient magnitude) image.
- ▷ **ImageEdgeDir** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : direction
Edge direction image.

Example

```

read_image(Image, 'fabrik')
kirsch_dir(Image, Kirsch_dirA, Kirsch_dirD)
threshold(Kirsch_dirA, Res, 128, 255).

```

Result

[kirsch_dir](#) always returns 2 (H_MSG_TRUE). If the input is empty the behaviour can be set via [set.system\('no_object_result', <Result>\)](#). If necessary, an exception handling is raised.

Parallelization Information

[kirsch_dir](#) is *reentrant* and automatically *parallelized* (on tuple level, channel level, domain level).

Possible Predecessors

[gauss_image](#), [sigma_image](#), [median_image](#), [smooth_image](#)

Possible Successors

[hysteresis_threshold](#), [threshold](#), [gray_skeleton](#), [nonmax_suppression_dir](#),
[close_edges](#), [close_edges_length](#)

	Alternatives
edges_image , sobel_dir , robinson_dir , prewitt_dir , frei_dir	
	See also
bandpass_image , laplace_of_gauss	
	Module
Image filters	

laplace (*Image* : *ImageLaplace* : *FilterType*, *Size*,
NeighbourhoodType :)

Calculate the Laplace operator by using finite differences.

[laplace](#) filters the input images [Image](#) using a Laplace operator. Depending on the parameter [NeighbourhoodType](#) the following simple approximations of the Laplace operator are used:

'n_4'

$$\begin{array}{ccc} & 1 & \\ 1 & -4 & 1 \\ & 1 & \end{array}$$

'n_8'

$$\begin{array}{ccc} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{array}$$

'n_8_isotrop'

$$\begin{array}{ccc} 10 & 22 & 10 \\ 22 & -128 & 22 \\ 10 & 22 & 10 \end{array}$$

The filter **'n_8'** corresponds to the filter mask used in [highpass_image\(0:R:3,3:\)](#). For a Laplace operator with size 3×3 , the corresponding filter is applied directly, while for larger filter sizes (**Size** = 5,7,9 and 11) the input image is first smoothed using a Gaussian filter of the selected size. Therefore,

```
laplace(0:R:int4,S,N:)
```

for **Size** > is equivalent to

```
gauss_image(0:G:S:) ▷
laplace(G:R:int4,3,N:).
```

[laplace](#) either returns the absolute value of the Laplace filtered image (**FilterType** **'abs'**) in a byte/uint2-image or the signed result (**FilterType** **'int4'**) in an int4-image.

	Parameter
▷ Image (input_object)	(multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / uint2 Input image.
▷ ImageLaplace (output_object)	(multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / uint2 / int4 Laplace-filtered result images.
▷ FilterType (input_control)	string $\leadsto$ <i>string</i> Calculate the absolute value of the filter to a byte/uint2-image or the signed result to an int4-image. Default Value : 'int4' List of values : FilterType $\in$ { 'abs' , 'int4' }
▷ Size (input_control)	integer $\leadsto$ <i>integer</i> Size of filter mask. Default Value : 3 List of values : Size $\in$ {3, 5, 7, 9, 11}

- ▷ **NeighbourhoodType** (input_control) string $\leadsto$ string
 Neighborhood used in the Laplace operator
Default Value : 'n_8_isotrop'
List of values : NeighbourhoodType $\in$ {'n_4', 'n_8', 'n_8_isotrop'}

Example

```
read_image(&Image, "mreut");
laplace(Image, &Laplace, "int4", 3, "n_8_isotrop");
zero_crossing(Laplace, &ZeroCrossings);
```

Result

laplace returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via **set_system('no-object-result', <Result>)**. If necessary, an exception handling is raised.

Parallelization Information

laplace is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Successors

zero_crossing, **dual_threshold**

Alternatives

diff_of_gauss, **laplace_of_gauss**

See also

highpass_image, **edges_image**

Module

Image filters

laplace_of_gauss (Image : ImageLaplace : Sigma :)

LoG-Operator (Laplace of Gaussian).

laplace_of_gauss calculates the Laplace-of-Gaussian operator, i.e., the Laplace operator on a Gaussian smoothed image, for arbitrary smoothing parameters **Sigma**. The Laplace operator is given by:

$$\Delta g(x, y) = \frac{\partial^2 g(x, y)}{\partial x^2} + \frac{\partial^2 g(x, y)}{\partial y^2}$$

The derivatives in **laplace_of_gauss** are calculated by appropriate derivatives of the Gaussian, resulting in the following formula for the convolution mask:

$$\Delta G_{\sigma}(x, y) = \frac{1}{2\pi\sigma^4} \left(\frac{x^2 + y^2}{2\sigma^2} - 1 \right) \left[\exp \left(-\frac{x^2 + y^2}{2\sigma^2} \right) \right]$$

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int1 / int2 / uint2 / int4 / real
 Input image.
- ▷ **ImageLaplace** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : int2
 Laplace filtered image.
- ▷ **Sigma** (input_control) number $\leadsto$ *real* / integer
 Smoothing parameter of the Gaussian.
Default Value : 2.0
Suggested values : Sigma $\in$ {0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 4.0, 5.0, 7.0}
Typical range of values : $0.7 \leq \text{Sigma} \leq 5.0$
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : (Sigma > 0.7) $\wedge$ (Sigma $\leq$ 25.0)

Example

```
read_image(&Image, "mreut");
laplace_of_gauss(Image, &Laplace, 2.0);
zero_crossing(Laplace, &ZeroCrossings);
```

Parallelization Information

[laplace_of_gauss](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Successors

[zero_crossing](#), [dual_threshold](#)

Alternatives

[laplace](#), [diff_of_gauss](#), [derivate_gauss](#)

See also

[derivate_gauss](#)

Module

Image filters

prewitt_amp (Image : ImageEdgeAmp : :)

Detect edges (amplitude) using the Prewitt operator.

[prewitt_amp](#) calculates an approximation of the first derivative of the image data and is used as an edge detector. The filter is based on the following filter masks:

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}$$

The result image contains the maximum response of the masks *A* and *B*.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Input image.
- ▷ **ImageEdgeAmp** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Edge amplitude (gradient magnitude) image.

Example

```
read_image(Image, 'fabrik')
prewitt_amp(Image, Prewitt)
threshold(Prewitt, Edges, 128, 255).
```

Result

[prewitt_amp](#) always returns 2 (H_MSG_TRUE). If the input is empty the behaviour can be set via [set_system\('no-object-result', <Result>\)](#). If necessary, an exception handling is raised.

Parallelization Information

[prewitt_amp](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Predecessors

[gauss_image](#), [sigma_image](#), [median_image](#), [smooth_image](#)

Possible Successors

[threshold](#), [gray_skeleton](#), [nonmax-suppression_amp](#), [close_edges](#), [close_edges_length](#)

	Alternatives
sobel_amp , kirsch_amp , frei_amp , robinson_amp , roberts	
	See also
bandpass_image , laplace_of_gauss	
	Module
Image filters	

prewitt_dir (Image : ImageEdgeAmp, ImageEdgeDir : :)

Detect edges (amplitude and direction) using the Prewitt operator.

[prewitt_dir](#) calculates an approximation of the first derivative of the image data and is used as an edge detector. The filter is based on the following filter masks:

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 0 & -1 \\ 1 & 0 & -1 \\ 1 & 0 & -1 \end{pmatrix}$$

The result image contains the maximum response of the masks A and B . The edge directions are returned in [ImageEdgeDir](#), and are stored in 2-degree steps, i.e., an edge direction of x degrees with respect to the horizontal axis is stored as $x/2$ in the edge direction image. Furthermore, the direction of the change of intensity is taken into account. Let $[E_x, E_y]$ denote the image gradient. Then the following edge directions are returned as $r/2$:

intensity increase	E_x/E_y
edge direction r	
from bottom to top	0/+ 0
from lower right to upper left	+/-]0, 90[
from right to left	+ / 0 90
from upper right to lower left	+ / +]90, 180[
from top to bottom	0 / + 180
from upper left to lower right	- / +]180, 270[
from left to right	+ / 0 270
from lower left to upper right	- / -]270, 360[

Points with edge amplitude 0 are assigned the edge direction 255 (undefined direction).

	Parameter
▷ Image (input_object)	(multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / uint2 Input image.
▷ ImageEdgeAmp (output_object)	(multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / uint2 Edge amplitude (gradient magnitude) image.
▷ ImageEdgeDir (output_object)	(multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : direction Edge direction image.

	Example

```
read_image(Image, 'fabrik')
prewitt_dir(Image, PrewittA, PrewittD)
threshold(PrewittA, Edges, 128, 255).
```

	Result

[prewitt_dir](#) always returns 2 (H_MSG_TRUE). If the input is empty the behaviour can be set via [set_system\('no_object_result', <Result>\)](#). If necessary, an exception handling is raised.

<i>Parallelization Information</i>
<code>prewitt_dir</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i> , <i>channel level</i> , <i>domain level</i>).
<i>Possible Predecessors</i>
<code>gauss_image</code> , <code>sigma_image</code> , <code>median_image</code> , <code>smooth_image</code>
<i>Possible Successors</i>
<code>hysteresis_threshold</code> , <code>threshold</code> , <code>gray_skeleton</code> , <code>nonmax_suppression_dir</code> , <code>close_edges</code> , <code>close_edges_length</code>
<i>Alternatives</i>
<code>edges_image</code> , <code>sobel_dir</code> , <code>robinson_dir</code> , <code>frei_dir</code> , <code>kirsch_dir</code>
<i>See also</i>
<code>bandpass_image</code> , <code>laplace_of_gauss</code>
<i>Module</i>
Image filters

roberts (Image : ImageRoberts : FilterType :)

Detect edges using the Roberts filter.

`roberts` calculates the first derivative of an image and is used as an edge operator. If the following mask describes a part of the image,

$$\begin{array}{cc} A & B \\ C & D \end{array}$$

the different filter types are defined as follows:

<code>'roberts_max'</code>	$\max(A - D , B - C)$
<code>'gradient_max'</code>	$\max(A + B - (C + D) , A + C - (B + D))$
<code>'gradient_sum'</code>	$ A + B - (C + D) + A + C - (B + D) $

If an overflow occurs the result is clipped. The result of the operator is stored at the pixel with the coordinates of “D”.

<i>Parameter</i>
▷ Image (input_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / uint2 Input image. ▷ ImageRoberts (output_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / uint2 Roberts-filtered result images. ▷ FilterType (input_control) string $\leadsto$ <i>string</i> Filter type. Default Value : <code>'gradient_sum'</code> List of values : <code>FilterType</code> $\in$ {<code>'roberts_max'</code>, <code>'gradient_max'</code>, <code>'gradient_sum'</code>}
<i>Example</i>

```
read_image(Image, 'fabrik')
roberts(Image, Roberts, 'roberts_max')
threshold(Roberts, Margin, 128, 255).
```

<i>Result</i>
<code>roberts</code> returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via <code>set_system('no_object_result', <Result>)</code> . If necessary, an exception handling is raised.
<i>Parallelization Information</i>
<code>roberts</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i> , <i>channel level</i> , <i>domain level</i>).
<i>Possible Predecessors</i>
<code>gauss_image</code>

	Possible Successors
threshold , skeleton	
	Alternatives
edges_image , sobel_amp , frei_amp , kirsch_amp , prewitt_amp	
	See also
laplace , highpass_image , bandpass_image	
	Module
Image filters	

robinson_amp (Image : ImageEdgeAmp : :)

Detect edges (amplitude) using the Robinson operator.

[robinson_amp](#) calculates an approximation of the first derivative of the image data and is used as an edge detector. In [robinson_amp](#) the following four of the originally proposed eight 3×3 filter masks are convolved with the image. The other four masks are obtained by a multiplication by -1. All masks contain only the values 0,1,-1,2,-2.

$$\begin{array}{ccc}
 -1 & 0 & 1 \\
 -2 & 0 & 2 \\
 -1 & 0 & 1 \\
 \\
 2 & 1 & 0 \\
 1 & 0 & -1 \\
 0 & -1 & -2 \\
 \\
 0 & 1 & 2 \\
 -1 & 0 & 1 \\
 -2 & -1 & 0 \\
 \\
 1 & 2 & 1 \\
 0 & 0 & 0 \\
 -1 & -2 & -1
 \end{array}$$

The result image contains the maximum response of all masks.

	Parameter
▷ Image (input_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / uint2	
Input image.	
▷ ImageEdgeAmp (output_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / uint2	
Edge amplitude (gradient magnitude) image.	
	Example

```

read_image(Image, 'fabrik')
robinson_amp(Image, Robinson_amp)
threshold(Robinson_amp, Edges, 128, 255).

```

	Result
robinson_amp always returns 2 (H_MSG_TRUE). If the input is empty the behaviour can be set via set_system('no_object_result', <Result>) . If necessary, an exception handling is raised.	

	Parallelization Information
robinson_amp is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level, channel level, domain level).	
	Possible Predecessors
gauss_image , sigma_image , median_image , smooth_image	
	Alternatives
sobel_amp , frei_amp , prewitt_amp , robinson_amp , roberts	

See also

[bandpass_image](#), [laplace_of_gauss](#)

Module

Image filters

robinson_dir (Image : ImageEdgeAmp, ImageEdgeDir : :)

Detect edges (amplitude and direction) using the Robinson operator.

[robinson_dir](#) calculates an approximation of the first derivative of the image data and is used as an edge detector. In [robinson_amp](#) the following four of the originally proposed eight 3×3 filter masks are convolved with the image. The other four masks are obtained by a multiplication by -1. All masks contain only the values 0,1,-1,2,-2.

```

-1  0  1
-2  0  2
-1  0  1

 2  1  0
 1  0 -1
 0 -1 -2

 0  1  2
-1  0  1
-2 -1  0

 1  2  1
 0  0  0
-1 -2 -1

```

The result image contains the maximum response of all masks. The edge directions are returned in [ImageEdgeDir](#), and are stored as $x/2$. They correspond to the direction of the mask yielding the maximum response.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Input image.
- ▷ **ImageEdgeAmp** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Edge amplitude (gradient magnitude) image.
- ▷ **ImageEdgeDir** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : direction
Edge direction image.

Example

```

read_image(Image, 'fabrik')
robinson_dir(Image, Robinson_dirA, Robinson_dirD)
threshold(Robinson_dirA, Res, 128, 255).

```

Result

[robinson_dir](#) always returns 2 (H_MSG.TRUE). If the input is empty the behaviour can be set via [set_system\('no.object.result', <Result>\)](#). If necessary, an exception handling is raised.

Parallelization Information

[robinson_dir](#) is *reentrant* and automatically *parallelized* (on tuple level, channel level, domain level).

Possible Predecessors

[gauss_image](#), [sigma_image](#), [median_image](#), [smooth_image](#)

Possible Successors

[hysteresis_threshold](#), [threshold](#), [gray_skeleton](#), [nonmax_suppression_dir](#),
[close_edges](#), [close_edges_length](#)

	Alternatives
edges_image , sobel_dir , kirsch_dir , prewitt_dir , frei_dir	
	See also
bandpass_image , laplace_of_gauss	
	Module
Image filters	

sobel_amp (Image : EdgeAmplitude : FilterType, Size :)

Detect edges (amplitude) using the Sobel operator.

sobel_amp calculates first derivative of an image and is used as an edge detector. The filter is based on the following filter masks:

$$A = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$

These masks are used differently, according to the selected filter type. (In the following, a and b denote the results of convolving an image with A and B for one particular pixel.)

'sum_sqrt'	$\sqrt{a^2 + b^2}$
'sum_abs'	$(a + b)/2$
'thin_sum_abs'	$(thin(a) + thin(b))/2$
'thin_max_abs'	$max(thin(a), thin(b))$
'x'	b
'y'	a

Here, $thin(x)$ is equal to x for a vertical maximum (mask A) and a horizontal maximum (mask B), respectively, and 0 otherwise. Thus, for 'thin_sum_abs' and 'thin_max_abs' the gradient image is thinned. For the filter types 'x' and 'y' if the input image is of type byte the output image is of type int1, of type int2 otherwise. For a Sobel operator with size 3×3 , the corresponding filters A and B are applied directly, while for larger filter sizes (**Size** = 5,7,9 and 11) the input image is first smoothed using a Gaussian filter of size **Size**-2. Therefore,

sobel_amp(I:E, Dir:FilterTyp, S:)

for **Size** > 3 is equivalent to

```
gauss_image(I:G:S-2) ▷
sobel_amp(G:E:FilterType, 3:).
```

	Parameter
▷ Image (input_object)	(multichannel-)image(-array) $\leadsto$ Hobject : byte / int2 / uint2 Input image.
▷ EdgeAmplitude (output_object)	(multichannel-)image(-array) $\leadsto$ Hobject : int1 / int2 / uint2 Edge amplitude (gradient magnitude) image.
▷ FilterType (input_control)	string $\leadsto$ string Filter type. Default Value : 'sum_abs' List of values : FilterType $\in$ {'sum_abs', 'thin_sum_abs', 'thin_max_abs', 'sum_sqrt', 'x', 'y'}
▷ Size (input_control)	integer $\leadsto$ integer Size of filter mask. Default Value : 3 List of values : Size $\in$ {3, 5, 7, 9, 11, 13}

Example

```
read_image(Image, 'fabrik')
sobel_amp(Image, Amp, 'sum_abs', 3)
threshold(Amp, Edg, 128, 255).
```

Result

`sobel_amp` returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via `set_system('no-object-result', <Result>)`. If necessary, an exception handling is raised.

Parallelization Information

`sobel_amp` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Predecessors

`gauss_image`, `mean_image`, `anisotrope_diff`, `sigma_image`

Possible Successors

`threshold`, `nonmax-suppression_amp`, `gray_skeleton`

Alternatives

`frei_amp`, `roberts`, `kirsch_amp`, `prewitt_amp`, `robinson_amp`

See also

`laplace`, `highpass_image`, `bandpass_image`

Module

Image filters

sobel_dir (Image : EdgeAmplitude, EdgeDirection : FilterType, Size :)

Detect edges (amplitude and direction) using the Sobel operator.

`sobel_dir` calculates first derivative of an image and is used as an edge detector. The filter is based on the following filter masks:

$$A = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix}$$

$$B = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$

These masks are used differently, according to the selected filter type. (In the following, a and b denote the results of convolving an image with A and B for one particular pixel.)

$$\begin{array}{ll} \text{'sum\_sqrt'} & \sqrt{a^2 + b^2} \\ \text{'sum\_abs'} & (|a| + |b|)/2 \end{array}$$

For a Sobel operator with size 3×3 , the corresponding filters A and B are applied directly, while for larger filter sizes (`Size` = 5,7,9 and 11) the input image is first smoothed using a Gaussian filter of size `Size-2`. Therefore,

```
sobel_dir(I:Amp, Dir:FilterTyp, S:)
```

for `Size` > 3 is equivalent to

```
gauss_image(I:G:S-2) >
sobel_dir(G:Amp, Dir:FilterType, 3:).
```

The edge directions are returned in [EdgeDirection](#), and are stored in 2-degree steps, i.e., an edge direction of x degrees with respect to the horizontal axis is stored as $x/2$ in the edge direction image. Furthermore, the direction of the change of intensity is taken into account. Let $[E_x, E_y]$ denote the image gradient. Then the following edge directions are returned as $r/2$:

intensity increase	E_x/E_y	
edge direction r		
from bottom to top	0/+	0
from lower right to upper left	+/-	]0, 90[
from right to left	+/0	90
from upper right to lower left	+/+	]90, 180[
from top to bottom	0/+	180
from upper left to lower right	-/+	]180, 270[
from left to right	+/0	270
from lower left to upper right	-/-	]270, 360[

Points with edge amplitude 0 are assigned the edge direction 255 (undefined direction).

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Input image.
- ▷ **EdgeAmplitude** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Edge amplitude (gradient magnitude) image.
- ▷ **EdgeDirection** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : direction
Edge direction image.
- ▷ **FilterType** (input_control) string $\leadsto$ *string*
Filter type.
Default Value : 'sum_abs'
List of values : FilterType $\in$ {'sum_abs', 'sum_sqrt'}
- ▷ **Size** (input_control) integer $\leadsto$ *integer*
Size of filter mask.
Default Value : 3
List of values : Size $\in$ {3, 5, 7, 9, 11, 13}

Example

```
read_image(Image, 'fabrik')
sobel_dir(Image, Amp, Dir, 'sum_abs', 3)
threshold(Amp, Edg, 128, 255).
```

Result

[sobel_dir](#) returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via [set_system\('no-object-result', <Result>\)](#). If necessary, an exception handling is raised.

Parallelization Information

[sobel_dir](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*).

Possible Predecessors

[gauss_image](#), [mean_image](#), [anisotrope_diff](#), [sigma_image](#)

Possible Successors

[nonmax-suppression_dir](#), [hysteresis_threshold](#), [threshold](#)

Alternatives

[edges_image](#), [frei_dir](#), [kirsch_dir](#), [prewitt_dir](#), [robinson_dir](#)

See also

[roberts](#), [laplace](#), [highpass_image](#), [bandpass_image](#)

Module

Image filters

5.6 Enhancement

emphasize (Image : ImageEmphasize : MaskWidth, MaskHeight, Factor :)

Enhance contrast of the image.

The operator **emphasize** emphasizes high frequency areas of the image (edges and corners). The resulting images appears sharper.

First the procedure carries out a filtering with the low pass (**mean_image**). The resulting gray values (*res*) are calculated from the obtained gray values (*mean*) and the original gray values (*orig*) as follows:

$$res := round((orig - mean) * Factor) + orig$$

Factor serves as measurement of the increase in contrast. The division frequency is determined via the size of the filter matrix: The larger the matrix, the lower the division frequency.

As an edge treatment the gray values are mirrored at the edges of the image. Overflow and/or underflow of gray values is clipped.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Image to be enhanced.
- ▷ **ImageEmphasize** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
contrast enhanced image.
- ▷ **MaskWidth** (input_control) extent.x $\leadsto$ *integer*
Width of low pass mask.
Default Value : 7
Suggested values : MaskWidth $\in \{3, 5, 7, 9, 11, 15, 21, 25, 31, 39\}$
Typical range of values : $3 \leq \text{MaskWidth} \leq 201$
Minimum Increment : 2
Recommended Increment : 2
- ▷ **MaskHeight** (input_control) extent.y $\leadsto$ *integer*
Height of the low pass mask.
Default Value : 7
Suggested values : MaskHeight $\in \{3, 5, 7, 9, 11, 15, 21, 25, 31, 39\}$
Typical range of values : $3 \leq \text{MaskHeight} \leq 201$
Minimum Increment : 2
Recommended Increment : 2
- ▷ **Factor** (input_control) real $\leadsto$ *real*
Intensity of contrast emphasis.
Default Value : 1.0
Suggested values : Factor $\in \{0.3, 0.5, 0.7, 1.0, 1.4, 1.8, 2.0\}$
Typical range of values : $0.0 \leq \text{Factor} \leq 20.0$ (sqrt)
Minimum Increment : 0.01
Recommended Increment : 0.2
Restriction : $(0 < \text{Factor}) \wedge (\text{Factor} < 20)$

Example

```
read_image(Image, 'meer_rot')
disp_image(Image, WindowHandle)
draw_region(Region, WindowHandle)
reduce_domain(Image, Region, Mask)
emphasize(Mask, Sharp, 7, 7, 2.0)
disp_image(Sharp, WindowHandle).
```

Result

If the parameter values are correct the operator **emphasize** returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator **set_system** (`::'no_object_result', <Result>:`). If necessary an exception handling is raised.

Parallelization Information

emphasize is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Successors

disp_image

Alternatives

mean_image, **sub_image**, **laplace**, **add_image**

See also

mean_image, **highpass_image**

Module

Image filters

equ_histo_image (Image : ImageEquHisto : :)

Histogram linearisation of images

The operator **equ_histo_image** enhances the contrast. The starting point is the histogram of the input images. The following simple gray value transformation $f(g)$ is carried out for byte images:

$$f(g) = 255 \sum_{x=0 \dots g} h(x)$$

$h(x)$ describes the relative frequency of the occurrence of the gray value x . For uint2 images, the only difference is that the value 255 is replaced with a different maximum value. The maximum value is computed from the number of significant bits stored with the input image, provided that this value is set. If not, the value of the system parameter 'int2_bits' is used (see **set_system**), if this value is set (i.e., different from -1). If none of the two values is set, the number of significant bits is set to 16.

This transformation linearises the cumulative histogram. Maxima in the original histogram are "spreaded" and thus the contrast in image regions with these frequently occurring gray values is increased. Supposedly homogenous regions receive more easily visible structures. On the other hand, of course, the noise in the image increases correspondingly. Minima in the original histogram are dually "compressed". The transformed histogram contains gaps, but the remaining gray values used occur approximately at the same frequency ("histogram equalization").

Attention

The operator **equ_histo_image** primarily serves for optical processing of images for a human viewer. For example, the (local) contrast spreading can lead to a detection of fictitious edges.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / uint2
Image to be enhanced.
- ▷ **ImageEquHisto** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / uint2
Image with linearized gray values.

Parallelization Information

equ_histo_image is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*).

Possible Successors

disp_image

Alternatives

scale_image, **scale_image_max**, **illuminate**

See also

scale_image

References

R.C. Gonzales, P. Wintz: "Digital Image Processing"; Second edition; Addison Wesley; 1987.

Module

Image filters

```
illuminate ( Image : ImageIlluminate : MaskWidth, MaskHeight,
Factor : )
```

Illuminate image.

The operator **illuminate** enhances contrast. Very dark parts of the image are "illuminated" more strongly, very light ones are "darkened". If *orig* is the original gray value and *mean* is the corresponding gray value of the low pass filtered image detected via the operators **mean_image** and filter size **MaskHeight** x **MaskWidth**. For byte-images val equals 127, for int2-images and uint2-images val equals the median value. The resulting gray value is *new*:

$$new := round((val - mean) * Factor + orig)$$

The low pass should have rather large dimensions (30 x 30 to 200 x 200). Reasonable parameter combinations might be:

MaskHeight	MaskWidth	Factor
40	40	0.55
100	100	0.7
150	150	0.8

i.e. the larger the low pass mask is chosen, the larger **Factor** should be as well.

The following "spotlight effect" should be noted: If, for example, a dark object is in front of a light wall the object as well as the wall, which is already light in the immediate proximity of the object contours, are lightened by the operator **illuminate**. This corresponds roughly to the effect that is produced when the object is illuminated by a strong spotlight. The same applies to light objects in front of a darker background. In this case, however, the fictitious "spotlight" darkens objects.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Image to be enhanced.
- ▷ **ImageIlluminate** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
"Illuminated" image.
- ▷ **MaskWidth** (input_control) extent.x $\leadsto$ *integer*
Width of low pass mask.
Default Value : 101
Suggested values : MaskWidth $\in$ {31, 41, 51, 71, 101, 121, 151, 201}
Typical range of values : $3 \leq \text{MaskWidth} \leq 299$
Minimum Increment : 2
Recommended Increment : 10
- ▷ **MaskHeight** (input_control) extent.y $\leadsto$ *integer*
Height of low pass mask.
Default Value : 101
Suggested values : MaskHeight $\in$ {31, 41, 51, 71, 101, 121, 151, 201}
Typical range of values : $3 \leq \text{MaskHeight} \leq 299$
Minimum Increment : 2
Recommended Increment : 10
- ▷ **Factor** (input_control) real $\leadsto$ *real*
Scales the "correction gray value" added to the original gray values.
Default Value : 0.7
Suggested values : Factor $\in$ {0.3, 0.5, 0.7, 1.0, 1.5, 2.0, 3.0, 5.0}
Typical range of values : $0.0 \leq \text{Factor} \leq 5.0$
Minimum Increment : 0.01
Recommended Increment : 0.2
Restriction : $(0 < \text{Factor}) \wedge (\text{Factor} < 5)$

Example

```
read_image(Image, 'fabrik')
disp_image(Image, WindowHandle)
illuminate(Image, Better, 40, 40, 0.55)
disp_image(Better, WindowHandle).
```

Result

If the parameter values are correct the operator `illuminate` returns the value 2 (H_MSG.TRUE). The behavior in case of empty input (no input images available) is set via the operator `set_system (::'no_object_result', <Result>:)`. If necessary an exception handling is raised.

Parallelization Information

`illuminate` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Successors

`disp_image`

Alternatives

`scale_image_max`, `equ_histo_image`, `mean_image`, `sub_image`

See also

`emphasize`, `gray_histo`

Module

Image filters

<code>scale_image_max</code> (Image : ImageScaleMax : :)

Maximum gray value spreading in the value range 0 bis 255.

The operator `scale_image_max` calculates the minimum and maximum and scales the image to the maximum value range of a byte image. This way the dynamics (value range) is fully exploited. The number of different gray scales does not change, but in general the visual impression is enhanced. The gray values of images of the *real*, *int2*, *uint2* and *int4* type are scaled to the range 0 to 255 and returned as *byte* images.

Attention

The output always is an image of the type *byte*.

Parameter

- ▷ **Image** (input_object)(multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4 / real
Image to be scaled.
- ▷ **ImageScaleMax** (output_object) image(-array) $\leadsto$ *Hobject* : byte
contrast enhanced image.

Parallelization Information

`scale_image_max` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*).

Possible Successors

`disp_image`

Alternatives

`equ_histo_image`, `scale_image`, `illuminate`, `convert_image_type`

See also

`min_max_gray`, `gray_histo`

Module

Image filters

5.7 FFT

<code>convol_fft</code> (ImageFFT, ImageFilter : ImageConvolve : :)

Convolve an image with a byte-mask in the frequency domain.

`convol_fft` convolves two (Fourier-transformed) images in the frequency domain, i.e., the pixels of the complex image `ImageFFT` are multiplied by the corresponding pixels of the filter `ImageFilter`. This image is a byte-image in which 0 corresponds to complete suppression of a frequency, and 255 corresponds to no suppression. The result image is of type '*complex*'.

Attention

The filtering is always done on the entire image, i.e., the region of the image is ignored.

Parameter

- ▷ **ImageFFT** (input_object) image(-array) $\leadsto$ *Hobject*: complex
Complex input image.
- ▷ **ImageFilter** (input_object) image $\leadsto$ *Hobject*: byte
Filter in frequency domain.
- ▷ **ImageConvolve** (output_object) image(-array) $\leadsto$ *Hobject*: complex
Result of applying the filter.

Example

```

my_highpass(Hobject Image, Hobject *Result, int frequency, int size)
{
    Hobject  FFT, Highpass, FFTConvolve;
    fft_image(Image,&FFT);
    gen_highpass(&Highpass,frequency,size);
    convolve_fft(FFT,Highpass,&FFTConvolve);
    clear_obj(Highpass); clear_obj(FFT);
    fft_image_inv(FFTConvolve,Result);
    clear_obj(FFTConvolve);
}

```

Result

convolve_fft returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via **set_system(, 'no_object_result', <Result>:)**. If necessary, an exception handling is raised.

Parallelization Information

convolve_fft is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[fft_image](#), [fft_generic](#), [gen_highpass](#), [gen_lowpass](#), [gen_bandpass](#), [gen_bandfilter](#)

Possible Successors

[power_byte](#), [power_real](#), [power_ln](#), [fft_image_inv](#), [fft_generic](#)

Alternatives

[convolve_gabor](#)

See also

[gen_gabor](#), [gen_highpass](#), [gen_lowpass](#), [gen_bandpass](#), [convolve_gabor](#), [fft_image_inv](#)

Module

Image filters

convolve_gabor (ImageFFT, GaborFilter : ImageResultGabor,
ImageResultHilbert : :)

Convolve an image with a Gabor filter in the frequency domain.

convolve_gabor convolves a Fourier-transformed image with a Gabor filter [GaborFilter](#) (see [gen_gabor](#)) and its Hilbert transform in the frequency domain. The result image is of type 'complex'.

Attention

The filtering is always done on the entire image, i.e., the region of the image is ignored.

Parameter

- ▷ **ImageFFT** (input_object) image(-array) $\leadsto$ *Hobject*: complex
Input image.
- ▷ **GaborFilter** (input_object) multichannel-image $\leadsto$ *Hobject*: real
Gabor/Hilbert-Filter.

- ▷ **ImageResultGabor** (output_object) image(-array) $\leadsto$ *Hobject* : complex
Result of the Gabor filter.
- ▷ **ImageResultHilbert** (output_object) image(-array) $\leadsto$ *Hobject* : complex
Result of the Hilbert filter.

Example

```
fft_image(Image,&FFT);
gen_gabor(&Filter,1.4,0.4,1.0,1.5,512);
convol_gabor(FFT,Filter,&Gabor,&Hilbert);
fft_image_inv(Gabor,&GaborInv);
fft_image_inv(Hilbert,&HilbertInv);
energy_gabor(GaborInv,HilbertInv,&Energy);
```

Result

convol_gabor returns 2 (H_MSG.TRUE) if all images are of correct type. If the input is empty the behaviour can be set via **set_system**(::'no_object_result',<Result>:). If necessary, an exception handling is raised.

Parallelization Information

convol_gabor is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

fft_image, **fft_generic**, **gen_gabor**

Possible Successors

power_byte, **power_real**, **power_ln**, **fft_image_inv**, **fft_generic**

Alternatives

convol_fft

See also

convol_image

Module

Image filters

energy_gabor (ImageGabor, ImageHilbert : Energy : :)

Calculate the energy of a two-channel image.

energy_gabor calculates the local contrast (**Energy**) of the two input images. The energy of the resulting image is then defined as

$$\text{Energy} = \text{channel1}^2 + \text{channel2}^2 .$$

Often the calculation of the energy is preceded by the convolution of an image with a Gabor filter and the Hilbert transform of the Gabor filter (see **convol_gabor**). In this case, the first channel of the image passed to **energy_gabor** is the Gabor-filtered image, transformed back into the spatial domain (see **fft_image_inv**), and the second channel the result of the convolution with the Hilbert transform, also transformed back into the spatial domain. The local energy is a measure for the local contrast of structures (e.g., edges and lines) in the image.

Parameter

- ▷ **ImageGabor** (input_object) image(-array) $\leadsto$ *Hobject* : byte / real
1st channel of input image (usually: Gabor image).
- ▷ **ImageHilbert** (input_object) image(-array) $\leadsto$ *Hobject* : byte / real
2nd channel of input image (usually: Hilbert image).
- ▷ **Energy** (output_object) image(-array) $\leadsto$ *Hobject* : real
Image containing the local energy.

Example

```
fft_image(Image,&FFT);
gen_gabor(&Filter,1.4,0.4,1.0,1.5,512);
convol_gabor(FFT,Filter,&Gabor,&Hilbert);
fft_image_inv(Gabor,&GaborInv);
fft_image_inv(Hilbert,&HilbertInv);
energy_gabor(GaborInv,HilbertInv,&Energy);
```

Result

energy_gabor returns 2 (H_MSG.TRUE) if all parameters are correct. If the input is empty the behaviour can be set via **set_system(: : 'no_object_result', <Result> :)**. If necessary, an exception handling is raised.

Parallelization Information

energy_gabor is *reentrant* and automatically *parallelized* (on *tuple level*, *domain level*).

Possible Predecessors

gen_gabor, **convol_gabor**, **fft_image_inv**

Module

Image filters

fft_generic (Image : ImageFFT : Direction, Exponent, Norm, Mode :)

Compute the fast Fourier transform of an image.

fft_generic computes the fast Fourier transform of the input image **Image**. Because several definitions of the forward and reverse transforms exist in the literature, this operator allows the user to select the most convenient definition.

The general definition of a Fourier transform is as follows:

$$F(m, n) = \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} e^{s2\pi i(km/M + ln/N)} f(k, l)$$

Opinions vary on whether the sign s in the exponent should be set to 1 or -1 for the forward transform, i.e. the for going to the frequency domain. There is also disagreement on the magnitude of the normalizing factor c . This is sometimes set to 1 for the forward transform, sometimes to MN , and sometimes (in case of the unitary FFT) to $\sqrt{MN}$. Especially in image processing applications the DC term is shifted to the center of the image.

fft_generic allows to select these choices individually. The parameter **Direction** allows to select the logical direction of the FFT. (This parameter is not unnecessary; it is needed to discern how to shift the image if the DC term should rest in the center of the image.) Possible values are *'to_freq'* and *'from_freq'*. The parameter **Exponent** is used to determine the sign of the exponent. It can be set to 1 or -1 . The normalizing factor can be set with **Norm**, and can take on the values *'none'*, *'sqrt'* and *'n'*. The parameter **Mode** determines the location of the DC term of the FFT. It can be set to *'dc_center'* or *'dc_edge'*.

In any case, the user must ensure the consistent use of the parameters. This means that the normalizing factors used for the forward and backward transform must yield MN when multiplied, the exponents must be of opposite sign, and **Mode** must be equal for both transforms.

A consistent combination is, for example (*'to_freq', -1, 'n', 'dc_edge'*) for the forward transform and (*'from_freq', 1, 'none', 'dc_edge'*) for the reverse transform. In this case, the FFT can be interpreted as interpolation with trigonometric basis functions. Another useful combination is (*'to_freq', 1, 'sqrt', 'dc_center'*) and (*'from_freq', -1, 'sqrt', 'dc_center'*).

Parameter

▷ **Image** (input_object) image(-array) $\rightsquigarrow$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex

Input image.

- ▷ **ImageFFT** (output_object) image(-array) $\leadsto$ *Hobject* : complex Fourier-transformed image.
- ▷ **Direction** (input_control) string $\leadsto$ *string*
Calculate forward or reverse transform.
Default Value : 'to-freq'
List of values : *Direction* $\in$ {'to-freq', 'from-freq'}
- ▷ **Exponent** (input_control) integer $\leadsto$ *integer*
Sign of the exponent.
Default Value : 1
List of values : *Exponent* $\in$ {-1, 1}
- ▷ **Norm** (input_control) string $\leadsto$ *string*
Normalizing factor of the transform.
Default Value : 'sqrt'
List of values : *Norm* $\in$ {'none', 'sqrt', 'n'}
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Location of the DC term in the frequency domain.
Default Value : 'dc_center'
List of values : *Mode* $\in$ {'dc_center', 'dc_edge'}

Example

```
/* simulation of fft */
my_fft(Hobject In, Hobject *Out)
{
    fft_generic(In, Out, 'to-freq', 1, 'sqrt', 'dc_center');
}

/* simulation of fft_image_inv */
my_fft_image_inv(Hobject In, Hobject *Out)
{
    Hobject Tmp;
    fft_generic(In, &Tmp, 'from-freq', 1, 'sqrt', 'dc_center');
    convert_image_type(Tmp, Out, "byte");
    clear_obj(Tmp);
}
```

Result

fft_generic returns 2 (H_MSG_TRUE) if the input image is of correct type, its width and height are a power of 2, and all parameters are correct. If the input is empty the behaviour can be set via **set_system** (::'no_object_result', <Result>:). If necessary, an exception handling is raised.

Parallelization Information

fft_generic is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

convol_fft, **convol_gabor**, **convert_image_type**, **power_byte**, **power_real**, **power_ln**, **phase_deg**, **phase_rad**, **energy_gabor**

Alternatives

fft_image, **fft_image_inv**

Module

Image filters

fft_image (Image : ImageFFT : :)

Compute the fast Fourier transform of an image.

fft_image calculates the Fourier transform of the input image (**Image**), i.e., it transforms the image into the frequency domain. The algorithm used is the fast Fourier transform. This corresponds to the call

```
fft_generic(Image, ImageFFT, 'to_freq', 1, 'n', 'dc_center');
```

The result image is of type 'complex'.

Attention

The filtering is always done on the entire image, i.e., the region of the image is ignored. The images must be quadratic and the width and height must be a power of 2.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject*: byte / real
Input image.
- ▷ **ImageFFT** (output_object) image(-array) $\leadsto$ *Hobject*: complex
Fourier-transformed image.

Example

```
fft_image(Image, &FFT);
gen_gabor(&Filter, 1.4, 0.4, 1.0, 1.5, 512);
convol_gabor(FFT, Filter, &Gabor, &Hilbert);
fft_image_inv(Gabor, &GaborInv);
fft_image_inv(Hilbert, &HilbertInv);
energy_gabor(GaborInv, HilbertInv, &Energy);
```

Result

fft_image returns 2 (H_MSG_TRUE) if the input image is of correct type and its width and height are a power of 2. If the input is empty the behaviour can be set via **set_system**(::'no_object_result', <Result>:). If necessary, an exception handling is raised.

Parallelization Information

fft_image is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

convol_fft, **convol_gabor**, **convert_image_type**, **power_byte**, **power_real**, **power_ln**, **phase_deg**, **phase_rad**

Alternatives

fft_generic

See also

fft_image_inv

Module

Image filters

fft_image_inv (Image : ImageFFTInv : :)

Compute the inverse fast Fourier transform of an image.

fft_image_inv calculates the inverse Fourier transform of the input image (**Image**), i.e., it transforms the image back into the spatial domain. This corresponds to the call

```
fft_generic(Image, ImageFFT, 'from_freq', -1, 'sqrt', 'dc_center')
convert_image_type(ImageFFT, ImageFFTInv, 'byte')
```

The result image is of type 'byte'.

Attention

The filtering is always done on the entire image, i.e., the region of the image is ignored. The images must be quadratic and the width and height must be a power of 2.

Parameter
▷ Image (input_object) image(-array) $\leadsto$ <i>Hobject</i> : complex Input image. ▷ ImageFFTInv (output_object) image(-array) $\leadsto$ <i>Hobject</i> : byte Inverse-Fourier-transformed image.
Example
<pre>fft_image(Image,&FFT); gen_gabor(&Filter,1.4,0.4,1.0,1.5,512); convol_gabor(FFT,Filter,&Gabor,&Hilbert); fft_image_inv(Gabor,&GaborInv); fft_image_inv(Hilbert,&HilbertInv); energy_gabor(GaborInv,HilbertInv,&Energy);</pre>
Result
fft_image_inv returns 2 (H_MSG_TRUE) if the input image is of correct type and its width and height are a power of 2. If the input is empty the behaviour can be set via set_system (::'no_object_result', <Result>:) . If necessary, an exception handling is raised.
Parallelization Information
fft_image_inv is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>).
Possible Predecessors
convol_fft , convol_gabor , fft_image
Possible Successors
convert_image_type , energy_gabor
Alternatives
fft_generic
See also
fft_image , fft_generic , energy_gabor
Module
Image filters

gen_bandfilter (: ImageFilter : MinFrequency, MaxFrequency, Size :)

Generate an ideal band filter.

gen_bandfilter generates an ideal band filter in the frequency domain. The DC term is assumed to lie in the center of the image. The parameters **MinFrequency** and **MaxFrequency** determine the cutoff frequencies of the filter (in pixels). The resulting image contains an annulus with the value 0, and the value 255 outside of this annulus.

Parameter
▷ ImageFilter (output_object) image $\leadsto$ <i>Hobject</i> : byte Band filter in the frequency domain. ▷ MinFrequency (input_control) real $\leadsto$ <i>real</i> Minimum frequency. Default Value : 20 Suggested values : $\text{MinFrequency} \in \{10, 20, 30, 40, 50, 60, 70, 100\}$ Typical range of values : $1 \leq \text{MinFrequency}$ Minimum Increment : 1 Recommended Increment : 1 Restriction : $\text{MinFrequency} > 0$

- ▷ **MaxFrequency** (input_control) real $\leadsto$ real
Maximum frequency.
Default Value : 40
Suggested values : MaxFrequency $\in \{10, 20, 30, 40, 50, 60, 70, 100\}$
Typical range of values : $1 \leq \text{MaxFrequency}$
Minimum Increment : 1
Recommended Increment : 1
Restriction : MaxFrequency > 0
- ▷ **Size** (input_control) integer $\leadsto$ integer
Size (dimension) of the image (filter).
Default Value : 512
List of values : Size $\in \{8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192\}$

Example

```
my_lowpass(Hobject Image, Hobject *Result)
{
    Hobject  FFT, Bandpass, FFTConvolve;
    fft_image(Image, &FFT);
    gen_bandfilter(&Bandpass, 20, 40, 512);
    convolve_fft(FFT, Bandpass, &FFTConvolve);
    clear_obj(Bandpass); clear_obj(FFT);
    fft_image_inv(FFTConvolve, Result);
    clear_obj(FFTConvolve);
}
```

Result

[gen_bandfilter](#) returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception handling is raised.

Parallelization Information

[gen_bandfilter](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[convolve_fft](#)

Alternatives

[gen_circle](#), [paint_region](#)

See also

[gen_highpass](#), [gen_lowpass](#), [gen_bandpass](#)

Module

Image filters

gen_bandpass (: ImageBandpass : MinFrequency, MaxFrequency, Size :)

Generate an ideal bandpass filter.

[gen_bandpass](#) generates an ideal bandpass filter in the frequency domain. The DC term is assumed to lie in the center of the image. The parameters [MinFrequency](#) and [MaxFrequency](#) determine the cutoff frequencies of the filter (in pixels). The resulting image contains an annulus with the value 255, and the value 0 outside of this annulus.

Parameter

- ▷ **ImageBandpass** (output_object) image $\leadsto$ Hobject : byte
Bandpass filter in the frequency domain.

- ▷ **MinFrequency** (input_control) real $\leadsto$ real
 Minimum frequency.
Default Value : 20
Suggested values : MinFrequency $\in \{10, 20, 30, 40, 50, 60, 70, 100\}$
Typical range of values : $1 \leq \text{MinFrequency} \leq 200$
Minimum Increment : 1
Recommended Increment : 1
Restriction : MinFrequency > 0
- ▷ **MaxFrequency** (input_control) real $\leadsto$ real
 Maximum frequency.
Default Value : 40
Suggested values : MaxFrequency $\in \{10, 20, 30, 40, 50, 60, 70, 100\}$
Typical range of values : $1 \leq \text{MaxFrequency} \leq 200$
Minimum Increment : 1
Recommended Increment : 1
Restriction : MaxFrequency > 0
- ▷ **Size** (input_control) integer $\leadsto$ integer
 Size (dimension) of the image (filter).
Default Value : 512
List of values : Size $\in \{8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192\}$

Example

```
my_lowpass(Hobject Image, Hobject *Result)
{
  Hobject  FFT, Bandpass, FFTConvolve;
  fft(Image,&FFT);
  gen_bandpass(&Bandpass, 20, 40, 512);
  convolve_fft(FFT, Bandpass, &FFTConvolve);
  clear_obj(Bandpass); clear_obj(FFT);
  fft_image_inv(FFTConvolve, Result);
  clear_obj(FFTConvolve);
}
```

Result

[gen_bandpass](#) returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception handling is raised.

Parallelization Information

[gen_bandpass](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[convolve_fft](#)

Alternatives

[gen_circle](#), [paint_region](#)

See also

[gen_highpass](#), [gen_lowpass](#), [gen_bandfilter](#)

Module

Image filters

gen_filter_mask (: ImageFilter : FilterMask, Scale, Size :)

Store a filter mask in the spatial domain as a real-image.

[gen_filter_mask](#) stores a filter mask in the spatial domain as a real-image. The center of the filter mask lies in the center of the resulting image. The parameter [Scale](#) determines by which amount the values of the filter mask are multiplied (this results in larger values of the Fourier transform of the filter). The corresponding filter matrix, which is given in [FilterMask](#) can be generated either from a file or a tuple. The format of the filter

matrix is described with the operator [convol_image](#). Example filter masks can be found in the directory “filter” in the HALCON home directory. This operator is useful for visualizing the frequency response of filter masks (by applying a Fourier transform to the result image of this operator).

Parameter

- ▷ **ImageFilter** (output_object) image(-array) $\leadsto$ *Hobject* : real
Filter in the spatial domain.
- ▷ **FilterMask** (input_control) string(-array) $\leadsto$ *string* / integer
Filter mask as file name or tuple.
Default Value : 'sobel'
Suggested values : FilterMask $\in$ {'laplace4', 'laplace8', 'lowpass_3_3'}
- ▷ **Scale** (input_control) real $\leadsto$ *real*
Scaling factor.
Default Value : 1.0
Suggested values : Scale $\in$ {0.3, 0.5, 0.75, 1.0, 1.25, 1.5, 2.0}
Typical range of values : $0.001 \leq \text{Scale} \leq 10.0$
Minimum Increment : 0.001
Recommended Increment : 0.1
Restriction : Scale > 0.0
- ▷ **Size** (input_control) integer $\leadsto$ *integer*
Size (dimension) of the image (filter).
Default Value : 512
List of values : Size $\in$ {8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192}

Example

```
* If the filter should be read from a file:
gen_filter_mask (Filter, 'lowpas_3_3', 1.0, 512)
* If the filter should be directly passed as a tuple:
gen_filter_mask (Filter, [3,3,9,1,1,1,1,1,1,1,1,1], 1.0, 512)
fft_image (Filter, FilterFFT)
set_paint (WindowHandle, '3D-plot_hidden')
disp_image (FilterFFT, WindowHandle)
```

Parallelization Information

[gen_filter_mask](#) is *reentrant* and processed *without parallelization*.

Possible Successors

[fft_image](#), [fft_generic](#)

See also

[convol_image](#)

Module

Image filters

gen_gabor (: ImageFilter : Angle, Frequency, Bandwidth, Orientation, Size :)

Generate a Gabor filter.

[gen_gabor](#) generates a Gabor filter with a user-definable bandpass frequency range and sign for the Hilbert transformation. This is done by calculating a symmetrical filter in the frequency domain, which can be adapted by the parameters [Angle](#), [Frequency](#), [Bandwidth](#) und [Orientation](#) such that a certain frequency band and a certain direction range in the spatial domain is filtered out in the frequency domain.

The parameters [Frequency](#) (central frequency = distance from the DC term) and [Orientation](#) (direction) determine the center of the filter. Larger values of [Frequency](#) result in higher frequencies being passed. A value of 0 for [Orientation](#) generates a horizontally oriented “crescent” (the bulge of the crescent points upward). Higher values of [Orientation](#) result in the counterclockwise rotation of the crescent.

The parameters **Angle** and **Bandwidth** are used to determine the range of frequencies and angles being passed by the filter. The larger **Angle** is, the smaller the range of angles passed by the filter gets (because the “crescent” gets narrower). The larger **Bandwidth** is, the smaller the frequency band being passed gets (because the “crescent” gets thinner).

The resulting image is a two-channel real-image, containing the Gabor filter in the first channel and the corresponding Hilbert filter in the second channel.

<i>Parameter</i>	
▷ ImageFilter (output_object)	multichannel-image(-array) $\leadsto$ <i>Hobject</i> : real Gabor and Hilbert filter.
▷ Angle (input_control)	real $\leadsto$ real Angle range, inversely proportional to the range of orientations. Default Value : 1.4 Suggested values : Angle $\in \{1.0, 1.2, 1.4, 1.6, 2.0, 2.5, 3.0, 5.0, 6.0, 10.0, 20.0, 30.0, 50.0, 70.0, 100.0\}$ Typical range of values : $1.0 \leq \text{Angle} \leq 500.0$ Minimum Increment : 0.001 Recommended Increment : 0.1
▷ Frequency (input_control)	real $\leadsto$ real Distance of the center of the filter to the DC term. Default Value : 0.4 Suggested values : Frequency $\in \{0.0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.50, 0.55, 0.60, 0.65, 0.699\}$ Typical range of values : $0.0 \leq \text{Frequency} \leq 0.7$ Minimum Increment : 0.00001 Recommended Increment : 0.005
▷ Bandwidth (input_control)	real $\leadsto$ real Bandwidth range, inversely proportional to the range of frequencies being passed. Default Value : 1.0 Suggested values : Bandwidth $\in \{0.1, 0.3, 0.7, 1.0, 1.5, 2.0, 3.0, 5.0, 7.0, 10.0, 15.0, 20.0, 30.0, 50.0\}$ Typical range of values : $0.05 \leq \text{Bandwidth} \leq 100.0$ Minimum Increment : 0.001 Recommended Increment : 0.1
▷ Orientation (input_control)	real $\leadsto$ real Angle of the principal orientation being passed. Default Value : 1.5 Suggested values : Orientation $\in \{0.0, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.0, 2.2, 2.4, 2.6, 2.8, 3.0, 3.14\}$ Typical range of values : $0.0 \leq \text{Orientation} \leq 3.1416$ Minimum Increment : 0.0001 Recommended Increment : 0.05
▷ Size (input_control)	integer $\leadsto$ integer Size (dimension) of the image (filter). Default Value : 512 List of values : Size $\in \{8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192\}$
<i>Example</i>	

```
fft_image(Image,&FFT);
gen_gabor(&Filter,1.4,0.4,1.0,1.5,512);
convol_gabor(FFT,Filter,&Gabor,&Hilbert);
fft_image_inv(Gabor,&GaborInv);
fft_image_inv(Hilbert,&HilbertInv);
energy_gabor(GaborInv,HilbertInv,&Energy);
```

Result
gen_gabor returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception handling is raised.

Parallelization Information
gen_gabor is *reentrant* and processed *without* parallelization.

fft_image , fft_generic	Possible Predecessors
convol_gabor	Possible Successors
gen_bandpass , gen_bandfilter , gen_highpass , gen_lowpass	Alternatives
fft_image_inv , energy_gabor	See also
Image filters	Module

gen_highpass (: ImageHighpass : Frequency, Size :)

Generate an ideal highpass filter.

[gen_highpass](#) generates an ideal highpass filter in the frequency domain. The DC term is assumed to lie in the center of the image. The parameter [Frequency](#) determines the cutoff frequency of the filter (in pixels). The resulting image contains a circle of radius [Frequency](#) with the value 0, and the value 255 outside of this circle.

	Parameter
▷ ImageHighpass (output_object) image $\rightsquigarrow$ <i>Hobject</i> : byte Highpass filter in the frequency domain. ▷ Frequency (input_control) real $\rightsquigarrow$ <i>real</i> Cutoff frequency. Default Value : 20 Suggested values : Frequency $\in \{10, 20, 30, 40, 50, 60, 70, 100\}$ Typical range of values : $1 \leq \text{Frequency} \leq 200$ Minimum Increment : 1 Recommended Increment : 1 Restriction : Frequency > 0 ▷ Size (input_control) integer $\rightsquigarrow$ <i>integer</i> Size (dimension) of the image (filter). Default Value : 512 List of values : Size $\in \{16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192\}$	

	Example

```
my_highpass(Hobject Image, Hobject *Result, int frequency, int size)
{
  Hobject FFT, Highpass, FFTConvol;
  fft_image(Image, &FFT);
  gen_highpass(&Highpass, frequency, size);
  convol_fft(FFT, Highpass, &FFTConvol);
  clear_obj(Highpass); clear_obj(FFT);
  fft_image_inv(FFTConvol, Result);
  clear_obj(FFTConvol);
}
```

	Result

[gen_highpass](#) returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception handling is raised.

	Parallelization Information

[gen_highpass](#) is *reentrant* and processed *without* parallelization.

	Possible Successors

[convol_fft](#)

gen_circle , paint_region	Alternatives
convol_fft , gen_lowpass , gen_bandpass , gen_bandfilter	See also
Image filters	Module

gen_lowpass (: ImageLowpass : Frequency, Size :)

Generate an ideal lowpass filter.

[gen_lowpass](#) generates an ideal lowpass filter in the frequency domain. The DC term is assumed to lie in the center of the image. The parameter [Frequency](#) determines the cutoff frequency of the filter (in pixels). The resulting image contains a circle of radius [Frequency](#) with the value 255, and the value 0 outside of this circle.

	Parameter
▷ ImageLowpass (output_object) image $\rightsquigarrow$ Hobject : byte Highpass filter in the frequency domain. ▷ Frequency (input_control) real $\rightsquigarrow$ real Cutoff frequency. Default Value : 20 Suggested values : Frequency $\in \{10, 20, 30, 40, 50, 60, 70, 100\}$ Typical range of values : $1 \leq \text{Frequency} \leq 200$ Minimum Increment : 1 Recommended Increment : 1 ▷ Size (input_control) integer $\rightsquigarrow$ integer Size (dimension) of the image (filter). Default Value : 512 List of values : Size $\in \{16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192\}$	
	Example

```
my_lowpass(Hobject Image, Hobject *Result, int frequency, int size)
{
    Hobject  FFT, Lowpass, FFTConvol;
    fft(Image, &FFT);
    gen_lowpass(&Lowpass, frequency, size);
    convol_fft(FFT, Lowpass, &FFTConvol);
    clear_obj(Lowpass); clear_obj(FFT);
    fft_image_inv(FFTConvol, Result);
    clear_obj(FFTConvol);
}
```

gen_lowpass returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception handling is raised.	Result

gen_lowpass is <i>reentrant</i> and processed <i>without</i> parallelization.	Parallelization Information

convol_fft	Possible Successors

gen_circle , paint_region	Alternatives

gen_highpass , gen_bandpass , gen_bandfilter	See also

Image filters	Module

gen_sin_bandpass (: ImageFilter : Frequency, Size :)

Generate a bandpass filter with sinusoidal shape.

[gen_sin_bandpass](#) generates a rotationally invariant bandpass filter with the response being a sinusoidal function in the frequency domain. The maximum of the sine (255) is determined by [Frequency](#) (distance from the DC term in pixels). The filter is always zero for the DC term, rises with the sine function up to [Frequency](#), and drops for higher frequencies accordingly. The range of the sine used is from 0 to π . All other points are set to zero.

Parameter

- ▷ **ImageFilter** (output_object) image(-array) $\leadsto$ *Hobject* : byte
Bandpass filter as image in the frequency domain.
- ▷ **Frequency** (input_control) real $\leadsto$ *real*
Distance of the filter's maximum from the DC term.
Default Value : 20
Suggested values : $\text{Frequency} \in \{0.0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 80, 100, 150, 200\}$
Typical range of values : $0.0 \leq \text{Frequency} \leq 1000$
Minimum Increment : 1
Recommended Increment : 2
- ▷ **Size** (input_control) integer $\leadsto$ *integer*
Size (dimension) of the image (filter).
Default Value : 512
List of values : $\text{Size} \in \{8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192\}$

Result

[gen_sin_bandpass](#) returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception handling is raised.

Parallelization Information

[gen_sin_bandpass](#) is *reentrant* and processed *without parallelization*.

Possible Predecessors

[fft_image](#), [fft_generic](#)

Possible Successors

[convol_fft](#)

Alternatives

[gen_std_bandpass](#), [gen_bandpass](#), [gen_bandfilter](#), [gen_highpass](#), [gen_lowpass](#)

See also

[fft_image_inv](#)

Module

Image filters

gen_std_bandpass (: ImageFilter : Frequency, Sigma, Size, Mode :)

Generate a bandpass filter with Gaussian or sinusoidal shape.

[gen_std_bandpass](#) generates a rotationally invariant bandpass filter with the response being determined by the parameters [Frequency](#) and [Sigma](#): [Frequency](#) determines the location of the maximum response with respect to the DC term, while [Sigma](#) determines the width of the frequency band that passes the filter. For [Mode](#) = 'gauss', a Gaussian response is generated with [Sigma](#) being the standard deviation. For [Mode](#) = 'sin', a sine function is generated with the maximum at [Frequency](#) and the extent [Sigma](#).

Parameter

- ▷ **ImageFilter** (output_object) image(-array) $\leadsto$ *Hobject* : byte
Bandpass filter as image in the frequency domain.

- ▷ **Frequency** (input_control) real $\leadsto$ real
Distance of the filter's maximum from the DC term.
Default Value : 20
Suggested values : Frequency $\in \{0.0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 80, 100, 150, 200\}$
Typical range of values : $0.0 \leq \text{Frequency} \leq 1000$
Minimum Increment : 1
Recommended Increment : 2
- ▷ **Sigma** (input_control) real $\leadsto$ real
Bandwidth of the filter (standard deviation).
Default Value : 1.0
Suggested values : Sigma $\in \{0.1, 0.3, 0.7, 1.0, 1.5, 2.0, 3.0, 5.0, 7.0, 10.0, 15.0, 20.0, 30.0, 50.0\}$
Typical range of values : $0.05 \leq \text{Sigma} \leq 100.0$
Minimum Increment : 0.001
Recommended Increment : 0.1
- ▷ **Size** (input_control) integer $\leadsto$ integer
Size (dimension) of the image (filter).
Default Value : 512
List of values : Size $\in \{8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096, 8192\}$
- ▷ **Mode** (input_control) string $\leadsto$ string
Filter type.
Default Value : 'sin'
List of values : Mode $\in \{'sin', 'gauss'\}$

Result

[gen_std_bandpass](#) returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception handling is raised.

Parallelization Information

[gen_std_bandpass](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[fft_image](#), [fft_generic](#)

Possible Successors

[convol_fft](#)

Alternatives

[gen_sin_bandpass](#), [gen_bandpass](#), [gen_bandfilter](#), [gen_highpass](#), [gen_lowpass](#)

See also

[fft_image_inv](#)

Module

Image filters

phase_deg (ImageComplex : ImagePhase : :)

Return the phase of a complex image in degrees.

[phase_deg](#) computes the phase of a complex image in degrees. The following formula is used:

$$phase = \frac{180 \tan^{-1}(imaginarypart/realpart)}{\pi} .$$

Parameter

- ▷ **ImageComplex** (input_object) image(-array) $\leadsto$ Hobject : complex
Input image in frequency domain.
- ▷ **ImagePhase** (output_object) image(-array) $\leadsto$ Hobject : direction
Phase of the image in degrees.

Example

```
read_image(&Image, "affe");
disp_image(Image, WindowHandle);
fft_image(Image, &FFT);
phase_deg(FFT, &Phase);
disp_image(Phase, WindowHandle);
```

Result

phase_deg returns 2 (H_MSG_TRUE) if the image is of correct type. If the input is empty the behaviour can be set via **set_system(, 'no_object_result', <Result>:)**. If necessary, an exception handling is raised.

Parallelization Information

phase_deg is *reentrant* and automatically *parallelized* (on tuple level, domain level).

Possible Predecessors

fft_image, **fft_generic**

Possible Successors

disp_image

Alternatives

phase_rad

See also

fft_image_inv

Module

Image filters

phase_rad (ImageComplex : ImagePhase : :)

Return the phase of a complex image in radians.

phase_rad computes the phase of a complex image in radians. The following formula is used:

$$phase = \tan^{-1}(\text{imaginarypart}/\text{realpart}) \text{ .}$$

Parameter

- ▷ **ImageComplex** (input_object) image(-array) $\leadsto$ *Hobject* : complex
Input image in frequency domain.
- ▷ **ImagePhase** (output_object) image(-array) $\leadsto$ *Hobject* : real
Phase of the image in radians.

Example

```
read_image(&Image, "affe");
disp_image(Image, WindowHandle);
fft_image(Image, &FFT);
phase_rad(FFT, &Phase);
disp_image(Phase, WindowHandle);
```

Result

phase_rad returns 2 (H_MSG_TRUE) if the image is of correct type. If the input is empty the behaviour can be set via **set_system(, 'no_object_result', <Result>:)**. If necessary, an exception handling is raised.

Parallelization Information

phase_rad is *reentrant* and automatically *parallelized* (on tuple level, domain level).

Possible Predecessors

fft_image, **fft_generic**

	Possible Successors
disp_image	
	Alternatives
phase_deg	
	See also
fft_image_inv	
	Module
Image filters	

power_byte (Image : PowerByte : :)

Return the power spectrum of a complex image.

[power_byte](#) computes the power spectrum from the real and imaginary parts of a Fourier-transformed image (see [fft_image](#)), i.e., the modulus of the frequencies. The result image is of type 'byte'. The following formula is used:

$$\sqrt{\text{realpart}^2 + \text{imaginarypart}^2} .$$

	Parameter
▷ Image (input_object) image(-array) $\leadsto$ Hobject : complex	
Input image in frequency domain.	
▷ PowerByte (output_object) image(-array) $\leadsto$ Hobject : byte	
Power spectrum of the input image.	
	Example

```
read_image(&Image, "affe");
disp_image(Image, WindowHandle);
fft_image(Image, &FFT);
power_byte(FFT, &Power);
disp_image(Power, WindowHandle);
```

	Result
power_byte returns 2 (H_MSG_TRUE) if the image is of correct type. If the input is empty the behaviour can be set via set_system(::'no_object_result', <Result>:) . If necessary, an exception handling is raised.	

	Parallelization Information
power_byte is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level, domain level).	
	Possible Predecessors
fft_image , fft_generic , convol_fft , convol_gabor	
	Possible Successors
disp_image	
	Alternatives
abs_image , convert_image_type , power_real , power_ln	
	See also
fft_image	
	Module
Image filters	

power_ln (Image : ImageResult : :)

Return the power spectrum of a complex image.

`power_ln` computes the power spectrum from the real and imaginary parts of a Fourier-transformed image (see `fft_image`), i.e., the modulus of the frequencies. Additionally, the natural logarithm is applied to the result. The result image is of type 'real'. The following formula is used:

$$\ln \left(\sqrt{\text{realpart}^2 + \text{imaginarypart}^2} \right) .$$

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : complex
Input image in frequency domain.
- ▷ **ImageResult** (output_object) image(-array) $\leadsto$ *Hobject* : real
Power spectrum of the input image.

Example

```
read_image(&Image, "monkey");
disp_image(Image, WindowHandle);
fft_image(Image, &FFT);
power_ln(FFT, &Power);
disp_image(Power, WindowHandle);
```

Result

`power_ln` returns 2 (H_MSG_TRUE) if the image is of correct type. If the input is empty the behaviour can be set via `set_system(::'no_object_result', <Result>:)`. If necessary, an exception handling is raised.

Parallelization Information

`power_ln` is *reentrant* and automatically *parallelized* (on tuple level, domain level).

Possible Predecessors

`fft_image`, `fft_generic`, `convol_fft`, `convol_gabor`

Possible Successors

`disp_image`, `convert_image_type`, `scale_image`

Alternatives

`abs_image`, `convert_image_type`, `power_real`, `power_byte`

See also

`fft_image`, `fft_generic`

Module

Image filters

power_real (Image : ImageResult : :)

Return the power spectrum of a complex image.

`power_real` computes the power spectrum from the real and imaginary parts of a Fourier-transformed image (see `fft_image`), i.e., the modulus of the frequencies. The result image is of type 'real'. The following formula is used:

$$\sqrt{\text{realpart}^2 + \text{imaginarypart}^2} .$$

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : complex
Input image in frequency domain.
- ▷ **ImageResult** (output_object) image(-array) $\leadsto$ *Hobject* : real
Power spectrum of the input image.

Example

```
read_image(&Image, "affe");
disp_image(Image, WindowHandle);
fft_image(Image, &FFT);
power_real(FFT, &Power);
disp_image(Power, WindowHandle);
```

Result

`power_real` returns 2 (H_MSG_TRUE) if the image is of correct type. If the input is empty the behaviour can be set via `set_system(:, 'no_object_result', <Result>:)`. If necessary, an exception handling is raised.

Parallelization Information

`power_real` is *reentrant* and automatically *parallelized* (on *tuple level*, *domain level*).

Possible Predecessors

`fft_image`, `fft_generic`, `convol_fft`, `convol_gabor`

Possible Successors

`disp_image`, `convert_image_type`, `scale_image`

Alternatives

`abs_image`, `convert_image_type`, `power_byte`, `power_ln`

See also

`fft_image`

Module

Image filters

5.8 Lines

bandpass_image (Image : ImageBandpass : FilterType :)

Edge extraction using bandpass filters.

`bandpass_image` serves as an edge filter. It applies a linear filter with the following convolution mask to `Image`:

FilterType: 'lines'

In contrast to the edge operator `sobel_amp` this filter detects lines instead of edges, i.e., two closely adjacent edges.

```

    0  -2  -2  -2   0
   -2   0   3   0  -2
   -2   3  12   3  -2
   -2   0   3   0  -2
    0  -2  -2  -2   0
```

At the border of the image the gray values are mirrored. Over- and underflows of gray values are clipped. The resulting images are returned in `ImageBandpass`.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject*: byte / uint2
Input images.
- ▷ **ImageBandpass** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject*: byte / uint2
Bandpass-filtered images.
- ▷ **FilterType** (input_control) string $\leadsto$ *string*
Filter type: currently only 'lines' is supported.
Default Value : 'lines'
List of values : FilterType $\in$ { 'lines' }

Example

```
bandpass_image(Image, &LineImage, "lines");
threshold(LineImage, &Lines, 60.0, 255.0);
skeleton(Lines, &ThinLines);
```

Result

[bandpass_image](#) returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via [set_system\(::'no_object_result', <Result>:\)](#). If necessary, an exception handling is raised.

Parallelization Information

[bandpass_image](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Successors

[threshold](#), [skeleton](#)

Alternatives

[convol_image](#), [topographic_sketch](#), [texture_laws](#)

See also

[highpass_image](#), [gray_skeleton](#)

Module

Image filters

lines_facet (Image : Lines : MaskSize, Low, High, LightDark :)

Detection of lines using the facet model.

The operator [lines_facet](#) can be used to extract lines (curvilinear structures) from the image [Image](#). The extracted lines are returned in [Lines](#) as sub-pixel precise XLD-contours. The parameter [LightDark](#) determines, whether bright or dark lines are extracted.

The extraction is done by using the facet model, i.e., a least squares fit, to determine the parameters of a quadratic polynomial in x and y for each point of the image. The parameter [MaskSize](#) determines the size of the window used for the least squares fit. Larger values of [MaskSize](#) lead to a larger smoothing of the image, but can lead to worse localization of the line. The parameters of the polynomial are used to calculate the line direction for each pixel. Pixels which exhibit a local maximum in the second directional derivative perpendicular to the line direction are marked as line points. The line points found in this manner are then linked to contours. This is done by immediately accepting line points that have a second derivative larger than [High](#). Points that have a second derivative smaller than [Low](#) are rejected. All other line points are accepted if they are connected to accepted points by a connected path. This is similar to a hysteresis threshold operation with infinite path length (see [hysteresis_threshold](#)). However, this function is not used internally since it does not allow the extraction of sub-pixel precise contours.

The gist of how to select the thresholds in the description of [lines_gauss](#) also holds for this operator. A value of Sigma = 1.5 there roughly corresponds to a [MaskSize](#) of 5 here.

The extracted lines are returned in a topologically sound data structure in [Lines](#). This means that lines are correctly split at junction points.

[lines_facet](#) defines the following attributes for each line point:

'angle' The angle of the direction perpendicular to the line
'response' The magnitude of the second derivative

These attributes can be queried via the operator [get_contour_attrib_xld](#).

Attention

The smaller the filter size [MaskSize](#) is chosen, the more short, fragmented lines will be extracted. This can lead to considerably longer execution times.

Parameter

- ▷ **Image** (input_object) singlechannel-image $\leadsto$ *Hobject* : byte / int1 / int2 / uint2 / int4 / real
Input image.
- ▷ **Lines** (output_object) xld_cont-array $\leadsto$ *Hobject*
Extracted lines.
- ▷ **MaskSize** (input_control) integer $\leadsto$ *integer*
Size of the facet model mask.
Default Value : 5
List of values : MaskSize $\in \{3, 5, 7, 9, 11\}$
- ▷ **Low** (input_control) number $\leadsto$ *real* / *integer*
Lower threshold for the hysteresis threshold operation.
Default Value : 3
Suggested values : Low $\in \{0, 0.5, 1, 2, 3, 4, 5, 8, 10\}$
Typical range of values : $0 \leq \text{Low} \leq 20$
Recommended Increment : 0.5
Restriction : Low ≥ 0
- ▷ **High** (input_control) number $\leadsto$ *real* / *integer*
Upper threshold for the hysteresis threshold operation.
Default Value : 8
Suggested values : High $\in \{0, 0.5, 1, 2, 3, 4, 5, 8, 10, 12, 15, 18, 20, 25\}$
Typical range of values : $0 \leq \text{High} \leq 35$
Recommended Increment : 0.5
Restriction : (High ≥ 0) $\wedge$ (High $\geq$ Low)
- ▷ **LightDark** (input_control) string $\leadsto$ *string*
Extract bright or dark lines.
Default Value : 'light'
List of values : LightDark $\in \{\text{'dark'}, \text{'light'}\}$

Example

```
/* Detection of lines in an aerial image */
read_image(Image, 'mreut4_3')
lines_facet(Image, Lines, 5, 3, 8, 'light')
disp_xld(Lines, WindowHandle).
```

Complexity

Let A be the number of pixels in the domain of **Image**. Then the runtime complexity is $O(A * \text{MaskSize})$.

Let $S = \text{Width} * \text{Height}$ be the number of pixels of **Image**. Then **lines_facet** requires at least $55 * S$ bytes of temporary memory during execution.

Result

lines_facet returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during execution. If the input is empty the behaviour can be set via **set_system(::'no_object_result', <Result>:)**. If necessary, an exception handling is raised.

Parallelization Information

lines_facet is *reentrant* and processed *without* parallelization.

Possible Successors

gen_polygons_xld

Alternatives

lines_gauss

See also

bandpass_image, **dyn_threshold**, **topographic_sketch**

References

A. Busch: "Fast Recognition of Lines in Digital Images Without User-Supplied Parameters". In H. Ebner, C. Heipke, K. Eder, eds., "Spatial Information from Digital Photogrammetry and Computer Vision", International Archives of Photogrammetry and Remote Sensing, Vol. 30, Part 3/1, pp. 91-97, 1994.

Module

Sub-pixel operators

lines_gauss (Image : Lines : Sigma, Low, High, LightDark, ExtractWidth, CorrectPositions, CompleteJunctions :)

Detect lines and their width.

The operator **lines_gauss** can be used to extract lines (curvilinear structures) from the image **Image**. The extracted lines are returned in **Lines** as sub-pixel precise XLD-contours. The parameter **LightDark** determines, whether bright or dark lines are extracted. If **ExtractWidth** is set to *'true'* the line width is extracted for each line point. If **CorrectPositions** is set to *'true'*, **lines_gauss** compensates the effect of asymmetrical lines (lines having different contrast on each side of the line), and corrects the position and width of the line. This parameter is only meaningful if **ExtractWidth**=*'true'*. Because the line extractor is unable to extract certain junctions because of differential geometric reasons, it tries to extract these by different means if **CompleteJunctions** is set to *'true'*.

The extraction is done by using partial derivatives of a Gaussian smoothing kernel to determine the parameters of a quadratic polynomial in x and y for each point of the image. The parameter **Sigma** determines the amount of smoothing to be performed. Larger values of **Sigma** lead to a larger smoothing of the image, but can lead to worse localization of the line. Generally, the localization will be much better than that of lines returned by **lines_facet** with comparable parameters. The parameters of the polynomial are used to calculate the line direction for each pixel. Pixels which exhibit a local maximum in the second directional derivative perpendicular to the line direction are marked as line points. The line points found in this manner are then linked to contours. This is done by immediately accepting line points that have a second derivative larger than **High**. Points that have a second derivative smaller than **Low** are rejected. All other line points are accepted if they are connected to accepted points by a connected path. This is similar to a hysteresis threshold operation with infinite path length (see **hysteresis_threshold**). However, this function is not used internally since it does not allow the extraction of sub-pixel precise contours.

For the choice of the thresholds **High** and **Low** one has to keep in mind that the second directional derivative depends on the amplitude and width of the line as well as the choice of **Sigma**. The value of the second derivative depends linearly on the amplitude, i.e., the larger the amplitude, the larger the response. For the width of the line there is an approximately inverse exponential dependence: The wider the line is, the smaller the response gets. This holds analogously for the dependence on **Sigma**: The larger **Sigma** is chosen, the smaller the second derivative will be. This means that for larger smoothing correspondingly smaller values for **High** and **Low** have to be chosen. Two examples help to illustrate this: If 5 pixel wide lines with an amplitude larger than 100 are to be extracted from an image with a smoothing of **Sigma** = 1.5, **High** should be chosen larger than 14. If, on the other hand, 10 pixel wide lines with an amplitude larger than 100 and a **Sigma** = 3 are to be detected, **High** should be chosen larger than 3.5. For the choice of **Low** values between 0.25 **High** and 0.5 **High** are appropriate.

The extracted lines are returned in a topologically sound data structure in **Lines**. This means that lines are correctly split at junction points.

lines_gauss defines the following attributes for each line point if **ExtractWidth** was set to *'false'*:

'angle' The angle of the direction perpendicular to the line
'response' The magnitude of the second derivative

If **ExtractWidth** was set to *'true'* and **CorrectPositions** to *'false'*, the following attributes are defined in addition to the above ones:

'width_left' The line width to the left of the line
'width_right' The line width to the right of the line

Finally, if **CorrectPositions** was set to *'true'*, additionally the following attributes are defined:

'asymmetry' The asymmetry of the line point
'contrast' The contrast of the line point

Here, the asymmetry is positive if the asymmetric part, i.e., the part with the weaker gradient, is on the right side of the line, while it is negative if the asymmetric part is on the left side of the line. All these attributes can be queried via the operator **get_contour_attr_xld**.

Attention

In general, but in particular if the line width is to be extracted, $\text{Sigma} \geq w/\sqrt{3}$ should be selected, where w is the width (half the diameter) of the lines in the image. As the lowest allowable value $\text{Sigma} \geq w/2.5$ must be selected. If, for example, lines with a width of 4 pixels (diameter 8 pixels) are to be extracted, $\text{Sigma} \geq 2.3$ should be selected.

Parameter	
▷ Image (input_object)	singlechannel-image $\leadsto$ <i>Hobject</i> : byte / int1 / int2 / uint2 / int4 / real Input image.
▷ Lines (output_object)	xld_cont-array $\leadsto$ <i>Hobject</i> Extracted lines.
▷ Sigma (input_control)	number $\leadsto$ <i>real</i> / integer Amount of Gaussian smoothing to be applied. Default Value : 1.5 Suggested values : $\text{Sigma} \in \{1, 1.2, 1.5, 1.8, 2, 2.5, 3, 4, 5\}$ Typical range of values : $0.7 \leq \text{Sigma} \leq 20$ Recommended Increment : 0.1
▷ Low (input_control)	number $\leadsto$ <i>real</i> / integer Lower threshold for the hysteresis threshold operation. Default Value : 3 Suggested values : $\text{Low} \in \{0, 0.5, 1, 2, 3, 4, 5, 8, 10\}$ Typical range of values : $0 \leq \text{Low} \leq 20$ Recommended Increment : 0.5 Restriction : $\text{Low} \geq 0$
▷ High (input_control)	number $\leadsto$ <i>real</i> / integer Upper threshold for the hysteresis threshold operation. Default Value : 8 Suggested values : $\text{High} \in \{0, 0.5, 1, 2, 3, 4, 5, 8, 10, 12, 15, 18, 20, 25\}$ Typical range of values : $0 \leq \text{High} \leq 35$ Recommended Increment : 0.5 Restriction : $(\text{High} \geq 0) \wedge (\text{High} \geq \text{Low})$
▷ LightDark (input_control)	string $\leadsto$ <i>string</i> Extract bright or dark lines. Default Value : 'light' List of values : $\text{LightDark} \in \{\text{'dark'}, \text{'light'}\}$
▷ ExtractWidth (input_control)	string $\leadsto$ <i>string</i> Should the line width be extracted? Default Value : 'true' List of values : $\text{ExtractWidth} \in \{\text{'true'}, \text{'false'}\}$
▷ CorrectPositions (input_control)	string $\leadsto$ <i>string</i> Should the line position and width be corrected? Default Value : 'true' List of values : $\text{CorrectPositions} \in \{\text{'true'}, \text{'false'}\}$
▷ CompleteJunctions (input_control)	string $\leadsto$ <i>string</i> Should junctions be added where they cannot be extracted? Default Value : 'true' List of values : $\text{CompleteJunctions} \in \{\text{'true'}, \text{'false'}\}$

Example

```
/* Detection of lines in an aerial image */
read_image(Image, 'mreut4_3')
lines_gauss(Image, Lines, 1.5, 3, 8, 'light', 'true', 'true', 'true')
disp_xld(Lines, WindowHandle).
```

Complexity

Let A be the number of pixels in the domain of **Image**. Then the runtime complexity is $O(A * \text{Sigma})$.

Let $S = \text{Width} * \text{Height}$ be the number of pixels of **Image**. Then **lines_gauss** requires at least $55 * S$ bytes of temporary memory during execution.

Result

lines_gauss returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during execution. If the input is empty the behaviour can be set via **set_system(::'no_object_result', <Result>:)**. If necessary, an exception handling is raised.

<i>Parallelization Information</i>
<code>lines_gauss</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
<i>Possible Successors</i>
<code>gen_polygons_xld</code>
<i>Alternatives</i>
<code>lines_facet</code>
<i>See also</i>
<code>bandpass_image</code> , <code>dyn_threshold</code> , <code>topographic_sketch</code>
<i>References</i>
C. Steger: "Extracting Curvilinear Structures: A Differential Geometric Approach". In B. Buxton, R. Cipolla, eds., "Fourth European Conference on Computer Vision", Lecture Notes in Computer Science, Volume 1064, Springer Verlag, pp. 630-641, 1996.
C. Steger: "Extraction of Curved Lines from Images". In "13th International Conference on Pattern Recognition", Volume II, pp. 251-255, 1996.
C. Steger: "An Unbiased Detector of Curvilinear Structures". Technical Report FGBV-96-03, Forschungsgruppe Bildverstehen (FG BV), Informatik IX, Technische Universität München, July 1996.
<i>Module</i>
Sub-pixel operators

5.9 Match

corner_response (Image : ImageCorner : Size, Weight :)

Searching corners in images.

The operator `corner_response` extracts gray value corners in an image. The formula for the calculation of the response is:

$$\begin{aligned}
 R(x, y) &= A(x, y) \cdot B(x, y) - C^2(x, y) - Weight \cdot (A(x, y) + B(x, y))^2 \\
 A(x, y) &= W(u, v) * (\nabla_x I(x, y))^2 \\
 B(x, y) &= W(u, v) * (\nabla_y I(x, y))^2 \\
 C(c, y) &= W(u, v) * (\nabla_x I(x, y) \nabla_y I(x, y))
 \end{aligned}$$

where I is the input image and R the output image of the filter. The operator `gauss_image` is used for smoothing (W), the operator `sobel_amp` is used for calculating the derivative (∇).

The corner response function is invariant with regard to rotation. In order to achieve a suitable dependency of the function $R(x, y)$ on the local gradient, the parameter `weight` must be set to 0.04. With this, only gray value corners will return positive values for $R(x, y)$, while straight edges will receive negative values. The output image type is identical to the input image type. Therefore, the negative output values are set to 0 if byte images are used as input images. If this is not desired, the input image should be converted into a real or int2 image with `convert_image_type`.

<i>Parameter</i>
▷ Image (input_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / real Input image.
▷ ImageCorner (output_object) multichannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / real Result of the filtering. Number of elements : ImageCorner = Image
▷ Size (input_control) integer $\leadsto$ <i>integer</i> Desired filtersize of the graymask. Default Value : 3 List of values : Size $\in$ {3, 5, 7, 9, 11}

- ▷ **Weight** (input_control) real $\leadsto$ real
 Weighting.
Default Value : 0.04
Typical range of values : $0.0 \leq \text{Weight} \leq 0.3$
Minimum Increment : 0.001
Recommended Increment : 0.01

Example

```
read_image(&Fabrik, "fabrik");
corner_response(Fabrik, &CornerResponse, 3, 0.04);
local_max(CornerResponse, &LocalMax);
disp_image(Fabrik, WindowHandle);
set_color(WindowHandle, "red");
disp_region(LocalMax, WindowHandle);
```

Parallelization Information

corner_response is reentrant and automatically parallelized (on tuple level, channel level, domain level).

Possible Successors

local_max, threshold

See also

gauss_image, sobel_amp, convert_image_type

References

C.G. Harris, M.J. Stephens, "A combined corner and edge detector"; Proc. of the 4th Alvey Vision Conference; August 1988; pp. 147-152.

H. Breit, "Bestimmung der Kameraeigenbewegung und Gewinnung von Tiefendaten aus monokularen Bildfolgen"; Diplomarbeit am Lehrstuhl für Nachrichtentechnik der TU München; 30. September 1990.

Module

Image filters

exhaustive_match (Image, RegionOfInterest,
 ImageTemplate : ImageMatch : Mode :)

Matching of a template and an image.

The operator **exhaustive_match** matches **ImageTemplate** and **Image** within the region of interest **RegionOfInterest**. Hereby the **ImageTemplate** will be moved over all points of **Image** which lie within the **RegionOfInterest**. With regard to the parameter **Mode**, a matching criterion will be calculated. The result values will be stored in **ImageMatch**.

The following matching criteria (**Mode**) are available:

'norm_correlation'

$$\text{ImageMatch}[i][j] = 255 \cdot \frac{\sum_{u,v} (\text{Image}[i-u][j-v] \cdot \text{ImageTemplate}[l-u][c-v])}{\sqrt{\sum_{u,v} (\text{Image}[i-u][j-v]^2) \cdot \sum_{u,v} (\text{ImageTemplate}[l-u][c-v]^2)}}$$

whereby $X[i][j]$ indicates the grayvalue in the i th column and j th row of the image X . (l, c) is the centre of the region of **ImageTemplate**. u and v are chosen so that all points of the template will be reached, i, j run across the **RegionOfInterest**. At the image frame only those parts of **ImageTemplate** will be considered which lie inside the image (i.e. u and v will be restricted correspondingly). Range of values: 0 - 255 (best fit).

'dfd' Calculating the average "displaced frame difference":

$$\text{ImageMatch}[i][j] = \frac{\sum_{u,v} |\text{Image}[i-u][j-v] - \text{ImageTemplate}[l-u][c-v]|}{\text{AREA}(\text{ImageTemplate})}$$

The terms are the same as in 'norm_correlation'. $\text{AREA}(X)$ means the area of the region X . Range of value 0 (best fit) - 255.

To calculate the normalized correlation as well as the “displaced frame difference” is (with regard to the area of [ImageTemplate](#)) very time consuming. Therefore it is important to restrict the input region ([RegionOfInterest](#) if possible, i.e. to apply the filter only in a very confined “region of interest”).

As far as quality is concerned, both modes return comparable results, whereby the mode ‘dfd’ is faster by a factor of about 3.5.

Parameter

- ▷ **Image** (input_object) image $\leadsto$ Hobject : byte
Input image.
- ▷ **RegionOfInterest** (input_object) region $\leadsto$ Hobject
Area to be searched in the input image.
- ▷ **ImageTemplate** (input_object) image $\leadsto$ Hobject : byte
This area will be “matched” by [Image](#) within the [RegionOfInterest](#).
- ▷ **ImageMatch** (output_object) image $\leadsto$ Hobject : byte
Result image: values of the matching criterion.
- ▷ **Mode** (input_control) string $\leadsto$ string
Desired matching criterion.
Default Value : ‘dfd’
List of values : Mode $\in$ {‘norm_correlation’, ‘dfd’}

Example

```
read_image(Image, 'monkey')
disp_image(Image, WindowHandle)
draw_rectangle2(WindowHandle, Row, Column, Phi, Length1, Length2)
gen_rectangle2(Rectangle, Row, Column, Phi, Length1, Length2)
reduce_domain(Image, Rectangle, Template)
exhaustive_match(Image, Image, Template, ImageMatch, 'dfd')
invert_image(ImageMatch, ImageInvert)
local_max(Image, Maxima)
union1(Maxima, AllMaxima)
add_channels(AllMaxima, ImageInvert, FitMaxima)
threshold(FitMaxima, BestFit, 230.0, 255.0)
disp_region(BestFit, WindowHandle).
```

Result

If the parameter values are correct, the operator [exhaustive_match](#) returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behaviour can be set via [set_system](#) (‘no_object_result’, <Result>). If necessary, an exception handling is raised.

Parallelization Information

[exhaustive_match](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[draw_region](#), [draw_rectangle1](#)

Possible Successors

[local_max](#), [threshold](#)

Alternatives

[exhaustive_match_mg](#)

Module

Image filters

exhaustive_match_mg (Image, ImageTemplate : ImageMatch : Mode,
Level, Threshold :)

Matching a template and an image in a resolution pyramid.

The operator [exhaustive_match_mg](#) is an additional option for the operator [exhaustive_match](#) performing a matching of the image [Image](#) and the template [ImageTemplate](#). Hereby [ImageTemplate](#) will be

moved over all points of the region of **Image**, a matching criterion will be calculated with regard to the parameter **Mode** and the result values will be stored in **ImageMatch**.

Of images having been filtered this way, normally only those areas with good matching results are of interest. The size of the area to be searched, i.e. the region of the input image **Image**, determines decisively the runtime of the matching filter. Therefore it is recommendable to use at first **exhaustive_match_mg** with reduced image resolution in order to determine a “region of interest” in which good matching results can be expected; then in this restricted area only the real matching (see also **exhaustive_match**) will be executed with normal resolution.

Hereby the Gauss-pyramids of **Image** and **ImageTemplate** will be composed (in particular the corresponding regions will be transformed as well). Then on each level of the resolution pyramids - starting with the startlevel **Level** - the matching inside the current “region of interest” will be executed. Whereby the “region of interest” on the startlevel is equivalent to the region of the input image **Image**. After the filtering, a new “region of interest” is determined with the help of a threshold operation and will be transformed on the next resolution level:

```
threshold(..0,Threshold..), if Mode = 'dfd'
threshold(..Threshold,255..), if Mode = 'crosscorrelation'
```

The final matching in the determined “region of interest” will then be calculated with the highest resolution (**Level** 0). The output image **ImageMatch** includes the corresponding filter result and the final “region of interest”, which is determined on the result image with the help of a threshold operation.

The operator **exhaustive_match_mg** therefore is not simply a filter, but can also be considered as a member of the class of region transformations.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte
Input image.
- ▷ **ImageTemplate** (input_object) image $\leadsto$ *Hobject* : byte
The domain of this image will be matched with **Image**.
- ▷ **ImageMatch** (output_object) image(-array) $\leadsto$ *Hobject* : byte
Result image and result region: values of the matching criterion within the determined “region of interest”.
Number of elements : ImageMatch = Image
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Desired matching criterion.
Default Value : 'dfd'
List of values : Mode $\in$ { 'crosscorrelation', 'dfd' }
- ▷ **Level** (input_control) integer $\leadsto$ *integer*
Startlevel in the resolution pyramid (highest resolution: Level 0).
Default Value : 1
List of values : Level $\in$ { 0, 1, 2, 3, 4, 5, 6, 7, 8 }
Restriction : (Level < Id(width(Image))) $\wedge$ (Level < Id(height(Image)))
- ▷ **Threshold** (input_control) integer $\leadsto$ *integer*
Threshold to determine the “region of interest”.
Default Value : 30
Suggested values : Threshold $\in$ { 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95, 100, 105, 110, 115, 120, 125, 130, 135, 140, 145, 150, 155, 160, 165, 170, 175, 180, 185, 190, 195, 200, 205, 210, 215, 220, 225, 230, 235, 240, 245, 250 }
Typical range of values : $0 \leq \text{Threshold} \leq 255$
Minimum Increment : 1
Recommended Increment : 5

Example

```
read_image(&Image, "monkey");
disp_image(Image, WindowHandle);
draw_rectangle2(WindowHandle, &Row, &Column, &Phi, &Length1, &Length2);
gen_rectangle2(&Rectangle, Row, Column, Phi, Length1, Length2);
reduce_domain(Image, Rectangle, &Template);
exhaustive_match_mg(Image, Template, &ImageMatch, 'dfd', 1, 30);
invert_image(ImageMatch, &ImageInvert);
local_max(ImageInvert, &BestFit);
disp_region(BestFit, WindowHandle);
```

Result

If the parameter values are correct, the operator `exhaustive_match_img` returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behaviour can be set via `set_system('no_object_result', <Result>)`. If necessary, an exception handling is raised.

Parallelization Information

`exhaustive_match_img` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`draw_region`, `draw_rectangle1`

Possible Successors

`threshold`, `local_max`

Alternatives

`exhaustive_match`

See also

`gen_gauss_pyramid`

Module

Image filters

fill_dvf (Image : ImageExpanded : Width, Height, Iterations :)

Expansion of a displacement vector field onto undefined areas.

The operator `fill_dvf` calculates a value for each undefined point in a displacement vector field by averaging its neighbors. The dimension of the neighborhood will be determined by the parameters `Width` and `Height`. As the gaps may be larger than the masks, the filter may be used iteratively (`Iterations`). Correctly defined values in the input image will not be modified.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : dvf
Input image.
- ▷ **ImageExpanded** (output_object) image(-array) $\leadsto$ *Hobject* : dvf
Result of adding the points which are missing.
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer*
Width of the filtermask.
Default Value : 5
Suggested values : $\text{Width} \in \{3, 5, 7, 9, 11, 15\}$
Typical range of values : $3 \leq \text{Width} \leq 201$
Minimum Increment : 2
Recommended Increment : 2
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer*
Height of the filtermask.
Default Value : 5
Suggested values : $\text{Height} \in \{3, 5, 7, 9, 11, 13, 15\}$
Typical range of values : $3 \leq \text{Height} \leq 201$
Minimum Increment : 2
Recommended Increment : 2
- ▷ **Iterations** (input_control) integer $\leadsto$ *integer*
Number of iterations.
Default Value : 3
Suggested values : $\text{Iterations} \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15\}$
Typical range of values : $1 \leq \text{Iterations} \leq 100$
Minimum Increment : 1
Recommended Increment : 1

Parallelization Information

`fill_dvf` is *reentrant* and automatically *parallelized* (on *tuple level*, *domain level*).

Possible Predecessors

[optical_flow_match](#)

Module

Image filters

gen_gauss_pyramid (Image : ImagePyramid : Mode, Scale :)

Calculating a Gauss pyramid.

The operator [gen_gauss_pyramid](#) calculates a pyramid of scaled down images. The scale by which the next image will be reduced is determined by the parameter [Scale](#). For instance, a value of 0.5 for [Scale](#) will shorten the edge length of [Image](#) by 50%. This is exactly equivalent to the “normal” pyramid.

The parameter [Mode](#) determines the way of averaging. For a more detailed description concerning this parameter see also [affine_trans_image](#). In the case that [Scale](#) is equal 0.5 there are the additional modes ‘min’ and ‘max’ available. In this case the minimum or the maximum of the four neighboring pixels is selected.

Please note that each level will be returned as an individual image, i.e. as one iconic objekt, with one matrix and its own domain. If a multichannel image is needed as a result, the operator [image_to_channels](#) has to be used. A single level or more than one level can be selected by using [select_obj](#) respectively [copy_obj](#).

Parameter

- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte / uint2 / real
Input image.
- ▷ **ImagePyramid** (output_object) image-array $\leadsto$ *Hobject* : byte / uint2 / real
Output images.
Number of elements : ImagePyramid > Image
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Kind of filtermask.
Default Value : ‘weighted’
List of values : Mode $\in$ { ‘none’, ‘constant’, ‘weighted’, ‘min’, ‘max’ }
- ▷ **Scale** (input_control) real $\leadsto$ *real*
Factor for scaling down.
Default Value : 0.5
Suggested values : Scale $\in$ { 0.2, 0.3, 0.4, 0.5, 0.6 }
Typical range of values : $0.1 \leq \text{Scale} \leq 0.9$
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : $(0.1 < \text{Scale}) \wedge (\text{Scale} < 0.9)$

Example

```
gen_gauss_pyramid(Image,Pyramid,"weighted",0.5);
count_obj(Pyramid,&num);
for (i=1; i<=num; i++)
{
    select_obj(Pyramid,&Single,i);
    disp_image(Single,windowHandle);
    clear(Single);
}
```

Parallelization Information

[gen_gauss_pyramid](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

[image_to_channels](#), [count_obj](#), [select_obj](#), [copy_obj](#)

Alternatives

[zoom_image_size](#), [zoom_image_factor](#)

See also

[affine_trans_image](#)

Module

Image filters

monotony (Image : ImageMonotony : :)

Calculating the monotony operation.

The operator [monotony](#) calculates the monotony operator. Thereby the points which are strictly smaller than the current grayvalue will be counted in the 8 neighborhood. This number will be entered into the output imaged.

If there is a strict maximum, the value 8 is returned; in case of a minimum or a plateau, the value 0 will be returned. A ridge or a slope will return the corresponding intermediate values.

The monotony operator is often used to prepare matching operations as it is invariant with regard to lightness modifications.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Input image.
- ▷ **ImageMonotony** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Result of the monotony operator.
- Number of elements** : ImageMonotony = Image

Example

```
/* searching the strict maximums */
gauss_image(Image,&Gauss,5);
monotony(Gauss,&Monotony);
threshold(Monotony,Maxima,8.0,8.0);
```

Parallelization Information

[monotony](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Predecessors

[gauss_image](#), [median_image](#), [mean_image](#), [smooth_image](#), [invert_image](#)

Possible Successors

[threshold](#), [exhaustive_match](#), [disp_image](#)

Alternatives

[local_max](#), [topographic_sketch](#), [corner_response](#)

Module

Image filters

optical_flow_match (Image1, Image2 : VectorField : Mode, Threshold, Step, SizeWindow, SizeSearch, Weights :)

Calculating the Displacement vector field by correlation methods.

The operator [optical_flow_match](#) determines a displacement vector field (DVF) with the help of correlation methods. Simple- and multichannel Images may be used. A window of the size [SizeWindow](#) will be moved inside a mask of the size [SizeSearch](#) over a second image and there the grayvalues will be compared with the ones from the first image. The correlation showing the slightest error will determine the vector. In case of two vectors having the same weighting, the shorter vector will be preferred. After having processed one image point, the next image point at a distance of [Step](#) will be selected. It is therefore not guaranteed to get a dense displacement vector field.

The parameter [Threshold](#) specifies the maximal divergence of previously estimated correlations and the best estimation of a displacement vector.

The method used by the operator is determined by the parameter **Mode**. The following values for **Mode** can be selected:

'dfd' Calculation of displaced frame difference.

'dfd_norm' Calculation of displaced frame difference normalized by the mean value of the matching windows.

If the image has more than one channel the parameter **Weights** indicates the weighting of the individual channels. For each channel one weight factor must be passed.

For the parameters **SizeWindow**, **SizeSearch** and **Step** optionally two values can be passed: The first value indicates hereby the width (respectively the row), the second value the height (respectively the column).

Parameter	
▷ Image1 (input_object)	image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic First input image (optionally with more than one channel).
▷ Image2 (input_object)	image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic Second input image (optionally with more than one channel). Number of elements : Image2 = Image1
▷ VectorField (output_object)	image(-array) $\leadsto$ <i>Hobject</i> : dvf Displacement vector field to be calculated. Number of elements : VectorField = Image1
▷ Mode (input_control)	string $\leadsto$ <i>string</i> Kind of correlation. Default Value : 'dfd' List of values : Mode $\in$ {'dfd', 'dfd_norm'}
▷ Threshold (input_control)	integer $\leadsto$ <i>integer</i> Maximal divergence of the previously estimated correlations and the best estimate of a displacement vector. Default Value : 10 Suggested values : Threshold $\in$ {1, 2, 3, 4, 5, 7, 10, 12, 15, 17, 20} Typical range of values : $1 \leq \text{Threshold} \leq 300$ Minimum Increment : 1 Recommended Increment : 5
▷ Step (input_control)	integer(-array) $\leadsto$ <i>integer</i> Step width. Default Value : 10 Suggested values : Step $\in$ {1, 2, 3, 5, 7, 10, 15, 20, 30, 50} Typical range of values : $1 \leq \text{Step} \leq 300$ Minimum Increment : 1 Recommended Increment : 5
▷ SizeWindow (input_control)	integer(-array) $\leadsto$ <i>integer</i> Size of the correlation window. Default Value : 11 Suggested values : SizeWindow $\in$ {3, 5, 7, 9, 11, 15, 20, 30} Typical range of values : $1 \leq \text{SizeWindow} \leq 30$ Minimum Increment : 1 Recommended Increment : 2
▷ SizeSearch (input_control)	integer(-array) $\leadsto$ <i>integer</i> Size of the area to be searched. Default Value : 20 Suggested values : SizeSearch $\in$ {3, 5, 7, 10, 20, 30, 40, 50, 60} Typical range of values : $1 \leq \text{SizeSearch} \leq 60$ Minimum Increment : 1 Recommended Increment : 2
▷ Weights (input_control)	integer(-array) $\leadsto$ <i>integer</i> Weighting of the channels. Default Value : 1 Suggested values : Weights $\in$ {1, 3, 5, 7, 10} Typical range of values : $1 \leq \text{Weights} \leq 10$ Minimum Increment : 1 Recommended Increment : 1

Example

```

read_image(B1, 'image1')
mean_image(B1, L1, 16, 16)
read_image(B2, 'image2')
mean(B2, L2, 16, 16)
optical_flow_match(L1, L2, VVF, 'dfd', 10, 10, 11, 20, 1)
disp_image(B1, WindowHandle)
set_color(WindowHandle, 'red')
disp_image(VVF, WindowHandle).

```

Parallelization Information

`optical_flow_match` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`mean_image`, `gauss_image`, `smooth_image`

Possible Successors

`fill_dvf`

See also

`dvf_to_int1`, `dvf_to_hom_mat2d`

Module

Image filters

5.10 Misc

convol_image (Image : ImageResult : FilterMask, Margin :)

Convolve an image with an arbitrary filter mask.

`convol_image` convolves the input image `Image` with an arbitrary linear filter. The corresponding filter mask, which is given in `FilterMask` can be generated either from a file or a tuple. Several options for the treatment at the image's borders can be chosen (`Margin`):

gray value	Pixels outside of the image edges are assumed to be constant (with the indicated gray value).
'continued'	Continuation of edge pixels.
'cyclic'	Cyclic continuation of image edges.
'mirrored'	Reflection of pixels at the image edges.

All image points are convolved with the filter mask. If an overflow or underflow occurs, the resulting gray value is clipped. Hence, if filters that result in negative output values are used (e.g., derivative filters) the input image should be of type `int2`. If a filename is given in `FilterMask` the filter mask is read from a text file with the following structure:

```

<Mask size>
<Inverse weight of the mask>
<Matrix>

```

The first line contains the size of the filter mask, given as two numbers separated by white space (e.g., 3 3 for 3×3). Here, the first number defines the height of the filter mask, while the second number defines its width. The next line contains the inverse weight of the mask, i.e., the number by which the convolution of a particular image point is divided. The remaining lines contain the filter mask as integer numbers (separated by white space), one line of the mask per line in the file. The file must have the extension ".fil". This extension must not be passed to the operator. If the filter mask is to be computed from a tuple, the tuple given in `FilterMask` must also satisfy the structure described above. However, in this case the line feed is omitted.

<i>Parameter</i>	
▷ Image (input_object)	(multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / uint2 Image to be convolved.
▷ ImageResult (output_object)	multichannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / uint2 Convolved result image.
▷ FilterMask (input_control)	string(-array) $\leadsto$ <i>string</i> / integer Filter mask as file name or tuple. Default Value : 'sobel' Suggested values : FilterMask $\in$ {'sobel', 'laplace4', 'lowpas.3.3'}
▷ Margin (input_control)	string $\leadsto$ <i>string</i> / integer / real Border treatment. Default Value : 'mirrored' Suggested values : Margin $\in$ {'mirrored', 'cyclic', 'continued', 0, 30, 60, 90, 120, 150, 180, 210, 240, 255}

Parallelization Information

`convol_image` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*).

Module

Image filters

expand_domain_gray (InputImage : ExpandedImage : ExpansionRange :)

Expand the domain of an image and set the gray values in the expanded domain.

`expand_domain_gray` expands the border gray values of the domain outwards. The width of the expansion is set by the parameter `ExpansionRange`. All filters in HALCON use gray values of the pixels outside the domain depending on the filter width. This may lead to undesirable side effects especially in the border region of the domain. For example, if the foreground (domain) and the background of the image differ strongly in brightness, the result of a filter operation may lead to undesired darkening or brightening at the border of the domain. In order to avoid this drawback, the domain is expanded by `expand_domain_gray` in a preliminary stage, copying the gray values of the border pixels to the outside of the domain. In addition, the domain itself is also expanded to reflect the newly set pixels. Therefore, in many cases it is reasonable to reduce the domain again (`reduce_domain` or `change_domain`) after using `expand_domain_gray` and call the filter operation afterwards. `ExpansionRange` should be set to the half of the filter width.

<i>Parameter</i>	
▷ InputImage (input_object)	image(-array) $\leadsto$ <i>Hobject</i> : byte / int1 / int2 / uint2 / int4 / real Input image with domain to be expanded.
▷ ExpandedImage (output_object)	image(-array) $\leadsto$ <i>Hobject</i> : byte / int1 / int2 / uint2 / int4 / real Output image with new gray values in the expanded domain.
▷ ExpansionRange (input_control)	integer $\leadsto$ <i>integer</i> Radius of the gray value expansion, measured in pixels. Default Value : 2 Suggested values : ExpansionRange $\in$ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 16} Restriction : ExpansionRange $\geq$ 1

Example

```
read_image(Fabrik, 'fabrik.tif');
gen_rectangle2(Rectangle_Label, 243, 320, -1.55, 62, 28);
reduce_domain(Fabrik, Rectangle_Label, Fabrik_Label);
/* Character extraction without gray value expansion: */
mean_image(Fabrik_Label, Label_Mean_normal, 31, 31);
dyn_threshold(Fabrik_Label, Label_Mean_normal, Characters_normal, 10, 'dark');
dev_display(Fabrik);
dev_display(Characters_normal);
/* The characters in the border region are not extracted ! */
stop();
```



```

/* Character extraction with gray value expansion: */
expand_domain_gray(Fabrik_Label, Label_expanded,15);
reduce_domain(Label_expanded,Rectangle_Label, Label_expanded_reduced);
mean_image(Label_expanded_reduced,Label_Mean_expanded,31,31);
dyn_threshold(Fabrik_Label,Label_Mean_expanded,Characters_expanded,10,'dark');
dev_display(Fabrik);
dev_display(Characters_expanded);
/* Now, even in the border region the characters are recognized */

```

Complexity

Let L the perimeter of the domain. Then the runtime complexity is approximately $O(L) * \text{ExpansionRange}$.

Result

[expand_domain_gray](#) returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception handling is raised.

Parallelization Information

[expand_domain_gray](#) is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

[reduce_domain](#)

Possible Successors

[reduce_domain](#), [mean_image](#), [dyn_threshold](#)

See also

[reduce_domain](#), [mean_image](#)

Module

Image filters

gray_inside (Image : ImageDist : :)

Calculate the lowest possible gray value on an arbitrary path to the image border for each point in the image.

[gray_inside](#) determines the “cheapest” path to the image border for each point in the image, i.e., the path on which the lowest gray values have to be overcome. The resulting image contains the difference of the gray value of the particular point and the maximum gray value on the path. Bright areas in the result image therefore signify that these areas (which are typically dark in the original image) are surrounded by bright areas. Dark areas in the result image signify that there are only small gray value differences between them and the image border (which doesn’t mean that they are surrounded by dark areas; a small “gap” of dark values suffices). The value 0 (black) in the result image signifies that only darker or equally bright pixels exist on the path to the image border.

The operator is implemented by first segmenting into basins and watersheds the image using the [watersheds](#) operator. If the image is regarded as a gray value mountain range, basins are the places where water accumulates and the mountain ridges are the watersheds. Then, the watersheds are distributed to adjacent basins, thus leaving only basins. The border of the domain (region) of the original image is now searched for the lowest gray value, and the region in which it resides is given its result values. If the lowest gray value resides on the image border, all result values can be calculated immediately using the gray value differences to the darkest point. If the smallest found gray value lies in the interior of a basin, the lowest possible gray value has to be determined from the already processed adjacent basins in order to compute the new values. An 8-neighborhood is used to determine adjacency. The found region is subtracted from the regions yet to process, and the whole process is repeated. Thus, the image is “stripped” from the outside.

Analogously to [watersheds](#), it is advisable to apply a smoothing operation before calling [watersheds](#), e.g., [gauss_image](#), in order to reduce the amount of regions that result from the watershed algorithm, and thus to speed up the processing time.

Parameter

- ▷ **Image** (input_object)(multichannel-)image(-array) $\leadsto$ *Hobject* : byte
Image being processed.
- ▷ **ImageDist** (output_object)(multichannel-)image(-array) $\leadsto$ *Hobject* : int2
Result image.

Example

```
read_image(Bild, 'coin')
gauss_image (Bild, G_Bild, 11)
open_window (0, 0, 512, 512, 0, 'visible', '', WindowHandle)
gray_inside(G_Bild, Ausgabebild)
disp_image (Ausgabebild, WindowHandle).
```

Result

[gray_inside](#) always returns 2 (H_MSG_TRUE).

Parallelization Information

[gray_inside](#) is *reentrant* and automatically *parallelized* (on tuple level, channel level).

Possible Predecessors

[gauss_image](#), [smooth_image](#), [mean_image](#), [median_image](#)

Possible Successors

[select_shape](#), [area_center](#), [count_obj](#)

See also

[watersheds](#)

Module

Image filters

gray_skeleton (Image : GraySkeleton : :)

Thinning of gray value images.

[gray_skeleton](#) applies a gray value thinning operation to the input image [Image](#). Figuratively, the gray value “mountain range” is reduced to its ridge lines by setting the gray value of “hillsides” to the gray value at the corresponding valley bottom. The resulting ridge lines are at most two pixels wide. This operator is especially useful for thinning edge images, and is thus an alternative to [nonmax_suppression_amp](#). In contrast to [nonmax_suppression_amp](#), [gray_skeleton](#) preserves contours, but is much slower. In contrast to [skeleton](#), this operator changes the gray values of an image while leaving its region unchanged.

Parameter

- ▷ **Image** (input_object)(multichannel-)image(-array) $\leadsto$ *Hobject* : byte
Image to be thinned.
- ▷ **GraySkeleton** (output_object)(multichannel-)image(-array) $\leadsto$ *Hobject* : byte
Thinned image.

Example

```
/* Seeking leafs of a beech tree in an aerial picture: */
read_image(Image, 'wald1')
gray_skeleton(Image, Skelett)
mean_image(Skelett, MeanSkelett, 7, 7)
dyn_threshold(Skelett, MeanSkelett, Leafs, 3, 'light').
```

Result

[gray_skeleton](#) returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via [set_system\('no_object_result', <Result>\)](#). If necessary, an exception is raised.

Parallelization Information

[gray_skeleton](#) is *reentrant* and automatically *parallelized* (on tuple level, channel level).

Possible Successors

[mean_image](#)

Alternatives

[nonmax_suppression_amp](#), [nonmax_suppression_dir](#), [local_max](#)

See also

[skeleton](#), [gray_dilation_rect](#)

Module

Image filters

lut_trans (Image : ImageResult : Lut :)

Transform an image with a gray-value look-up-table

lut_trans transforms an image **Image** by using a gray value look-up-table **Lut**. This table acts as a transformation function. In the case of byte-images, **Lut** has to be a tuple of length 256. In the case of int2-images, **Lut** has to be a tuple of length 256 $\neq$ length $\neq$ 65536. If the length of the **Lut** is $\neq$ 32768, the transformation is applied to the positive gray values only, i.e., the first element of the **Lut** specifies the new gray value for the gray value 0. If the **Lut** is longer than 32768, exactly 65536 must be passed. In this case, the positive and negative gray values are transformed. In this case, the first element indicates the new gray value for the gray value -32768 of the input image, while the last element of the tuple indicates the new gray value for the gray value 32767. In all cases, the gray values of values outside the range of **Lut** are set to 0. In the case of uint2-images, **Lut** has to be a tuple of length 256 $\neq$ length $\neq$ 65536. Gray values outside the range of **Lut** are set to 0.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Image whose gray values are to be transformed.
- ▷ **ImageResult** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Transformed image.
- ▷ **Lut** (input_control) integer-array $\leadsto$ *integer*
Table containing the transformation.

Example

```
/* To get the inverse of an image: */
read_image(Image, 'wald1')
def_tab(Tab, 0)
lut_trans(Image, Invers, Tab, 1, 1)

def_tab(Tab, I) :- I=255
                  Tab = 0
def_tab([Tk|Ts], I) :-
    Tk is 255 - I
    Iw is I - 1
    def_tab(Ts, Iw)
```

Result

The operator **lut_trans** returns the value 2 (H_MSG_TRUE) if the parameters are correct. Otherwise an exception is raised.

Parallelization Information

lut_trans is *reentrant* and automatically *parallelized* (on tuple level, channel level, domain level).

Module

Image filters

map_image (Image, Map : ImageMapped : :)

Apply a general transformation to an image.

map_image transforms an image **Image** using an arbitrary transformation **Map**, which, for example, was previously generated using [gen_image_to_world_plane_map](#) or [gen_radial_distortion_map](#). The multi-channel image **Map** must be organized as follows:

The height and the width of **Map** define the size of the output image **ImageMapped**. The number of channels in **Map** defines whether no interpolation or bilinear interpolation should be used. If **Map** only consists of one channel, no interpolation is applied during the transformation. This channel contains 'int4' values that describe the geometric transformation: For each pixel in the output image **ImageMapped** the linearized coordinate of the pixel in the input image **Image** from which the gray value should be taken is stored.

If bilinear interpolation between the pixels in the input image should be applied, **Map** must consist of 5 channels. The first channel again consists of an 'int4' image and describes the geometric transformation. The channels 2-5 consist of an 'uint2' image each and contain the weights [0...1] of the four neighboring pixels that are used during bilinear interpolation. If the overall brightness of the output image **ImageMapped** should not differ from the overall brightness of the input image **Image**, the sum of the four unscaled weights must be 1 for each pixel. The weights [0...1] are scaled to the range of values of the 'uint2' image and therefore hold integer values from 0 bis 65535.

Furthermore, the weights must be chosen in a way that the range of values of the output image **ImageMapped** is not exceeded. The geometric relation between the four channels 2-5 is illustrated in the following sketch:

2	3
4	5

The reference point of the four pixels is the upper left pixel. The linearized coordinate of the reference point is stored in the first channel.

Attention

The weights must be chosen in a way that the range of values of the output image **ImageMapped** is not exceeded.

For runtime reasons during the mapping process, it is not checked whether the linearized coordinates, which are stored in the first channel of **Map**, lie inside the input image. Thus, it must be ensured by the user that this constraint is fulfilled. Otherwise, the program may crash!

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / uint2
Image to be mapped.
- ▷ **Map** (input_object) (multichannel-)image $\leadsto$ *Hobject* : int4 / uint2
Image containing the mapping data.
- ▷ **ImageMapped** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / uint2
Mapped image.

Result

map_image returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception handling is raised.

Parallelization Information

map_image is *reentrant* and processed *without* parallelization.

Possible Predecessors

gen_image_to_world_plane_map, **gen_radial_distortion_map**

See also

affine_trans_image, **rotate_image**

Module

Image filters

principal_comp (MultichannelImage : PCAImage : : InfoPerComp)

Compute the principal components of multi-channel images.

principal_comp does a principal components analysis of multi-channel images. This is useful for images obtained, e.g., with the thematic mapper of the Landsat satellite. Because the spectral bands are highly correlated it is desirable to transform them to uncorrelated images. This can be used to save storage, since the bands containing little information can be discarded, and with respect to a later classification step.

The operator **principal_comp** takes a (multi-channel) image **MultichannelImage** and transforms it to the output image **PCAImage**, which contains the same number of channels, using the principal components analysis. The parameter **InfoPerComp** contains the relative information content of each output channel.

Parameter	
▷ MultichannelImage (input_object)	multichannel-image $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Multi-channel input image.	
▷ PCAIImage (output_object)	multichannel-image $\leadsto$ <i>Hobject</i> : real
Multi-channel output image.	
▷ InfoPerComp (output_control)	real(-array) $\leadsto$ <i>real</i>
Information content of each output channel.	

Result

The operator **principal_comp** returns the value 2 (H_MSG_TRUE) if the parameters are correct. Otherwise an exception is raised.

Parallelization Information

principal_comp is *reentrant* and processed *without* parallelization.

Module

Image filters

symmetry (Image : ImageSymmetry : MaskSize, Direction, Exponent :)

Symmetry of gray values along a row.

symmetry calculates the symmetry along a line. For each pixel the gray values of both sides of the line are compared: The absolute value of the differences of gray values with same distance to the pixel is computed. Each of these differences is weighted by the exponent (after division by 255) and the summed up.

$$sym := 255 - \frac{255}{\text{MaskSize}} \sum_{i=1}^{\text{MaskSize}} \left(\frac{|g(i) - g(-i)|}{255} \right)^{\text{Exponent}}$$

Pixels with a high symmetry have large gray values.

Attention

Currently only horizontal search lines are implemented

Parameter	
▷ Image (input_object)	(multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte
Input image.	
▷ ImageSymmetry (output_object)	(multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte
Symmetry image.	
▷ MaskSize (input_control)	number $\leadsto$ <i>integer</i>
Extension of search area.	
Default Value : 40	
Suggested values : MaskSize $\in \{3, 5, 7, 10, 15, 20, 25, 30, 40, 50, 60, 70, 80, 100, 120, 140, 180\}$	
Typical range of values : $3 \leq \text{MaskSize} \leq 1000$	
Minimum Increment : 1	
Recommended Increment : 2	
▷ Direction (input_control)	number $\leadsto$ <i>real</i>
Angle of test direction.	
Default Value : 0.0	
Suggested values : Direction $\in \{0.0\}$	
Typical range of values : $0.0 \leq \text{Direction} \leq 0.0$	
▷ Exponent (input_control)	number $\leadsto$ <i>real</i>
Exponent for weighting.	
Default Value : 0.5	
Suggested values : Exponent $\in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.7, 0.8, 0.9, 1.0\}$	
Typical range of values : $0.05 \leq \text{Exponent} \leq 1.0$	
Minimum Increment : 0.01	
Recommended Increment : 0.1	
Restriction : $(0 < \text{Exponent}) \wedge (\text{Exponent} \leq 1)$	

Example

```
read_image(Image, 'monkey')
symmetry(Image, ImageSymmetry, 70, 0.0, 0.5)
threshold(ImageSymmetry, SymmPoints, 170, 255)
```

Result

If the parameter values are correct the operator `symmetry` returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator `set_system (::'no_object_result', <Result>:)`. If necessary an exception handling is raised.

Parallelization Information

`symmetry` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Successors

`threshold`

Module

Image filters

topographic_sketch (Image : Sketch : :)

Compute the topographic primal sketch of an image.

`topographic_sketch` computes the topographic primal sketch of the input image `Image`. This is done by approximating the image locally by a bicubic polynomial ("facet model"). It serves to calculate the first and second partial derivatives of the image, and thus to classify the image into 11 classes. These classes are coded in the output image `Sketch` as numbers from 1 to 11. The classes are as follows:

Peak	1
Pit	2
Ridge	3
Ravine	4
Saddle	5
Flat	6
Hillside Slope	7
Hillside Convex	8
Hillside Concave	9
Hillside Saddle	10
Hillside Inflection	11

In order to obtain the separate classes as regions, a threshold operation has to be applied to the result image with the appropriate thresholds.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte
Image for which the topographic primal sketch is to be computed.
- ▷ **Sketch** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte
Label image containing the 11 classes.

Example

```
/* To extract the Ridges from a Image */
read_image(Image, 'sinus')
topographic_sketch(Image, Sketch)
threshold(Sketch, Ridges, 3, 3).
```

Complexity

Let n be the number of pixels in the image. Then $O(n)$ operations are performed.

Result

`topographic_sketch` returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via `set_system('no_object_result', <Result>:)`. If necessary, an exception is raised.

Parallelization Information

[topographic_sketch](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*).

Possible Successors

[threshold](#)

References

R. Haralick, L. Shapiro: "Computer and Robot Vision, Volume I"; Reading, Massachusetts, Addison-Wesley; 1992; Kapitel 8.13.

Module

Image filters

5.11 Noise

add_noise_distribution (Image : ImageNoise : Distribution :)

Add noise to an image.

[add_noise_distribution](#) adds noise distributed according to [Distribution](#) to the image [Image](#). Resulting gray values are clipped to the range [0,255].

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2
Input image.
- ▷ **ImageNoise** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2
Noisy image.
Number of elements : ImageNoise = Image
- ▷ **Distribution** (input_control) distribution.values-array $\leadsto$ *real*
Noise distribution.
Number of elements : 513

Example

```

read_image(Image, 'meer_rot')
disp_image(Image, WindowHandle)
sp_distribution(30, 30, Dist)
add_noise_distribution(Image, ImageNoise, Dist)
disp_image(ImageNoise, WindowHandle).

```

Result

[add_noise_distribution](#) returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via [set_system\('no_object_result', <Result>\)](#). If necessary, an exception handling is raised.

Parallelization Information

[add_noise_distribution](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Predecessors

[gauss_distribution](#), [sp_distribution](#), [noise_distribution_mean](#)

Alternatives

[add_noise_white](#)

See also

[sp_distribution](#), [gauss_distribution](#), [noise_distribution_mean](#), [add_noise_white](#)

Module

Image filters

add_noise_white (Image : ImageNoise : Amp :)

Add noise to an image.

add_noise_white adds noise to the image **Image**. The noise is white noise, equally distributed in the interval $[-\text{Amp}, \text{Amp}]$, and is generated by using the C function “drand48” with an initial time dependent seed. Resulting gray values are clipped to the range $[0, 255]$.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Input image.
- ▷ **ImageNoise** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Noisy image.
Number of elements : ImageNoise = Image
- ▷ **Amp** (input_control) real $\leadsto$ *real*
Maximum noise amplitude.
Default Value : 60.0
Suggested values : $\text{Amp} \in \{1.0, 2.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0\}$
Typical range of values : $1.0 \leq \text{Amp} \leq 1000.0$
Minimum Increment : 1.0
Recommended Increment : 10.0
Restriction : $\text{Amp} > 0$

Example

```
read_image(Image, 'fabrik')
disp_image(Image, WindowHandle)
add_noise_white(Image, ImageNoise, 90)
disp_image(ImageNoise, WindowHandle).
```

Result

add_noise_white returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via **set_system('no_object_result', <Result>)**. If necessary, an exception handling is raised.

Parallelization Information

add_noise_white is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Alternatives

add_noise_distribution

See also

add_noise_distribution, **noise_distribution_mean**, **gauss_distribution**,
sp_distribution

Module

Image filters

gauss_distribution (: : Sigma : Distribution)

Generate a Gaussian noise distribution.

gauss_distribution generates a Gaussian noise distribution. The parameter **Sigma** determines the noise's standard deviation. Usually, the result **Distribution** is used as input for the operator **add_noise_distribution**.

Parameter
▷ Sigma (input_control) real $\leadsto$ real Standard deviation of the Gaussian noise distribution. Default Value : 2.0 Suggested values : Sigma $\in$ {1.5, 2.0, 3.0, 5.0, 10.0} Typical range of values : $0.0 \leq \text{Sigma} \leq 100.0$ Minimum Increment : 0.1 Recommended Increment : 1.0 ▷ Distribution (output_control) distribution.values-array $\leadsto$ real Resulting Gaussian noise distribution. Number of elements : 513

Example

```
read_image(Image, 'fabrik')
disp_image(Image, WindowHandle)
gauss_distribution(30, Dist)
add_noise_distribution(Image, ImageNoise, Dist)
disp_image(ImageNoise, WindowHandle).
```

Parallelization Information
gauss_distribution is <i>reentrant</i> and processed <i>without</i> parallelization.

Possible Successors
add_noise_distribution

Alternatives
sp_distribution , noise_distribution_mean

See also
sp_distribution , add_noise_white , noise_distribution_mean

Module
Image filters

noise_distribution_mean (ConstRegion,
Image : : FilterSize : Distribution)

Determine the noise distribution of an image.

[noise_distribution_mean](#) calculates the noise distribution in a region of the image [Image](#). The parameter [ConstRegion](#) determines a region of the image with approximately constant gray values. Ideally, the changes in gray values should only be caused by noise in this region. From this region the noise distribution is determined by using the [mean_image](#) operator to smooth the image, and to use the gray value differences in this area as an estimate for the noise distribution, which is returned in [Distribution](#).

Attention
It is important to ensure that the region ConstRegion is not too close to a large gradient in the image, because the gradient values are then used for calculating the mean. This means the the distance of ConstRegion must be at least as large as the filter size FilterSize used for calculating the mean.

Parameter
▷ ConstRegion (input_object) region(-array) $\leadsto$ Hobject Region from which the noise distribution is to be estimated. ▷ Image (input_object) image $\leadsto$ Hobject : byte Corresponding image. ▷ FilterSize (input_control) integer $\leadsto$ integer Size of the mean filter. Default Value : 21 Suggested values : FilterSize $\in$ {5, 11, 15, 21, 31, 51, 101} Typical range of values : $3 \leq \text{FilterSize} \leq 501$ (lin) Minimum Increment : 2 Recommended Increment : 2

- ▷ **Distribution** (output_control) distribution.values-array $\leadsto$ real
Noise distribution of all input regions.

Parallelization Information

noise_distribution_mean is *reentrant* and processed *without* parallelization.

Possible Predecessors

draw_region, **gen_circle**, **gen_ellipse**, **gen_rectangle1**, **gen_rectangle2**, **threshold**,
erosion_circle, **gauss_image**, **smooth_image**, **sub_image**

Possible Successors

add_noise_distribution, **disp_distribution**

See also

mean_image, **gauss_distribution**

Module

Image filters

sp_distribution (: : PercentSalt, PercentPepper : Distribution)

Generate a salt-and-pepper noise distribution.

sp_distribution generates a noise distribution with the values 0 and 255. The parameters **PercentSalt** and **PercentPepper** determine the percentage of white and black noise pixels, respectively. The sum of these parameters must be smaller than 100. Usually, the result **Distribution** is used as input for the operator **add_noise_distribution**.

Parameter

- ▷ **PercentSalt** (input_control) number $\leadsto$ real / integer
Percentage of salt (white noise pixels).
Default Value : 5.0
Suggested values : PercentSalt $\in \{1.0, 2.0, 5.0, 7.0, 10.0, 15.0, 20.0, 30.0\}$
Typical range of values : $0.0 \leq \text{PercentSalt} \leq 100.0$
Minimum Increment : 0.1
Recommended Increment : 1.0
Restriction : $(0.0 \leq \text{PercentSalt}) \wedge (\text{PercentSalt} \leq 100.0)$
- ▷ **PercentPepper** (input_control) number $\leadsto$ real / integer
Percentage of pepper (black noise pixels).
Default Value : 5.0
Suggested values : PercentPepper $\in \{1.0, 2.0, 5.0, 7.0, 10.0, 15.0, 20.0, 30.0\}$
Typical range of values : $0.0 \leq \text{PercentPepper} \leq 100.0$
Minimum Increment : 0.1
Recommended Increment : 1.0
Restriction : $(0.0 \leq \text{PercentPepper}) \wedge (\text{PercentPepper} \leq 100.0)$
- ▷ **Distribution** (output_control) distribution.values-array $\leadsto$ real
Resulting noise distribution.
Number of elements : 513

Example

```
read_image(Image, 'fabrik')
disp_image(Image, WindowHandle)
sp_distribution(30, 30, Dist)
add_noise_distribution(Image, ImageNoise, Dist)
disp_image(ImageNoise, WindowHandle)
```

Parallelization Information

sp_distribution is *reentrant* and processed *without* parallelization.

Possible Successors

add_noise_distribution

Alternatives

[gauss_distribution](#), [noise_distribution_mean](#)

See also

[gauss_distribution](#), [noise_distribution_mean](#), [add_noise_white](#)

Module

Image filters

5.12 Smoothing

```
anisotrope_diff ( Image : ImageAniso : Percent, Mode, Iteration,
neighborhoodType : )
```

Smooth an image by edge-preserving anisotropic diffusion.

The operator [anisotrope_diff](#) carries out an iterative, anisotropic smoothing process on the mathematical basis of physical diffusion. In analogy to the physical diffusion process describing the concentration balance between molecules dependent on the density gradient, the diffusion filter carries out a smoothing of the gray values dependent on the local gray value gradients.

For iterative calculation of the gray value of a pixel the gray value differences in relation to the four or eight neighbors, respectively, are used. These gray value differences, however, are evaluated differently, i.e., a non-linear diffusion process is carried out.

The evaluation is carried out by using a diffusion function (two different functions were implemented, namely [Mode](#) = 1 and/or 2), which — depending on the gradient — ensures that within homogenous regions the smoothing is stronger than over the margins of regions so that the edges remain sharp. The diffusion function is adjusted to the noise ratio of the image by a histogram analysis in the gradient image (according to Canny). A high value for [Percent](#) increases the smoothing effect but blurs the edges a little more (values from 80 - 90 percent are typical).

The parameter [Iteration](#) determines the number of iterations (typically 3–7).

Parameter

- ▷ **Image** (input_object)(multichannel-)image(-array) $\rightsquigarrow$ *Hobject* : byte
Image to be smoothed.
- ▷ **ImageAniso** (output_object)(multichannel-)image(-array) $\rightsquigarrow$ *Hobject* : byte
Smoothed image.
- ▷ **Percent** (input_control) integer $\rightsquigarrow$ *integer*
For histogram analysis; higher values increase the smoothing effect, typically: 80-90.
Default Value : 80
Suggested values : Percent $\in \{65, 70, 75, 80, 85, 90\}$
Typical range of values : $50 \leq \text{Percent} \leq 100$
Minimum Increment : 1
Recommended Increment : 5
- ▷ **Mode** (input_control) integer $\rightsquigarrow$ *integer*
Selection of diffusion function.
Default Value : 1
List of values : Mode $\in \{1, 2\}$
- ▷ **Iteration** (input_control) integer $\rightsquigarrow$ *integer*
Number of iterations, typical values: 3-7.
Default Value : 5
Suggested values : Iteration $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
Typical range of values : $1 \leq \text{Iteration} \leq 30$
Minimum Increment : 1
Recommended Increment : 1
- ▷ **neighborhoodType** (input_control) integer $\rightsquigarrow$ *integer*
Required neighborhood type.
Default Value : 8
List of values : neighborhoodType $\in \{4, 8\}$

Example

```
read_image(Image, 'fabrik')
anisotrope_diff(Image, Aniso, 80, 1, 5, 8)
sub_image(Image, Aniso, Sub, 2.0, 127)
disp_image(Sub, WindowHandle).
```

Complexity

For each pixel: $O(\text{Iterations} * 18)$.

Result

If the parameter values are correct the operator `anisotrope_diff` returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator `set_system('no-object-result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`anisotrope_diff` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*).

Possible Predecessors

`read_image`, `grab_image`

Possible Successors

`regiongrowing`, `threshold`, `sub_image`, `dyn_threshold`, `auto_threshold`

Alternatives

`sigma_image`, `rank_image`

See also

`smooth_image`, `gauss_image`, `sigma_image`, `rank_image`, `eliminate_min_max`

References

P. Perona, J. Malik: "Scale-space and edge detection using anisotropic diffusion", IEEE transaction on pattern analysis and machine intelligence, Vol. 12, No. 7, July 1990.

Module

Image filters

eliminate_min_max (Image : filteredImage : MaskWidth, MaskHeight, Gap, Mode :)

Smooth an image in the spatial domain to suppress noise.

`eliminate_min_max` smooths an image by replacing gray values with neighboring mean values, or local minima/maxima. In order to prevent edges and lines from being smoothed, only those gray values that represent local minima or maxima are replaced (if there is a line or edge within an image there will be at least one neighboring pixel with a comparable gray value). `Gap` controls the strictness of replacement: Only gray values that exceed all other values within their local neighborhood more than `Gap` and all values that fall below their neighboring more than `Gap` are replaced: $E(x, y)$ represents a $N \times M$ sized rectangular neighborhood of an pixel at position (x, y) , containing all pixels within the neighborhood except the pixel itself;

- if $grayvalue(x, y) \geq \text{Gap} + \text{maximum}(E(x, y))$ then replacement;
- else if $grayvalue(x, y) + \text{Gap} \leq \text{minimum}(E(x, y))$ then replacement;
- else adopt $grayvalue(x, y)$ without change;

`Mode` specifies how to perform the new value in case of a replacement.

`Mode` = 1 → replace a local maximum with next minor local maximum and replace a local minimum with next bigger local minimum

`Mode` = 2 → replace with mean value of all pixels within the local neighborhood (including the replaced pixel)

`Mode` = 3 → replace with median value of all pixels within the local neighborhood (including the replaced pixel (this is default and used if `Mode` has got any other value than 1 or 2)

MaskWidth and **MaskHeight** specify the width and height of the rectangular neighborhood. Border treatment: Pixels outside the image border are not considered (e.g.: With a local 3×3 -mask the neighborhood of a pixel at $(0, 0)$ reduces to the pixels at $(1, 0)$, $(0, 1)$ and $(1, 1)$).

Attention

eliminate_min_max only can work on byte images (HALCON image type `BYTE_IMAGE`). If **MaskWidth** or **MaskHeight** is an even number, it is replaced by the next higher odd number (this allows the unique extraction of the center of the filter mask). Width/height of the mask may not exceed the image width/height.

Parameter

- ▷ **Image** (input_object)(multichannel-)image $\leadsto$ *Hobject* : byte
Image to smooth.
- ▷ **filteredImage** (output_object)(multichannel-)image $\leadsto$ *Hobject* : byte
Smoothed image.
- ▷ **MaskWidth** (input_control) extent.x $\leadsto$ *integer*
Width of filter mask.
Default Value : 3
Suggested values : $\text{MaskWidth} \in \{3, 5, 7, 9\}$
Typical range of values : $3 \leq \text{MaskWidth} \leq \text{width}(\text{Image})$
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskWidth)
- ▷ **MaskHeight** (input_control) extent.y $\leadsto$ *integer*
Height of filter mask.
Default Value : 3
Suggested values : $\text{MaskHeight} \in \{3, 5, 7, 9\}$
Typical range of values : $3 \leq \text{MaskHeight} \leq \text{width}(\text{Image})$
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskWidth)
- ▷ **Gap** (input_control) number $\leadsto$ *real*
Gap between local maximum/minimum and all other gray values of the neighborhood.
Default Value : 1.0
Suggested values : $\text{Gap} \in \{1.0, 2.0, 5.0, 10.0\}$
- ▷ **Mode** (input_control) integer $\leadsto$ *integer*
Replacement rule (1 = next minimum/maximum, 2 = average, 3 =median).
Default Value : 3
List of values : $\text{Mode} \in \{1, 2, 3\}$

Result

eliminate_min_max returns 2 (`H_MSG_TRUE`) if all parameters are correct. If the input is empty **eliminate_min_max** returns with an error message.

Parallelization Information

eliminate_min_max is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Successors

wiener_filter, **wiener_filter_ni**

See also

mean_sp, **mean_image**, **median_image**, **median_weighted**, **gauss_image**, **smooth_image**

References

M. Imme:“A Noise Peak Elimination Filter”; S. 204-211 in CVGIP Graphical Models and Image Processing, Vol. 53, No. 2, March 1991

M. Lückenhaus:“Grundlagen des Wiener-Filters und seine Anwendung in der Bildanalyse”; Diplomarbeit; Technische Universität München, Institut für Informatik; Lehrstuhl Prof. Radig; 1995.

Module

Image filters

eliminate_sp (Image : ImageFillSP : MaskWidth, MaskHeight, MinThresh, MaxThresh :)

Replace values outside of thresholds with average value.

The operator **eliminate_sp** replaces all gray values outside the indicated gray value intervals (**MinThresh** to **MaxThresh**) with the neighboring mean values. Only those neighboring pixels which also fall within the gray value interval are used for averaging. If no such pixel is present in the vicinity the original gray value is used. The gray values in the input image falling within the gray value interval are also adopted without change.

Attention

If even values instead of odd values are given for **MaskHeight** or **MaskWidth**, the routine uses the next larger odd values instead (this way the center of the filter mask is always explicitly determined).

Parameter

- ▷ **Image** (input_object)(multichannel-)image(-array) $\leadsto$ *Hobject* : byte
Input image.
- ▷ **ImageFillSP** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte
Smoothed image.
- ▷ **MaskWidth** (input_control) extent.x $\leadsto$ *integer*
Width of filter mask.
Default Value : 3
Suggested values : MaskWidth $\in \{3, 5, 7, 9, 11\}$
Typical range of values : $3 \leq \text{MaskWidth} \leq 512$ (lin)
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskWidth)
- ▷ **MaskHeight** (input_control) extent.y $\leadsto$ *integer*
Height of filter mask.
Default Value : 3
Suggested values : MaskHeight $\in \{3, 5, 7, 9, 11\}$
Typical range of values : $3 \leq \text{MaskHeight} \leq 512$ (lin)
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskHeight)
- ▷ **MinThresh** (input_control) integer $\leadsto$ *integer*
Minimum gray value.
Default Value : 1
Suggested values : MinThresh $\in \{1, 5, 7, 9, 11, 15, 23, 31, 43, 61, 101\}$
Typical range of values : $0 \leq \text{MinThresh} \leq 254$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **MaxThresh** (input_control) integer $\leadsto$ *integer*
Maximum gray value.
Default Value : 254
Suggested values : MaxThresh $\in \{5, 7, 9, 11, 15, 23, 31, 43, 61, 101, 200, 230, 250, 254\}$
Typical range of values : $1 \leq \text{MaxThresh} \leq 255$ (lin)
Minimum Increment : 1
Recommended Increment : 1
Restriction : MinThresh $\leq$ MaxThresh

Example

```
read_image(Image, 'meer_rot')
disp_image(Image, WindowHandle)
eliminate_sp(Image, ImageMeansp, 3, 3, 101, 201)
disp_image(ImageMeansp, WindowHandle).
```

Parallelization Information

eliminate_sp is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

disp_image	Possible Successors
mean_sp , mean_image , median_image , eliminate_min_max	Alternatives
gauss_image , smooth_image , anisotrope_diff , sigma_image , eliminate_min_max	See also
Image filters	Module

fill_interlace (ImageCamera : ImageFilled : Mode :)

Interpolate 2 video half images.

The operator [fill_interlace](#) calculates an interpolated full image or removes odd/even lines from a video image composed of two half images. If an image is recorded with a video camera it consists of two half images recorded at different times but stored in one image in the digital form. This can lead to several errors in further processing. In order to reduce these errors the video image is modified. Every second line is re-calculated or removed. The parameter **Mode** determines whether this must be done for even ('even', 'rmeven') or odd ('odd', 'rmodd') line numbers. If you choose 'even' or 'odd' the gray values in the generated lines are calculated as mean values from the direct neighbors above or below the current pixel, respectively. If you choose 'rmeven' or 'rmodd' the even or odd lines numbers are removed (in that case the resulting image is only half as high as the input image). The value 'switch' for **Mode** cause the odd and even lines to be exchanged.

	Parameter
▷ ImageCamera (input_object)(multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte	
Gray image consisting of two half images.	
▷ ImageFilled (output_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte	
Full image with interpolated/removed lines.	
▷ Mode (input_control) string $\leadsto$ <i>string</i>	
Instruction whether even or odd lines should be replaced/removed.	
Default Value : 'odd'	
List of values : Mode $\in$ {'odd', 'even', 'rmodd', 'rmeven', 'switch'}	

	Example

```

read_image(Image, 'video_bild')
fill_interlace(Image, New, 'odd')
sobel_amp(New, Sobel, 'sum_abs', 3).
  
```

	Complexity

For each pixel: $O(2)$.

	Result

If the parameter values are correct the operator [fill_interlace](#) returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator [set_system](#) ('no-object-result', <Result>). If necessary an exception handling is raised.

	Parallelization Information

[fill_interlace](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

	Possible Predecessors

[read_image](#), [grab_image](#)

	Possible Successors

[sobel_amp](#), [edges_image](#), [regiongrowing](#), [diff_of_gauss](#), [threshold](#), [dyn_threshold](#), [auto_threshold](#), [mean_image](#), [gauss_image](#), [anisotrope_diff](#), [sigma_image](#), [median_image](#)

	See also

[median_image](#), [gauss_image](#), [crop_part](#)

Module

Image filters

gauss_image (Image : ImageGauss : Size :)

Smooth using discrete gauss functions.

The operator [gauss_image](#) smoothes images using the discrete Gaussian. The smoothing effect increases with increasing filter size. The following filter sizes ([Size](#)) are supported (the sigma value of the gauss function is indicated in brackets):

3	(0.81)
5	(0.93)
7	(1.50)
9	(2.00)
11	(2.45)

For border treatment the gray values of the images are reflected at the image borders.

Parameter

- ▷ **Image** (input_object)(multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4
Image to be smoothed.
- ▷ **ImageGauss** (output_object)(multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4
Filtered image.
- ▷ **Size** (input_control) integer $\leadsto$ *integer*
Required filter size.
Default Value : 5
List of values : Size $\in \{3, 5, 7, 9, 11\}$

Example

```
gauss_image(Input, Gauss, 7)
regiongrowing(Gauss, Segments, 7, 7, 5, 100).
```

Complexity

For each pixel: $O(Size * 2)$.

Result

If the parameter values are correct the operator [gauss_image](#) returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator [set_system](#) ('no_object_result', <Result>). If necessary an exception handling is raised.

Parallelization Information

[gauss_image](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Predecessors

[read_image](#), [grab_image](#)

Possible Successors

[regiongrowing](#), [threshold](#), [sub_image](#), [dyn_threshold](#), [auto_threshold](#)

Alternatives

[smooth_image](#), [derivate_gauss](#)

See also

[mean_image](#), [anisotrope_diff](#), [sigma_image](#), [gen_lowpass](#)

Module

Image filters

info_smooth (: : Filter, Alpha : Size, Coeffs)

Information on smoothing filter [smooth_image](#).

The operator [info_smooth](#) returns an estimation of the width of the smoothing filters used in routine [smooth_image](#). For this purpose the underlying continuous impulse answers of [Filter](#) are scanned until a filter coefficient is smaller than five percent of the maximum coefficient (at the origin). [Alpha](#) is the filter parameter (see [smooth_image](#)). Currently four filters are supported (parameter [Filter](#)):

'deriche1', 'deriche2', 'shen' und 'gauss'.

The gauss filter was conventionally implemented with filter masks (the other three are recursive filters). In the case of the gauss filter the filter coefficients (of the one-dimensional impulse answer $f(n)$ with $n \geq 0$) are returned in [Coeffs](#) in addition to the filter size.

Parameter	
▷ Filter (input_control)	string $\leadsto$ string
Name of required filter.	
Default Value : 'deriche2'	
List of values : $\text{Filter} \in \{\text{'deriche1'}, \text{'deriche2'}, \text{'shen'}, \text{'gauss'}\}$	
▷ Alpha (input_control)	real $\leadsto$ real
Filter parameter: small values effect strong smoothing (reversed in case of 'gauss').	
Default Value : 0.5	
Suggested values : $\text{Alpha} \in \{0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 4.0, 5.0, 7.0, 10.0\}$	
Typical range of values : $0.01 \leq \text{Alpha} \leq 50.0$	
Minimum Increment : 0.01	
Recommended Increment : 0.1	
Restriction : $\text{Alpha} > 0.0$	
▷ Size (output_control)	integer $\leadsto$ integer
Width of filter is approx. size x size pixels.	
▷ Coeffs (output_control)	integer-array $\leadsto$ integer
In case of gauss filter: coefficients of the "positive" half of the 1D impulse answer.	

Example

```
info_smooth('deriche2', 0.5, Size, Coeffs)
smooth_image(Input, Smooth, 'deriche2', 7).
```

Result

If the parameter values are correct the operator [info_smooth](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

[info_smooth](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[read_image](#)

Possible Successors

[smooth_image](#)

See also

[smooth_image](#)

Module

Image filters

mean_image (Image : ImageMean : MaskWidth, MaskHeight :)

Smooth by averaging.

The operator `mean_image` carries out a linear smoothing with the gray values of all input images (`Image`). The filter matrix consists of ones (evaluated equally) and has the size `MaskHeight` $\times$ `MaskWidth`. The result of the convolution is divided by `MaskHeight` $\times$ `MaskWidth`. For border treatment the gray values are reflected at the image edges.

Attention

If even values instead of odd values are given for `MaskHeight` or `MaskWidth`, the routine uses the next larger odd values instead (this way the center of the filter mask is always explicitly determined).

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4 / real / dvf
Image to be smoothed.
- ▷ **ImageMean** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4 / real / dvf
Smoothed image.
- ▷ **MaskWidth** (input_control) extent.x $\leadsto$ *integer*
Width of filter mask.
Default Value : 9
Suggested values : `MaskWidth` $\in$ {3, 5, 7, 9, 11, 15, 23, 31, 43, 61, 101}
Typical range of values : $1 \leq \text{MaskWidth} \leq 501$
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(`MaskWidth`)
- ▷ **MaskHeight** (input_control) extent.y $\leadsto$ *integer*
Height of filter mask.
Default Value : 9
Suggested values : `MaskHeight` $\in$ {3, 5, 7, 9, 11, 15, 23, 31, 43, 61, 101}
Typical range of values : $1 \leq \text{MaskHeight} \leq 501$
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(`MaskHeight`)

Example

```
read_image(Image, 'fabrik')
mean_image(Image, Mean, 3, 3)
disp_image(Mean, WindowHandle).
```

Complexity

For each pixel: O(15).

Result

If the parameter values are correct the operator `mean_image` returns the value 2 (`H_MSG_TRUE`). The behavior in case of empty input (no input images available) is set via the operator `set_system('no_object_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`mean_image` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Successors

`dyn_threshold`, `regiongrowing`

Alternatives

`gauss_image`, `smooth_image`

See also

`anisotrope_diff`, `sigma_image`, `convol_image`, `gen_lowpass`

Module

Image filters

mean_n (Image : ImageMean : :)

Average gray values over several channels.

The operator **mean_n** generates the pixel-by-pixel mean value of all channels. For each coordinate point the sum of all gray values at this coordinate is calculated. The result is the mean of the gray values (sum divided by the number of channels). The output image has one channel.

Parameter

- ▷ **Image** (input_object) multichannel-image(-array) $\leadsto$ *Hobject* : byte / int4 / uint2 / int4 / real
Multichannel gray image.
- ▷ **ImageMean** (output_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / int4 / uint2 / int4 / real
Result of averaging.

Example

```
compose3(Channel1, Channel2, Channel3, &MultiChannel);
mean_n(MultiChannel, &Mean);
```

Parallelization Information

mean_n is *reentrant* and automatically *parallelized* (on *tuple level*, *domain level*).

Possible Predecessors

[compose2](#), [compose3](#), [compose4](#), [add_channels](#)

Possible Successors

[disp_image](#)

See also

[count_channels](#)

Module

Image filters

mean_sp (Image : ImageSPMean : MaskWidth, MaskHeight, MinThresh, MaxThresh :)

Suppress salt and pepper noise.

The operator **mean_sp** carries out a smoothing by averaging the values. Only the gray values within the interval from **MinThresh** to **MaxThresh** are averaged. Gray values which are too light or too dark are ignored during summation. If no gray value lies within the default interval during summation the original gray value is adopted. If the thresholds are set at 0 or 255, respectively, the operator **mean_sp** behaves like **mean_image** except for the running time.

The operator **mean_sp** is used to suppress extreme gray values (salt and pepper noise = white and black dots).

Attention

If even values instead of odd values are given for **MaskHeight** or **MaskWidth**, the routine uses the next larger odd values instead (this way the center of the filter mask is always explicitly determined).

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte
Input image.
- ▷ **ImageSPMean** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte
Smoothed image.
- ▷ **MaskWidth** (input_control) extent.x $\leadsto$ *integer*
Width of filter mask.
Default Value : 3
Suggested values : MaskWidth $\in$ {3, 5, 7, 9, 11}
Typical range of values : $3 \leq \text{MaskWidth} \leq 512$ (lin)
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskWidth)

- ▷ **MaskHeight** (input_control) extent.y $\leadsto$ integer
 Height of filter mask.
Default Value : 3
Suggested values : MaskHeight $\in \{3, 5, 7, 9, 11\}$
Typical range of values : $3 \leq \text{MaskHeight} \leq 512$ (lin)
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskHeight)
- ▷ **MinThresh** (input_control) integer $\leadsto$ integer
 Minimum gray value.
Default Value : 1
Suggested values : MinThresh $\in \{1, 5, 7, 9, 11, 15, 23, 31, 43, 61, 101\}$
Typical range of values : $0 \leq \text{MinThresh} \leq 255$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **MaxThresh** (input_control) integer $\leadsto$ integer
 Maximum gray value.
Default Value : 254
Suggested values : MaxThresh $\in \{5, 7, 9, 11, 15, 23, 31, 43, 61, 101, 200, 230, 250, 254\}$
Typical range of values : $0 \leq \text{MaxThresh} \leq 255$ (lin)
Minimum Increment : 1
Recommended Increment : 1
Restriction : MinThresh $\leq$ MaxThresh

Example

```
read_image(Image, 'meer_rot')
disp_image(Image, WindowHandle)
mean_sp(Image, ImageMeansp, 3, 3, 101, 201)
disp_image(ImageMeansp, WindowHandle).
```

Parallelization Information

[mean_sp](#) is *reentrant* and automatically *parallelized* (on tuple level, channel level, domain level).

Possible Successors

[disp_image](#)

Alternatives

[mean_image](#), [median_image](#), [median_separate](#), [eliminate_min_max](#)

See also

[anisotrope_diff](#), [sigma_image](#), [gauss_image](#), [smooth_image](#), [eliminate_min_max](#)

Module

Image filters

median_image (Image : ImageMedian : MaskType, Radius, Margin :)

Median filtering with different rank masks.

The operator [median_image](#) carries out a non-linear smoothing of the gray values of all input images ([Image](#)). The shift mask ([MaskType](#)) is transmitted in the form of an object (more precisely: its region). Several border treatments can be chosen for filtering ([Margin](#)):

gray value	Pixels outside of the image edges are assumed to be constant (with the indicated gray value).
'continued'	Continuation of edge pixels.
'cyclic'	Cyclic continuation of image edges.
'mirrored'	Reflection of pixels at the image edges.

The indicated mask (= region of the mask image) is put over the image to be filtered in such a way that the center of the mask touches all pixels of the objects once. For each of these pixels all neighboring pixels covered by the mask are sorted in an ascending sequence according to their gray values. Thus, each of these sorted gray value sequences contains exactly as many gray values as the mask has pixels. From these sequences the median is selected and entered as resulting gray value at the corresponding output image.

Parameter

- ▷ **Image** (input_object)(multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4 / real
Image to be filtered.
- ▷ **ImageMedian** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4 / real
Median filtered image.
- ▷ **MaskType** (input_control) string $\leadsto$ *string*
Type of median mask.
Default Value : 'circle'
List of values : MaskType $\in$ {'circle', 'rectangle'}
- ▷ **Radius** (input_control) integer $\leadsto$ *integer*
Radius of median mask.
Default Value : 1
Suggested values : Radius $\in$ {1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 15, 19, 25, 31, 39, 47, 59}
Typical range of values : $1 \leq \text{Radius} \leq 101$
Minimum Increment : 1
Recommended Increment : 2
- ▷ **Margin** (input_control) string $\leadsto$ *string* / integer / real
Border treatment.
Default Value : 'mirrored'
Suggested values : Margin $\in$ {'mirrored', 'cyclic', 'continued', 0, 30, 60, 90, 120, 150, 180, 210, 240, 255}

Example

```
read_image(Image, 'fabrik')
median_image(Image, Median, 'circle', 3, 'continued')
disp_image(MedianWeighted, WindowHandle).
```

Complexity

For each pixel: $O(\sqrt{F} * 5)$ with F = area of **MaskType**.

Result

If the parameter values are correct the operator **median_image** returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator **set_system('no_object_result', <Result>)**. If necessary an exception handling is raised.

Parallelization Information

median_image is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Predecessors

read_image

Possible Successors

threshold, **dyn_threshold**, **regiongrowing**

Alternatives

rank_image

See also

gen_circle, **gen_rectangle1**, **gray_erosion_rect**, **gray_dilation_rect**

References

R. Haralick, L. Shapiro; "Computer and Robot Vision"; Addison-Wesley, 1992, Seite 318-319

Module

Image filters

median_separate (Image : ImageSMedian : MaskWidth, MaskHeight, Margin :)

Separated median filtering with rectangle masks.

The operator `median_separate` carries out a variation of the median filtering: First two auxiliary images are created. The first one originates from a median filtering with a horizontal mask with a height of one pixel and the width `MaskWidth` followed by filtering with a mask with the height `MaskHeight`. The second auxiliary image is created by filtering with the same masks, but with a reversed sequence of the operation: first the vertical, then the horizontal mask. The output image results from averaging the two auxiliary images pixel by pixel.

The operator `median_separate` is clearly faster than the normal operator `median_image` because both masks are one pixel wide, facilitating a very efficient processing. The runtime is practically *independent* of the size of the mask. For example, the operator `median_separate` can be well used after texture filters, where large masks are needed.

The filter can also be used several times in a row in order to enhance the smoothing.

Parameter

- ▷ **Image** (input_object)(multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4 / real
Image to be filtered.
- ▷ **ImageSMedian** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4 / real
Median filtered image.
- ▷ **MaskWidth** (input_control) extent.x $\leadsto$ *integer*
Width of rank mask.
Default Value : 25
Suggested values : MaskWidth $\in \{1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 27, 43, 51, 67, 91, 121, 151\}$
Typical range of values : $1 \leq \text{MaskWidth} \leq 401$
Minimum Increment : 2
Recommended Increment : 2
- ▷ **MaskHeight** (input_control) extent.y $\leadsto$ *integer*
Height of rank mask.
Default Value : 25
Suggested values : MaskHeight $\in \{1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 27, 43, 51, 67, 91, 121, 151\}$
Typical range of values : $1 \leq \text{MaskHeight} \leq 401$
Minimum Increment : 2
Recommended Increment : 2
- ▷ **Margin** (input_control) string $\leadsto$ *string* / *integer* / *real*
Border treatment.
Default Value : 'mirrored'
Suggested values : Margin $\in \{\text{'mirrored'}, \text{'cyclic'}, \text{'continued'}, 0, 30, 60, 90, 120, 150, 180, 210, 240, 255\}$

Example

```
read_image(Image, 'fabrik')
median_separate(Image, MedianSeparate, 5, 5, 3)
disp_image(MedianSeparate, WindowHandle).
```

Complexity

For each pixel: $O(40)$.

Parallelization Information

`median_separate` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Predecessors

`texture_laws`, `sobel_amp`, `deviation_image`

Possible Successors

`learn_ndim_norm`, `learn_ndim_box`, `median_separate`, `regiongrowing`, `auto_threshold`

Alternatives

`median_image`

See also

[rank_image](#)

References

R. Haralick, L. Shapiro; "Computer and Robot Vision"; Addison-Wesley, 1992, Seite 319

Module

Image filters

median_weighted (Image : ImageWMedian : MaskType, MaskSize :)
Weighted median filtering with different rank masks.

The operator [median_weighted](#) calculates the median of the gray values within a local environment. In contrast to [median_image](#), which uses all gray values within the environment exactly once, the operator [median_weighted](#) weights all gray values several times depending on their position. A gray value is received into the field to be sorted several times according to its weighting. The following masks are available:

'gauss' ([MaskSize](#) = 3)

```

1  2  1
2  4  2
1  2  1

```

'inner' ([MaskSize](#) = 3)

```

1  1  1
1  3  1
1  1  1

```

The operator [median_weighted](#) means that, contrary to [median_image](#), gray value corners remain.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2
Image to be filtered.
- ▷ **ImageWMedian** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2
Median filtered image.
- ▷ **MaskType** (input_control) string $\leadsto$ *string*
Type of median mask.
Default Value : 'inner'
List of values : MaskType $\in$ {'inner', 'gauss'}
- ▷ **MaskSize** (input_control) integer $\leadsto$ *integer*
mask size.
Default Value : 3
List of values : MaskSize $\in$ {3}

Example

```

read_image(Image, 'fabrik')
median_weighted(Image, MedianWeighted, 'gauss', 3)
disp_image(MedianWeighted, WindowHandle).

```

Complexity

For each pixel: $O(F * \log F)$ with F = area of [MaskType](#).

Parallelization Information

[median_weighted](#) is *reentrant* and automatically *parallelized* (on tuple level, channel level, domain level).

Possible Predecessors

[read_image](#)

Possible Successors

[threshold](#), [dyn_threshold](#), [regiongrowing](#)

Alternatives

[median_image](#), [trimmed_mean](#), [sigma_image](#)

References

R. Haralick, L. Shapiro; "Computer and Robot Vision"; Addison-Wesley, 1992, Seite 319

Module

Image filters

midrange_image (Image, Mask : ImageMidrange : Margin :)*Calculate the average of maximum and minimum inside any mask.*

The operator [midrange_image](#) forms the average of maximum and minimum inside the indicated mask in the whole image. Several border treatments ([Margin](#)) can be chosen for filtering:

gray value	Pixels outside of the image edges are assumed to be constant (with the indicated gray value).
'continued'	Continuation of edge pixels.
'cyclic'	Cyclic continuation of image edges.
'mirrored'	Reflection of pixels at the image edges.

The indicated mask (= region of the mask image) is put over the image to be filtered in such a way that the center of the mask touches all pixels once.

Parameter

- ▷ **Image** (input_object)(multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4 / real
Image to be filtered.
- ▷ **Mask** (input_object) region $\leadsto$ *Hobject* : byte
Region serving as filter mask.
- ▷ **ImageMidrange** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4 / real
Filtered output image.
- ▷ **Margin** (input_control) string $\leadsto$ *string* / integer / real
Border treatment.
Default Value : 'mirrored'
Suggested values : Margin $\in$ {'mirrored', 'cyclic', 'continued', 0, 30, 60, 90, 120, 150, 180, 210, 240, 255}

Example

```
read_image(Image, 'fabrik')
draw_region(Region, WindowHandle)
midrange_image(Image, Region, Midrange, 'mirrored')
disp_image(Midrange, WindowHandle).
```

Complexity

For each pixel: $O(\sqrt{F} * 5)$ with F = area of [Mask](#).

Result

If the parameter values are correct the operator [midrange_image](#) returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator [set_system](#) ('no_object_result', <Result>). If necessary an exception handling is raised.

Parallelization Information

[midrange_image](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Predecessors

[read_image](#), [draw_region](#), [gen_circle](#), [gen_rectangle1](#)

Possible Successors

[threshold](#), [dyn_threshold](#), [regiongrowing](#)

Alternatives

[sigma_image](#)

See also

[gen_circle](#), [gen_rectangle1](#), [gray_erosion_rect](#), [gray_dilation_rect](#), [gray_range_rect](#)

References

R. Haralick, L. Shapiro; "Computer and Robot Vision"; Addison-Wesley, 1992, Seite 319

Module

Image filters

rank_image (Image, Mask : ImageRank : Rank, Margin :)*Smooth an image with an arbitrary rank mask.*

The operator [rank_image](#) carries out a non-linear smoothing of the gray values of all input images ([Image](#)). The filter mask ([Mask](#)) is transmitted as a region. In contrast to many other filters you can choose an arbitrary shape, e.g., by using operators like [gen_circle](#) or [draw_region](#). The position of the mask region has no influence on the result; the center of gravity of the region is used as the reference point of the filter mask.

The specified mask is moved over the image to be filtered in such a way that the reference point of the mask touches all pixels once. At each position a histogram is calculated from the gray values of all pixels covered by the mask. By specifying [Rank](#) = 1 the lowest (= darkest) gray value appearing in the histogram is selected and entered as resulting gray value in the output image [ImageRank](#); if [Rank](#) corresponds to the number of pixels of the filter mask, i.e., its area, the brightest gray value is selected. This behavior is identical to the erosion/dilation operators in gray morphology ([gray_erosion](#), [gray_dilation](#)). If you use a rank that is equal to half of the pixels of the filter mask you get the same behavior as for the the median filter ([median_image](#)).

You can use [rank_image](#) to eliminate noise, to eliminate structures with a given orientation (use [gen_rectangle2](#) to create the mask region), or as an advanced gray morphologic operator that is more robust against noise. In this case you will not use 1 or the mask area as rank values, but a slightly higher or lower value, respectively.

Several border treatments can be chosen for filtering ([Margin](#)):

gray value	Pixels outside of the image edges are assumed to be constant (with the indicated gray value).
'continued'	Continuation of edge pixels.
'cyclic'	Cyclic continuation of image edges.
'mirrored'	Reflection of pixels at the image edges.

Parameter

- ▷ **Image** (input_object) multichannel-image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4 / real
Image to be filtered.
- ▷ **Mask** (input_object) region $\leadsto$ *Hobject* : byte
Region serving as filter mask.
- ▷ **ImageRank** (output_object) multichannel-image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4 / real
Filtered image.
- ▷ **Rank** (input_control) integer $\leadsto$ *integer*
Rank of the output gray value in the sorted sequence of input gray values inside the filter mask. Typical value (median): $\text{area}(\text{mask}) / 2$.
Default Value : 5
Suggested values : $\text{Rank} \in \{1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31\}$
Typical range of values : $1 \leq \text{Rank} \leq 512$
Minimum Increment : 1
Recommended Increment : 2
- ▷ **Margin** (input_control) string $\leadsto$ *string* / integer / real
Border treatment.
Default Value : 'mirrored'
Suggested values : $\text{Margin} \in \{\text{'mirrored'}, \text{'cyclic'}, \text{'continued'}, 0, 30, 60, 90, 120, 150, 180, 210, 240, 255\}$

Example

```
read_image(Image, 'fabrik')
draw_region(Region, WindowHandle)
rank_image(Image, Region, ImageRank, 5, 'mirrored')
disp_image(ImageRank, WindowHandle).
```

Complexity

For each pixel: $O(\sqrt{F} * 5)$ with F = area of **Mask**.

Result

If the parameter values are correct the operator **rank_image** returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator **set_system** ('no_object_result', <Result>). If necessary an exception handling is raised.

Parallelization Information

rank_image is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Predecessors

read_image, **draw_region**, **gen_circle**, **gen_rectangle1**

Possible Successors

threshold, **dyn_threshold**, **regiongrowing**

Alternatives

sigma_image

See also

gen_circle, **gen_rectangle1**, **gray_erosion_rect**, **gray_dilation_rect**

References

R. Haralick, L. Shapiro; "Computer and Robot Vision"; Addison-Wesley, 1992, Seite 318-320

Module

Image filters

sigma_image (Image : ImageSigma : MaskHeight, MaskWidth, Sigma :)

Non-linear smoothing with the sigma filter.

The operator **sigma_image** carries out a non-linear smoothing of the gray values of all input images (**Image**). All pixels are checked in a rectangular window (**MaskHeight** × **MaskWidth**). All pixels of the window which differ from the current pixel by less than **Sigma** are used for calculating the new pixel, which is the average of the chosen pixels. If all differences are larger than **Sigma** the gray value is adapted unchanged.

Attention

The filter is implemented for images of the 'byte' type only. If even values instead of odd values are given for **MaskHeight** or **MaskWidth**, the routine uses the next larger odd values instead (this way the center of the filter mask is always explicitly determined).

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / cyclic / int1 / int2 / uint2 / int4 / real
Image to be smoothed.
- ▷ **ImageSigma** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / cyclic / int1 / int2 / uint2 / int4 / real
Smoothed image.
- ▷ **MaskHeight** (input_control) extent.y $\leadsto$ *integer*
Height of the mask (number of lines).
Default Value : 5
Suggested values : MaskHeight $\in$ {3, 5, 7, 9, 11, 13, 15}
Typical range of values : $3 \leq \text{MaskHeight} \leq 101$
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskHeight)

- ▷ **Maskwidth** (input_control) extent.x $\leadsto$ integer
Width of the mask (number of columns).
Default Value : 5
Suggested values : MaskWidth $\in \{3, 5, 7, 9, 11, 13, 15\}$
Typical range of values : $3 \leq \text{MaskWidth} \leq 101$
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskWidth)
- ▷ **Sigma** (input_control) integer $\leadsto$ integer
Max. deviation to the average.
Default Value : 3
Suggested values : Sigma $\in \{3, 5, 7, 9, 11, 20, 30, 50\}$
Typical range of values : $0 \leq \text{Sigma} \leq 255$
Minimum Increment : 1
Recommended Increment : 2

Example

```
read_image(Image, 'fabrik')
sigma_image(Image, ImageSigma, 5, 5, 3)
disp_image(ImageSigma, WindowHandle).
```

Complexity

For each pixel: $O(\text{MaskHeight} \times \text{MaskWidth})$.

Result

If the parameter values are correct the operator `sigma_image` returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator `set_system('no_object_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`sigma_image` is *reentrant* and automatically *parallelized* (on tuple level, channel level, domain level).

Possible Predecessors

`read_image`

Possible Successors

`threshold`, `dyn_threshold`, `regiongrowing`

Alternatives

`anisotrope_diff`, `rank_image`

See also

`smooth_image`, `gauss_image`, `mean_image`

References

R. Haralick, L. Shapiro; "Computer and Robot Vision"; Addison-Wesley, 1992, Seite 325

Module

Image filters

smooth_image (Image : ImageSmooth : Filter, Alpha :)

Smooth an image using recursive filters.

`smooth_image` smooths gray images using recursive filters originally developed by Deriche and Shen and using the non-recursive Gaussian filter. The following filters can be chosen via the parameter `Filter`:

'deriche1', 'deriche2', 'shen' und 'gauss'.

The "filter width" (i.e., the range of the filter and thereby result of the filter) can be of any size. In the case that the Deriche or Shen is chosen it decreases by increasing the filter parameter `Alpha` and increases in the case of the Gauss filter (and `Alpha` corresponds to the standard deviation of the Gaussian function). An approximation of the appropriate size of the filterwidth `Alpha` is performed by the operator `info_smooth`.

Non-recursive filters like the Gaussian filter are often implemented using filter-masks. In this case the runtime of the operator increases with increasing size of the filter mask. The runtime of the recursive filters remains constant; except the border treatment becomes a little bit more time consuming. The Gaussian filter becomes slow in comparison to the recursive ones but is in contrast to them isotropic (the filter 'deriche2' is only weakly direction sensitive). A comparable result of the smoothing is achieved by choosing the following values for the parameter:

$$\begin{aligned}\text{Alpha}('deriche2') &= \frac{\text{Alpha}('deriche1')}{2} \\ \text{Alpha}('shen') &= \frac{\text{Alpha}('deriche1')}{2} \\ \text{Alpha}('gauss') &= \frac{1.77}{\text{Alpha}('deriche1')}$$

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / uint2
Image to be smoothed.
- ▷ **ImageSmooth** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / uint2
Smoothed image.
- ▷ **Filter** (input_control) string $\leadsto$ *string*
Filter.
Default Value : 'deriche2'
List of values : Filter $\in$ {'deriche1', 'deriche2', 'shen', 'gauss'}
- ▷ **Alpha** (input_control) real $\leadsto$ *real*
Filterparameter: small values cause strong smoothing (vice versa by using bei 'gauss').
Default Value : 0.5
Suggested values : Alpha $\in$ {0.1, 0.2, 0.3, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 4.0, 5.0, 7.0, 10.0}
Typical range of values : $0.01 \leq \text{Alpha} \leq 50.0$
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : Alpha > 0

Example

```
info_smooth('deriche2', 0.5, Size, Coeffs)
smooth_image(Input, Smooth, 'deriche2', 7)
```

Result

If the parameter values are correct the operator **smooth_image** returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator **set.system** ('no_object_result', <Result>). If necessary an exception handling is raised.

Parallelization Information

smooth_image is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*).

Possible Predecessors

read_image

Possible Successors

threshold, **dyn_threshold**, **regiongrowing**

Alternatives

gauss_image, **mean_image**, **derivate_gauss**

See also

info_smooth, **median_image**, **sigma_image**, **anisotrope_diff**

References

R.Deriche: "Fast Algorithms for Low-Level Vision"; IEEE Transactions on Pattern Analysis and Machine Intelligence; PAMI-12, no. 1; S. 78-87; 1990.

Module

Image filters

trimmed_mean (Image, Mask : ImageTMean : Number, Margin :)

Smooth an image with an arbitrary rank mask.

The operator **trimmed_mean** carries out a non-linear smoothing of the gray values of all input images (**Image**). The filter mask (**Mask**) is passed in the form of a region. The average of **Number** gray values located near the median is calculated. Several border treatments can be chosen for filtering (**Margin**):

gray value	Pixels outside of the image edges are assumed to be constant (with the indicated gray value).
'continued'	Continuation of edge pixels.
'cyclic'	Cyclic continuation of image edges.
'mirrored'	Reflection of pixels at the image edges.

The indicated mask (= region of the mask image) is put over the image to be filtered in such a way that the center of the mask touches all pixels once. For each of these pixels all neighboring pixels covered by the mask are sorted in an ascending sequence according to their gray values. Thus, each of these sorted gray value sequences contains exactly as many gray values as the mask has pixels. If F is the area of the mask the average of these sequences is calculated as follows: The first $(F - \text{Number})/2$ gray values are ignored. Then the following **Number** gray values are summed up and divided by **Number**. Again the remaining $(F - \text{Number})/2$ gray values are ignored.

Parameter

- ▷ **Image** (input_object) multichannel-image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4 / real
Image to be filtered.
- ▷ **Mask** (input_object) region $\leadsto$ *Hobject*
Image whose region serves as filter mask.
- ▷ **ImageTMean** (output_object) multichannel-image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4 / real
Filtered output image.
- ▷ **Number** (input_control) integer $\leadsto$ *integer*
Number of averaged pixels. Typical value: $\text{Surface}(\text{Mask}) / 2$.
Default Value : 5
Suggested values : $\text{Number} \in \{1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25, 27, 29, 31\}$
Typical range of values : $1 \leq \text{Number} \leq 401$
Minimum Increment : 1
Recommended Increment : 2
- ▷ **Margin** (input_control) string $\leadsto$ *string* / integer / real
Border treatment.
Default Value : 'mirrored'
Suggested values : $\text{Margin} \in \{\text{'mirrored'}, \text{'cyclic'}, \text{'continued'}, 0, 30, 60, 90, 120, 150, 180, 210, 240, 255\}$

Example

```
read_image(Image, 'fabrik')
draw_region(Region, WindowHandle)
trimmed_mean(Image, Region, TrimmedMean, 5, 'mirrored')
disp_image(TrimmedMean, WindowHandle).
```

Result

If the parameter values are correct the operator **trimmed_mean** returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator **set_system** (**'no_object_result'**, <Result>). If necessary an exception handling is raised.

Parallelization Information

trimmed_mean is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Predecessors

read_image, **draw_region**, **gen_circle**, **gen_rectangle1**

Possible Successors

threshold, **dyn_threshold**, **regiongrowing**

<i>Alternatives</i>
sigma_image , median_weighted , median_image
<i>See also</i>
gen_circle , gen_rectangle1 , gray_erosion_rect , gray_dilation_rect
<i>References</i>
R. Haralick, L. Shapiro; “Computer and Robot Vision”; Addison-Wesley, 1992, Seite 320
<i>Module</i>
Image filters

5.13 Texture

deviation_image (Image : ImageDeviation : Width, Height :)

Calculate the standard deviation of gray values within rectangular windows.

deviation_image calculates the standard deviation of gray values in the image **Image** within a rectangular mask of size (**Height**, **Width**). The resulting image is returned in **ImageDeviation**. To better use the range of gray values available in the output image, the result is multiplied by 2. If the parameters **Height** and **Width** are even, they are changed to the next larger odd value. At the image borders the gray values are mirrored.

<i>Parameter</i>
▷ Image (input_object)(multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / int4 / real / int2 / uint2 Image for which the standard deviation is to be calculated. ▷ ImageDeviation (output_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / int4 / real / int2 / uint2 Image containing the standard deviation. ▷ Width (input_control) extent.x $\leadsto$ <i>integer</i> Width of the mask in which the standard deviation is calculated. Default Value : 11 List of values : Width $\in \{3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25\}$ Restriction : $(3 \leq \text{Width}) \wedge \text{odd}(\text{Width})$ ▷ Height (input_control) extent.y $\leadsto$ <i>integer</i> Height of the mask in which the standard deviation is calculated. Default Value : 11 List of values : Height $\in \{3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25\}$ Restriction : $(3 \leq \text{Height}) \wedge \text{odd}(\text{Height})$
<i>Example</i>

```
read_image(Image, 'fabrik')
disp_image(Image, WindowHandle)
deviation_image(Image, Deviation, 9, 9)
disp_image(Deviation, WindowHandle).
```

<i>Result</i>
deviation_image returns 2 (H_MSG.TRUE) if all parameters are correct. If the input is empty the behaviour can be set via set_system('no-object-result', <Result>) . If necessary, an exception handling is raised.

<i>Parallelization Information</i>
deviation_image is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level, channel level).
<i>Possible Successors</i>

disp_image

<i>Alternatives</i>
entropy_image , entropy_gray
<i>See also</i>
convol_image , texture_laws , intensity
<i>Module</i>
Image filters

entropy_image (Image : ImageEntropy : Width, Height :)

Calculate the entropy of gray values within a rectangular window.

entropy_image calculates the entropy of gray values in the image **Image** within a rectangular mask of size (**Height**, **Width**). The resulting image is returned in **ImageEntropy**, in which the entropy is multiplied by 32. If the parameters **Height** and **Width** are even, they are changed to the next larger odd value. At the image borders the gray values are mirrored.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte
Image for which the entropy is to be calculated.
- ▷ **ImageEntropy** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte
Entropy image.
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer*
Width of the mask in which the entropy is calculated.
Default Value : 9
List of values : Width $\in \{3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25\}$
Suggested values : Width $\in \{3, 5, 7, 9, 11, 13, 15\}$
Restriction : $(3 \leq \text{Width}) \wedge \text{odd}(\text{Width})$
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer*
Height of the mask in which the entropy is calculated.
Default Value : 9
List of values : Height $\in \{3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23, 25\}$
Suggested values : Height $\in \{3, 5, 7, 9, 11, 13, 15\}$
Restriction : $(3 \leq \text{Height}) \wedge \text{odd}(\text{Height})$

Example

```
read_image(Image, 'fabrik')
disp_image(Image, WindowHandle)
entropy_image(Image, Entropy1, 9, 9)
disp_image(Entropy1, WindowHandle).
```

Result

entropy_image returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via **set_system('no_object_result', <Result>)**. If necessary, an exception handling is raised.

Parallelization Information

entropy_image is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Successors

disp_image

Alternatives

entropy_gray

See also

energy_gabor, **entropy_gray**

Module

Image filters

texture_laws (Image : ImageTexture : FilterTypes, Shift, FilterSize :)

Filter an image using a Laws texture filter.

texture_laws applies one or more texture transformations (according to Laws) to an image. This is done by convolving the input image with one (or more) filter masks. The filters are:

9 different 3x3 matrices obtainable from the following three vectors:

$$\begin{array}{lcl} l & = & \begin{bmatrix} 1 & 2 & 1 \end{bmatrix} \\ e & = & \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \\ s & = & \begin{bmatrix} -1 & 2 & -1 \end{bmatrix} \end{array}$$

25 different 5x5 matrices obtainable from the following five vectors:

$$\begin{array}{lcl} l & = & \begin{bmatrix} 1 & 4 & 6 & 4 & 1 \end{bmatrix} \\ e & = & \begin{bmatrix} -1 & -2 & 0 & 2 & 1 \end{bmatrix} \\ s & = & \begin{bmatrix} -1 & 0 & 2 & 0 & -1 \end{bmatrix} \\ r & = & \begin{bmatrix} 1 & -4 & 6 & -4 & 1 \end{bmatrix} \\ w & = & \begin{bmatrix} -1 & 2 & 0 & -2 & 1 \end{bmatrix} \end{array}$$

36 different 7x7 matrices obtainable from the following six vectors:

$$\begin{array}{lcl} l & = & \begin{bmatrix} 1 & 6 & 15 & 20 & 15 & 6 & 1 \end{bmatrix} \\ e & = & \begin{bmatrix} -1 & -4 & -5 & 0 & 5 & 4 & 1 \end{bmatrix} \\ s & = & \begin{bmatrix} -1 & -2 & 1 & 4 & 1 & -2 & -1 \end{bmatrix} \\ r & = & \begin{bmatrix} -1 & -2 & -1 & 4 & -1 & -2 & -1 \end{bmatrix} \\ w & = & \begin{bmatrix} -1 & 0 & 3 & 0 & -3 & 0 & 1 \end{bmatrix} \\ o & = & \begin{bmatrix} -1 & 6 & -15 & 20 & -15 & 6 & -1 \end{bmatrix} \end{array}$$

For most of the filters the resulting gray values must be modified by a [Shift](#). This makes the different textures in the output image more comparable to each other, provided suitable filters are used.

The name of the filter is composed of the letters of the two vectors used, where the first letter denotes convolution in the column direction while the second letter denotes convolution in the row direction.

If more than one filter type is passed, a multi-channel image is returned for each (single-channel) input image, with each channel corresponding to a particular filter.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Images to which the texture transformation is to be applied.
- ▷ **ImageTexture** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Texture images.
- ▷ **FilterTypes** (input_control) string(-array) $\leadsto$ *string*
Desired filters (name or number).
Default Value : 'el'
Suggested values : FilterTypes $\in$ {'ll', 'le', 'ls', 'lr', 'lw', 'lo', 'el', 'ee', 'es', 'er', 'ew', 'eo', 'sl', 'se', 'ss', 'sr', 'sw', 'so', 'rl', 're', 'rs', 'rr', 'rw', 'ro', 'wl', 'we', 'ws', 'wr', 'ww', 'wo', 'ol', 'oe', 'os', 'or', 'ow', 'oo'}
- ▷ **Shift** (input_control) integer $\leadsto$ *integer*
Shift to reduce the gray value dynamics.
Default Value : 2
List of values : Shift $\in$ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- ▷ **FilterSize** (input_control) integer $\leadsto$ *integer*
Size of the filter kernel.
Default Value : 5
List of values : FilterSize $\in$ {3, 5, 7}

Example

```
/* 2 dimensional pixel classification */
read_image(Image, 'combine')
open_window(0, 0, -1, -1, 'root', 'visible', '', WindowHandle)
disp_image(Image, WindowHandle)
texture_laws(Image, Texture1, 'es', 2, 5)
texture_laws(Image, Texture2, 'le', 2, 5)
```



```

mean_image(Texture1,H1,51,51)
mean_image(Texture2,H2,51,51)
fwrite_string(FileId,'mark desired image section')
fnew_line(FileId)
set_color(WindowHandle,'green')
draw_region(Region,WindowHandle)
reduce_domain(H1,Region,Foreground1)
reduce_domain(H2,Region,Foreground2)
histo_2dim(Region,Foreground1,Foreground2,Histo)
threshold(Histo,Characteristic_area,1,1000000)
set_color(WindowHandle,'blue')
disp_region(Characteristic_area,WindowHandle)
class_2dim_sup(H1,H2,Characteristic_area,Seg,4,5)
set_color(WindowHandle,'red')
disp_region(Seg,WindowHandle).

```

Result

`texture_laws` returns 2 (H_MSG.TRUE) if all parameters are correct. If the input is empty the behaviour can be set via `set_system('no_object_result', <Result>)`. If necessary, an exception handling is raised.

Parallelization Information

`texture_laws` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Possible Successors

`mean_image`, `gauss_image`, `median_image`, `histo_2dim`, `learn_ndim_norm`, `learn_ndim_box`, `threshold`

Alternatives

`convol_image`

See also

`class_2dim_sup`, `class_ndim_norm`

References

Laws, K.I. "Textured image segmentation"; Ph.D. dissertation, Dept. of Engineering, Univ. Southern California, 1980

Module

Image filters

5.14 Wiener-Filter

gen_psf_defocus (: Psf : PSFwidth, PSFheight, Blurring :)

Generate an impulse response of an uniform out-of-focus blurring.

`gen_psf_defocus` generates an impulse response (spatial domain) of an uniform out-of-focus blurring and writes it into an image of HALCON image type 'real'. `Blurring` specifies the extent of blurring by defining the "blur radius" (out-of-focus blurring maps each image pixel on a small circle with a radius of `Blurring` - specified in "number of pixels"). If specified less than zero, the absolute value of `Blurring` is used. The result image of `gen_psf_defocus` encloses an spatial domain impulse response of the specified blurring. Its representation presumes the origin in the upper left corner. This results in the following disposition of an $N \times M$ sized image:

- first rectangle ("upper left"): (image coordinates $xb = 0..(N/2) - 1$, $yb = 0..(M/2) - 1$)
- conforms to the fourth quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = 0..N/2$ and $y = 0..M/2$
- second rectangle ("upper right"): (image coordinates $xb = N/2..N - 1$, $yb = 0..(M/2) - 1$)
- conforms to the third quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = -N/2..-1$ and $y = -1..-M/2$
- third rectangle ("lower left"): (image coordinates $xb = 0..(N/2) - 1$, $yb = M/2..M - 1$)
- conforms to the first quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = 1..N/2$ and $y = M/2..0$

- fourth rectangle (“lower right”): (image coordinates $xb = N/2..N - 1$, $yb = M/2..M - 1$)
- conforms to the second quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = -N/2.. - 1$ und $y = M/2..1$

This representation conforms to that of the impulse response parameter of the HALCON-operator [wiener_filter](#). So one can use [gen_psf_defocus](#) to generate an impulse response for Wiener filtering.

Parameter	
▷ Psf (output_object)	image $\leadsto$ <i>Hobject</i> : real Impuls response of uniform out-of-focus blurring.
▷ PSFwidth (input_control)	integer $\leadsto$ <i>integer</i> Width of result image. Default Value : 256 Suggested values : PSFwidth $\in \{128, 256, 512, 1024\}$ Typical range of values : $1 \leq \text{PSFwidth}$
▷ PSFheight (input_control)	integer $\leadsto$ <i>integer</i> Height of result image. Default Value : 256 Suggested values : PSFheight $\in \{128, 256, 512, 1024\}$ Typical range of values : $1 \leq \text{PSFheight}$
▷ Blurring (input_control)	real $\leadsto$ <i>real</i> Degree of Blurring. Default Value : 5.0 Suggested values : Blurring $\in \{1.0, 5.0, 10.0, 15.0, 18.0\}$

Result
[gen_psf_defocus](#) returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information
[gen_psf_defocus](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors
[simulate_motion](#), [gen_psf_motion](#)

Possible Successors
[simulate_defocus](#), [wiener_filter](#), [wiener_filter_ni](#)

See also
[simulate_defocus](#), [gen_psf_motion](#), [simulate_motion](#), [wiener_filter](#),
[wiener_filter_ni](#)

References
Reginald L. Lagendijk, Jan Biemond: Iterative Identification and Restoration of Images, Kluwer Academic Publishers Boston/Dordrecht/London, 1991
M. L"uckenhaus:"Grundlagen des Wiener-Filters und seine Anwendung in der Bildanalyse"; Diplomarbeit; Technische Universit"at M"unchen, Institut f"ur Informatik; Lehrstuhl Prof. Radig; 1995.

Module
Wiener filter

gen_psf_motion (: Psf : PSFwidth, PSFheight, Blurring, Angle, Type :)

Generate an impulse response of a (linearly) motion blurring.

[gen_psf_motion](#) generates an impulse response (spatial domain) of a blurring caused by a relative motion between the object and the camera during exposure. The generated impulse response is output into an image of HALCON image type 'real'. [PSFwidth](#) and [PSFheight](#) define the width and height of the output image. The blurring motion moves along an even. [Angle](#) fixes its direction by specifying the angle between the motion direction and the x-axis (anticlockwise, measured in degrees). To specify different velocity behaviour five PSF prototypes can be generated. [Type](#) switches between the following prototypes:

1. reverse ramp (crude model for acceleration)

2. reverse trapezoid (crude model for high acceleration)
3. square pulse (exact model for constant velocity), this is default
4. forward trapezoid (crude model for deceleration)
5. forward ramp (crude model for high deceleration)

The blurring affects all part of the image uniformly. **Blurring** controls the extent of blurring. It specifies the number of pixels (lying one after another) that are affected by the blurring. This number is determined by velocity of the motion and exposure time. If **Blurring** is a negative number, an adequate blurring in reverse direction is simulated. If **Angle** is a negative number, it is interpreted clockwise. If **Angle** exceeds 360 or falls below -360, it is transformed modulo(360) in an adequate number between $[0..360]$ resp. $[-360..0]$. The result image of **gen_psf_motion** encloses an spatial domain impulse response of the specified blurring. Its representation presumes the origin in the upper left corner. This results in the following disposition of an $N \times M$ sized image:

- first rectangle ("upper left"): (image coordinates $xb = 0..(N/2) - 1$, $yb = 0..(M/2) - 1$)
- conforms to the fourth quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = 0..N/2$ and $y = 0..M/2$
- second rectangle ("upper right"): (image coordinates $xb = N/2..N - 1$, $yb = 0..(M/2) - 1$)
- conforms to the third quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = -N/2..-1$ and $y = -1..-M/2$
- third rectangle ("lower left"): (image coordinates $xb = 0..(N/2) - 1$, $yb = M/2..M - 1$)
- conforms to the first quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = 1..N/2$ and $y = M/2..0$
- fourth rectangle ("lower right"): (image coordinates $xb = N/2..N - 1$, $yb = M/2..M - 1$)
- conforms to the second quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = -N/2..-1$ and $y = M/2..1$

This representation conforms to that of the impulse response parameter of the HALCON-operator **wiener_filter**. So one can use **gen_psf_motion** to generate an impulse response for Wiener filtering a motion blurred image.

Parameter

- ▷ **Psf** (output_object) image $\leadsto$ *Hobject*: real
Impulse response of motion-blur.
- ▷ **PSFwidth** (input_control) integer $\leadsto$ *integer*
Width of impulse response image.
Default Value : 256
Suggested values : $PSFwidth \in \{128, 256, 512, 1024\}$
Typical range of values : $1 \leq PSFwidth$
- ▷ **PSFheight** (input_control) integer $\leadsto$ *integer*
Height of impulse response image.
Default Value : 256
Suggested values : $PSFheight \in \{128, 256, 512, 1024\}$
Typical range of values : $1 \leq PSFheight$
- ▷ **Blurring** (input_control) real $\leadsto$ *real*
Degree of motion-blur.
Default Value : 20.0
Suggested values : $Blurring \in \{5.0, 10.0, 20.0, 30.0, 40.0\}$
- ▷ **Angle** (input_control) integer $\leadsto$ *integer*
Angle between direction of motion and x-axis (anticlockwise).
Default Value : 0
Suggested values : $Angle \in \{0, 45, 90, 180, 270\}$
- ▷ **Type** (input_control) integer $\leadsto$ *integer*
PSF prototype resp. type of motion.
Default Value : 3
List of values : $Type \in \{1, 2, 3, 4, 5\}$

Result

gen_psf_motion returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

[gen_psf_motion](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_psf_motion](#), [simulate_defocus](#), [gen_psf_defocus](#)

Possible Successors

[simulate_motion](#), [wiener_filter](#), [wiener_filter_ni](#)

See also

[simulate_motion](#), [simulate_defocus](#), [gen_psf_defocus](#), [wiener_filter](#),
[wiener_filter_ni](#)

References

Anil K. Jain: Fundamentals of Digital Image Processing, Prentice-Hall International Inc., Englewood Cliffs, New Jersey, 1989

M. L"uckenhaus: "Grundlagen des Wiener-Filters und seine Anwendung in der Bildanalyse"; Diplomarbeit; Technische Universit"at M"unchen, Institut f"ur Informatik; Lehrstuhl Prof. Radig; 1995.

Kha-Chye Tan, Hock Lim, B. T. G. Tan: "Restoration of Real-World Motion-Blurred Images"; S. 291-299 in: CVGIP Graphical Models and Image Processing, Vol. 53, No. 3, May 1991

Module

Wiener filter

simulate_defocus (Image : DefocusedImage : Blurring :)

Simulate an uniform out-of-focus blurring of an image.

[simulate_defocus](#) simulates out-of-focus blurring of an image. All parts of the image are blurred uniformly. [Blurring](#) specifies the extent of blurring by defining the "blur radius" (out-of-focus blurring maps each image pixel on a small circle with a radius of [Blurring](#) - specified in "number of pixels"). If specified less than null, the absolute value of [Blurring](#) is used. Simulation of blurring is done by a convolution of the image with a blurring specific impulse response. The convolution is realized by multiplication in the Fourier domain.

Attention

As the simulation of the blurring is realized based on the Fourier transform, image height and width must be a power of 2, e.g. 128, 128, 256, 512,...

Parameter

- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Image to blur.
- ▷ **DefocusedImage** (output_object) image $\leadsto$ *Hobject* : real
Blurred image.
- ▷ **Blurring** (input_control) real $\leadsto$ *real*
Degree of blurring.
Default Value : 5.0
Suggested values : [Blurring](#) $\in$ {1.0, 5.0, 10.0, 15.0, 18.0}

Result

[simulate_defocus](#) returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty [simulate_defocus](#) returns with an error message.

Parallelization Information

[simulate_defocus](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_psf_defocus](#), [simulate_motion](#), [gen_psf_motion](#)

Possible Successors

[wiener_filter](#), [wiener_filter_ni](#)

See also

[gen_psf_defocus](#), [simulate_motion](#), [gen_psf_motion](#)

References

Reginald L. Lagendijk, Jan Biemond: Iterative Identification and Restoration of Images, Kluwer Academic Publishers Boston/Dordrecht/London, 1991

M. L"uckenhaus:"Grundlagen des Wiener-Filters und seine Anwendung in der Bildanalyse"; Diplomarbeit; Technische Universit"at M"unchen, Institut f"ur Informatik; Lehrstuhl Prof. Radig; 1995.

Module

Wiener filter

simulate_motion (Image : MovedImage : Blurring, Angle, Type :)

Simulation of (linearly) motion blur.

simulate_motion simulates blurring caused by a relative motion between the object and the camera during exposure. The simulated motion moves along an even. **Angle** fixes its direction by specifying the angle between the motion direction and the x-axis (anticlockwise, measured in degrees). Simulation is done by a convolution of the image with a blurring specific impulse response. The convolution is realized by multiplication in the Fourier domain. **simulate_motion** offers five prototypes of impulse responses conforming to different acceleration behaviours. **Type** allows to choose one of the following PSF prototypes:

1. reverse ramp (crude model for acceleration)
2. reverse trapezoid (crude model for high acceleration)
3. square pulse (exact model for constant velocity), this is default
4. forward trapezoid (crude model for deceleration)
5. forward ramp (crude model for high deceleration)

The simulated blurring affects all part of the image uniformly. **Blurring** controls the extent of blurring. It specifies the number of pixels (lying one after another) that are affected by the blurring. This number is determined by velocity of the motion and exposure time. If **Blurring** is a negative number, an adequate blurring in reverse direction is simulated. If **Angle** is a negative number, it is interpreted clockwise. If **Angle** exceeds 360 or falls below -360, it is transformed modulo(360) in an adequate number between [0..360] resp. [-360..0].

Attention

As the simulation of the blurring is realized based on the Fourier transform, image height and width must be a power of 2, e.g. 128, 128, 256, 512,...

Parameter

- ▷ **Image** (input_object) image $\rightsquigarrow$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
image to be blurred.
- ▷ **MovedImage** (output_object) image $\rightsquigarrow$ *Hobject* : real
motion blurred image.
- ▷ **Blurring** (input_control) real $\rightsquigarrow$ *real*
extent of blurring.
Default Value : 20.0
Suggested values : Blurring $\in$ {5.0, 10.0, 20.0, 30.0, 40.0}
- ▷ **Angle** (input_control) integer $\rightsquigarrow$ *integer*
Angle between direction of motion and x-axis (anticlockwise).
Default Value : 0
Suggested values : Angle $\in$ {0, 45, 90, 180, 270}
- ▷ **Type** (input_control) integer $\rightsquigarrow$ *integer*
impulse response of motion blur.
Default Value : 3
List of values : Type $\in$ {1, 2, 3, 4, 5}

Result

simulate_motion returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty **simulate_motion** returns with an error message.

<i>Parallelization Information</i>
<code>simulate_motion</code> is <i>reentrant</i> and processed <i>without parallelization</i> .
<i>Possible Predecessors</i>
<code>gen_psf_motion</code> , <code>gen_psf_motion</code>
<i>Possible Successors</i>
<code>simulate_defocus</code> , <code>wiener_filter</code> , <code>wiener_filter_ni</code>
<i>See also</i>
<code>gen_psf_motion</code> , <code>simulate_defocus</code> , <code>gen_psf_defocus</code>
<i>References</i>
Anil K. Jain: Fundamentals of Digital Image Processing, Prentice-Hall International Inc., Englewood Cliffs, New Jersey, 1989
M. L"uckenhaus: "Grundlagen des Wiener-Filters und seine Anwendung in der Bildanalyse"; Diplomarbeit; Technische Universit"at M"unchen, Institut f"ur Informatik; Lehrstuhl Prof. Radig; 1995.
Kha-Chye Tan, Hock Lim, B. T. G. Tan: "Restoration of Real-World Motion-Blurred Images"; S. 291-299 in: CVGIP Graphical Models and Image Processing, Vol. 53, No. 3, May 1991
<i>Module</i>
Wiener filter

wiener_filter (Image, Psf, FilteredImage : RestoredImage : :)

Image restoration by Wiener filtering.

`wiener_filter` produces an estimate of the original image (= image without noise and blurring) by minimizing the mean square error between estimated and original image. `wiener_filter` can be used to restore images corrupted by noise and/or blurring (e.g. motion blur, atmospheric turbulence or out-of-focus blur). Method and realisation of this restoration technique bases on the following model: The corrupted image is interpreted as the output of a (disturbed) linear system. Functionality of a linear system is determined by its specific impuls response. So the convolution of original image and impulse response results in the corrupted image. The specific impulse response describes image acquisition and the occurred degradations. In the presence of additive noise an additional noise term must be considered. So the corrupted image can be modeled as the result of

$[convolution(impulse\_response, original\_image)] + noise\_term$

The noise term encloses two different terms describing image-dependent and image-independent noise. According to this model, two terms must be known for restoration by Wiener filtering:

1. degradation-specific impulse response
2. noise term

So `wiener_filter` needs a smoothed version of the input image to estimate the power spectral density of noise and original image. One can use one of the smoothing HALCON-filters (e.g. `eliminate_min_max`) to get this version. `wiener_filter` needs further the impulse response that describes the specific degradation. This impulse response (represented in spatial domain) must fit into an image of HALCON image type 'real'. There exist two HALCON-operators for generation of an impulse response for motion blur and out-of-focus (see `gen_psf_motion`, `gen_psf_defocus`). The representation of the impulse response presumes the origin in the upper left corner. This results in the following disposition of an $N \times M$ sized image:

- first rectangle ("upper left"): (image coordinates $xb = 0..(N/2) - 1$, $yb = 0..(M/2) - 1$)
- conforms to the fourth quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = 0..N/2$ and $y = 0..M/2$
- second rectangle ("upper right"): (image coordinates $xb = N/2..N - 1$, $yb = 0..(M/2) - 1$)
- conforms to the third quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = -N/2..-1$ and $y = -1..-M/2$
- third rectangle ("lower left"): (image coordinates $xb = 0..(N/2) - 1$, $yb = M/2..M - 1$)
- conforms to the first quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = 1..N/2$ and $y = M/2..0$

- fourth rectangle (“lower right”): (image coordinates $xb = N/2..N - 1$, $yb = M/2..M - 1$)
- conforms to the second quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = -N/2..-1$ und $y = M/2..1$

As the Wiener filter is realized using the Fourier transform, only images with height and width fitting a power of 2 can be processed. [wiener_filter](#) works as follows:

- estimation of the power spectrum density of the original image by using the smoothed version of the corrupted image,
- estimation of the power spectrum density of each pixel by subtracting smoothed version from unsmoothed version,
- building the Wiener filter kernel with the quotient of power spectrum densities of noise and original image and with the impulse response,
- processing the convolution of image and Wiener filter frequency response.

The result image has got image type 'real'.

Attention

Psf must be of image type 'real' and conform to [Image](#) and [FilteredImage](#) in image width and height.

Parameter

- ▷ **Image** (input_object) image $\leadsto$ *Hobject*: byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Corrupted image.
- ▷ **Psf** (input_object) image $\leadsto$ *Hobject*: real
impulse response (PSF) of degradation (in spatial domain).
- ▷ **FilteredImage** (input_object) image $\leadsto$ *Hobject*: byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Smoothed version of corrupted image.
- ▷ **RestoredImage** (output_object) image $\leadsto$ *Hobject*: real
Restored image.

Example

```
/* Restoration of a noisy image (size=256x256), that was blurred by motion*/
Hobject object;
Hobject restored;
Hobject psf;
Hobject noisefiltered;
/* 1. Generate a Point-Spread-Function for a motion-blur with          */
/*    parameter a=10 and direction along the x-axis                    */
gen_psf(&psf,256,256,10,0,3);
/* 2. Noisefiltering of the image                                     */
median_image(object,&noisefiltered,"circle",2,0);
/* 3. Wiener-filtering                                              */
wiener_filter(object,psf,noisefiltered,&restored);
```

Result

[wiener_filter](#) returns 2 (H_MSG.TRUE) if all parameters are correct. If the input is empty [wiener_filter](#) returns with an error message.

Parallelization Information

[wiener_filter](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_psf_motion](#), [simulate_motion](#), [simulate_defocus](#), [gen_psf_defocus](#)

Alternatives

[wiener_filter_ni](#)

See also

[simulate_motion](#), [gen_psf_motion](#), [simulate_defocus](#), [gen_psf_defocus](#)

References

M. L"uckenhaus:"Grundlagen des Wiener-Filters und seine Anwendung in der Bildanalyse"; Diplomarbeit; Technische Universit"at M"unchen, Institut f"ur Informatik; Lehrstuhl Prof. Radig; 1995

Azriel Rosenfeld, Avinash C. Kak: Digital Picture Processing, Computer Science and Aplied Mathematics, Academic Press New York/San Francisco/London 1982

Module

Wiener filter

wiener_filter_ni (Image, Psf,
NoiseRegion : RestoredImage : MaskWidth, MaskHeight :)

Image restoration by Wiener filtering.

wiener_filter_ni (ni = noise-estimation integrated) produces an estimate of the original image (= image without noise and blurring) by minimizing the mean square error between estimated and original image. **wiener_filter** can be used to restore images corrupted by noise and/or blurring (e.g. motion blur, atmospheric turbulence or out-of-focus blur). Method and realisation of this restoration technique bases on the following model: The corrupted image is interpreted as the output of a (disturbed) linear system. Functionality of a linear system is determined by its specific impuls response. So the convolution of original image and impulse response results in the corrupted image. The specific impulse response describes image acquisition and the occured degradations. In the presence of additive noise an additional noise term must be considered. So the corrupted image can be modeled as the result of

$[convolution(impulse\_response, original\_image)] + noise\_term$

The noise term encloses two different terms describing image-dependent and image-independent noise. According to this model, two terms must be known for restoration by Wiener filtering:

1. degradation-specific impulse response
2. noise term

wiener_filter_ni estimates the noise term as follows: The user defines a region that is suitable for noise estimation within the image (homogeneous as possible, as edges or textures aggravate noise estimation). After smoothing within this region by an (unweighted) median filter and subtracting smoothed version from unsmoothed, the average noise amplitude of the region is processed within **wiener_filter_ni**. This amplitude together with the average gray value within the region allows estimating the quotient of the power spectral densities of noise and original image (in contrast to **wiener_filter** **wiener_filter_ni** assumes a rather constant quotient within the whole image). The user can define width and height of the rectangular (median-)filter mask to influence the noise estimation (**MaskWidth**, **MaskHeight**). **wiener_filter_ni** needs further the impulse response that describes the specific degradation. This impulse response (represented in spatial domain) must fit into an image of HALCON image type 'real'. There exist two HALCON-operators for generation of an impulse response for motion blur and out-of-focus (see **gen_psf_motion**, **gen_psf_defocus**). The representation of the impulse response presumes the origin in the upper left corner. This results in the following disposition of an $N \times M$ sized image:

- first rectangle ("upper left"): (image coordinates $xb = 0..(N/2) - 1$, $yb = 0..(M/2) - 1$)
- conforms to the fourth quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = 0..N/2$ and $y = 0.. - M/2$
- second rectangle ("upper right"): (image coordinates $xb = N/2..N - 1$, $yb = 0..(M/2) - 1$)
- conforms to the third quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = -N/2.. - 1$ and $y = -1.. - M/2$
- third rectangle ("lower left"): (image coordinates $xb = 0..(N/2) - 1$, $yb = M/2..M - 1$)
- conforms to the first quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = 1..N/2$ and $y = M/2..0$
- fourth rectangle ("lower right"): (image coordinates $xb = N/2..N - 1$, $yb = M/2..M - 1$)
- conforms to the second quadrant of the Cartesian coordinate system, encloses values of the impulse response at position $x = -N/2.. - 1$ und $y = M/2..1$

As the Wiener filter is realized using the Fourier transform, only images with height and width fitting a power of 2 can be processed. `wiener_filter` works as follows:

- estimating the quotient of the power spectrum densities of noise and original image,
- building the Wiener filter kernel with the quotient of power spectrum densities of noise and original image and with the impulse response,
- processing the convolution of image and Wiener filter frequency response.

The result image has got image type 'real'.

Attention

`Psf` must be of image type 'real' and conform to `Image` in width and height. The Region used for noise estimation must lie completely within the image. If `MaskWidth` or `MaskHeight` is an even number, it is replaced by the next higher odd number (this allows the unique extraction of the center of the filter mask). Width/height of the mask may not exceed the image width/height or be less than null.

Parameter

- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Corrupted image.
- ▷ **Psf** (input_object) image $\leadsto$ *Hobject* : real
impulse response (PSF) of degradation (in spatial domain).
- ▷ **NoiseRegion** (input_object) region $\leadsto$ *Hobject*
Region for noise estimation.
- ▷ **RestoredImage** (output_object) image $\leadsto$ *Hobject* : real
Restored image.
- ▷ **MaskWidth** (input_control) integer $\leadsto$ *integer*
Width of filter mask.
Default Value : 3
Suggested values : `MaskWidth` $\in$ {3, 5, 7, 9}
Typical range of values : $0 \leq \text{MaskWidth} \leq \text{width}(\text{Image})$
- ▷ **MaskHeight** (input_control) integer $\leadsto$ *integer*
Height of filter mask.
Default Value : 3
Suggested values : `MaskHeight` $\in$ {3, 5, 7, 9}
Typical range of values : $0 \leq \text{MaskHeight} \leq \text{height}(\text{Image})$

Example

```
/* Restoration of a noisy image (size=256x256), that was blurred by motion*/
Hobject object;
Hobject restored;
Hobject psf;
Hobject noise_region;
/* 1. Generate a Point-Spread-Function for a motion-blur with          */
/*    parameter a=10 and direction of the x-axis                      */
gen_psf(&psf,256,256,10,0,3);
/* 2. Segmentation of a region for the noise-estimation                */
open_window(0,0,256,256,0,"visible",&WindowHandle);
disp_image(object,WindowHandle);
draw_region(&noise_region,draw_region);
/* 3. Wiener-filtering                                                */
wiener_filter_ni(object,psf,noise_region,&restored,3,3);
```

Result

`wiener_filter_ni` returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty `wiener_filter_ni` returns with an error message.

Parallelization Information

`wiener_filter_ni` is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_psf_motion](#), [simulate_motion](#), [simulate_defocus](#), [gen_psf_defocus](#)

Alternatives

[wiener_filter](#)

See also

[simulate_motion](#), [gen_psf_motion](#), [simulate_defocus](#), [gen_psf_defocus](#)

References

M. Lückenhaus: "Grundlagen des Wiener-Filters und seine Anwendung in der Bildanalyse"; Diplomarbeit; Technische Universität München, Institut für Informatik; Lehrstuhl Prof. Radig; 1995

Azriel Rosenfeld, Avinash C. Kak: Digital Picture Processing, Computer Science and Applied Mathematics, Academic Press New York/San Francisco/London 1982

Module

Wiener filter

Chapter 6

Graphics

6.1 Drawing

drag_region1 (SourceRegion : DestinationRegion : WindowHandle :)

Interactive moving of a region.

[drag_region1](#) is used to move a region on the display by mouse. Calling [drag_region1](#) turns the region visible as soon as the left mouse button is pressed. Therefore the region's edges are displayed only. As representation mode the mode 'not' (see [set_draw](#)) is used during procedure's permanence. During the movement the cursor resides in the region's barycenter. If you move the mouse with pressed left mouse button, the depicted region follows - delayed - this movement. If you press the right mouse button you terminate [drag_region1](#). The depicted region disappears from the display. Output is a region which corresponds to the last position on the display. You may pass even several regions at once. Procedure [affine_trans_image](#) moves the gray values.

Attention

Gray values of regions are not moved. With moving the input region it is not sure whether the gray values of the output regions are filled reasonable. This may occur if the gray values of the input regions do not comprise the whole image.

Parameter

- ▷ **SourceRegion** (input_object) region-array $\leadsto$ *Hobject*
Regions to move.
- ▷ **DestinationRegion** (output_object) region-array $\leadsto$ *Hobject*
Moved Regions.
- ▷ **WindowHandle** (input_control) window $\leadsto$ *integer*
Window_id.

Example

```
draw_region(Obj, WindowHandle)
drag_region1(Obj, New, WindowHandle)
disp_region(New, WindowHandle)
position(Obj, _, Row1, Column1, _, _, _, _)
position(New, _, Row2, Column2, _, _, _, _)
disp_arrow(WindowHandle, Row1, Column1, Row2, Column2, 1.0)
fwrite_string(['Transformation: (', Row2-Row1, ', ', Column2-Column1, ')'])
fnew_line().
```

Result

[drag_region1](#) returns 2 (H_MSG_TRUE), if a region is entered, the window is valid and the needed drawing mode (see [set_insert](#)) is available. If necessary, an exception handling is raised. You may determine the behavior after an empty input with [set_system\(:, 'no.object.result', <Result>:\)](#).

Parallelization Information

[drag_region1](#) is *reentrant*, *local*, and processed *without* parallelization.

	<i>Possible Predecessors</i>	
open_window		
	<i>Possible Successors</i>	
reduce_domain , disp_region , set_colored , set_line_width , set_draw , set_insert		
	<i>Alternatives</i>	
get_mposition , move_region		
	<i>See also</i>	
set_insert , set_draw , affine_trans_image		
	<i>Module</i>	
System		

drag_region2 (SourceRegion : DestinationRegion : WindowHandle, Row, Column :)

Interactive movement of a region with fixpoint specification.

You use [drag_region2](#) to move a region on the display by mouse. It corresponds to the procedure [drag_region1](#) with the difference, that the position of the mouse cursor can be determined.

	<i>Attention</i>	
Gray values of the regions are not moved. With moving the input region it is not sure whether the gray values of the output regions are filled reasonable. This may occur if the gray values of the input regions do not comprise the whole image.		

	<i>Parameter</i>	
▷ SourceRegion (input_object)	region-array $\leadsto$ <i>Hobject</i>	
Regions to move.		
▷ DestinationRegion (output_object)	region-array $\leadsto$ <i>Hobject</i>	
Moved regions.		
▷ WindowHandle (input_control)	window $\leadsto$ <i>integer</i>	
Window_id.		
▷ Row (input_control)	point.y $\leadsto$ <i>integer</i>	
Row index of the reference point.		
Default Value : 100		
Suggested values : Row $\in$ {0, 64, 128, 256, 512}		
Typical range of values : $0 \leq \text{Row} \leq 1024$		
▷ Column (input_control)	point.x $\leadsto$ <i>integer</i>	
Column index of the reference point.		
Default Value : 100		
Suggested values : Column $\in$ {0, 64, 128, 256, 512}		
Typical range of values : $0 \leq \text{Column} \leq 1024$		

	<i>Result</i>	
drag_region2 returns 2 (H_MSG_TRUE), if a region is entered, the window is valid and the needed drawing mode (see set_insert) is available. If necessary, an exception handling is raised. You may determine the behavior after an empty input with set_system (::'no.object.result', <Result>:).		

	<i>Parallelization Information</i>	
drag_region2 is <i>reentrant</i> , <i>local</i> , and processed <i>without parallelization</i> .		

	<i>Possible Predecessors</i>	
open_window		
	<i>Possible Successors</i>	
reduce_domain , disp_region , set_colored , set_line_width , set_draw , set_insert , affine_trans_image		
	<i>Alternatives</i>	
get_mposition , move_region , drag_region1 , drag_region3		
	<i>See also</i>	
set_insert , set_draw , affine_trans_image		

Module

System

drag_region3 (SourceRegion,
MaskRegion : DestinationRegion : WindowHandle, Row, Column :)

Interactive movement of a region with restriction of positions.

You use **drag_region3** to move a region on the display by mouse. It corresponds to the procedure **drag_region2** with the enhancement, that all points are specified which can be entered by mouse. If you move the mouse outside of this area (**MaskRegion**), the region on the point with the smallest distance inside **MaskRegion** will be displayed.

Attention

The region's gray values are not moved. With moving the input region it is not sure whether the gray values of the output regions are filled reasonable. This may occur if the gray values of the input regions do not comprise the whole image.

Parameter

- ▷ **SourceRegion** (input_object) region-array $\leadsto$ Hobject
Regions to move.
- ▷ **MaskRegion** (input_object) region-array $\leadsto$ Hobject
Points on which it is allowed for a region to move.
- ▷ **DestinationRegion** (output_object) region-array $\leadsto$ Hobject
Moved regions.
- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window_id.
- ▷ **Row** (input_control) point.y $\leadsto$ integer
Row index of the reference point.
Default Value : 100
Suggested values : Row $\in$ {0, 64, 128, 256, 512}
Typical range of values : $0 \leq \text{Row} \leq 1024$
- ▷ **Column** (input_control) point.x $\leadsto$ integer
Column index of the reference point.
Default Value : 100
Suggested values : Column $\in$ {0, 64, 128, 256, 512}
Typical range of values : $0 \leq \text{Column} \leq 1024$

Result

drag_region3 returns 2 (H_MSG_TRUE), if a region is entered, if the window is valid and the needed drawing mode (see **set_insert**) is available. If necessary, an exception handling is raised. You may determine the behavior after an empty input with **set_system (:: 'no.object.result', <Result>:)**.

Parallelization Information

drag_region3 is *reentrant*, *local*, and processed without parallelization.

Possible Predecessors

open_window, **get_mposition**

Possible Successors

reduce_domain, **disp_region**, **set_colored**, **set_line_width**, **set_draw**, **set_insert**, **affine_trans_image**

Alternatives

get_mposition, **move_region**, **drag_region1**, **drag_region2**

See also

set_insert, **set_draw**, **affine_trans_image**

Module

System

draw_circle (: : WindowHandle : Row, Column, Radius)

Interactive drawing of a circle.

draw_circle produces the parameter for a circle created interactive by the user in the window.

To create a circle you have to press the mouse button at the location which is used as the center of that circle. While keeping the mouse button pressed, the **Radius**'s length can be modified through moving the mouse. After another mouse click in the created circle center you can move it. A clicking close to the circular arc you can modify the **Radius** of the circle. Pressing the right mousebutton terminates the procedure. After terminating the procedure the circle is not visible in the window any longer.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window_id.
- ▷ **Row** (output_control) circle.center.y $\leadsto$ real
Barycenter's row index.
- ▷ **Column** (output_control) circle.center.x $\leadsto$ real
Barycenter's column index.
- ▷ **Radius** (output_control) circle.radius $\leadsto$ real
Circle's radius.

Example

```
read_image(Image, 'affe')
draw_circle(WindowHandle, Row, Column, Radius)
gen_circle(Circle, Row, Column, Radius, )
reduce_domain(Image, Circle, GrayCircle)
disp_image(GrayCircle, WindowHandle).
```

Result

draw_circle returns 2 (H_MSG_TRUE) if the window is valid and the needed drawing mode (see **set_insert**) is available. If necessary, an exception handling is raised.

Parallelization Information

draw_circle is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

open_window

Possible Successors

reduce_domain, **disp_region**, **set_colored**, **set_line_width**, **set_draw**, **set_insert**

Alternatives

draw_circle_mod, **draw_ellipse**, **draw_region**

See also

gen_circle, **draw_rectangle1**, **draw_rectangle2**, **draw_polygon**, **set_insert**

Module

System

draw_circle_mod (: : WindowHandle, RowIn, ColumnIn, RadiusIn : Row, Column, Radius)

Interactive drawing of a circle.

draw_circle_mod produces the parameter for a circle created interactive by the user in the window.

To create a circle are expected the coordinates **RowIn** and **ColumnIn** of the center of a circle with radius **RadiusIn**. After another mouse click in the created circle center you can move it. A clicking close to the circular arc you can modify the **Radius** of the circle. Pressing the right mousebutton terminates the procedure. After terminating the procedure the circle is not visible in the window any longer.

Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window_id.
▷ RowIn (input_control)	circle.center.y $\leadsto$ real Row index of the center.
▷ ColumnIn (input_control)	circle.center.x $\leadsto$ real Column index of the center.
▷ RadiusIn (input_control)	circle.radius1 $\leadsto$ real Radius of the circle.
▷ Row (output_control)	circle.center.y $\leadsto$ real Barycenter's row index.
▷ Column (output_control)	circle.center.x $\leadsto$ real Barycenter's column index.
▷ Radius (output_control)	circle.radius $\leadsto$ real Circle's radius.
Example	

```
read_image(Image, 'affe')
draw_circle_mod(WindowHandle, 20, 20, 15, Row, Column, Radius)
gen_circle(Circle, Row, Column, Radius, )
reduce_domain(Image, Circle, GrayCircle)
disp_image(GrayCircle, WindowHandle).
```

draw_circle_mod returns 2 (H_MSG_TRUE) if the window is valid and the needed drawing mode (see **set_insert**) is available. If necessary, an exception handling is raised.

draw_circle_mod is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

open_window

Possible Successors

reduce_domain, **disp_region**, **set_colored**, **set_line_width**, **set_draw**, **set_insert**

Alternatives

draw_circle, **draw_ellipse**, **draw_region**

See also

gen_circle, **draw_rectangle1**, **draw_rectangle2**, **draw_polygon**, **set_insert**

Module

System

draw_ellipse (: : WindowHandle : Row, Column, Phi, Radius1, Radius2)

Interactive drawing of an ellipse.

draw_ellipse returns the parameter for any orientated ellipse, which has been created interactively by the user in the window.

The created ellipse is described by its center, its two half axes and the angle between the first half axis and the horizontal coordinate axis.

To create an ellipse you have to determine the center of the ellipse with the left mouse button. Keeping the button pressed determines the length (**Radius1**) and the orientation (**Phi**) of the first half axis. In doing so a temporary default length for the second half axis is assumed, which may be modified afterwards on demand. After another mouse click in the center of the created ellipse you can move it. A mouse click close to a vertex “grips” it to modify the length of the appropriate half axis. You may modify the orientation only, if a vertex of the first half axis is gripped.

Pressing the right mouse button terminates the procedure. After terminating the procedure the ellipse is not visible in the window any longer.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window_id.
- ▷ **Row** (output_control) ellipse.center.y $\leadsto$ real
Row index of the center.
- ▷ **Column** (output_control) ellipse.center.x $\leadsto$ real
Column index of the center.
- ▷ **Phi** (output_control) ellipse.angle.rad $\leadsto$ real
Orientation of the first half axis in radians.
- ▷ **Radius1** (output_control) ellipse.radius1 $\leadsto$ real
First half axis.
- ▷ **Radius2** (output_control) ellipse.radius2 $\leadsto$ real
Second half axis.

Example

```
read_image(Image, 'affe')
draw_ellipse(WindowHandle, Row, Column, Phi, Radius1, Radius2)
gen_ellipse(Ellipse, Row, Column, Phi, Radius1, Radius2)
reduce_domain(Image, Ellipse, GrayEllipse)
sobel_amp(GrayEllipse, Sobel, 'sum_abs', 3)
disp_image(Sobel, WindowHandle).
```

Result

draw_ellipse returns 2 (H_MSG_TRUE), if the window is valid and the needed drawing mode (see **set_insert**) is available. If necessary, an exception handling is raised.

Parallelization Information

draw_ellipse is *reentrant*, *local*, and processed without parallelization.

Possible Predecessors

open_window

Possible Successors

reduce_domain, **disp_region**, **set_colored**, **set_line_width**, **set_draw**, **set_insert**

Alternatives

draw_ellipse_mod, **draw_circle**, **draw_region**

See also

gen_ellipse, **draw_rectangle1**, **draw_rectangle2**, **draw_polygon**, **set_insert**

Module

System

draw_ellipse_mod (: : WindowHandle, RowIn, ColumnIn, PhiIn, Radius1In, Radius2In : Row, Column, Phi, Radius1, Radius2)

Interactive drawing of an ellipse.

draw_ellipse_mod returns the parameter for any orientated ellipse, which has been created interactively by the user in the window.

The created ellipse is described by its center, its two half axes and the angle between the first half axis and the horizontal coordinate axis.

To create an ellipse are expected the parameters **RowIn**, **ColumnIn**, **PhiIn**, **Radius1In**, **Radius2In**. Keeping the button pressed determines the length (**Radius1**) and the orientation (**Phi**) of the first half axis. In doing so a temporary default length for the second half axis is assumed, which may be modified afterwards on demand. After another mouse click in the center of the created ellipse you can move it. A mouse click close to a vertex

“grips” it to modify the length of the appropriate half axis. You may modify the orientation only, if a vertex of the first half axis is gripped.

Pressing the right mouse button terminates the procedure. After terminating the procedure the ellipse is not visible in the window any longer.

Parameter

- ▷ **WindowHandle** (input_control) window $\rightsquigarrow$ integer
Window_id.
- ▷ **RowIn** (input_control) ellipse.center.y $\rightsquigarrow$ real
Row index of the barycenter.
- ▷ **ColumnIn** (input_control) ellipse.center.x $\rightsquigarrow$ real
Column index of the barycenter.
- ▷ **PhiIn** (input_control) ellipse.angle.rad $\rightsquigarrow$ real
Orientation of the bigger half axis in radians.
- ▷ **Radius1In** (input_control) ellipse.radius1 $\rightsquigarrow$ real
Bigger half axis.
- ▷ **Radius2In** (input_control) ellipse.radius1 $\rightsquigarrow$ real
Smaller half axis.
- ▷ **Row** (output_control) ellipse.center.y $\rightsquigarrow$ real
Row index of the center.
- ▷ **Column** (output_control) ellipse.center.x $\rightsquigarrow$ real
Column index of the center.
- ▷ **Phi** (output_control) ellipse.angle.rad $\rightsquigarrow$ real
Orientation of the first half axis in radians.
- ▷ **Radius1** (output_control) ellipse.radius1 $\rightsquigarrow$ real
First half axis.
- ▷ **Radius2** (output_control) ellipse.radius2 $\rightsquigarrow$ real
Second half axis.

Example

```
read_image(Image, 'affe')
draw_ellipse_mod(WindowHandle, RowIn, ColumnIn, PhiIn, Radius1In, Radius2In, Row, Column, Phi, Row, Column, Phi, Radius1, Radius2)
gen_ellipse(Ellipse, Row, Column, Phi, Radius1, Radius2)
reduce_domain(Image, Ellipse, GrayEllipse)
sobel_amp(GrayEllipse, Sobel, 'sum_abs', 3)
disp_image(Sobel, WindowHandle).
```

Result

[draw_ellipse_mod](#) returns 2 (H.MSG.TRUE), if the window is valid and the needed drawing mode (see [set_insert](#)) is available. If necessary, an exception handling is raised.

Parallelization Information

[draw_ellipse_mod](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[open_window](#)

Possible Successors

[reduce_domain](#), [disp_region](#), [set_colored](#), [set_line_width](#), [set_draw](#), [set_insert](#)

Alternatives

[draw_ellipse](#), [draw_circle](#), [draw_region](#)

See also

[gen_ellipse](#), [draw_rectangle1](#), [draw_rectangle2](#), [draw_polygon](#), [set_insert](#)

Module

System

draw_line (: : WindowHandle : Row1, Column1, Row2, Column2)

Draw a line.

draw_line returns the parameter for a line, which has been created interactively by the user in the window.

To create a line you have to press the left mouse button determining a start point of the line. While keeping the button pressed you may “drag” the line in any direction. After another mouse click in the middle of the created line you can move it. If you click on one end point of the created line, you may move this point. Pressing the right mousebutton terminates the procedure.

After terminating the procedure the line is not visible in the window any longer.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window_id.
- ▷ **Row1** (output_control) line.begin.y $\leadsto$ real
Row index of the first point of the line.
- ▷ **Column1** (output_control) line.begin.x $\leadsto$ real
Column index of the first point of the line.
- ▷ **Row2** (output_control) line.end.y $\leadsto$ real
Row index of the second point of the line.
- ▷ **Column2** (output_control) line.end.x $\leadsto$ real
Column index of the second point of the line.

Example

```
get_system('width',Width)
get_system('height',Height)
set_part(WindowHandle,0,0,Width-1,Height-1)
read_image(Image,'affe')
disp_image(Image,WindowHandle)
draw_line(WindowHandle,Row1,Column1,Row2,Column2)
set_part(WindowHandle,Row1,Column1,Row2,Column2)
disp_image(Image,WindowHandle)
fwrite_string(['Clipping = (',Row1,',',',',Column1,')'])
fwrite_string(['(',',',Row2,',',',',Column2,')'])
fnew_line().
```

Result

draw_line returns 2 (H_MSG_TRUE), if the window is valid and the needed drawing mode (see [set_insert](#)) is available. If necessary, an exception handling is raised.

Parallelization Information

draw_line is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[open_window](#)

Possible Successors

[reduce_domain](#), [disp_line](#), [set_colored](#), [set_line_width](#), [set_draw](#), [set_insert](#)

See also

[draw_line_mod](#), [gen_rectangle1](#), [draw_circle](#), [draw_ellipse](#), [set_insert](#)

Module

System

draw_line_mod (: : WindowHandle, Row1In, Column1In, Row2In, Column2In : Row1, Column1, Row2, Column2)

Draw a line.

draw_line_mod returns the parameter for a line, which has been created interactively by the user in the window. To create a line are expected the coordinates of the start point **Row1In**, **Column1In** and of the end point **Row2In**, **Column2In**. If you click on one end point of the created line, you may move this point. After another mouse click in the middle of the created line you can move it.

Pressing the right mousebutton terminates the procedure.

After terminating the procedure the line is not visible in the window any longer.

<i>Parameter</i>	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window_id.
▷ Row1In (input_control)	line.begin.y $\leadsto$ real Row index of the first point of the line.
▷ Column1In (input_control)	line.begin.x $\leadsto$ real Column index of the first point of the line.
▷ Row2In (input_control)	line.end.y $\leadsto$ real Row index of the second point of the line.
▷ Column2In (input_control)	line.end.x $\leadsto$ real Column index of the second point of the line.
▷ Row1 (output_control)	line.begin.y $\leadsto$ real Row index of the first point of the line.
▷ Column1 (output_control)	line.begin.x $\leadsto$ real Column index of the first point of the line.
▷ Row2 (output_control)	line.end.y $\leadsto$ real Row index of the second point of the line.
▷ Column2 (output_control)	line.end.x $\leadsto$ real Column index of the second point of the line.

Example

```
get_system('width',Width)
get_system('height',Height)
set_part(WindowHandle,0,0,Width-1,Height-1)
read_image(Image,'affe')
disp_image(Image,WindowHandle)
draw_line_mod(WindowHandle,10,20,55,124,Row1,Column1,Row2,Column2)
set_part(WindowHandle,Row1,Column1,Row2,Column2)
disp_image(Image,WindowHandle)
fwrite_string(['Clipping = (' ,Row1,',',',Column1,')'])
fwrite_string(['(',',',Row2,',',',Column2,')'])
fnew_line().
```

Result

draw_line_mod returns 2 (H_MSG_TRUE), if the window is valid and the needed drawing mode is available. If necessary, an exception handling is raised.

Parallelization Information

draw_line_mod is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

open_window

Possible Successors

reduce_domain, **disp_line**, **set_colored**, **set_line_width**, **set_draw**, **set_insert**

Alternatives

draw_line, **draw_ellipse**, **draw_region**

See also

gen_circle, **draw_rectangle1**, **draw_rectangle2**

Module

System

draw_point (: : WindowHandle : Row, Column)

Draw a point.

draw_point returns the parameter for a point, which has been created interactively by the user in the window.

To create a point you have to press the left mouse button. While keeping the button pressed you may “drag” the point in any direction. Pressing the right mousebutton terminates the procedure.

After terminating the procedure the point is not visible in the window any longer.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window_id.
- ▷ **Row** (output_control) point.y $\leadsto$ real
Row index of the point.
- ▷ **Column** (output_control) point.x $\leadsto$ real
Column index of the point.

Example

```
get_system('width',Width)
get_system('height',Height)
set_part(WindowHandle,0,0,Width-1,Height-1)
read_image(Image,'affe')
disp_image(Image,WindowHandle)
draw_point(WindowHandle,Row1,Column1)
disp_line(WindowHandle,Row1-2,Column1,Row1+2,Column1)
disp_line(WindowHandle,Row1,Column1-2,Row1,Column1+2)
disp_image(Image,WindowHandle)
fwrite_string(['Clipping = (',Row1,',',Column1,')'])
fnew_line().
```

Result

draw_point returns 2 (H_MSG_TRUE), if the window is valid and the needed drawing mode is available. If necessary, an exception handling is raised.

Parallelization Information

draw_point is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

open_window

Possible Successors

reduce_domain, **disp_line**, **set_colored**, **set_line_width**, **set_draw**, **set_insert**

See also

draw_point_mod, **draw_circle**, **draw_ellipse**, **set_insert**

Module

System

draw_point_mod (: : WindowHandle, RowIn, ColumnIn : Row, Column)

Draw a point.

draw_point_mod returns the parameter for a point, which has been created interactively by the user in the window.

To create a point are expected the coordinates **RowIn** and **ColumnIn**. While keeping the button pressed you may “drag” the point in any direction. Pressing the right mousebutton terminates the procedure.

After terminating the procedure the point is not visible in the window any longer.

Parameter
▷ WindowHandle (input_control) window $\leadsto$ integer Window_id.
▷ RowIn (input_control) point.y $\leadsto$ real Row index of the point.
▷ ColumnIn (input_control) point.x $\leadsto$ real Column index of the point.
▷ Row (output_control) point.y $\leadsto$ real Row index of the point.
▷ Column (output_control) point.x $\leadsto$ real Column index of the point.
Example
<pre> get_system('width',Width) get_system('height',Height) set_part(WindowHandle,0,0,Width-1,Height-1) read_image(Image,'affe') disp_image(Image,WindowHandle) draw_point_mod(WindowHandle,Row1,Column1) disp_line(WindowHandle,Row1-2,Column1,Row1+2,Column1) disp_line(WindowHandle,Row1,Column1-2,Row1,Column1+2) disp_image(Image,WindowHandle) fwrite_string(['Clipping = (',Row1,',',Column1,')']) fnew_line(). </pre>
Result
draw_point_mod returns 2 (H_MSG_TRUE), if the window is valid and the needed drawing mode is available. If necessary, an exception handling is raised.
Parallelization Information
draw_point_mod is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.
Possible Predecessors
open_window
Possible Successors
reduce_domain , disp_line , set_colored , set_line_width , set_draw , set_insert
See also
draw_point , draw_circle , draw_ellipse , set_insert
Module
System

draw_polygon (: PolygonRegion : WindowHandle :)

Interactive drawing of a polygon row.

draw_polygon produces an image. The region of that image spans exactly the imagepoints entered interactively by mouse clicks (gray values remain undefined).

Painting in the output window happens while pressing the left mouse button. Releasing the left mouse button and repressing it at another position effects drawing a line between these two points. Pressing the right mouse button terminates the input. Painting uses that color which has been set by **set_color**, **set_rgb**, etc. .

To put gray values on the created **PolygonRegion** for further processing, you may use the procedure **reduce_domain**.

Attention

The painted contour is not closed automatically, in particular it is not “filled up” either.

Output object’s gray values are not defined.

Parameter
▷ PolygonRegion (output_object) region $\leadsto$ <i>Hobject</i> Region, which encompasses all painted points. ▷ WindowHandle (input_control) window $\leadsto$ <i>integer</i> Window_id.
Example
<pre>draw_polygon(Polygon, WindowHandle) convex(Polygon, Filled) disp_region(Filled, WindowHandle).</pre>
Result
If the window is valid, draw_polygon returns 2 (H_MSG_TRUE). If necessary, an exception handling is raised.
Parallelization Information
draw_polygon is <i>reentrant</i> , <i>local</i> , and processed without parallelization.
Possible Predecessors
open_window
Possible Successors
reduce_domain , disp_region , set_colored , set_line_width , set_draw
Alternatives
draw_region , draw_circle , draw_rectangle1 , draw_rectangle2 , boundary
See also
reduce_domain , fill_up , set_color
Module
System

draw_rectangle1 (: : WindowHandle : Row1, Column1, Row2, Column2)

Draw a rectangle parallel to the coordinate axis.

draw_rectangle1 returns the parameter for a rectangle parallel to the coordinate axes, which has been created interactively by the user in the window.

To create a rectangle you have to press the left mouse button determining a corner of the rectangle. While keeping the button pressed you may “drag” the rectangle in any direction. After another mouse click in the middle of the created rectangle you can move it. A click close to one side “grips” it to modify the rectangle’s dimension in perpendicular direction to this side. If you click on one corner of the created rectangle, you may move this corner. Pressing the right mousebutton terminates the procedure.

After terminating the procedure the rectangle is not visible in the window any longer.

Parameter
▷ WindowHandle (input_control) window $\leadsto$ <i>integer</i> Window_id. ▷ Row1 (output_control) rectangle.origin.y $\leadsto$ <i>real</i> Row index of the left upper corner. ▷ Column1 (output_control) rectangle.origin.x $\leadsto$ <i>real</i> Column index of the left upper corner. ▷ Row2 (output_control) rectangle.corner.y $\leadsto$ <i>real</i> Row index of the right lower corner. ▷ Column2 (output_control) rectangle.corner.x $\leadsto$ <i>real</i> Column index of the right lower corner.

Example

```

get_system('width',Width)
get_system('height',Height)
set_part(WindowHandle,0,0,Width-1,Height-1)
read_image(Image,'affe')
disp_image(Image,WindowHandle)
draw_rectangle1(WindowHandle,Row1,Column1,Row2,Column2)
set_part(WindowHandle,Row1,Column1,Row2,Column2)
disp_image(Image,WindowHandle)
fwrite_string(['Clipping = (' ,Row1,' ,',Column1,')'])
fwrite_string([' ,(' ,Row2,' ,',Column2,')'])
fnew_line().

```

Result

[draw_rectangle1](#) returns 2 (H_MSG_TRUE), if the window is valid and the needed drawing mode (see [set_insert](#)) is available. If necessary, an exception handling is raised.

Parallelization Information

[draw_rectangle1](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[open_window](#)

Possible Successors

[reduce_domain](#), [disp_region](#), [set_colored](#), [set_line_width](#), [set_draw](#), [set_insert](#)

Alternatives

[draw_rectangle1_mod](#), [draw_rectangle2](#), [draw_region](#)

See also

[gen_rectangle1](#), [draw_circle](#), [draw_ellipse](#), [set_insert](#)

Module

System

draw_rectangle1_mod (: : WindowHandle, Row1In, Column1In, Row2In, Column2In : Row1, Column1, Row2, Column2)

Draw a rectangle parallel to the coordinate axis.

[draw_rectangle1_mod](#) returns the parameter for a rectangle parallel to the coordinate axes, which has been created interactively by the user in the window.

To create a rectangle are expected the parameters [Row1In](#), [Column1In](#), [Row2In](#) und [Column2In](#). After a mouse click in the middle of the created rectangle you can move it. A click close to one side “grips” it to modify the rectangle’s dimension in perpendicular direction to this side. If you click on one corner of the created rectangle, you may move this corner. Pressing the right mousebutton terminates the procedure.

After terminating the procedure the rectangle is not visible in the window any longer.

Parameter

- ▷ **WindowHandle** (input_control) window $\rightsquigarrow$ *integer*
Window_id.
- ▷ **Row1In** (input_control) rectangle.origin.y $\rightsquigarrow$ *real*
Row index of the left upper corner.
- ▷ **Column1In** (input_control) rectangle.origin.x $\rightsquigarrow$ *real*
Column index of the left upper corner.
- ▷ **Row2In** (input_control) rectangle.corner.y $\rightsquigarrow$ *real*
Row index of the right lower corner.
- ▷ **Column2In** (input_control) rectangle.corner.x $\rightsquigarrow$ *real*
Column index of the right lower corner.
- ▷ **Row1** (output_control) rectangle.origin.y $\rightsquigarrow$ *real*
Row index of the left upper corner.

- ▷ **Column1** (output_control) rectangle.origin.x $\leadsto$ *real*
Column index of the left upper corner.
- ▷ **Row2** (output_control) rectangle.corner.y $\leadsto$ *real*
Row index of the right lower corner.
- ▷ **Column2** (output_control) rectangle.corner.x $\leadsto$ *real*
Column index of the right lower corner.

Example

```

get_system('width',Width)
get_system('height',Height)
set_part(WindowHandle,0,0,Width-1,Height-1)
read_image(Image,'affe')
disp_image(Image,WindowHandle)
draw_rectangle1_mod(WindowHandle,Row1In,Column1In,Row2In,Column2In,Row1,Column1,Row2,Column2)
set_part(WindowHandle,Row1,Column1,Row2,Column2)
disp_image(Image,WindowHandle)
fwrite_string(['Clipping = (',Row1,',',',',Column1,',')'])
fwrite_string(['(',',',Row2,',',',',Column2,',')'])
fnew_line().

```

Result

draw_rectangle1_mod returns 2 (H_MSG_TRUE), if the window is valid and the needed drawing mode (see **set_insert**) is available. If necessary, an exception handling is raised.

Parallelization Information

draw_rectangle1_mod is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

open_window

Possible Successors

reduce_domain, **disp_region**, **set_colored**, **set_line_width**, **set_draw**, **set_insert**

Alternatives

draw_rectangle1, **draw_rectangle2**, **draw_region**

See also

gen_rectangle1, **draw_circle**, **draw_ellipse**, **set_insert**

Module

System

draw_rectangle2 (: : WindowHandle : Row, Column, Phi, Length1, Length2)

Interactive drawing of any orientated rectangle.

draw_rectangle2 returns the parameter for any orientated rectangle, which has been created interactively by the user in the window.

The created rectangle is described by its center, its two half axes and the angle between the first half axis and the horizontal coordinate axis.

To create a rectangle you have to press the left mouse button for the center of the rectangle. While keeping the button pressed you may dimension the length (**Length1**) and the orientation (**Phi**) of the first half axis. In doing so a temporary default length for the second half axis is assumed, which may be modified afterwards on demand. After another mouse click in the middle of the created rectangle, you can move it. A click close to one side “grips” it to modify the rectangle’s dimension in perpendicular direction to this side. You only can modify the orientation, if you grip a side perpendicular to the first half axis. Pressing the right mouse button terminates the procedure.

After terminating the procedure the rectangle is not visible in the window any longer.

Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window_id.
▷ Row (output_control)	rectangle2.center.y $\leadsto$ real Row index of the barycenter.
▷ Column (output_control)	rectangle2.center.x $\leadsto$ real Column index of the barycenter.
▷ Phi (output_control)	rectangle2.angle.rad $\leadsto$ real Orientation of the bigger half axis in radians.
▷ Length1 (output_control)	rectangle2.hwidth $\leadsto$ real Bigger half axis.
▷ Length2 (output_control)	rectangle2.hheight $\leadsto$ real Smaller half axis.
Result	
draw_rectangle2 returns 2 (H_MSG_TRUE), if the window is valid and the needed drawing mode (see set_insert) is available. If necessary, an exception handling is raised.	
Parallelization Information	
draw_rectangle2 is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.	
Possible Predecessors	
open_window	
Possible Successors	
reduce_domain , disp_region , set_colored , set_line_width , set_draw , set_insert	
Alternatives	
draw_rectangle2_mod , draw_rectangle1 , draw_region	
See also	
gen_rectangle2 , draw_circle , draw_ellipse , set_insert	
Module	
System	

draw_rectangle2_mod (: : WindowHandle, RowIn, ColumnIn, PhiIn, Length1In, Length2In : Row, Column, Phi, Length1, Length2)

Interactive drawing of any orientated rectangle.

draw_rectangle2_mod returns the parameter for any orientated rectangle, which has been created interactively by the user in the window.

The created rectangle is described by its center, its two half axes and the angle between the first half axis and the horizontal coordinate axis.

To create a rectangle are expected the parameters **RowIn**, **ColumnIn**, **PhiIn**, **Length1In**, **Length2In**. A click close to one side “grips” it to modify the rectangle’s dimension in perpendicular direction (**Length2**) to this side. You only can modify the orientation (**Phi**), if you grip a side perpendicular to the first half axis. After another mouse click in the middle of the created rectangle, you can move it. Pressing the right mouse button terminates the procedure.

After terminating the procedure the rectangle is not visible in the window any longer.

Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window_id.
▷ RowIn (input_control)	rectangle2.center.y $\leadsto$ real Row index of the barycenter.
▷ ColumnIn (input_control)	rectangle2.center.x $\leadsto$ real Column index of the barycenter.
▷ PhiIn (input_control)	rectangle2.angle.rad $\leadsto$ real Orientation of the bigger half axis in radians.

- ▷ **Length1In** (input_control) rectangle2.hwidth $\leadsto$ *real*
Bigger half axis.
- ▷ **Length2In** (input_control) rectangle2.hheight $\leadsto$ *real*
Smaller half axis.
- ▷ **Row** (output_control) rectangle2.center.y $\leadsto$ *real*
Row index of the barycenter.
- ▷ **Column** (output_control) rectangle2.center.x $\leadsto$ *real*
Column index of the barycenter.
- ▷ **Phi** (output_control) rectangle2.angle.rad $\leadsto$ *real*
Orientation of the bigger half axis in radians.
- ▷ **Length1** (output_control) rectangle2.hwidth $\leadsto$ *real*
Bigger half axis.
- ▷ **Length2** (output_control) rectangle2.hheight $\leadsto$ *real*
Smaller half axis.

Result

[draw_rectangle2_mod](#) returns 2 (H_MSG_TRUE), if the window is valid and the needed drawing mode (see [set_insert](#)) is available. If necessary, an exception handling is raised.

Parallelization Information

[draw_rectangle2_mod](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[open_window](#)

Possible Successors

[reduce_domain](#), [disp_region](#), [set_colored](#), [set_line_width](#), [set_draw](#), [set_insert](#)

Alternatives

[draw_rectangle2](#), [draw_rectangle1](#), [draw_rectangle2](#), [draw_region](#)

See also

[gen_rectangle2](#), [draw_circle](#), [draw_ellipse](#), [set_insert](#)

Module

System

draw_region (: Region : WindowHandle :)

Interactive drawing of a closed region.

[draw_region](#) produces an image. The region of that image spans exactly the image region entered interactively by mouse clicks (gray values remain undefined). Painting happens in the output window while keeping the left mouse button pressed. The left mouse button even operates by clicking in the output window; through this a line between the previous clicked points is drawn. Clicking the right mouse button terminates input and closes the outline. Subsequently the image is “filled up”. Also it contains the whole image area enclosed by the mouse.

Painting uses that color which has been set by [set_color](#), [set_rgb](#), etc. .

Attention

The output object’s gray values are not defined.

Parameter

- ▷ **Region** (output_object) region $\leadsto$ *Hobject*
Interactive created region.
- ▷ **WindowHandle** (input_control) window $\leadsto$ *integer*
Window_id.

Example

```
read_image(Image, 'fabrik')
disp_image(Image, WindowHandle)
draw_region(Region, WindowHandle)
```

```

reduce_domain(Image, Region, New)
regiongrowing(New, Segmente, 5, 5, 6, 50)
set_colored(WindowHandle, 12)
disp_region(Segmente, WindowHandle).

```

Result

If the window is valid, [draw_region](#) returns 2 (H_MSG_TRUE). If necessary, an exception handling is raised.

Parallelization Information

[draw_region](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[open_window](#)

Possible Successors

[reduce_domain](#), [disp_region](#), [set_colored](#), [set_line_width](#), [set_draw](#)

Alternatives

[draw_circle](#), [draw_ellipse](#), [draw_rectangle1](#), [draw_rectangle2](#)

See also

[draw_polygon](#), [reduce_domain](#), [fill_up](#), [set_color](#)

Module

System

6.2 Gnuplot

gnuplot_close (: : GnuplotFileID :)

Close all open gnuplot files or terminate an active gnuplot sub-process.

[gnuplot_close](#) closes all gnuplot files opened by [gnuplot_open_file](#) or terminates the gnuplot sub-process created with [gnuplot_open_pipe](#). In the latter case, all temporary files used to display images and control values are deleted. This means that [gnuplot_close](#) must be called after such a plot sequence. [GnuplotFileID](#) is the identifier of the corresponding gnuplot output stream.

Parameter

- ▷ **GnuplotFileID** (input_control) gnuplot_id $\leadsto$ *integer*
 Identifier for the gnuplot output stream.

Result

[gnuplot_close](#) returns 2 (H_MSG_TRUE) if [GnuplotFileID](#) is a valid gnuplot output stream. Otherwise, an exception handling is raised.

Parallelization Information

[gnuplot_close](#) is processed *completely exclusively* without parallelization.

Possible Predecessors

[gnuplot_open_pipe](#), [gnuplot_open_file](#), [gnuplot_plot_image](#)

See also

[gnuplot_open_pipe](#), [gnuplot_open_file](#), [gnuplot_plot_image](#)

Module

System

gnuplot_open_file (: : FileName : GnuplotFileID)

Open a gnuplot file for visualization of images and control values.

[gnuplot_open_file](#) allows the output of images and control values in a format which can be later processed by gnuplot. The parameter [FileName](#) determines the base-name of the files to be created by calls to [gnuplot_plot_image](#). [gnuplot_open_file](#) generates a gnuplot control file with

the name `<FileName>.gp`, in which the respective plot commands are written. Each image plotted by `gnuplot_plot_image` (or control values plotted by `gnuplot_plot_ctrl`) creates a data file with the name `<FileName>.dat.<Number>`, where Number is the number of the plot in the current sequence. The generated control file can later be edited to create the desired effect. After the last plot `gnuplot_close` has to be called in order to close all open files. The corresponding identifier for the gnuplot output stream is returned in `GnuplotFileID`.

Parameter	
▷ FileName (input_control)	filename $\leadsto$ <i>string</i> Base name for control and data files.
▷ GnuplotFileID (output_control)	gnuplot_id $\leadsto$ <i>integer</i> Identifier for the gnuplot output stream.
Result	
<code>gnuplot_open_file</code> returns the value 2 (H_MSG_TRUE) if the control file could be opened. Otherwise, an exception handling is raised.	
Parallelization Information	
<code>gnuplot_open_file</code> is processed <i>completely exclusively</i> without parallelization.	
Possible Successors	
<code>gnuplot_plot_image</code> , <code>gnuplot_close</code>	
Alternatives	
<code>gnuplot_open_pipe</code>	
See also	
<code>gnuplot_open_pipe</code> , <code>gnuplot_close</code> , <code>gnuplot_plot_image</code>	
Module	
System	

gnuplot_open_pipe (: : : GnuplotFileID)

Open a pipe to a gnuplot process for visualization of images and control values.

`gnuplot_open_pipe` opens a pipe to a gnuplot sub-process with which subsequently images can be visualized as 3D-plots (`gnuplot_plot_image`) or control values can be visualized as 2D-plots (`gnuplot_plot_ctrl`). The sub-process must be terminated after displaying the last plot by calling `gnuplot_close`. The corresponding identifier for the gnuplot output stream is returned in `GnuplotFileID`.

Attention
`gnuplot_open_pipe` is only implemented for Unix because gnuplot for Windows (wgnuplot) cannot be controlled by an external process.

Parameter	
▷ GnuplotFileID (output_control)	gnuplot_id $\leadsto$ <i>integer</i> Identifier for the gnuplot output stream.
Result	
<code>gnuplot_open_pipe</code> returns the value 2 (H_MSG_TRUE) if the sub-process could be created. Otherwise, an exception handling is raised.	
Parallelization Information	
<code>gnuplot_open_pipe</code> is processed <i>completely exclusively</i> without parallelization.	
Possible Successors	
<code>gnuplot_plot_image</code> , <code>gnuplot_plot_ctrl</code> , <code>gnuplot_close</code>	
Alternatives	
<code>gnuplot_open_file</code>	
Module	
System	

gnuplot_plot_ctrl (: : GnuplotFileID, Values :)

Plot control values using gnuplot.

[gnuplot_plot_ctrl](#) displays a tuple of control values using gnuplot. If there is an active gnuplot sub-process (started with [gnuplot_open_pipe](#)), the image is displayed in a gnuplot window. Otherwise, the image is output to a file, which can be later read by gnuplot. In both cases the gnuplot output stream is identified by [GnuplotFileID](#).

Parameter

- ▷ **GnuplotFileID** (input_control) gnuplot_id $\leadsto$ integer
Identifier for the gnuplot output stream.
- ▷ **Values** (input_control) number-array $\leadsto$ real / integer
Control values to be plotted (y-values).

Result

[gnuplot_plot_ctrl](#) returns the value if [GnuplotFileID](#) is a valid gnuplot output stream, if the data file for the current plot could be opened, and if only integer or floating point values were passed. Otherwise, an exception handling is raised.

Parallelization Information

[gnuplot_plot_ctrl](#) is processed *completely exclusively* without parallelization.

Possible Predecessors

[gnuplot_open_pipe](#), [gnuplot_open_file](#)

Possible Successors

[gnuplot_close](#)

See also

[gnuplot_open_pipe](#), [gnuplot_open_file](#), [gnuplot_close](#)

Module

System

gnuplot_plot_func_1d (: : GnuplotFileID, Function :)

Plot a function using gnuplot.

[gnuplot_plot_ctrl](#) displays a function of control values using gnuplot. If there is an active gnuplot sub-process (started with [gnuplot_open_pipe](#)), the image is displayed in a gnuplot window. Otherwise, the image is output to a file, which can be later read by gnuplot. In both cases the gnuplot output stream is identified by [GnuplotFileID](#).

Parameter

- ▷ **GnuplotFileID** (input_control) gnuplot_id $\leadsto$ integer
Identifier for the gnuplot output stream.
- ▷ **Function** (input_control) function_1d-array $\leadsto$ real / integer
Function to be plotted.

Result

[gnuplot_plot_ctrl](#) returns 2 (H_MSG_TRUE) if [GnuplotFileID](#) is a valid gnuplot output stream, and if the data file for the current plot could be opened. Otherwise, an exception handling is raised.

Parallelization Information

[gnuplot_plot_func_1d](#) is processed *completely exclusively* without parallelization.

Possible Predecessors

[gnuplot_open_pipe](#), [gnuplot_open_file](#)

Possible Successors

[gnuplot_close](#)

Alternatives

[gnuplot_plot_ctrl](#)

See also [gnuplot_open_pipe](#), [gnuplot_open_file](#), [gnuplot_close](#)

Module

System

gnuplot_plot_image (Image : : GnuplotFileID, SamplesX, SamplesY, ViewRotX, ViewRotZ, Hidden3D :)

Visualize images using *gnuplot*.

[gnuplot_plot_image](#) displays an image as a 3D-plot using *gnuplot*. If there is an active *gnuplot* sub-process (started with [gnuplot_open_pipe](#)), the image is displayed in a *gnuplot* window. Otherwise, the image is output to a file, which can be later read by *gnuplot*. In both cases the *gnuplot* output stream is identified by [GnuplotFileID](#). The parameters [SamplesX](#) and [SamplesY](#) determine the number of data points in the x- and y-direction, respectively, which *gnuplot* should use to display the image. They are the equivalent of the *gnuplot* variables *samples* and *isosamples*. The parameters [ViewRotX](#) and [ViewRotZ](#) determine the rotation of the plot with respect to the viewer. [ViewRotX](#) is the rotation of the coordinate system about the x-axis, while [ViewRotZ](#) is the rotation of the plot about the z-axis. These two parameters correspond directly to the first two parameters of the 'set view' command in *gnuplot*. The parameter [Hidden3D](#) determines whether hidden surfaces should be removed. This is equivalent to the 'set hidden3d' command in *gnuplot*. If a single image is passed to the operator, it is displayed in a separate plot. If multiple images are passed, they are displayed in the same plot.

-
- Parameter
-
- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Image to be plotted.
 - ▷ **GnuplotFileID** (input_control) gnuplot_id $\leadsto$ integer
Identifier for the *gnuplot* output stream.
 - ▷ **SamplesX** (input_control) integer $\leadsto$ integer
Number of samples in the x-direction.
Default Value : 64
Typical range of values : $2 \leq \text{SamplesX} \leq 10000$
Restriction : $\text{SamplesX} \geq 2$
 - ▷ **SamplesY** (input_control) integer $\leadsto$ integer
Number of samples in the y-direction.
Default Value : 64
Typical range of values : $2 \leq \text{SamplesY} \leq 10000$
Restriction : $\text{SamplesY} \geq 2$
 - ▷ **ViewRotX** (input_control) number $\leadsto$ real / integer
Rotation of the plot about the x-axis.
Default Value : 60
Typical range of values : $0 \leq \text{ViewRotX} \leq 180$
Minimum Increment : 0.01
Recommended Increment : 10
Restriction : $(0 \leq \text{ViewRotX}) \wedge (\text{ViewRotX} \leq 180)$
 - ▷ **ViewRotZ** (input_control) number $\leadsto$ real / integer
Rotation of the plot about the z-axis.
Default Value : 30
Typical range of values : $0 \leq \text{ViewRotZ} \leq 360$
Minimum Increment : 0.01
Recommended Increment : 10
Restriction : $(0 \leq \text{ViewRotZ}) \wedge (\text{ViewRotZ} \leq 360)$
 - ▷ **Hidden3D** (input_control) string $\leadsto$ string
Plot the image with hidden surfaces removed.
Default Value : 'hidden3d'
List of values : $\text{Hidden3D} \in \{\text{'hidden3d'}, \text{'nohidden3d'}\}$

Result

[gnuplot_plot_image](#) returns the value if [GnuplotFileID](#) is a valid *gnuplot* output stream, and if the data file for the current plot could be opened. Otherwise, an exception handling is raised.

<i>Parallelization Information</i>
<code>gnuplot_plot_image</code> is processed <i>completely exclusively</i> without parallelization.
<i>Possible Predecessors</i>
<code>gnuplot_open_pipe</code> , <code>gnuplot_open_file</code>
<i>Possible Successors</i>
<code>gnuplot_close</code>
<i>See also</i>
<code>gnuplot_open_pipe</code> , <code>gnuplot_open_file</code> , <code>gnuplot_close</code>
<i>Module</i>
System

6.3 LUT

disp_lut (: : WindowHandle, Row, Column, Scale :)

Graphical view of the look-up-table (lut).

`disp_lut` displays a graphical view of the look-up-table (lut) in the valid window. A look-up-table defines the transformation of image gray values to colors/gray levels on the screen. On most systems this can be modified. `disp_lut` creates a graphical view of the table assigned to the output window with the logical window number `WindowHandle` and displays it for every basic color (red, green, blue). `Row` and `Column` define the position of the centre of the graphic. `Scale` allows scaling of the graphic, whereas 1 means displaying all 256 values, 2 means displaying 128 values, 3 means displaying only 64 values, etc. Tables for monochrome-representations are displayed in the currently set color (see `set_color`, `set_rgb`, etc.). Tables for displaying "false colors" are viewed with red, green and blue for each color component.

<i>Attention</i>
<code>draw_lut</code> can only be used on hardware supporting look-up-tables for the output.
<i>Parameter</i>
▷ WindowHandle (input_control) window $\rightsquigarrow$ integer Window identifier.
▷ Row (input_control) point.y $\rightsquigarrow$ integer Row of centre of the graphic. Default Value : 128 Typical range of values : $0 \leq \text{Row} \leq 511$
▷ Column (input_control) point.x $\rightsquigarrow$ integer Column of centre of the graphic. Default Value : 128 Typical range of values : $0 \leq \text{Column} \leq 511$
▷ Scale (input_control) integer $\rightsquigarrow$ integer Scaling of the graphic. Default Value : 1 List of values : $\text{Scale} \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ Typical range of values : $0 \leq \text{Scale} \leq 20$

<i>Example</i>

```
set_lut(WindowHandle, 'color')
disp_lut(WindowHandle, 256, 256, 1)
get_mbutton(WindowHandle, _, _, _)
set_lut(WindowHandle, 'sqrt')
disp_lut(WindowHandle, 128, 128, 2).
```

<i>Result</i>

`disp_lut` returns 2 (H_MSG_TRUE) if the hardware supports a look-up-table, the window is valid and the parameters are correct. Otherwise an exception handling is raised.

Parallelization Information

`disp_lut` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`set_lut`

See also

`open_window`, `open_textwindow`, `draw_lut`, `set_lut`, `set_fix`, `set_pixel`, `write_lut`, `get_lut`, `set_color`

Module

System

<code>draw_lut</code> (: : WindowHandle :)

Manipulate look-up-table (lut) interactively.

`draw_lut` allows interactive manipulation of the look-up-table of the device currently displaying the output window.

By pressing and holding down the left mouse button one can change (from "left to right") the red-, green- and blue-intensity displayed in a 2 dimensional diagram with the gray values on the x-axis. The left mouse button also is used for choosing the color channel that should be changed. As an alternative, one can map pure gray levels (gray "color channel") to the gray values on the x-axis. The right mouse button is used for terminating the change-process. The modified look-up-table can be saved by `write_lut` and reloaded later by `set_lut`. `get_lut` succeeding `draw_lut` returns directly the RGB tuple of the look-up-table. These are suitable as input of `set_lut`.

Attention

`draw_lut` can only be used on hardware supporting look-up-tables for the output and allow dynamic changing of the tables.

Parameter

▷ **WindowHandle** (input_control) window $\rightsquigarrow$ integer
Window identifier.

Example

```
read_image(Image, 'fabrik')
disp_image(Image, WindowHandle)
draw_lut(WindowHandle)
write_lut(WindowHandle, 'my_lut').
...
read_image(Image, 'fabrik')
set_lut(WindowHandle, 'my_lut').
```

Result

`draw_lut` returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.

Parallelization Information

`draw_lut` is *reentrant*, *local*, and processed *without* parallelization.

Possible Successors

`set_lut.style`, `set_lut`, `write_lut`, `disp_lut`

Alternatives

`set_fix`, `set_rgb`

See also

`write_lut`, `set_lut`, `get_lut`, `disp_lut`

Module

System

get_fixed_lut (: : WindowHandle : Mode)

Get fixing of "look-up-table" (lut) for "real color images"

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window identifier.
- ▷ **Mode** (output_control) string $\leadsto$ string
Mode of fixing.
Default Value : 'true'
List of values : Mode $\in$ { 'true', 'false' }

Parallelization Information

[get_fixed_lut](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Successors

[set_fixed_lut](#)

Module

System

get_lut (: : WindowHandle : LookUpTable)

Get current look-up-table (lut).

[get_lut](#) returns the name or the values of the look-up-table (lut) of the window, currently used by [disp_image](#) (or indirectly by [disp_region](#), etc.) for output. To set a look-up-table use [set_lut](#). If the current table is a system table without any modification (by [set_fix](#)), the name of the table is returned. If it is a modified table, a table read from a file or a table for output with pseudo real colors, the RGB-values of the table are returned.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window identifier.
- ▷ **LookUpTable** (output_control) string-array $\leadsto$ string / integer
Name of look-up-table or tuple of RGB-values.

Result

[get_lut](#) returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.

Parallelization Information

[get_lut](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Successors

[draw_lut](#), [set_lut](#)

Alternatives

[set_fix](#), [get_pixel](#)

See also

[set_lut](#), [draw_lut](#)

Module

System

get_lut_style (: : WindowHandle : Hue, Saturation, Intensity)

Get modification parameters of look-up-table (lut).

[get_lut_style](#) returns the values that were set with [set_lut_style](#). Default is:

Hue: 0.0

Saturation 1.0**Intensity** 1.0

Parameter

- ▷ **WindowHandle** (input_control) window $\rightsquigarrow$ *integer*
Window identifier.
- ▷ **Hue** (output_control) real $\rightsquigarrow$ *real*
Modification of color value.
- ▷ **Saturation** (output_control) real $\rightsquigarrow$ *real*
Modification of saturation.
- ▷ **Intensity** (output_control) real $\rightsquigarrow$ *real*
Modification of intensity.

Result

get_lut_style returns 2 (H_MSG_TRUE) if the window is valid and the parameter is correct. Otherwise an exception handling is raised.

Parallelization Information

get_lut_style is *reentrant*, *local*, and processed *without* parallelization.

Possible Successors

set_lut_style, **set_lut**

See also

set_lut_style

Module

System

query_lut (: : WindowHandle : LookUpTable)

Query all available look-up-tables (*lut*).

query_lut returns the names of all look-up-tables available on the current used device. These tables can be set with **set_lut**. An table named 'default' is always available.

Parameter

- ▷ **WindowHandle** (input_control) window $\rightsquigarrow$ *integer*
Window identifier.
- ▷ **LookUpTable** (output_control) string-array $\rightsquigarrow$ *string*
Names of look-up-tables.

Result

query_lut returns 2 (H_MSG_TRUE) if a window is valid. Otherwise an exception handling is raised.

Parallelization Information

query_lut is *reentrant*, *local*, and processed *without* parallelization.

Possible Successors

set_lut_style, **set_lut**, **write_lut**, **disp_lut**

See also

set_lut

Module

System

set_fixed_lut (: : WindowHandle, Mode :)

Fix "look-up-table" (*lut*) for "real color images".

Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window identifier.
▷ Mode (input_control)	string $\leadsto$ string Mode of fixing. Default Value : 'true' List of values : Mode $\in$ {'true', 'false'}
Parallelization Information	
set_fixed_lut is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.	
Possible Predecessors	
get_fixed_lut	
Module	
System	

set_lut (: : WindowHandle, LookUpTable :)

Set "look-up-table" (lut).

set_lut sets look-up-table of the device (monitor) displaying the output window. A look-up-table defines the transformation of a "gray value" within an image into a gray value or color on the screen. It describes the screen gray value/color as a combination of red, green and blue for any image gray value (0..255) (so it is a 'table' to 'look up' the screen gray value/color for each image gray value: look-up-table). Transformation into screen-colors is performed in real-time at every time the screen is displayed new (typically this happens about 60 - 70 times per second). So it is possible to change the look-up-table to get a new look of images or regions. Please remind that not all machines support changing the look-up-table (e.g. monochrome resp. real color).

Look-up-tables within HALCON (and on a machine that supports 256 colors) are disposed into three areas:

- S:** system area resp. user area,
- G:** graphic colors,
- B:** image data.

Colors in S descend from applications that were active before starting HALCON and should not get lost. Graphic colors in G are used for operators such as **disp_region**, **disp_circle** etc. and are set unique within all look-up-tables. An output in a graphic color has always got the same (color-)look, even if different look-up-tables are used. **set_color** and **set_rgb** set graphic colors. Gray values resp. colors in B are used by **disp_image** to display an image. They can change according to the current look-up-table. There exist two exceptions to this concept:

- **set_gray** allows setting of colors of the area B for operators such as **disp_region**,
- **set_fix** that allows modification of graphic colors.

For common monitors only one look-up-table can be loaded per screen. Whereas **set_lut** can be activated separately for each window. There is the following solution for this problem: It will always be activated the look-up-table that is assigned to the "active window" (a window is set into the state "active" by the window manager).

look-up-table can also be used with truecolor displays. In this case the look-up-table will be simulated in software. This means, that the look-up-table will be used each time an image is displayed.

WindowsNT specific: if the graphiccard is used in mode different from truecolor, you must display the image after setting the look-up-table.

query_lut lists the names of all look-up-tables. They differ from each other in the area used for gray values. Within this area the following behaviour is defined:

gray value tables (1-7 image levels)

'default': Only the two basic colors (generally black and white) are used.

color tables (Real color, static gray value steps)

'default': Table proposed by the hardware.

gray value tables (256 colors)

'default': As 'linear'.

'linear': Linear increasing of gray values from 0 (black) to 255 (white).

'inverse': Inverse function of 'linear'.

'sqr': Gray values increase according to square function.

'inv_sqr': Inverse function of 'sqr'.

'cube': Gray values increase according to cubic function.

'inv_cube': Inverse function of 'cube'.

'sqrt': Gray values increase according to square-root function.

'inv_sqrt': Inverse Function of 'sqrt'.

'cubic_root': Gray values increase according to cubic-root function.

'inv_cubic_root': Inverse Function of 'cubic_root'.

color tables (256 colors)

'color1': Linear transition from red via green to blue.

'color2': Smooth transition from yellow via red, blue to green.

'color3': Smooth transition from yellow via red, blue, green, red to blue.

'color4': Smooth transition from yellow via red to blue.

'three': Displaying the three colors red, green and blue.

'six': Displaying the six basic colors yellow, red, magenta, blue, cyan and green.

'twelve': Displaying 12 colors.

'twenty_four': Displaying 24 colors.

'rainbow': Displaying the spectral colors from red via green to blue.

'temperature': Temperature table from black via red, yellow to white.

'change1': Color changement after every pixel within the table alternating the six basic colors.

'change2': Fivefold color changement from green via red to blue.

'change3': Threefold color changement from green via red to blue.

A look-up-table can be read from a file. Every line of such a file must contain three numbers in the range of 0 to 255, with the first number describing the amount of red, the second the amount of green and the third the amount of blue of the represented display color. The number of lines can vary. The first line contains information for the first gray value and the last line for the last value. If there are less lines than gray values, the available information values are distributed over the whole interval. If there are more lines than gray values, a number of (uniformly distributed) lines is ignored. The file-name must conform to `"LookUpTable.lut"`. Within the parameter the name is specified without file extension. HALCON will search for the file in the current directory and after that in a specified directory (see `set_system( : : 'lut_dir', <Pfad> : )`). It is also possible to call `set_lut` with a tuple of RGB-Values. These will be set directly. The number of parameter values must conform to the number of pixels currently used within the look-up-table.

Attention

`set_lut` can only be used with monitors supporting 256 gray levels/colors.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window identifier.
- ▷ **LookUpTable** (input_control) string(-array) $\leadsto$ string / integer
Name of look-up-table, values of look-up-table (RGB) or file name.
Default Value : 'default'
Suggested values : `LookUpTable` $\in$ { 'default', 'linear', 'inverse', 'sqr', 'inv_sqr', 'cube', 'inv_cube', 'sqrt', 'inv_sqrt', 'cubic_root', 'inv_cubic_root', 'color1', 'color2', 'color3', 'color4', 'three', 'six', 'twelve', 'twenty_four', 'rainbow', 'temperature', 'cyclic_gray', 'cyclic_temperature', 'hsi', 'change1', 'change2', 'change3' }

Example

```

read_image(Image, 'affe')
query_lut(WindowHandle, LUTs)
for(1, |LUTs|, i)
    set_lut(WindowHandle, LUTs[i])
    fwrite_string(['current table ', LUTs[i]])
    fnew_line()
    get_mbutton(WindowHandle, _, _, _)
loop().

```

Result

[set_lut](#) returns 2 (H_MSG_TRUE) if the hardware supports a look-up-table and the parameter is correct. Otherwise an exception handling is raised.

Parallelization Information

[set_lut](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[query_lut](#), [draw_lut](#), [get_lut](#)

Possible Successors

[write_lut](#)

Alternatives

[draw_lut](#), [set_fix](#), [set_pixel](#)

See also

[get_lut](#), [query_lut](#), [draw_lut](#), [set_fix](#), [set_color](#), [set_rgb](#), [set_hsi](#), [write_lut](#)

Module

System

set_lut_style (: : WindowHandle, Hue, Saturation, Intensity :)

Changing the look-up-table (lut).

[set_lut_style](#) changes the look-up-table (lut) of the device displaying the valid output window. It has got three parameters:

Hue: Rotation of color space, Hue = 1.9 conforms to a one-time rotation of the color space. No change: Hue = 0.0 Complement colors: Hue = 0.5

Saturation: Change of saturation, No change: Saturation = 1.0 Gray value image: Saturation = 0.0

Intensity: Change of intensity, No change: Intensity = 1.0 Black image: Intensity = 0.0

Change affects only the part of a look-up-table that is used for displaying images. The parameter of modification remains until the next call of [set_lut_style](#). Calling [set_lut](#) has got no effect on these parameters.

Parameter

- ▷ **WindowHandle** (input_control) window $\rightsquigarrow$ integer
Window identifier.
- ▷ **Hue** (input_control) real $\rightsquigarrow$ real
Modification of color value.
Default Value : 0.0
Typical range of values : $0.0 \leq \text{Hue} \leq 1.0$
Restriction : $(0.0 \leq \text{Hue}) \wedge (\text{Hue} \leq 1.0)$
- ▷ **Saturation** (input_control) real $\rightsquigarrow$ real
Modification of saturation.
Default Value : 1.5
Typical range of values : $0.0 \leq \text{Saturation}$
Restriction : $0.0 \leq \text{Saturation}$

- ▷ **Intensity** (input_control) real $\leadsto$ real
 Modification of intensity.
Default Value : 1.5
Typical range of values : $0.0 \leq \text{Intensity}$
Restriction : $0.0 \leq \text{Intensity}$

Example

```
read_image(Image, 'affe')
set_lut(WindowHandle, 'color')
repeat( ::: ) >
get_mbutton(WindowHandle, Row, Column, Button)
eval(Row/300.0, Saturation)
eval(Column/512.0, Hue)
set_lut_style(WindowHandle, Hue, Saturation, 1.0)
until(Button = 1).
```

Result

[set_lut_style](#) returns 2 (H_MSG_TRUE) if the window is valid and the parameter is correct. Otherwise an exception handling is raised.

Parallelization Information

[set_lut_style](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[get_lut_style](#)

Possible Successors

[set_lut](#)

Alternatives

[set_lut](#), [scale_image](#)

See also

[get_lut_style](#)

Module

System

write_lut (: : WindowHandle, FileName :)

Write look-up-table (lut) as file.

[write_lut](#) saves the look-up-table (resp. the part of it that is relevant for displaying image gray values) of the valid output window into a file named '[FileName.lut](#)'. It can be read again later with [set_lut](#).

Attention

[write_lut](#) is only suitable for systems using 256 colors.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
 Window identifier.
- ▷ **FileName** (input_control) filename $\leadsto$ string
 File name (of file containing the look-up-table).
Default Value : '/tmp/lut'

Example

```
read_image(Image, 'affe')
disp_image(Image, WindowHandle)
draw_lut(WindowHandle)
write_lut(WindowHandle, 'test_lut').
```

Result

[disp_lut](#) returns 2 (H_MSG_TRUE) if the window with the demanded properties (256 colors) is valid and the parameter (file-name) is correct. It returns 5 (H_MSG_FAIL) if the specified file can't be opened. Otherwise an exception handling is raised.

Parallelization Information

[write_lut](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[draw_lut](#), [set_lut](#)

See also

[set_lut](#), [draw_lut](#), [set_pixel](#), [get_pixel](#)

Module

System

6.4 Mouse

get_mbutton (: : WindowHandle : Row, Column, Button)

Wait until a mouse button is pressed.

[get_mbutton](#) returns the coordinates of the mouse pointer in the output window and the mouse button pressed (**Button**):

- 1: Left button,
- 2: Middle button,
- 4: Right button.

The operator waits until a button is pressed in the output window. If more than one button is pressed, the sum of the individual buttons' values is returned. The origin of the coordinate system is located in the left upper corner of the window. The row coordinates increase towards the bottom, while the column coordinates increase towards the right. For graphics windows, the coordinates of the lower right corner are (image height-1,image width-1) (see [open_window](#), [reset_obj_db](#)), while for text windows they are (window height-1,window width-1) (see [open_textwindow](#)).

Attention

[get_mbutton](#) only returns if a mouse button is pressed in the window.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window identifier.
- ▷ **Row** (output_control) point.y $\leadsto$ integer
Row coordinate of the mouse position in the window.
- ▷ **Column** (output_control) point.x $\leadsto$ integer
Column coordinate of the mouse position in the window.
- ▷ **Button** (output_control) integer $\leadsto$ integer
Mouse button(s) pressed.

Result

[get_mbutton](#) returns the value 2 (H_MSG_TRUE).

Parallelization Information

[get_mbutton](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[open_window](#), [open_textwindow](#)

Alternatives

[get_mposition](#)

See also

[open_window](#), [open_textwindow](#)

Module

System

get_mposition (: : WindowHandle : Row, Column, Button)

Query the mouse position.

get_mposition returns the coordinates of the mouse pointer in the output window and the mouse button pressed. These values are returned regardless of the state of the mouse buttons (pressed or not pressed). If more than one button is pressed, the sum of the individual buttons' values is returned. The possible values for **Button** are:

- 0:** No button,
- 1:** Left button,
- 2:** Middle button,
- 4:** Right button.

The origin of the coordinate system is located in the left upper corner of the window. The row coordinates increase towards the bottom, while the column coordinates increase towards the right. For graphics windows, the coordinates of the lower right corner are (image height-1,image width-1) (see [open_window](#), [reset_obj_db](#)), while for text windows they are (window height-1,window width-1) (see [open_textwindow](#)).

Attention

get_mposition fails (returns FAIL) if the mouse pointer is not located within the window. In this case, no values are returned.

Parameter

- ▷ **WindowHandle** (input_control) window $\rightsquigarrow$ integer
Window identifier.
- ▷ **Row** (output_control) point.y $\rightsquigarrow$ integer
Row coordinate of the mouse position in the window.
- ▷ **Column** (output_control) point.x $\rightsquigarrow$ integer
Column coordinate of the mouse position in the window.
- ▷ **Button** (output_control) integer $\rightsquigarrow$ integer
Mouse button(s) pressed or 0.

Result

get_mposition returns the value 2 (H_MSG_TRUE). If the mouse pointer is not located within the window, 5 (H_MSG_FAIL) is returned.

Parallelization Information

get_mposition is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[open_window](#), [open_textwindow](#)

Alternatives

[get_mbutton](#)

See also

[open_window](#), [open_textwindow](#)

Module

System

get_mshape (: : WindowHandle : Cursor)

Query the current mouse pointer shape.

get_mshape returns the name of the pointer shape set for the window. The mouse pointer shape can be used in the operator [set_mshape](#).

Parameter
▷ WindowHandle (input_control) window $\leadsto$ integer Window identifier.
▷ Cursor (output_control) string $\leadsto$ string Mouse pointer name.
Result
get_mshape returns the value 2 (H_MSG_TRUE).
Parallelization Information
get_mshape is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.
Possible Predecessors
open_window , open_textwindow , query_mshape
Possible Successors
set_mshape
See also
set_mshape , query_mshape
Module
System

query_mshape (: : WindowHandle : ShapeNames)

Query all available mouse pointer shapes.

[query_mshape](#) returns the names of all available mouse pointer shapes for the window. These can be used in the operator [set_mshape](#). If no mouse pointers are available, the empty tuple is returned.

Parameter
▷ WindowHandle (input_control) window $\leadsto$ integer Window identifier.
▷ ShapeNames (output_control) string-array $\leadsto$ string Available mouse pointer names.
Result
query_mshape returns the value 2 (H_MSG_TRUE).
Parallelization Information
query_mshape is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.
Possible Predecessors
open_window , open_textwindow , get_mshape
Possible Successors
set_mshape
See also
set_mshape , get_mshape
Module
System

set_mshape (: : WindowHandle, Cursor :)

Set the current mouse pointer shape.

[set_mshape](#) sets the shape of the mouse pointer for the window. A list of the names of all available mouse pointer shapes can be obtained by calling [query_mshape](#). The mouse pointer shape given by [Cursor](#) is used if the mouse pointer enters the window, irrespective of which window is the output window at present.

Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window identifier.
▷ Cursor (input_control)	string $\leadsto$ string Mouse pointer name. Default Value : 'arrow'
Result	
set_mshape returns the value 2 (H_MSG_TRUE) if the mouse pointer shape Cursor is defined for this window. Otherwise, an exception handling is raised.	
Parallelization Information	
set_mshape is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.	
Possible Predecessors	
open_window , open_textwindow , query_mshape , get_mshape	
See also	
get_mshape , query_mshape	
Module	
System	

6.5 Output

disp_arc (: : WindowHandle, CenterRow, CenterCol, Angle, BeginRow, BeginCol :)

Displays circular arcs in a window.

disp_arc displays one or several circular arcs in the output window. An arc is described by its center point (**CenterRow**, **CenterCol**), the angle between start and end of the arc (**Angle** in radians) and the first point of the arc (**BeginRow**, **BeginCol**). The arc is displayed in clockwise direction. The parameters for output can be determined - as with the output of regions - with the procedures **set_color**, **set_gray**, **set_draw**, etc. It is possible to draw several arcs with one call by using tuple parameters. For the use of colors with several arcs, see **set_color**.

Attention	
The center point has to be within the window. The radius of the arc has to be at least 2 pixel.	
Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window identifier.
▷ CenterRow (input_control)	arc.center.y $\leadsto$ real / integer Row coordinate of center point. Default Value : 64 Suggested values : CenterRow $\in$ {0, 64, 128, 256} Typical range of values : $0 \leq \text{CenterRow} \leq 511$ (lin) Minimum Increment : 1 Recommended Increment : 1
▷ CenterCol (input_control)	arc.center.x $\leadsto$ real / integer Column coordinate of center point. Default Value : 64 Suggested values : CenterCol $\in$ {0, 64, 128, 256} Typical range of values : $0 \leq \text{CenterCol} \leq 511$ (lin) Minimum Increment : 1 Recommended Increment : 1

- ▷ **Angle** (input_control) arc.angle.rad $\leadsto$ *real* / integer
 Angle between start and end of the arc (in radians).
Default Value : 3.1415926
Suggested values : Angle $\in \{0.0, 0.785398, 1.570796, 3.1415926, 6.283185\}$
Typical range of values : $0.0 \leq \text{Angle} \leq 6.283185$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : Angle > 0.0
- ▷ **BeginRow** (input_control) arc.begin.y(-array) $\leadsto$ *integer* / real
 Row coordinate of the start of the arc.
Default Value : 32
Suggested values : BeginRow $\in \{0, 64, 128, 256\}$
Typical range of values : $0 \leq \text{BeginRow} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **BeginCol** (input_control) arc.begin.x(-array) $\leadsto$ *integer* / real
 Column coordinate of the start of the arc.
Default Value : 32
Suggested values : BeginCol $\in \{0, 64, 128, 256\}$
Typical range of values : $0 \leq \text{BeginCol} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1

Example

```
open_window(0,0,-1,-1,'root','visible','',WindowHandle)
set_draw(WindowHandle,'fill')
set_color(WindowHandle,'white')
set_insert(WindowHandle,'not')
Row = 100
Column = 100
disp_arc(WindowHandle,Row,Column,3.14,Row+10,Column+10)
close_window(WindowHandle).
```

Result

`disp_arc` returns 2 (H_MSG_TRUE).

Parallelization Information

`disp_arc` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`open_window`, `set_draw`, `set_color`, `set_colored`, `set_line_width`, `set_rgb`, `set_hsi`

Alternatives

`disp_circle`, `disp_ellipse`, `disp_region`, `gen_circle`, `gen_ellipse`

See also

`open_window`, `open_textwindow`, `set_color`, `set_draw`, `set_rgb`, `set_hsi`

Module

System

disp_arrow (: : WindowHandle, Row1, Column1, Row2, Column2, Size :)

Displays arrows in a window.

`disp_arrow` displays one or several arrows in the output window. An arrow is described by the coordinates of the start (`Row1,Column1`) and the end (`Row2,Column2`). An arrowhead is displayed at the end of the arrow. The size of the arrowhead is specified by the parameter `Size`. If the arrow consists of just one point (start = end) nothing is displayed. The procedures used to control the display of regions (e.g. `set_draw`, `set_color`, `set_line_width`) can also be used with arrows. Several arrows can be displayed with one call by using tuple parameters. For the use of colors with several arcs, see `set_color`.

Attention

The start and the end of the arrows must fall within the window.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window identifier.
- ▷ **Row1** (input_control) line.begin.y(-array) $\leadsto$ real / integer
Row index of the start.
Default Value : 10.0
Suggested values : Row1 $\in \{0.0, 64.0, 128.0, 256.0\}$
Typical range of values : $0.0 \leq \text{Row1} \leq 511.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 1.0
- ▷ **Column1** (input_control) line.begin.x(-array) $\leadsto$ real / integer
Column index of the start.
Default Value : 10.0
Suggested values : Column1 $\in \{0.0, 64.0, 128.0, 256.0\}$
Typical range of values : $0.0 \leq \text{Column1} \leq 511.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 1.0
- ▷ **Row2** (input_control) line.end.y(-array) $\leadsto$ real / integer
Row index of the end.
Default Value : 118.0
Suggested values : Row2 $\in \{0.0, 64.0, 128.0, 256.0\}$
Typical range of values : $0.0 \leq \text{Row2} \leq 511.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 1.0
- ▷ **Column2** (input_control) line.end.x(-array) $\leadsto$ real / integer
Column index of the end.
Default Value : 118.0
Suggested values : Column2 $\in \{0.0, 64.0, 128.0, 256.0\}$
Typical range of values : $0.0 \leq \text{Column2} \leq 511.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 1.0
- ▷ **Size** (input_control) number $\leadsto$ real / integer
Size of the arrowhead.
Default Value : 1.0
Suggested values : Size $\in \{1.0, 2.0, 3.0, 5.0\}$
Typical range of values : $0.0 \leq \text{Size} \leq 20.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 1.0
Restriction : Size > 0.0

Example

```
set_color(WindowHandle, ['red', 'green'])
disp_arrow(WindowHandle, [10, 10], [10, 10], [118, 110], [118, 118], 1.0).
```

Result

`disp_arrow` returns 2 (H_MSG_TRUE).

Parallelization Information

`disp_arrow` is *reentrant*, *local*, and processed *without parallelization*.

Possible Predecessors

`open_window`, `set_draw`, `set_color`, `set_colored`, `set_line_width`, `set_rgb`, `set_hsi`

Alternatives

`disp_line`, `gen_region_polygon`, `disp_region`

See also

`open_window`, `open_textwindow`, `set_color`, `set_draw`, `set_line_width`

Module

System

disp_channel (MultichannelImage : : WindowHandle, Channel :)

Displays images with several channels.

disp_channel displays an image in the output window. It is possible to display several images with one call. In this case the images are displayed one after another. If the definition domains of the images overlap only the last image is visible. The parameter **Channel** defines the number of the channel that is displayed. For RGB-images the three color channels have to be used within a tuple parameter. For more information see [disp_image](#).

Parameter

- ▷ **MultichannelImage** (input_object) image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
 Multichannel images to be displayed.
- ▷ **WindowHandle** (input_control) window $\leadsto$ *integer*
 Window identifier.
- ▷ **Channel** (input_control) integer(-array) $\leadsto$ *integer*
 Number of channel or the numbers of the RGB-channels
Default Value : 1
List of values : Channel $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$

Example

```
/* Tranformation from rgb to gray */
read_image(Image,'patras')
disp_color(Image,WindowHandle)
rgb1_to_gray(Image,GrayImage)
disp_image(GrayImage,WindowHandle).
```

Result

If the used images contain valid values and a correct output mode is set, **disp_channel** returns 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

disp_channel is *reentrant*, *local*, and processed *without parallelization*.

Possible Predecessors

[open_window](#), [set_rgb](#), [set_lut](#), [set_hsi](#)

Alternatives

[disp_image](#), [disp_color](#)

See also

[open_window](#), [open_textwindow](#), [reset_obj_db](#), [set_lut](#), [draw_lut](#), [dump_window](#)

Module

System

disp_circle (: : WindowHandle, Row, Column, Radius :)

Displays circles in a window.

disp_circle displays one or several circles in the output window. A circle is described by the center (**Row**, **Column**) and the radius **Radius**. If the used coordinates are not within the window the circle is clipped accordingly.

The procedures used to control the display of regions (e.g. [set_draw](#), [set_gray](#), [set_draw](#)) can also be used with circles. Several circles can be displayed with one call by using tuple parameters. For the use of colors with several circles, see [set_color](#).

Attention

The center of the circle must be within the window.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window identifier.
- ▷ **Row** (input_control) circle.center.y(-array) $\leadsto$ real / integer
Row index of the center.
Default Value : 64
Suggested values : Row $\in \{0, 64, 128, 256\}$
Typical range of values : $0 \leq \text{Row} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Column** (input_control) circle.center.x(-array) $\leadsto$ real / integer
Column index of the center.
Default Value : 64
Suggested values : Column $\in \{0, 64, 128, 256\}$
Typical range of values : $0 \leq \text{Column} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Radius** (input_control) circle.radius(-array) $\leadsto$ real / integer
Radius of the circle.
Default Value : 64
Suggested values : Radius $\in \{0, 64, 128, 256\}$
Typical range of values : $0 \leq \text{Radius} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
Restriction : Radius > 0.0

Example

```
open_window(0,0,-1,-1,'root','visible','',WindowHandle)
set_draw(WindowHandle,'fill')
set_color(WindowHandle,'white')
set_insert(WindowHandle,'not')
repeat()
  get_mbutton(WindowHandle,Row,Column,Button)
  disp_circle(WindowHandle,Row,Column,(Row + Column) mod 50)
until(Button = 1)
close_window(WindowHandle).
```

Result

disp_circle returns 2 (H_MSG_TRUE).

Parallelization Information

disp_circle is *reentrant*, *local*, and processed without parallelization.

Possible Predecessors

open_window, **set_draw**, **set_color**, **set_colored**, **set_line_width**, **set_rgb**, **set_hsi**

Alternatives

disp_ellipse, **disp_region**, **gen_circle**, **gen_ellipse**

See also

open_window, **open_textwindow**, **set_color**, **set_draw**, **set_rgb**, **set_hsi**

Module

System

disp_color (ColorImage : : WindowHandle :)

Displays a color (RGB) image

disp_color displays the three channels of a color image in the output window. The channels are ordered in the sequence (red,green,blue). **disp_color** can be simulated by **disp_channel**.

Attention

Due to the restricted number of available colors the color appearance is usually different from the original.

Parameter

- ▷ **ColorImage** (input_object) image $\leadsto$ Hobject: byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Color image to display.
- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window identifier.

Example

```
/* disp_color(ColorImage) is identical to: */
Herror my_disp_color(Hobject ColorImage, Htuple *WindowHandle) {
    Htuple Tupel;
    create_tuple(&Tupel,3);
    set_i(Tupel,1,0);
    set_i(Tupel,2,1);
    set_i(Tupel,3,2);
    T_disp_channel(ColorImage, *WindowHandle, Tupel);
    destroy_tuple(Tupel);
}
```

Result

If the used image contains valid values and a correct output mode is set, **disp_color** returns 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

disp_color is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

open_window, **set_rgb**, **set_lut**, **set_hsi**

Alternatives

disp_channel, **disp_obj**

See also

disp_image, **open_window**, **open_textwindow**, **reset_obj_db**, **set_lut**, **draw_lut**, **dump_window**

Module

System

disp_distribution (: : WindowHandle, Distribution, Row, Column, Scale :)

Displays a noise distribution.

disp_distribution displays a distribution in the window. The parameters are the same as in **set_paint** (**WindowHandle**, 'histogram') or **gen_region_histo**. Noise distributions can be generated with operations like **gauss_distribution** or **noise_distribution_mean**.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window identifier.
- ▷ **Distribution** (input_control) real-array $\leadsto$ real
Gray value distribution (513 values).

- ▷ **Row** (input_control) point.y $\leadsto$ *integer*
 Row index of center.
Default Value : 256
Suggested values : Row $\in \{0, 64, 128, 256\}$
Typical range of values : $0 \leq \text{Row} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Column** (input_control) point.x $\leadsto$ *integer*
 Column index of center.
Default Value : 256
Suggested values : Column $\in \{0, 64, 128, 256\}$
Typical range of values : $0 \leq \text{Column} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Scale** (input_control) integer $\leadsto$ *integer*
 Size of display.
Default Value : 1
Suggested values : Scale $\in \{1, 2, 3, 4, 5, 6\}$

Example

```
open_window(0,0,-1,-1,'root','visible','',WindowHandle)
set_draw(WindowHandle,'fill')
set_color(WindowHandle,'white')
set_insert(WindowHandle,'not')
read_image(Image,'affe')
draw_region(Region,WindowHandle)
noise_distribution_mean(Region,Image,21,Distribution)
disp_distribution (WindowHandle,Distribution,100,100,3).
```

Parallelization Information

disp_distribution is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[open_window](#), [set_draw](#), [set_color](#), [set_colored](#), [set_line_width](#), [set_rgb](#), [set_hsi](#),
[noise_distribution_mean](#), [gauss_distribution](#)

See also

[gen_region_histo](#), [set_paint](#), [gauss_distribution](#), [noise_distribution_mean](#)

Module

System

disp_ellipse (: : WindowHandle, CenterRow, CenterCol, Phi, Radius1,
 Radius2 :)

Displays ellipses.

disp_ellipse displays one or several ellipses in the output window. An ellipse is described by the center (**CenterRow**, **CenterCol**), the orientation **Phi** (in radians) and the radii of the major and the minor axis (**Radius1** and **Radius2**).

The procedures used to control the display of regions (e.g. [set_draw](#), [set_gray](#), [set_draw](#)) can also be used with ellipses. Several ellipses can be displayed with one call by using tuple parameters. For the use of colors with several ellipses, see [set_color](#).

Attention

The center of the ellipse must be within the window.

<i>Parameter</i>	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window identifier.
▷ CenterRow (input_control)	ellipse.center.y(-array) $\leadsto$ integer Row index of center. Default Value : 64 Suggested values : CenterRow $\in \{0, 64, 128, 256\}$ Typical range of values : $0 \leq \text{CenterRow} \leq 511$ (lin) Minimum Increment : 1 Recommended Increment : 10
▷ CenterCol (input_control)	ellipse.center.x(-array) $\leadsto$ integer Column index of center. Default Value : 64 Suggested values : CenterCol $\in \{0, 64, 128, 256\}$ Typical range of values : $0 \leq \text{CenterCol} \leq 511$ (lin) Minimum Increment : 1 Recommended Increment : 10
▷ Phi (input_control)	ellipse.angle.rad(-array) $\leadsto$ real / integer Orientation of the ellipse in radians Default Value : 0.0 Suggested values : Phi $\in \{0.0, 0.785398, 1.570796, 3.1415926, 6.283185\}$ Typical range of values : $0.0 \leq \text{Phi} \leq 6.283185$ (lin) Minimum Increment : 0.01 Recommended Increment : 0.1
▷ Radius1 (input_control)	ellipse.radius1(-array) $\leadsto$ real / integer Radius of major axis. Default Value : 24.0 Suggested values : Radius1 $\in \{0.0, 64.0, 128.0, 256.0\}$ Typical range of values : $0.0 \leq \text{Radius1} \leq 511.0$ (lin) Minimum Increment : 1.0 Recommended Increment : 10.0
▷ Radius2 (input_control)	ellipse.radius2(-array) $\leadsto$ real / integer Radius of minor axis. Default Value : 14.0 Suggested values : Radius2 $\in \{0.0, 64.0, 128.0, 256.0\}$ Typical range of values : $0.0 \leq \text{Radius2} \leq 511.0$ (lin) Minimum Increment : 1.0 Recommended Increment : 10.0

Example

```

set_color(WindowHandle, 'red')
draw_region(MyRegion, WindowHandle)
elliptic_axis(MyRegionRa, Rb, Phi)
area_center(MyRegion, _, Row, Column)
disp_ellipse(WindowHandle, Row, Column, Phi, Ra, Rb) .

```

Result

disp_ellipse returns 2 (H_MSG_TRUE), if the parameters are correct. Otherwise an exception handling is raised.

Parallelization Information

disp_ellipse is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[open_window](#), [set_draw](#), [set_color](#), [set_colored](#), [set_line_width](#), [set_rgb](#), [set_hsi](#), [elliptic_axis](#), [area_center](#)

Alternatives

[disp_circle](#), [disp_region](#), [gen_ellipse](#), [gen_circle](#)

See also

[open_window](#), [open_textwindow](#), [set_color](#), [set_rgb](#), [set_hsi](#), [set_draw](#), [set_line_width](#)

Module

System

disp_image (Image : : WindowHandle :)

Displays gray value images.

[disp_image](#) displays the gray values of an image in the output window. The gray value pixels of the definition domain ([set_comprise \(::WindowHandle, 'object' :\)](#)) or of the whole image ([set_comprise \(::WindowHandle, 'image' :\)](#)) are used. Restriction to the definition domain is the default.

For the display of gray value images the number of gray values is usually reduced. This is due to the fact that colors have to be reserved for the display of graphics (e.g. [set_color](#)) and the window manager. Also depending on the number of bitplanes on the used output device often less than 256 colors (eight bitplanes) are available. The number of "colors" actually reserved for the display of gray values can be queried by [get_system](#). Before opening the first window this value can be modified by [set_system](#). For instance for 8 bitplanes 200 real gray values are the default.

The reduction of the number of gray values does not pose problems as long as only gray value information is displayed, humans cannot distinguish 256 different shades of gray. If certain gray values are used for the representation of region information (which is not the style commonly used in HALCON), confusions might be the result, since different numerical values are displayed on the screen with the same gray value. The procedure [label_to_region](#) should be used on these images in order to transform the label data into HALCON objects.

If images of type 'int2', 'int4', 'real' or 'complex' are displayed, the smallest and largest gray value is computed. Afterwards the pixel data is rescaled according to the number of available gray values (depending on the output device, e.g. 200). It is possible that some pixels have a very different value than the other pixels. This might lead to the display of an (almost) completely white or black image. In order to decide if the current image is a binary image [min_max_gray](#) can be used. If necessary the image can be transformed or converted by [scale_image](#) and [convert_image_type](#) before it is displayed.

Attention

If a wrong output mode was set by [set_paint](#), the error will be reported when [disp_image](#) is used.

Parameter

▷ **Image** (input_object) image $\rightsquigarrow$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

Gray value image to display.

▷ **WindowHandle** (input_control) window $\rightsquigarrow$ *integer*
Window identifier.

Example

```
/* Output of a gray image: */
read_image(Image1, 'affe')
disp_image(Image1, WindowHandle).
```

Result

If the used image contains valid values and a correct output mode is set, [disp_image](#) returns 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

[disp_image](#) is *reentrant*, *local*, and processed *without parallelization*.

Possible Predecessors

[open_window](#), [set_rgb](#), [set_lut](#), [set_hsi](#), [scale_image](#), [convert_image_type](#),
[min_max_gray](#)

Alternatives

[disp_obj](#), [disp_color](#)

See also

[open_window](#), [open_textwindow](#), [reset_obj_db](#), [set_comprise](#), [set_paint](#), [set_lut](#),
[draw_lut](#), [paint_gray](#), [scale_image](#), [convert_image_type](#), [dump_window](#)

Module

System

disp_line (: : WindowHandle, Row1, Column1, Row2, Column2 :)

Draws lines in a window.

disp_line displays one or several lines in the output window. A line is described by the coordinates of the start (**Row1,Column1**) and the coordinates of the end (**Row2,Column2**). The procedures used to control the display of regions (e.g. [set_color](#), [set_gray](#), [set_draw](#), [set_line_width](#)) can also be used with lines. Several lines can be displayed with one call by using tuple parameters. For the use of colors with several lines, see [set_color](#).

Attention

The starting points and the ending points of the lines must be in the window.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ *integer*
 Window identifier.
- ▷ **Row1** (input_control) line.begin.y(-array) $\leadsto$ *real*
 Row index of the start.
Default Value : 32
Suggested values : Row1 $\in$ {0, 64, 128, 256, 511}
Typical range of values : $0 \leq \text{Row1} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Column1** (input_control) line.begin.x(-array) $\leadsto$ *real*
 Column index of the start.
Default Value : 32
Suggested values : Column1 $\in$ {0, 64, 128, 256, 511}
Typical range of values : $0 \leq \text{Column1} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Row2** (input_control) line.end.y(-array) $\leadsto$ *real*
 Row index of end.
Default Value : 64
Suggested values : Row2 $\in$ {0, 64, 128, 256, 511}
Typical range of values : $0 \leq \text{Row2} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Column2** (input_control) line.end.x(-array) $\leadsto$ *real*
 Column index of end.
Default Value : 64
Suggested values : Column2 $\in$ {0, 64, 128, 256, 511}
Typical range of values : $0 \leq \text{Column2} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10

Example

```
/* Prozedur zur Ausgabe der Kontur eines Rechtecks: */
```

```
disp_rectangle1_margin(WindowHandle, Row1, Column1, Row2, Column2):
    disp_line(WindowHandle, Row1, Column1, Row1, Column2)
    disp_line(WindowHandle, Row1, Column2, Row2, Column2)
```

```
disp_line(WindowHandle, Row2, Column2, Row2, Column1)
disp_line(WindowHandle, Row2, Column1, Row1, Column1).
```

Result

`disp_line` returns 2 (H_MSG_TRUE).

Parallelization Information

`disp_line` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`open_window`, `set_rgb`, `set_lut`, `set_hsi`, `set_draw`, `set_color`, `set_colored`,
`set_line_width`

Alternatives

`disp_arrow`, `disp_rectangle1`, `disp_rectangle2`, `disp_region`, `gen_region_polygon`,
`gen_region_points`

See also

`open_window`, `open_textwindow`, `set_color`, `set_rgb`, `set_hsi`, `set_insert`,
`set_line_width`

Module

System

disp_obj (Object : : WindowHandle :)

Displays image objects (image, region, XLD).

`disp_obj` displays objects depending of their kind. `disp_obj` is equivalent to `disp_image` for one channel images, equivalent to `disp_color` for three channel images, equivalent to `disp_region` for regions and equivalent to `disp_xld` for XLDs.

Parameter

- ▷ **Object** (input_object) object(-array) $\leadsto$ *Hobject*
Image object to be displayed.
- ▷ **WindowHandle** (input_control) window $\leadsto$ *integer*
Window identifier.

Example

```
/* Output of a gray image: */
read_image(Image1, 'affe')
disp_obj(Image1, WindowHandle)
threshold(Image, Region, 0, 128)
disp_obj(Region, WindowHandle)
```

Result

If the used object is valid and a correct output mode is set, `disp_obj` returns 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

`disp_obj` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`open_window`, `set_rgb`, `set_lut`, `set_hsi`, `scale_image`, `convert_image_type`,
`min_max_gray`

Alternatives

`disp_color`, `disp_image`, `disp_xld`, `disp_region`

See also

`open_window`, `open_textwindow`, `reset_obj_db`, `set_comprise`, `set_paint`, `set_lut`,
`draw_lut`, `paint_gray`, `scale_image`, `convert_image_type`, `dump_window`

Module

System

disp_polygon (: : WindowHandle, Row, Column :)

Displays a polyline.

disp_polygon displays a polyline with the row coordinates **Row** and the column coordinates **Column** in the output window. The parameters **Row** and **Column** have to be provided as tuples. Straight lines are drawn between the given points. The start and the end of the polyline are not connected.

The procedures used to control the display of regions (e.g. **set_color**, **set_gray**, **set_draw**, **set_line_width**) can also be used with polylines.

Attention

The given coordinates must lie within the window.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window identifier.
- ▷ **Row** (input_control) polygon.y-array $\leadsto$ integer / real
Row index
Default Value : '[16,80,80]'
Suggested values : Row $\in$ {0, 64, 128, 256, 511}
Typical range of values : $0 \leq \text{Row} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Column** (input_control) polygon.x-array $\leadsto$ integer / real
Column index
Default Value : '[48,16,80]'
Suggested values : Column $\in$ {0, 64, 128, 256, 511}
Typical range of values : $0 \leq \text{Column} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10

Example

```
/* display a rectangle */

disp_rectangle1_margin1(long WindowHandle,
    long Row1, long Column1,
    long Row2, long Column2)
{
    Htuple Row, Col;
    create_tuple(&Row, 4) ;
    create_tuple(&Col, 4) ;

    set_i(Row, Row1, 0) ;
    set_i(Col, Column1, 0) ;

    set_i(Row, Row1, 1) ;
    set_i(Col, Column2, 1) ;

    set_i(Row, Row2, 2) ;
    set_i(Col, Column2, 2) ;

    set_i(Row, Row2, 3) ;
    set_i(Col, Column1, 3) ;

    set_i(Row, Row1, 4) ;
    set_i(Col, Column1, 4) ;

    T_disp_polygon(WindowHandle, Row, Col) ;
}
```

```
}

```

Result

`disp_polygon` returns 2 (H_MSG_TRUE).

Parallelization Information

`disp_polygon` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`open_window`, `set_rgb`, `set_lut`, `set_hsi`, `set_draw`, `set_color`, `set_colored`,
`set_line_width`

Alternatives

`disp_line`, `gen_region_polygon`, `disp_region`

See also

`open_window`, `open_textwindow`, `set_color`, `set_rgb`, `set_hsi`, `set_insert`,
`set_line_width`

Module

System

`disp_rectangle1` (: : WindowHandle, Row1, Column1, Row2, Column2 :)

Display of rectangles aligned to the coordinate axes.

`disp_rectangle1` displays one or several rectangles in the output window. A rectangle is described by the upper left corner (`Row1,Column1`) and the lower right corner (`Row2,Column2`). If the given coordinates are not within the boundary of the window the rectangle is clipped accordingly. The procedures used to control the display of regions (e.g. `set_color`, `set_gray`, `set_draw`, `set_line_width`) can also be used with rectangles. Several rectangles can be displayed with one call by using tuple parameters.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window identifier.
- ▷ **Row1** (input_control) rectangle.origin.y(-array) $\leadsto$ real / integer
Row index of the upper left corner.
Default Value : 16
Suggested values : Row1 $\in$ {0, 64, 128, 256, 511}
Typical range of values : $0 \leq \text{Row1} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Column1** (input_control) rectangle.origin.x(-array) $\leadsto$ real / integer
Column index of the upper left corner.
Default Value : 16
Suggested values : Column1 $\in$ {0, 64, 128, 256, 511}
Typical range of values : $0 \leq \text{Column1} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Row2** (input_control) rectangle.corner.y(-array) $\leadsto$ real / integer
Row index of the lower right corner.
Default Value : 48
Suggested values : Row2 $\in$ {0, 64, 128, 256, 511}
Typical range of values : $0 \leq \text{Row2} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : Row2 $\geq$ Row1

- ▷ **Column2** (input_control) rectangle.corner.x(-array) $\leadsto$ *real* / integer
 Column index of the lower right corner.
Default Value : 80
Suggested values : Column2 $\in$ {0, 64, 128, 256, 511}
Typical range of values : $0 \leq \text{Column2} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : Column2 $\geq$ Column1

Example

```
set_color(WindowHandle, 'green')
draw_region(MyRegion, WindowHandle)
smallest_rectangle1(MyRegion, R1, C1, R2, C2)
disp_rectangle1(WindowHandle, R1, C1, R2, C2).
```

Result

`disp_rectangle1` returns 2 (H_MSG_TRUE).

Parallelization Information

`disp_rectangle1` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`open_window`, `set_rgb`, `set_lut`, `set_hsi`, `set_draw`, `set_color`, `set_colored`,
`set_line_width`

Alternatives

`disp_rectangle2`, `gen_rectangle1`, `disp_region`, `disp_line`, `set_shape`

See also

`open_window`, `open_textwindow`, `set_color`, `set_draw`, `set_line_width`

Module

System

disp_rectangle2 (: : WindowHandle, CenterRow, CenterCol, Phi,
 Length1, Length2 :)

Displays arbitrarily oriented rectangles.

`disp_rectangle2` draws one or several arbitrarily oriented rectangles in the output window. A rectangle is described by the center (**CenterRow**, **CenterCol**), the orientation **Phi** (in radians) and half the lengths of the edges **Length1** and **Length2**. The procedures used to control the display of regions (e.g. `set_draw`, `set_gray`, `set_draw`) can also be used with rectangles. Several rectangles can be displayed with one call by using tuple parameters. For the use of colors with several rectangles, see `set_color`.

Attention

The center must lie within the window boundaries.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ *integer*
 Window identifier.
- ▷ **CenterRow** (input_control) rectangle2.center.y(-array) $\leadsto$ *real* / integer
 Row index of the center.
Default Value : 48
Suggested values : CenterRow $\in$ {0, 64, 128, 256, 511}
Typical range of values : $0 \leq \text{CenterRow} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10

- ▷ **CenterCol** (input_control) rectangle2.center.x(-array) $\leadsto$ *real* / integer
Column index of the center.
Default Value : 64
Suggested values : CenterCol $\in \{0, 64, 128, 256, 511\}$
Typical range of values : $0 \leq \text{CenterCol} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Phi** (input_control) rectangle2.angle.rad(-array) $\leadsto$ *real* / integer
Orientation of rectangle in radians.
Default Value : 0.0
Suggested values : Phi $\in \{0.0, 0.785398, 1.570796, 3.1415926, 6.283185\}$
Typical range of values : $0.0 \leq \text{Phi} \leq 6.283185$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
- ▷ **Length1** (input_control) rectangle2.hwidth(-array) $\leadsto$ *real* / integer
Half of the length of the longer side.
Default Value : 48
Suggested values : Length1 $\in \{0, 64, 128, 256, 511\}$
Typical range of values : $0 \leq \text{Length1} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Length2** (input_control) rectangle2.hheight(-array) $\leadsto$ *real* / integer
Half of the length of the shorter side.
Default Value : 32
Suggested values : Length2 $\in \{0, 64, 128, 256, 511\}$
Typical range of values : $0 \leq \text{Length2} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : Length2 < Length1

Example

```
set_color(WindowHandle, 'green')
draw_region(MyRegion:WindowHandle)
elliptic_axis(MyRegion, Ra, Rb, Phi)
area_center(MyRegion, _, Row, Column)
disp_rectangle2(WindowHandle, Row, Column, Phi, Ra, Rb) .
```

Result

`disp_rectangle2` returns 2 (H_MSG_TRUE), if the parameters are correct. Otherwise an exception handling is raised.

Parallelization Information

`disp_rectangle2` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`open_window`, `set_rgb`, `set_lut`, `set_hsi`, `set_draw`, `set_color`, `set_colored`,
`set_line_width`

Alternatives

`disp_region`, `gen_rectangle2`, `disp_rectangle1`, `set_shape`

See also

`open_window`, `open_textwindow`, `disp_region`, `set_color`, `set_draw`, `set_line_width`

Module

System

disp_region (DispRegions : : WindowHandle :)

Displays regions in a window.

disp_region displays the regions in **DispRegions** in the output window. The parameters for output can be set with the procedures **set_color**, **set_gray**, **set_draw**, **set_line_width**, etc.

The color(s) for the display of the regions are determined with **set_color**, **set_rgb**, **set_gray** or **set_colored**. If more than one region is displayed and more than one color is set, the colors are assigned in a cyclic way to the regions.

The form of the region for output can be modified by **set_paint** (e.g. encompassing circle, convex hull). The command **set_draw** determines if the region is filled or only the boundary is drawn. If only the boundary is drawn, the thickness of the boundary will be determined by **set_line_width** and the style by **set_line_style**.

Parameter	
▷ DispRegions (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Regions to display.
▷ WindowHandle (input_control)	window $\leadsto$ <i>integer</i> Window identifier.
Example	

```
/* Output with 12 colors: */
set_colored(WindowHandle,12)
disp_region(SomeSegments,WindowHandle).

/* Symbolic representation: */
set_draw(WindowHandle,'margin')
set_color(WindowHandle,'red')
set_shape(WindowHandle,'ellipse')
disp_region(SomeSegmentsWindowHandle).

/* Representation of a margin with pattern: */
set_draw(WindowHandle,'margin')
set_color(WindowHandle,'blue')
set_line_style(WindowHandle,[12,3])
disp_region(Segments,WindowHandle).
```

Result
disp_region returns 2 (H_MSG_TRUE).

Parallelization Information
disp_region is <i>reentrant</i> , <i>local</i> , and processed without parallelization.

Possible Predecessors
open_window , set_rgb , set_lut , set_hsi , set_shape , set_line_style , set_insert , set_fix , set_draw , set_color , set_colored , set_line_width

Alternatives
disp_obj , disp_arrow , disp_line , disp_circle , disp_rectangle1 , disp_rectangle2 , disp_ellipse

See also
open_window , open_textwindow , set_color , set_colored , set_draw , set_shape , set_paint , set_gray , set_rgb , set_hsi , set_pixel , set_line_width , set_line_style , set_insert , set_fix , paint_region , dump_window

Module
System

disp_xld (XLDObject : : WindowHandle :)

Display an XLD object.

disp_xld serves to display an XLD object of arbitrary type.

Parameter	
▷ XLDObject (input_object)	xld-array $\leadsto$ <i>Hobject</i> XLD object to display.
▷ WindowHandle (input_control)	window $\leadsto$ <i>integer</i> Window id.
Parallelization Information	
disp_xld is <i>local</i> and processed <i>completely exclusively</i> without parallelization.	
See also	
disp_image , disp_region , disp_channel , disp_color , disp_line , disp_arc	
Module	
Sub-pixel operators	

6.6 Parameters

get_comprise (: : WindowHandle : Mode)

Get the output treatment of an image matrix.

get_comprise returns the output mode of grayvalues in the window [WindowHandle](#) that is used by [disp_image](#) and [disp_color](#). The output mode defines whether only the grayvalues of objects are displayed or the whole image is displayed. The query is used for temporary mode settings, i.e., the current mode is queried, then overwritten with ([set_comprise](#)) and finally reset.

Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ <i>integer</i> Window_id.
▷ Mode (output_control)	string $\leadsto$ <i>string</i> Display mode for images.
Result	
get_comprise returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.	
Parallelization Information	
get_comprise is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Successors	
set_comprise , disp_image , disp_image	
See also	
set_comprise , disp_image , disp_color	
Module	
System	

get_draw (: : WindowHandle : Mode)

Get the current region fill mode.

get_draw returns the region fill mode of the output window. It is used by operators as [disp_region](#), [disp_circle](#), [disp_arrow](#), [disp_rectangle1](#), [disp_rectangle2](#) etc. The region fill mode is set with [set_draw](#).

Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ <i>integer</i> Window_id.
▷ Mode (output_control)	string $\leadsto$ <i>string</i> Current region fill mode.

Result

[get_draw](#) returns 2 (H_MSG_TRUE), if the window is valid. Otherwise an exception handling is raised.

Parallelization Information

[get_draw](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[set_draw](#), [disp_region](#)

See also

[set_draw](#), [disp_region](#), [set_paint](#)

Module

System

get_fix (: : WindowHandle : Mode)

Get mode of fixing of current look-up-table (lut).

Use [get_fix](#) to get mode of fixing of current look-up-table (look-up-table of valid window) set before by [set_fix](#).

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ *integer*
Window identifier.
- ▷ **Mode** (output_control) string $\leadsto$ *string*
Current Mode of fixing.

Result

[get_fix](#) returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.

Parallelization Information

[get_fix](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Successors

[set_fix](#), [set_pixel](#), [set_rgb](#)

See also

[set_fix](#)

Module

System

get_hsi (: : WindowHandle : Hue, Saturation, Intensity)

Get the HSI coding of the current color.

[get_hsi](#) returns the output color or grayvalues, respectively, for the window, described in [Hue](#), [Saturation](#) and [Intensity](#). [get_hsi](#) corresponds to the procedure [get_pixel](#) but returns the entries of the color lookup table instead of its indices. The values returned by [get_hsi](#) can be set with [set_hsi](#).

Attention

The values returned by [get_hsi](#) may be inaccurate due to rounding errors. They do not necessarily match the values set with [set_hsi](#) exactly (colors are stored in RGB internally).

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ *integer*
Window_id.
- ▷ **Hue** (output_control) integer-array $\leadsto$ *integer*
Hue (color value) of the current color.
- ▷ **Saturation** (output_control) integer-array $\leadsto$ *integer*
Saturation of the current color.

- ▷ **Intensity** (output_control) integer-array $\leadsto$ integer
Intensity of the current color.

Result

`get_hsi` returns 2 (H_MSG_TRUE), if the window is valid. Otherwise an exception handling is raised.

Parallelization Information

`get_hsi` is *reentrant* and processed *without* parallelization.

Possible Successors

`set_hsi`, `set_rgb`, `disp_image`

See also

`set_hsi`, `set_color`, `set_rgb`, `trans_to_rgb`, `trans_from_rgb`

Module

System

get_icon (: Icon : WindowHandle :)

Query the icon for region output

`get_icon` queries the icon that was set with `set_icon`.

Parameter

- ▷ **Icon** (output_object) region $\leadsto$ Hobject
Icon for the regions center of gravity.
- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window_id.

Example

```
/* draw a region and an icon. */
/* set it and get it again. */
draw_region(&Region,WindowHandle) ;
draw_region(&Icon,WindowHandle) ;
set_icon(Icon) ;
set_shape(WindowHandle,"icon") ;
disp_region(Region,WindowHandle) ;
get_icon(&OldIcon) ;
disp_region(OldIcon,WindowHandle) ;
```

Result

`get_icon` always returns 2 (H_MSG_TRUE).

Parallelization Information

`get_icon` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`set_icon`

Possible Successors

`disp_region`

Module

System

get_insert (: : WindowHandle : Mode)

Get the current display mode.

`get_insert` returns the display mode of the output window. It is used by procedures like `disp_region`, `disp_line`, `disp_rectangle1`, etc. The mode is set with `set_insert`. Possible values for `Mode` can be queried with the procedure `query_insert`.

Parameter
▷ WindowHandle (input_control) window $\leadsto$ integer Window_id.
▷ Mode (output_control) string $\leadsto$ string Display mode.
Result
get_insert returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.
Parallelization Information
get_insert is <i>reentrant</i> and processed <i>without</i> parallelization.
Possible Predecessors
query_insert
Possible Successors
set_insert , disp_image
See also
set_insert , query_insert , disp_region , disp_line
Module
System

get_line_approx (: : WindowHandle : Approximation)

Get the current approximation error for contour display.

[get_line_approx](#) returns a parameter that controls the approximation error for region contour display in the window. It is used by the procedure [disp_region](#). [Approximation](#) controls the polygon approximation for contour display (0 $\Leftrightarrow$ no approximation). [Approximation](#) is only important for displaying the contour of objects, especially if a line style was set with [set_line_style](#).

Parameter
▷ WindowHandle (input_control) window $\leadsto$ integer Window_id.
▷ Approximation (output_control) string $\leadsto$ integer Current approximation error for contour display.
Result
get_line_approx returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.
Parallelization Information
get_line_approx is <i>reentrant</i> and processed <i>without</i> parallelization.
Possible Successors
set_line_approx , set_line_style , disp_region
See also
get_region_polygon , set_line_approx , set_line_style , disp_region
Module
System

get_line_style (: : WindowHandle : Style)

Get the current graphic mode for contours.

[get_line_style](#) returns the display mode for contours when displaying regions. It is used by procedures like [disp_region](#), [disp_line](#), [disp_polygon](#), etc. [Style](#) is set with the procedure [set_line_style](#). [Style](#) is only important for displaying the contour of objects, especially if a line style was set with [set_line_style](#).

Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window_id.
▷ Style (output_control)	integer-array $\leadsto$ integer Template for contour display.
Result	
get_line_style returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.	
Parallelization Information	
get_line_style is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.	
See also	
set_line_style , disp_region	
Module	
System	

get_line_width (: : WindowHandle : Width)

Get the current line width for contour display.

get_line_width returns the line width for region display in the window. It is used by procedures like **disp_region**, **disp_line**, **disp_polygon**, etc. **Width** is set with the procedure **set_line_width**. **Width** is only important for displaying the contour of objects.

Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window_id.
▷ Width (output_control)	integer $\leadsto$ integer Current line width for contour display.
Result	
get_line_width returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.	
Parallelization Information	
get_line_width is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Successors	
set_line_width , set_line_style , disp_region	
See also	
set_line_width , disp_region	
Module	
System	

get_paint (: : WindowHandle : Mode)

Get the current display mode for grayvalues.

get_paint returns the display mode for grayvalues in the window. **Mode** is used by the procedure **disp_image**. **get_paint** is used for temporary changes of the grayvalue display mode. The current value is queried, then changed (with procedure **set_paint**) and finally the old value is written back. The available modes can be viewed with the procedure **query_paint**. **Mode** is the name of the display mode. If a mode can be customized with parameters, the parameter values are passed in a tuple after the mode name. The order of values is the same as in **set_paint**.

Parameter
▷ WindowHandle (input_control) window $\leadsto$ integer Window_id. ▷ Mode (output_control) string-array $\leadsto$ string / integer Name and parameter values of the current display mode.
Result
get_paint returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.
Parallelization Information
get_paint is <i>reentrant</i> and processed <i>without</i> parallelization.
Possible Predecessors
query_paint
Possible Successors
set_paint , disp_region , disp_image
See also
set_paint , query_paint , disp_image
Module
System

get_part (: : WindowHandle : Row1, Column1, Row2, Column2)

Get the image part.

get_part returns the upper left and lower right corner of the image part shown in the window. The image part can be changed with the procedure **set_part** (Default is the whole image).

Parameter
▷ WindowHandle (input_control) window $\leadsto$ integer Window_id. ▷ Row1 (output_control) rectangle.origin.y $\leadsto$ integer Row index of the image part's upper left corner. ▷ Column1 (output_control) rectangle.origin.x $\leadsto$ integer Column index of the image part's upper left corner. ▷ Row2 (output_control) rectangle.corner.y $\leadsto$ integer Row index of the image part's lower right corner. ▷ Column2 (output_control) rectangle.corner.x $\leadsto$ integer Column index of the image part's lower right corner.
Result
get_part returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.
Parallelization Information
get_part is <i>reentrant</i> and processed <i>without</i> parallelization.
Possible Successors
set_part , disp_region , disp_image
See also
set_part , disp_image , disp_region , disp_color
Module
System

get_part_style (: : WindowHandle : Style)

Get the current interpolation mode for grayvalue display.

[get_part_style](#) returns the interpolation mode used for displaying an image part in the window. An interpolation takes place if the output window is larger than the image format or the image output format (see [set_part](#)). HALCON supports three interpolation modes:

- 0 no interpolation (low quality, very fast).
- 1 unweighted interpolation (average quality and computation time)
- 2 weighted interpolation (high quality, slow)

The current mode can be changed with [set_part_style](#).

Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window_id.
▷ Style (output_control)	integer $\leadsto$ integer Interpolation mode for image display: 0 (fast, low quality) to 2 (slow, high quality). List of values : Style $\in$ {0, 1, 2}
Result	
get_part_style returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.	
Parallelization Information	
get_part_style is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Successors	
set_part_style , disp_region , disp_image	
See also	
set_part_style , set_part , disp_image , disp_color	
Module	
System	

get_pixel (: : WindowHandle : Pixel)

Get the current color lookup table index.

[get_pixel](#) returns the internal coding of the output grayvalue or color, respectively, for the window. If the output mode is set to color(s) or grayvalue(s) (see [set_color](#) or [set_gray](#)), then the color- or grayvalues are transformed for internal use. The internal code is then used for (physical) screen display. The transformation depends on the mapping characteristics and the condition of the output device and can be different in different program runs. Don't confuse the term "pixel" with the term "pixel" in image processing (the other procedure is [get_grayval](#)). Here a pixel is meant to be the color lookup table index.

With [get_pixel](#) it is possible to save the output mode without knowing whether colors or grayvalues are used. **Pixel** is set with the procedure [set_pixel](#).

Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window_id.
▷ Pixel (output_control)	string-array $\leadsto$ integer Index of the current color loopup table.
Result	
get_part_style returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.	
Parallelization Information	
get_pixel is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Successors	
set_pixel , disp_region , disp_image	
See also	
set_pixel , set_fix	
Module	
System	

get_rgb (: : WindowHandle : Red, Green, Blue)

Get the current color in RGB-coding.

get_rgb returns the output colors or grayvalues, respectively, for the output window. They are defined by the three color components red, green and blue.

get_rgb is like **get_pixel** but returns the entries of the color lookup table rather than the indices. The values returned by **get_rgb** can be set with **set_rgb**.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window_id.
- ▷ **Red** (output_control) integer-array $\leadsto$ integer
The current color's red value.
- ▷ **Green** (output_control) integer-array $\leadsto$ integer
The current color's green value.
- ▷ **Blue** (output_control) integer-array $\leadsto$ integer
The current color's blue value.

Result

get_rgb returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.

Parallelization Information

get_rgb is *reentrant* and processed *without* parallelization.

Possible Successors

set_rgb, **disp_region**, **disp_image**

See also

set_rgb

Module

System

get_shape (: : WindowHandle : DisplayShape)

Get the current region output shape.

get_shape returns the shape in which regions are displayed. The available shapes can be queried with **query_shape** and then changed with **set_shape**.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window_id.
- ▷ **DisplayShape** (output_control) string $\leadsto$ string
Current region output shape.

Result

get_shape returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.

Parallelization Information

get_shape is *reentrant* and processed *without* parallelization.

Possible Predecessors

query_shape

Possible Successors

set_shape, **disp_region**

See also

set_shape, **query_shape**, **disp_region**

Module

System

query_all_colors (: : WindowHandle : Colors)

Query all color names.

query_all_colors returns the names of all colors that are known to HALCON. That doesn't mean, that these colors are available for specific screens. On some screens there may only be a subset of colors available (see **query_color**). Before opening the first window, **set_system** can be used to define which and how many colors should be used. The HALCON colors are used to display regions (**disp_region**, **disp_polygon**, **disp_circle**, etc.). They can be defined with **set_color**.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window_id.
- ▷ **Colors** (output_control) string-array $\leadsto$ string
Color names.

Example

```
query_all_colors(WindowHandle,Colors)
<interactive selection from Colors provide ActColors> >
set_system('graphic_colors',ActColors)
open_window(0,0,1,1,'root','invisible','',WindowHandle)
query_color(WindowHandle,F)
close_window(WindowHandle)
fwrite_string(['Setting Colors: ',F]).
```

Result

query_all_colors always returns 2 (H_MSG_TRUE)

Parallelization Information

query_all_colors is *reentrant*, *local*, and processed *without* parallelization.

Possible Successors

set_system, **set_color**, **disp_region**

See also

query_color, **set_system**, **set_color**, **disp_region**, **open_window**, **open_textwindow**

Module

System

query_color (: : WindowHandle : Colors)

Query all color names displayable in the window.

query_color returns the names of all colors that are usable for region output (**disp_region**, **disp_polygon**, **disp_circle**, etc.). On a b/w screen **query_color** returns 'black' and 'white'. These two "colors" are displayable on any screen. In addition to 'black' and 'white' several grayvalues (e.g. 'dim gray') are returned on screens capable of grayvalues. A list of all displayable colors is returned for screens with color lookup table. The returned tuple of colors begins with b/w, followed by the three primaries ('red','green','blue') and several grayvalues. Before opening the first window it is furthermore possible to define the color list with **set_system(:,:, 'graphic_colors',...)**. **query_all_colors(:,:,WindowHandle:Colors)** returns a list of all available colors for the **set_system(:,:, 'graphic_colors',...)** call. For screens with truecolor output the same list is returned by **query_color**. The list of available colors (to HALCON) must not be confused with the list of displayable colors. For screens with truecolor output the available colors are only a small subset of the displayable colors. Colors that are not directly available to HALCON can be chosen manually with **set_rgb** or **set_hsi**. If colors are chosen that are known to HALCON but cannot be displayed, HALCON can choose a similar color. To use this feature, **set_check(:,:, '~color':)** must be set.

<i>Parameter</i>	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window_id.
▷ Colors (output_control)	string-array $\leadsto$ string Color names.

Example

```
open_window(0,0,-1,-1,'root','invisible','',WindowHandle)
query_color(WindowHandle,Colors)
close_window(WindowHandle)
fwrite_string(['Displayable colors: ',Farben]).
```

Result

[query_color](#) returns 2 (H_MSG_TRUE), if the window is valid. Otherwise an exception handling is raised.

Parallelization Information

[query_color](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Successors

[set_color](#), [disp_region](#)

See also

[query_all_colors](#), [set_color](#), [disp_region](#), [open_window](#), [open_textwindow](#)

Module

System

query_colored (: : : PossibleNumberOfColors)

Query the number of colors for color output.

[query_colored](#) returns all possible parameter values for [set_colored](#). [set_colored](#) defines how many colors are used for region or graphics output.

<i>Parameter</i>	
▷ PossibleNumberOfColors (output_control)	integer-array $\leadsto$ integer Tuple of the possible numbers of colors.

Example

```
regiongrowing(Image,Seg,5,5,6,100)
query_colored(Colors)
set_colored(WindowHandle,Colors[1])
disp_region(Seg,WindowHandle).
```

Result

[query_colored](#) always returns 2 (H_MSG_TRUE).

Parallelization Information

[query_colored](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[set_colored](#), [set_color](#), [disp_region](#)

Alternatives

[query_color](#)

See also

[set_colored](#), [set_color](#)

Module

System

query_gray (: : WindowHandle : Grayval)

Query the displayable grayvalues.

query_gray returns all grayvalues that are used for grayvalue output (**disp_image**) and that can be reproduced exactly in the window. They can be set with the **set_gray** call. The number of displayable grayvalues can be set with **set_system (:: 'num_gray_*', ...:)** before opening the first window.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ *integer*
Window_id.
- ▷ **Grayval** (output_control) integer-array $\leadsto$ *integer*
Tuple of all displayable grayvalues.

Result

query_gray returns 2 (H_MSG_TRUE), if the window is valid. Otherwise an exception handling is raised.

Parallelization Information

query_gray is *reentrant*, *local*, and processed *without parallelization*.

Possible Successors

set_gray, **disp_region**

See also

set_gray, **disp_image**

Module

System

query_insert (: : WindowHandle : Mode)

Query the possible graphic modes.

query_insert returns the possible modes pixels can be displayed in the output window. New pixels may e.g. overwrite old ones. In most of the cases there is a functional relationship between old and new values.

Possible display functions:

- 'copy': overwrite displayed pixels
- 'xor': display old "xor" new pixels
- 'complement': complement displayed pixels

"copy" is always available.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ *integer*
Window_id.
- ▷ **Mode** (output_control) string-array $\leadsto$ *string*
Display function name.

Result

query_insert returns 2 (H_MSG_TRUE), if the window is valid. Otherwise an exception handling is raised.

Parallelization Information

query_insert is *reentrant*, *local*, and processed *without parallelization*.

Possible Successors

set_insert, **disp_region**

See also

set_insert, **get_insert**

Module

System

query_line_width (: : : Min, Max)

Query the possible line widths.

query_line_width returns the minimal (**Min**) and maximal (**Max**) values of widths of region border which can be displayed. Setting of the border width is done with **set_line_width**. Border width is used by operators like **disp_region**, **disp_line**, **disp_circle**, **disp_rectangle1**, **disp_rectangle2** etc. if the drawing mode is "margin" (**set_draw**(:WindowHandle,'margin':)).

Parameter

- ▷ **Min** (output_control) integer $\leadsto$ integer
Displayable minimum width.
- ▷ **Max** (output_control) integer $\leadsto$ integer
Displayable maximum width.

Result

query_line_width returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.

Parallelization Information

query_line_width is *reentrant* and processed *without* parallelization.

Possible Successors

get_line_width, **set_line_width**, **set_line_style**, **disp_line**

See also

disp_circle, **disp_line**, **disp_rectangle1**, **disp_rectangle2**, **disp_region**,
set_line_width, **get_line_width**, **set_line_style**

Module

System

query_paint (: : WindowHandle : Mode)

Query the grayvalue display modes.

query_paint returns the names of all grayvalue display modes (e.g. 'gray', '3D-plot', 'contourline', etc.) for the output window. These modes are used by **set_paint**. **query_paint** only returns the names of the display values, not the additional parameters that may be necessary for some modes.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window_id.
- ▷ **Mode** (output_control) string-array $\leadsto$ string
Grayvalue display mode names.

Result

query_paint returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.

Parallelization Information

query_paint is *reentrant*, *local*, and processed *without* parallelization.

Possible Successors

get_paint, **set_paint**, **disp_image**

See also

set_paint, **get_paint**, **disp_image**

Module

System

query_shape (: : : DisplayShape)

Query the region display modes.

query_shape returns the names of all region display modes (e.g. 'original', 'circle', 'rectangle1', 'rectangle2', 'ellipse', etc.) for the window. They are used by **set_shape**.

Parameter

- ▷ **DisplayShape** (output_control) string-array $\leadsto$ *string*
 region display mode names.

Result

query_shape returns 2 (H_MSG_TRUE), if the window is valid. Otherwise an exception handling is raised.

Parallelization Information

query_shape is *reentrant* and processed *without* parallelization.

Possible Successors

get_shape, **set_shape**, **disp_region**

See also

set_shape, **get_shape**, **disp_region**

Module

System

set_color (: : WindowHandle, Color :)

Set output color.

set_color defines the colors for region output in the window. The available colors can be queried with the procedure **query_color**. The "colors" 'black' and 'white' are available for all screens. If colors are used that are not displayable on the screen, HALCON can choose a similar, displayable color of the output. For this, **set_check(::'~color':)** must be called. Furthermore, the list of available colors can be set with the procedure **set_system(::'graphic_colors',...)**. That must be done before opening the first output window. If only a single color is passed, all output is in this color. If a tuple of colors is passed, the output color of regions is modulo to the number of colors. In the example below, the first circle is displayed red, the second in green and the third in red again. HALCON always begins output with the first color passed. Note, that the number of output colors depends on the number of objects that are displayed in one procedure call. If only single objects are displayed, they always appear in the first color, even if the consist of more than one connected components.

The defined colors are used until **set_color**, **set_pixel**, **set_rgb**, **set_hsi** or **set_gray** is called again.

Colors are defined separately for each window. They can only be changed for the valid window.

Color is used in procedures with region output like **disp_region**, **disp_line**, **disp_rectangle1**, **disp_arrow** etc. It is also used by procedures with grayvalue output in certain output modes (e.g. '3D-plot', 'histogram', 'contourline', etc. See **set_paint**).

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ *integer*
 Window_id.
 ▷ **Color** (input_control) string(-array) $\leadsto$ *string*
 Output color names.
Default Value : 'white'

Example

```
set_color(WindowHandle, ['red', 'green'])
disp_circle(WindowHandle, [100, 200, 300], [200, 300, 100], [100, 100, 100]).
```

Result

set_color returns 2 (H_MSG_TRUE) if the window is valid and the passed colors are displayable on the screen. Otherwise an exception handling is raised.

Parallelization Information

set_color is *reentrant*, *local*, and processed *without* parallelization.

query_color	Possible Predecessors
disp_region	Possible Successors
set_rgb, set_hsi	Alternatives
get_rgb, disp_region, set_fix, set_paint	See also
System	Module

set_colored (: : WindowHandle, NumberOfColors :)

Set multiple output colors.

set_colored is a shortcut for certain **set_color** calls. It allows the user to display a region set in different colors. **NumberOfColors** defines the number of colors that are used. Valid values for **NumberOfColors** can be queried with **query_colored**. Furthermore, the list of available colors can be set with the procedure **set_system (:: 'graphic_colors', ... :)**. This must be done before opening the first output window.

	Parameter
▷ WindowHandle (input_control)	window $\leadsto$ integer Window_id.
▷ NumberOfColors (input_control)	integer $\leadsto$ integer Number of output colors. Default Value : 12 List of values : NumberOfColors $\in$ {3, 6, 12}

	Result
set_colored returns 2 (H_MSG_TRUE) if NumberOfColors is correct and the window is valid. Otherwise an exception handling is raised.	

set_colored is <i>reentrant</i> , <i>local</i> , and processed <i>without parallelization</i> .	Parallelization Information
query_colored, set_color	Possible Predecessors
disp_region	Possible Successors
query_colored, set_color, disp_region	See also
System	Module

set_comprise (: : WindowHandle, Mode :)

Define the image matrix output clipping.

set_comprise defines the image matrix output clipping. If **Mode** is set to 'object', only grayvalues belonging to the output object are displayed. If set to 'image', the whole image matrix is displayed. Default is 'object'.

	Attention
If Mode was set to 'image', undefined grayvalues may be displayed. Depending on the context they are black or can have random content. See the examples.	

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window_id.
- ▷ **Mode** (input_control) string $\leadsto$ string
Clipping mode for grayvalue output.
Default Value : 'object'
List of values : Mode $\in$ {'image', 'object'}

Example

```
open_window(0,0,-1,-1,'root','visible','',WindowHandle)
read_image(Image,'fabrik')
threshold(Image,Seg,100,255)
set_system('init_new_image','false')
sobel_amp(Image,Sob,'sum_abs',3)
disp_image(Sob,WindowHandle)
get_comprise(Mode)
fwrite_string(['Current mode for gray values: ',Mode])
fnew_line()
set_comprise(WindowHandle,'image')
get_mbutton(WindowHandle,_,_,_)
disp_image(Sob,WindowHandle)
fwrite_string(['Current mode for gray values: image'])
fnew_line().
```

Result

set_comprise returns 2 (H_MSG_TRUE) if **Mode** is correct and the window is valid. Otherwise an exception handling is raised.

Parallelization Information

set_comprise is *reentrant* and processed *without* parallelization.

Possible Predecessors

[get_comprise](#)

Possible Successors

[disp_image](#)

See also

[get_comprise](#), [disp_image](#), [disp_color](#)

Module

System

set_draw (: : WindowHandle, Mode :)

Define the region fill mode.

set_draw defines the region fill mode. If **Mode** is set to 'fill', output regions are filled, if set to 'margin', only contours are displayed. Setting **Mode** only affects the valid window. It is used by procedures with region output like [disp_region](#), [disp_circle](#), [disp_rectangle1](#), [disp_rectangle2](#), [disp_arrow](#) etc. It is also used by procedures with grayvalue output for some grayvalue output modes (e.g. 'histogram', see [set_paint](#)). If the mode is 'margin', the contour can be affected with [set_line_width](#), [set_line_approx](#) and [set_line_style](#).

Attention

If the output mode is 'margin' and the line width is more than one, objects may not be displayed.

Parameter
▷ WindowHandle (input_control) window $\leadsto$ integer Window_id. ▷ Mode (input_control) string $\leadsto$ string Fill mode for region output. Default Value : 'fill' List of values : Mode $\in$ {'fill', 'margin'}
Result
set_draw returns 2 (H_MSG_TRUE) if Mode is correct and the window is valid. Otherwise an exception handling is raised.
Parallelization Information
set_draw is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.
Possible Predecessors
get_draw
Possible Successors
disp_region
See also
get_draw , disp_region , set_paint , disp_image , set_line_width , set_line_style
Module
System

set_fix (: : WindowHandle, Mode :)

Set fixing of "look-up-table" (lut)

Behaviour for **Mode** = 'true': **set_fix** fixes that pixel lastly ascertained by one of the operators **set_gray**, **set_color**, **set_hsi** or **set_rgb** (Remark: Here a pixel is the index within the current look-up-table). To assign a new color to a fixed pixel set a color or gray value by using **set_color**, **set_rgb**, **set_hsi** or **set_gray**. This makes it possible to define any color (**set_color**), any gray value (**set_gray**) and any color combination (**set_rgb**, **set_hsi**) at any position of the look-up-table.

Mode set to 'false' reset the fixing. To modify or create a look-up-table process **set_pixel**, **set_fix** (: : WindowHandle, 'true':), **set_rgb** and **set_fix** (: : WindowHandle, 'false':) one after another.

Attention
As a side effect set_fix can change colors of "non-HALCON windows".
Parameter
▷ WindowHandle (input_control) window $\leadsto$ integer Window identifier. ▷ Mode (input_control) string $\leadsto$ string Mode of fixing. Default Value : 'true' List of values : Mode $\in$ {'true', 'false'}
Result
set_fix returns 2 (H_MSG_TRUE) if the window is valid, the hardware supports a look-up-table and all parameters are correct. Otherwise an exception handling is raised.
Parallelization Information
set_fix is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.
Possible Predecessors
get_fix
Possible Successors
set_pixel , set_rgb

See also

[get_fix](#), [set_pixel](#), [set_rgb](#), [set_color](#), [set_hsi](#), [set_gray](#)

Module

System

set_gray (: : WindowHandle, GrayValues :)

Define grayvalues for region output.

[set_gray](#) defines the grayvalues for region output. Grayvalues are defined as the range of the color lookup table that is used for grayvalue output with [disp_image](#) in conjunction with [set_paint](#) (: : WindowHandle, 'gray':). These entries can be modified by [set_lut](#). So a 'grayvalue' is the color in which a pixel with the same value is displayed (not necessarily really gray). In general, when changing the color lookup table with [set_lut](#), the colors of the displayed image will change too.

If a grayvalue is needed as a color for image output (i.e. no color changes with [set_lut](#) are possible), it can be set with [set_color](#) (: : WindowHandle, 'gray':).

If only a single grayvalue is passed, all output will take place in that grayvalue. If a tuple of grayvalues is passed, all output will take place in grayvalues modulo the number of tuple elements. In the example below, the first circle is displayed with grayvalue 100, the second with 200 and the third with 100 again. Every output procedure starts with the first grayvalue. Note, that the number of output grayvalues depends on the number of objects that are displayed in one procedure call. If only single objects are displayed, they always appear in the first grayvalue, even if they consist of more than one connected components.

When the procedures [set_gray](#), [set_color](#), [set_rgb](#), [set_hsi](#) are called, they overwrite the existing values. If not all grayvalues are displayable on the output device, the number range of [GrayValues](#) (0..255) is dithered to the range of displayable grayvalues. In any case 0 is displayed as black and 255 as white. The displayable grayvalues can be queried with the procedure [query_gray](#). Furthermore, the number of actually displayed grayvalues can be changed with [set_system](#) (: : 'num_gray_*', ... :). This must be done before opening the first window. With [set_check](#) (: : '~color':) error messages can be suppressed if a grayvalue can't be displayed on the screen. In that case, a similar grayvalue is displayed.

Parameter

▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window_id.

▷ **GrayValues** (input_control) integer(-array) $\leadsto$ integer
Grayvalues for region output.
Default Value : 255
Suggested values : $\text{GrayValues} \in \{0, 1, 2, 10, 16, 32, 64, 100, 120, 128, 250, 251, 252, 253, 254, 255\}$
Typical range of values : $0 \leq \text{GrayValues} \leq 255$

Example

```
set_gray(WindowHandle, [100, 200])
disp_circle(WindowHandle, [100, 200, 300], [200, 300, 100], [100, 100, 100]).
```

Result

[set_gray](#) returns 2 (H_MSG_TRUE) if [GrayValues](#) is displayable and the window is valid. Otherwise an exception handling is raised.

Parallelization Information

[set_gray](#) is *reentrant*, *local*, and processed *without parallelization*.

Possible Successors

[disp_region](#)

See also

[get_pixel](#), [set_color](#)

Module

System

set_hsi (: : WindowHandle, Hue, Saturation, Intensity :)

Define output colors (HSI-coded).

set_hsi sets the region output color(s)/grayvalue(s) for the valid window. Colors are passed as **Hue**, **Saturation**, and **Intensity**. Transformation from HSI to RGB is done with:

$$\begin{aligned}
 H &= (2\pi \text{Hue})/255 \\
 I &= (\sqrt{6} \text{Intensity})/255 \\
 M1 &= (\sin(H) \text{Saturation})/(255\sqrt{6}) \\
 M2 &= (\cos(H) \text{Saturation})/(255\sqrt{2}) \\
 R &= (2M1 + I)/(4\sqrt{6}) \\
 G &= (-M1 + M2 + I)/(4\sqrt{6}) \\
 B &= (-M1 - M2 + I)/(4\sqrt{6}) \\
 Red &= R * 255 \\
 Green &= G * 255 \\
 Blue &= B * 255
 \end{aligned}$$

If only one combination is passed, all output will take place in that color. If a tuple of colors is passed, the output color of regions and geometric objects is modulo to the number of colors. HALCON always begins output with the first color passed. Note, that the number of output colors depends on the number of objects that are displayed in one procedure call. If only single objects are displayed, they always appear in the first color, even if the consist of more than one connected components.

Selected colors are used until the next call of **set_color**, **set_pixel**, **set_rgb** or **set_gray**. Colors are relevant to windows, i.e. only the colors of the valid window can be set. Region output colors are used by operators like **disp_region**, **disp_line**, **disp_rectangle1**, **disp_rectangle2**, **disp_arrow**, etc. It is also used by procedures with grayvalue output in certain output modes (e.g. '3D-plot', 'histogram', 'contourline', etc. See **set_paint**).

Attention

The selected intensities may not be available for the selected hues. In that case, the intensities will be lowered automatically.

Parameter

- ▷ **WindowHandle** (input_control) window $\rightsquigarrow$ integer
Window_id.
- ▷ **Hue** (input_control) integer(-array) $\rightsquigarrow$ integer
Hue for region output.
Default Value : 30
Typical range of values : $0 \leq \text{Hue} \leq 255$
Restriction : $(0 \leq \text{Hue}) \wedge (\text{Hue} \leq 255)$
- ▷ **Saturation** (input_control) integer(-array) $\rightsquigarrow$ integer
Saturation for region output.
Default Value : 255
Typical range of values : $0 \leq \text{Saturation} \leq 255$
Restriction : $(0 \leq \text{Saturation}) \wedge (\text{Saturation} \leq 255)$
- ▷ **Intensity** (input_control) integer(-array) $\rightsquigarrow$ integer
Intensity for region output.
Default Value : 84
Typical range of values : $0 \leq \text{Intensity} \leq 255$
Restriction : $(0 \leq \text{Intensity}) \wedge (\text{Intensity} \leq 255)$

Result

set_hsi returns 2 (H_MSG.TRUE) if the window is valid and the output colors are displayable. Otherwise an exception handling is raised.

Parallelization Information

set_hsi is *reentrant*, *local*, and processed *without* parallelization.

get_hsi	Possible Predecessors
disp_region	Possible Successors
get_hsi , get_pixel , trans_from_rgb , trans_to_rgb , disp_region	See also
System	Module

set_icon (Icon : : WindowHandle :)

Icon definition for region output.

[set_icon](#) defines an icon for region output ([disp_region](#)). It is displayed in the regions center of gravity. The use of this icon is activated with [set_shape](#).

	Parameter
▷ Icon (input_object) region $\leadsto$ Hobject Icon for center of gravity.	
▷ WindowHandle (input_control) window $\leadsto$ integer Window_id.	
	Example

```
/* draw a region and an icon */
draw_region(&Region,WindowHandle) ;
draw_region(&Icon,WindowHandle) ;
set_icon(Icon) ;
set_shape(WindowHandle,"icon") ;
disp_region(Region,WindowHandle) ;
```

	Result
set_icon returns 2 (H_MSG_TRUE) if exactly one region is passed. Otherwise an exception handling is raised.	

set_icon is <i>reentrant</i> and processed <i>without</i> parallelization.	Parallelization Information

gen_circle , gen_ellipse , gen_rectangle1 , gen_rectangle2 , draw_region	Possible Predecessors
set_shape , disp_region	Possible Successors

System	Module

set_insert (: : WindowHandle, Mode :)

Define the pixel output function.

[set_insert](#) defines the function, with which pixels are displayed in the output window. It is e.g. possible for a pixel to overwrite the old value. In most of the cases there is a functional relationship between old and new values. The definition value is only valid for the valid window. Output procedures that honor **Mode** are e.g. [disp_region](#), [disp_polygon](#), [disp_circle](#).

Possible display functions are:

'copy': overwrite displayed pixels
'xor': display old "xor" new pixels

'complement': complement displayed pixels

There may not be all functions available, depending on the physical display. However, "copy" is always available.

Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window_id.
▷ Mode (input_control)	string $\leadsto$ string Name of the display function. Default Value : 'copy' List of values : Mode $\in$ {'copy', 'xor', 'complement'}
Result	
set_insert returns 2 (H_MSG_TRUE) if the parameter is correct and the window is valid. Otherwise an exception handling is raised.	
Parallelization Information	
set_insert is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.	
Possible Predecessors	
query_insert , get_insert	
Possible Successors	
disp_region	
See also	
get_insert , query_insert	
Module	
System	

set_line_approx (: : WindowHandle, Approximation :)

Define the approximation error for contour display.

set_line_approx defines the approximation error for region contour display in the window. **Approximation** values greater than zero cause a polygon approximation $\approx$ smoothing (with a maximum polygon/contour deviation of **Approximation** pixel). The approximation algorithm is the same as in **get_region_polygon**. **set_line_approx** is important for contour output via **set_line_style**.

Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window_id.
▷ Approximation (input_control)	integer $\leadsto$ integer Maximum deviation from the original contour. Default Value : 0 Typical range of values : $0 \leq \text{Approximation}$ Restriction : $\text{Approximation} \geq 0$
Example	

```
/* Calling */
set_line_approx(WindowHandle, Approximation)
set_draw(WindowHandle, 'margin')
disp_region(Obj, WindowHandle).

/* correspond with */
get_region_polygon(Obj, Approximation, Row, Col)
disp_polygon(WindowHandle, Row, Col).
```

Result	
set_line_approx returns 2 (H_MSG_TRUE) if the parameter is correct and the window is valid. Otherwise an exception handling is raised.	

	<i>Parallelization Information</i>	
set_line_approx is <i>reentrant</i> and processed <i>without</i> parallelization.		
	<i>Possible Predecessors</i>	
get_line_approx		
	<i>Possible Successors</i>	
disp_region		
	<i>Alternatives</i>	
get_region_polygon , disp_polygon		
	<i>See also</i>	
get_line_approx , set_line_style , set_draw , disp_region		
	<i>Module</i>	
System		

set_line_style (: : WindowHandle, Style :)

Define a contour output pattern.

[set_line_style](#) defines the output pattern of region contours. The information is used by procedures like [disp_region](#), [disp_line](#), [disp_polygon](#) etc. The current value can be queried with [get_line_style](#). **Style** contains up to five pairs of values. The first value is the length of the visible contour part, the second is the length of the invisible part. The value pairs are used cyclical for contour output.

	<i>Attention</i>	
set_line_style does an implicit polygon approximation (see set_line_approx (: : WindowHandle, 3:)). It is only possible to enlarge it with set_line_approx .		

	<i>Parameter</i>	
▷ WindowHandle (input_control) window $\leadsto$ <i>integer</i> Window_id. ▷ Style (input_control) integer-array $\leadsto$ <i>integer</i> Contour pattern. Default Value : '[]'		

	<i>Example</i>	

```
* stroke line: X-Windows
set_line_style(WindowHandle, [20,7])
* point-stroke line: X-Windows
set_line_style(WindowHandle, [20,7,3,7])
* passing line (standard)
set_line_style(WindowHandle, [])
```

	<i>Result</i>	
set_line_style returns 2 (H_MSG_TRUE) if the parameter is correct and the window is valid. Otherwise an exception handling is raised.		

	<i>Parallelization Information</i>	
set_line_style is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.		
	<i>Possible Predecessors</i>	
get_line_style		
	<i>Possible Successors</i>	
disp_region		
	<i>See also</i>	
get_line_style , set_line_approx , disp_region		
	<i>Module</i>	
System		

set_line_width (: : WindowHandle, Width :)

Define the line width for region contour output.

set_line_width defines the line width (in pixel) in which a region contour is displayed (e.g. with **disp_region**, **disp_line**, **disp_polygon**, etc.) The procedure **get_line_width** returns the current value for the window. Some output devices may not allow to change the contour width. If it is possible for the current device, it can be queried with **query_line_width**.

Attention

The line width is important if the output mode was set to 'margin' (see **set_draw**). If the line width is greater than one, regions may not always be displayed correctly.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window_id.
- ▷ **Width** (input_control) integer $\leadsto$ integer
Line width for region output in contour mode.
Default Value : 1
Restriction : $(\text{Width} \geq 1) \wedge (\text{Width} \leq 2000)$

Result

set_line_width returns 2 (H_MSG_TRUE) if the parameter is correct and the window is valid. Otherwise an exception handling is raised.

Parallelization Information

set_line_width is *reentrant* and processed *without parallelization*.

Possible Predecessors

query_line_width, **get_line_width**

Possible Successors

disp_region

See also

get_line_width, **query_line_width**, **set_draw**, **disp_region**

Module

System

set_paint (: : WindowHandle, Mode :)

Define the grayvalue output mode.

set_paint defines the output mode for grayvalue display (single- or multichannel) in the window. The mode is used by **disp_image**.

This page describes the different modes, that can be used for grayvalue output. It should be noted, that the mode 'default' is the most suitable.

The hardware characteristics determine how grayvalues can be displayed. On a screen with one to seven bit planes, only binary data can be displayed. On screens with at least eight bit planes, it is possible to display multiple grayvalues. For binary displays HALCON includes algorithms with dithering matrix (fast, but low resolution), minimal error (good, but slow) and thresholding. Using the thresholding algorithm, the threshold can be passed as a second parameter (a tuple with the string 'threshold' and the actual threshold, e.g.: ['threshold', 100]).

Displays with eight bit planes use approximately 200 grayvalues for output. Of course it is still possible to use a binary display on those displays.

A different way to display grayvalues is the histogram (mode: 'histogram'). This mode has two additional parameter values: Row (second value) and column (third value). They denote row and column of the histogram center for positioning on the screen. The scale factor (fourth value) determines the histogram size: a scale factor of 1 distinguishes 256 grayvalues, 2 distinguishes 128 grevalues, and so on. The four values are passed as a tuple, e.g. ['histogram', 256, 256, 1]. If only the first value is passed ('histogram'), the other values are set to defaults or

the last values, respectively. For histogram computation see `gray_histo`. Histogram output honors the same parameters as procedures like `disp_region` etc. (e.g. `set_color`, `set_draw`, etc.)

Yet another mode is the display of relative frequencies of the number of connection components (""component_histogram"). For informations on computing the component histogram see `shape_histo_all`). Positioning and resolution are exactly as in the mode 'histogram'.

In mode 'mean', all object regions are displayed in their mean grayvalue.

The modes 'row' and 'column' allow the display of lines or columns, respectively. The position (line- and columnindex) is passed with the second parameter value. The third parameter value is the scale factor in percent (100 means 1 pixel per grayvalue, 50 means one pixel per two grayvalues).

Gray images can also be interpreted as 3d data, depending on the grayvalue. To view these 3d plots, select the modes 'contourline', '3D-plot' or '3D-plot_hidden'.

Two-channel images can be viewed as 'vectorfield'.

Three-channel images are interpreted as RGB images. They can be displayed in three different modes. Two of them can be optimized by Floyd-Steinberg dithering.

Parameters for modes that need more than one parameter can be passed the following ways:

- Only the name of the mode is passed: the defaults or the last values are used, respectively. Example: `set_paint(::WindowHandle, 'contourline':)`
- All values are passed: all output characteristics can be set. Example: `set_paint(::WindowHandle, ['contourline', 10, 1]:)`
- Only the first n values are passed: only the passed values are changed. Example: `set_paint(::WindowHandle, ['contourline', 10]:)`
- Some of the values are replaced by an asterisk (*): The value of the replaced parameters is not changed. Example: `set_paint(::WindowHandle, ['contourline', '*', 1]:)`

If the current mode is 'default', HALCON chooses a suitable algorithm for the output of 2- and 3-channel images.. No `set_paint` call is necessary in this case.

Apart from `set_paint` there are other procedures that affect the output of grayvalues. The most important of them are `set_part`, `set_part_style`, `set_lut` and `set_lut_style`. Some output modes display grayvalues using region output (e.g. 'histogram', 'contourline', '3D-plot', etc.). In these modes, parameters set with `set_color`, `set_rgb`, `set_hsi`, `set_pixel`, `set_shape`, `set_line_width` and `set_insert` influence grayvalue output. This can lead to unexpected results when using `set_shape(::'convex':)` and `set_paint(::WindowHandle, 'histogram':)`. Here the convex hull of the histogram is displayed.

Modes:

- one-channel images:
 - 'default' optimal display on given hardware
 - 'gray' grayvalue output
 - 'mean' mean grayvalue
 - 'dither4_1' binary image, dithering matrix 4x4
 - 'dither4_2' binary image, dithering matrix 4x4
 - 'dither4_3' binary image, dithering matrix 4x4
 - 'dither8_1' binary image, dithering matrix 8x8
 - 'floyd_steinberg' binary image, optimal grayvalue simulation
 - ['threshold', Threshold]
 - 'threshold' binary image, threshold: 128 (default)
 - ['threshold', 200] binary image, any threshold: (here: 200)
 - ['histogram', Line, Column, Scale]
 - 'histogram' grayvalue output as histogram.
 - position default: max. size, in the window center
 - ['histogram', 256, 256, 2] grayvalue output as histogram, any parameter values.
 - positioning: window center (here (256,256))
 - size: (here 2, half the max. size)

['component_histogram', Line, Column, Scale]

'component_histogram' output as histogram of the connection components.

Positioning: default

['component_histogram', 256, 256, 1] output as histogram of the connection components.

Positioning: (here (256, 256))

Scaling: (here 1, max. size)

['row', Line, Scale]

'row' output of the grayvalue profile along the given line.

line: image center (default)

Scaling: 50

['row', 100, 20] output of the grayvalue profile of line 100 with a scaling of 0.2 (20)

['column', Column, Scale]

'column' output of the grayvalue profile along the given column.

column: image center (default)

Scaling: 50

['column', 100, 20] output of the grayvalue profile of column 100 with a scaling of 0.2 (20)

['contourline', Step, Colored]

'contourline' grayvalue output as contour lines: the grayvalue difference per line is defined with the parameter 'Step' (default: 30, i.e. max. 8 lines for 256 grayvalues). The line can be displayed in a given color (see set_color) or in the grayvalue they represent. This behaviour is defined with the parameter 'Colored' (0 = color, 1 = grayvalues). Default is color.

['contourline', 15, 1] grayvalue output as contour lines with a step of 15 and gray output.

['3D-plot', Step, Colored, EyeHeight, EyeDistance, ScaleGray, LinePos, ColumnPos]

'3D-plot' grayvalues are interpreted as 3d data: the greater the value, the 'higher' the assumed mountain. Lines with step 2 (second parameter value) are drawn along the x- and y-axes. The third parameter (Colored) determines, if the output should be in color (default) or grayvalues. To define the projection of the 3d data, use the parameters EyeHeight and EyeDistance. The projection parameters take values from 0 to 255. ScaleGray defines a factor, by which the grayvalues are multiplied for 'height' interpretation (given in percent. 100EyeHeight and EyeDistance the image can be shifted out of place. Use RowPos and ColumnPos to move the whole output. Values from -127 to 127 are possible.

['3D-plot', 5, 1, 110, 160, 150, 70, -10] line step: 5 pixel

Colored: yes (1)

EyeHeight: 110

EyeDistance: 160

ScaleGray: 1.5 (150)

RowPos: 70 pixel down

ColumnPos: 10 pixel right

['3D-plot_hidden', Step, Colored, EyeHeight, EyeDistance, ScaleGray, LinePos, ColumnPos]

'3D-plot_hidden' like '3D-plot', but computes hidden lines.

- Two-channel images:

'default' output as vector field.

['vectorfield', Step, MinLength, ScaleLength]

'vectorfield' output of the two channels as displacement vector field. Procedures like optical_flow* produce images of pixel displacements (first channel: x-displacement, second channel: y-displacement).

In this mode, an arrow is drawn for each vector. The parameter Step has to be set in correspondence to optical_flow*. Short vectors can be suppressed by the third parameter value (MinLength). The fourth parameter value scales the vector length. Vectors, that exceed the window boundaries are not drawn.

['vectorfield', 16, 2, 3] Output of every 16. vector, that is longer than 2 pixel. Each vector is multiplied by 3 for output.

- Three-channel images:

'default' output as RGB image with 'median_cut'.

- 'television' color addition algorithm for RGB images: (three components necessary for `disp_image`). Images are displayed via a fixed color lookup table. Fast, but non-optimal color resolution. Only recommended on bright screens.
- 'grid_scan' grid-scan algorithm for RGB images (three components necessary for `disp_image`). An optimized color lookup table is generated for each image. Slower than 'television'. Disadvantages: Hard color boundaries (no dithering). Different color lookup table for every image.
- 'grid_scan_floyd_steinberg' grid-scan with Floyd-Steinberg dithering for smooth color boundaries.
- 'median_cut' median-cut algorithm for RGB images (three components necessary for `disp_image`). Similar to grid-scan. Disadvantages: Hard color boundaries (no dithering). Different color lookup table for every image.
- 'median_cut_floyd_steinberg' median-cut algorithm with Floyd-Steinberg dithering for smooth color boundaries.

Attention

- Display of color images ('television', 'grid_scan', etc.) changes the color lookup tables.
- If a wrong color mode is set, the error message may appear not until the `disp_image` call.
- Grayvalue output may be influenced by region output parameters. This can yield unexpected results.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window_id.
- ▷ **Mode** (input_control) string-array $\leadsto$ string / integer
Output mode. Additional parameters possible.
Default Value : 'default'
List of values : Mode $\in$ {"default", "histogram", "row", "column", "contourline", "3D-plot", "3D-plot_hidden", "3D-plot_point"}

Example

```
read_image(Image, 'fabrik')
open_window(0, 0, -1, -1, 'root', 'visible', "", WindowHandle)
query_paint(WindowHandleModi)
fwrite_string(['available gray value modes: ', Modi])
fnew_line()
disp_image(Image, WindowHandle)
get_mbutton(WindowHandle, __, __, __)
set_color(WindowHandle, 'red')
set_draw(WindowHandle, 'margin')
set_paint(WindowHandle, 'histogram')
disp_image(Image, WindowHandle)
set_color(WindowHandle, 'blue')
set_paint(WindowHandle, ['histogram', 100, 100, 3])
disp_image(Image, WindowHandle)
set_color(WindowHandle, 'yellow')
set_paint(WindowHandle, ['row', 100])
disp_image(Image, WindowHandle)
get_mbutton(WindowHandle, __, __, __)
clear_window(WindowHandle)
set_paint(WindowHandle, ['contourline', 10, 1])
disp_image(Image, WindowHandle)
set_lut(WindowHandle, 'color')
get_mbutton(WindowHandle, __, __, __)
clear_window(WindowHandle)
set_part(WindowHandle, 100, 100, 300, 300)
set_paint(WindowHandle, '3D-plot')
disp_image(Image, WindowHandle).
```

	<i>Result</i>
<code>set_paint</code>	returns 2 (H_MSG_TRUE) if the parameter is correct and the window is valid. Otherwise an exception handling is raised.
	<i>Parallelization Information</i>
<code>set_paint</code>	is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.
	<i>Possible Predecessors</i>
<code>query_paint</code> , <code>get_paint</code>	
	<i>Possible Successors</i>
<code>disp_image</code>	
	<i>See also</i>
<code>get_paint</code> , <code>query_paint</code> , <code>disp_image</code> , <code>set_shape</code> , <code>set_rgb</code> , <code>set_color</code> , <code>set_gray</code>	
	<i>Module</i>
System	

set_part (: : WindowHandle, Row1, Column1, Row2, Column2 :)

Modify the displayed image part.

`set_part` modifies the image part that is displayed in the window. (`Row1,Column1`) denotes the upper left corner and (`Row2,Column2`) the lower right corner of the image part to display. The changed values are used by grayvalue output operators (`disp_image`, `disp_color`) as well as region output operators (`disp_region`).

If only part of an image is displayed, it will be zoomed to full window size. The zooming interpolation method can be set with `set_part_style`. `get_part` returns the values of the image part to display.

Beside setting the image part directly, the following special modes are supported:

Row1 = Column1 = Row2 = Column2 = -1: The window size is chosen as the image part, i.e. no zooming of the image will be performed.

Row1, Column1 > -1 and Row2 = Column2 = -1: The size of the last displayed image (in this window) is chosen as the image part, i.e. the image can completely be displayed in the image. For this the image will be zoomed if necessary.

	<i>Attention</i>
After a call of <code>set_part</code> , some operators like <code>draw_region</code> , <code>draw_ellipse</code> , etc. can no longer be used.	
	<i>Parameter</i>

- ▷ **WindowHandle** (input_control) window $\rightsquigarrow$ integer
Window_id.
- ▷ **Row1** (input_control) rectangle.origin.y $\rightsquigarrow$ integer
Row of the upper left corner of the chosen image part.
Default Value : 0
- ▷ **Column1** (input_control) rectangle.origin.x $\rightsquigarrow$ integer
Column of the upper left corner of the chosen image part.
Default Value : 0
- ▷ **Row2** (input_control) rectangle.corner.y $\rightsquigarrow$ integer
Row of the lower right corner of the chosen image part.
Default Value : -1
Restriction : ($\text{Row2} \geq \text{Row1}$) $\vee$ ($\text{Row2} = -1$)
- ▷ **Column2** (input_control) rectangle.corner.x $\rightsquigarrow$ integer
Column of the lower right corner of the chosen image part.
Default Value : -1
Restriction : ($\text{Column2} \geq \text{Column1}$) $\vee$ ($\text{Column2} = -1$)

	<i>Example</i>

```
get_system('width',,Width)
```

```

get_system('height', Height)
set_part(WindowHandle, 0, 0, Height-1, Width-1)
disp_image(Image, WindowHandle)
draw_rectangle1(WindowHandle:Row1, Column1, Row2, Column2)
set_part(WindowHandle, Row1, Column1, Row2, Column2)
disp_image(Image, WindowHandle).

```

	<i>Result</i>
<code>set_part</code>	returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.
	<i>Parallelization Information</i>
<code>set_part</code>	is <i>reentrant</i> and processed <i>without</i> parallelization.
	<i>Possible Predecessors</i>
<code>get_part</code>	
	<i>Possible Successors</i>
<code>set_part_style</code> , <code>disp_image</code> , <code>disp_region</code>	
	<i>Alternatives</i>
<code>affine_trans_image</code>	
	<i>See also</i>
<code>get_part</code> , <code>set_part_style</code> , <code>disp_region</code> , <code>disp_image</code> , <code>disp_color</code>	
	<i>Module</i>
System	

set_part_style (: : WindowHandle, Style :)

Define an interpolation method for grayvalue output.

`set_part_style` defines the interpolation method to zoom an image part which is displayed in the window. Interpolation takes place, if the output window has different size than the image to display (e.g. after a call to `set_part` or a window resize). Three modes are supported:

- 0 no interpolation (low quality, very fast).
- 1 unweighted interpolation (medium quality and run time)
- 2 weighted interpolation (high quality, slow)

The current value can be queried with `get_part_style`.

	<i>Parameter</i>
▷ WindowHandle (input_control)	window $\rightsquigarrow$ integer Window_id.
▷ Style (input_control)	integer $\rightsquigarrow$ integer Interpolation method for image output: 0 (fast, low quality) to 2 (slow, high quality). Default Value : 0 List of values : Style $\in$ {0, 1, 2}

	<i>Result</i>
<code>set_part_style</code>	returns 2 (H_MSG_TRUE) if the parameter is correct and the window is valid. Otherwise an exception handling is raised.
	<i>Parallelization Information</i>
<code>set_part_style</code>	is <i>reentrant</i> and processed <i>without</i> parallelization.
	<i>Possible Predecessors</i>
<code>get_part_style</code>	
	<i>Possible Successors</i>
<code>set_part</code> , <code>disp_image</code> , <code>disp_region</code>	
	<i>Alternatives</i>
<code>affine_trans_image</code>	

See also

[get_part_style](#), [set_part](#), [disp_image](#), [disp_color](#)

Module

System

set_pixel (: : WindowHandle, Pixel :)

Define a color lookup table index.

[set_pixel](#) sets pixel values: colors ([set_color](#), [set_rgb](#), etc.) and grayvalues ([set_gray](#)) are coded together into a number, called pixel. This 'pixel' is an index in the color lookup table. It ranges from 0 to 1 in b/w images and 0 to 255 color images with 8 bit planes. It is different from the 'pixel' ("picture element") in image processing. Therefore HALCON distinguishes between pixel and image element (or grayvalue).

The current value can be queried with [get_pixel](#).

Parameter

▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window_id.

▷ **Pixel** (input_control) integer(-array) $\leadsto$ integer
Color lookup table index.

Default Value : 128

Typical range of values : $0 \leq \text{Pixel} \leq 255$

Result

[set_pixel](#) returns 2 (H_MSG_TRUE) if the parameter is correct and the window is valid. Otherwise an exception handling is raised.

Parallelization Information

[set_pixel](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[get_pixel](#)

Possible Successors

[disp_image](#), [disp_region](#)

Alternatives

[set_rgb](#), [set_color](#), [set_hsi](#)

See also

[get_pixel](#), [set_lut](#), [disp_region](#), [disp_image](#), [disp_color](#)

Module

System

set_rgb (: : WindowHandle, Red, Green, Blue :)

Set the color definition via RGB values.

[set_rgb](#) sets the output color(s) or the grayvalues, respectively, for region output for the window. The colors are defined with the red, green and blue components. If only one combination is passed, all output takes place in that color. If a tuple is passed, region output and output of geometric objects takes place modulo the passed colors.

For every call of an output procedure, output is started with the first color. If only one object is displayed per call, it will always be displayed in the first color. This is even true for objects with multiple connection components. If multiple objects are displayed per procedure call, multiple colors are used. The defined colors are used until [set_color](#), [set_pixel](#), [set_rgb](#) or [set_gray](#) is called again. The values are used by procedures like [disp_region](#), [disp_line](#), [disp_rectangle1](#), [disp_rectangle2](#), [disp_arrow](#), etc.

Attention

If a passed is not available, an exception handling is raised. If [set_check\(:: '~color' :\)](#) was called before, HALCON uses a similar color and suppresses the error.

Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window_id.
▷ Red (input_control)	integer(-array) $\leadsto$ integer Red component of the color. Default Value : 255 Typical range of values : $0 \leq \text{Red} \leq 255$ Restriction : $(0 \leq \text{Red}) \wedge (\text{Red} \leq 255)$
▷ Green (input_control)	integer(-array) $\leadsto$ integer Green component of the color. Default Value : 0 Typical range of values : $0 \leq \text{Green} \leq 255$ Restriction : $(0 \leq \text{Green}) \wedge (\text{Green} \leq 255)$
▷ Blue (input_control)	integer(-array) $\leadsto$ integer Blue component of the color. Default Value : 0 Typical range of values : $0 \leq \text{Blue} \leq 255$ Restriction : $(0 \leq \text{Blue}) \wedge (\text{Blue} \leq 255)$
Result	
set_rgb returns 2 (H_MSG_TRUE) if the window is valid and all passed colors are available and displayable. Otherwise an exception handling is raised.	
Parallelization Information	
set_rgb is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.	
Possible Successors	
disp_image , disp_region	
Alternatives	
set_hsi , set_color , set_gray	
See also	
set_fix , disp_region	
Module	
System	

set_shape (: : WindowHandle, Shape :)

Define the region output shape.

set_shape defines the shape for region output. It is only valid for the window with the logical window number **WindowHandle**. The output shape is used by **disp_region**. The available shapes can be queried with **query_shape**.

Available modes:

'original': The shape is displayed unchanged. Nevertheless modifications via parameters like **set_line_width** or **set_line_approx** can take place. This is also true for all other modes.

'outer_circle': Each region is displayed by the smallest surrounding circle. (See **smallest_circle**.)

'inner_circle': Each region is displayed by the largest included circle. (See **inner_circle**.)

'ellipse': Each region is displayed by an ellipse with the same moments and orientation (See **elliptic_axis**.)

'rectangle1': Each region is displayed by the smallest surrounding rectangle parallel to the coordinate axes. (See **smallest_rectangle1**.)

'rectangle2': Each region is displayed by the smallest surrounding rectangle. (See **smallest_rectangle2**.)

'convex': Each region is displayed by its convex hull (See **convexity**.)

'icon': Each region is displayed by the icon set with **set_icon** in the center of gravity.

Attention

Caution is advised for grayvalue output procedures with output parameter settings that use region output, e.g. `disp_image` with `set_paint(::WindowHandle,'histogram')` and `set_shape(::WindowHandle,'convex')`. In that case the convex hull of the grayvalue histogram is displayed.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ *integer*
Window_id.
- ▷ **Shape** (input_control) string $\leadsto$ *string*
Region output mode.
Default Value : 'original'
List of values : Shape $\in$ {'original', 'convex', 'outer_circle', 'inner_circle', 'rectangle1', 'rectangle2', 'ellipse', 'icon'}

Example

```
read_image(Image,'fabrik')
regiongrowing(Image,Seg,5,5,6,100)
set_colored(WindowHandle,12)
set_shape(WindowHandle,'rectangle2')
disp_region(Seg,WindowHandle).
```

Result

`set_shape` returns 2 (H_MSG_TRUE) if the parameter is correct and the window is valid. Otherwise an exception handling is raised.

Parallelization Information

`set_shape` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`set_icon`, `query_shape`, `get_shape`

Possible Successors

`disp_region`

See also

`get_shape`, `query_shape`, `disp_region`

Module

System

6.7 Text

get_font (: : WindowHandle : Font)

Get the current font.

`get_font` queries the name of the font used in the output window. The font is used by the operators `write_string`, `read_string` etc. The font is set by the operator `set_font`. Text windows as well as windows for image display use fonts. Both types of windows have a default font that can be modified with `set_system('default.font',Fontname)` prior to opening the window. A list of all available fonts can be obtained using `query_font`.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ *integer*
Window identifier.
- ▷ **Font** (output_control) string $\leadsto$ *string*
Name of the current font.

Example

```

get_font(WindowHandle,CurrentFont)
set_font(WindowHandle,MyFont)
write_string(WindowHandle,['The name of my Font is:',Myfont])
new_line(WindowHandle)
set_font(WindowHandle,CurrentFont)

```

Result

[get_font](#) returns 2 (H_MSG_TRUE).

Parallelization Information

[get_font](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[open_window](#), [open_textwindow](#), [query_font](#)

Possible Successors

[set_font](#)

See also

[set_font](#), [query_font](#), [open_window](#), [open_textwindow](#), [set_system](#)

Module

System

get_string_extents (: : WindowHandle, Values : Ascent, Descent, Width, Height)

Get the spatial size of a string.

[get_string_extents](#) queries width and height of the output size of a string using the font of the window. In addition the extension above and below the current baseline for writing is returned ([Ascent](#) bzw. [Descent](#)).

The sizes are measured in the coordinate system of the window (for text windows in pixels). Using [get_string_extents](#) it is possible to determine text output and input independently from the used font. The conversion from integer numbers and floating point numbers to text strings is the same as in [write_string](#).

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window identifier.
- ▷ **Values** (input_control) string(-array) $\leadsto$ string / real / integer
Values to consider.
Default Value : 'test_string'
- ▷ **Ascent** (output_control) integer $\leadsto$ integer
Maximum height above baseline for writing.
- ▷ **Descent** (output_control) integer $\leadsto$ integer
Maximum extension below baseline for writing.
- ▷ **Width** (output_control) integer $\leadsto$ integer
Text width.
- ▷ **Height** (output_control) integer $\leadsto$ integer
Text height.

Result

[get_string_extents](#) returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.

Parallelization Information

[get_string_extents](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[open_window](#), [open_textwindow](#), [set_font](#)

Possible Successors

[set_tposition](#), [write_string](#), [read_string](#), [read_char](#)

See also

[set_tposition](#), [set_font](#)

Module

System

get_tposition (: : WindowHandle : Row, Column)

Get cursor position.

get_tposition queries the current position of the text cursor in the output window. The position is measured in the coordinate system of the window (in pixels for text windows). The next output of text in this window starts at the cursor position. The left end of the baseline for writing the next string (not considering descenders) is placed on this position. The position is changed by the output or input of text (**write_string**, **read_string**) or by an explicit change of position by (**set_tposition**, **new_line**).

Attention

If the output text does not fit completely into the window, an exception handling is raised. This can be avoided by **set_check**('~text').

Parameter

- ▷ **WindowHandle** (input_control) window $\rightsquigarrow$ integer
Window identifier.
- ▷ **Row** (output_control) point.y $\rightsquigarrow$ integer
Row index of text cursor position.
- ▷ **Column** (output_control) point.x $\rightsquigarrow$ integer
Column index of text cursor position.

Result

get_tposition returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.

Parallelization Information

get_tposition is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

open_window, **open_textwindow**, **set_font**

Possible Successors

set_tposition, **write_string**, **read_string**, **read_char**

See also

new_line, **read_string**, **set_tposition**, **write_string**, **set_check**

Module

System

get_tshape (: : WindowHandle : TextCursor)

Get the shape of the text cursor.

get_tshape queries the shape of the text cursor for the output window. A new cursor shape is set by the operator **set_tshape**.

A text cursor marks the current position for text output (which can also be invisible). It is different from the mouse cursor (although both will be called "cursor" if the context makes misconceptions impossible). The available shapes for the text cursor can be queried with **query_tshape**.

Parameter

- ▷ **WindowHandle** (input_control) window $\rightsquigarrow$ integer
Window identifier.
- ▷ **TextCursor** (output_control) string $\rightsquigarrow$ string
Name of the current text cursor.

Result

get_tshape returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.

Parallelization Information

`get_tshape` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`open_window`, `open_textwindow`, `set_font`

Possible Successors

`set_tshape`, `set_tposition`, `write_string`, `read_string`, `read_char`

See also

`set_tshape`, `query_tshape`, `write_string`, `read_string`

Module

System

<code>new_line</code> (: : <code>WindowHandle</code> :)

Set the position of the text cursor to the beginning of the next line.

`new_line` sets the position of the text cursor to the beginning of the next line. The new position depends on the current font. The left end of the baseline for writing the following text string (not considering descenders) is placed on this position.

If the next line does not fit into the window the content of the window is scrolled by the height of one line in the upper direction. In order to reach the correct new cursor position the font used in the next line must be set before `new_line` is called. The position is changed by the output or input of text (`write_string`, `read_string`) or by an explicit change of position by (`set_tposition`).

Parameter

- ▷ **`WindowHandle`** (input_control) window $\rightsquigarrow$ *integer*
Window identifier.

Result

`new_line` returns 2 (H_MSG_TRUE) if the window is valid. Otherwise an exception handling is raised.

Parallelization Information

`new_line` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`open_window`, `open_textwindow`, `set_font`, `write_string`

Alternatives

`get_tposition`, `get_string_extents`, `set_tposition`, `move_rectangle`

See also

`write_string`, `set_font`

Module

System

<code>query_font</code> (: : <code>WindowHandle</code> : <code>Font</code>)

Query the available fonts.

`query_font` queries the fonts available for text output in the output window. They can be set with the operator `set_font`. Fonts are used by the operators `write_string`, `read_char`, `read_string` and `new_line`.

Attention

For different machines the available fonts may differ a lot. Therefore `query_font` will return different fonts on different machines.

Parameter

- ▷ **`WindowHandle`** (input_control) window $\rightsquigarrow$ *integer*
Window identifier.
- ▷ **`Font`** (output_control) string-array $\rightsquigarrow$ *string*
Tupel with available font names.

Example

```

open_window(0,0,-1,-1,'root','visible','',WindowHandle)
set_check('~text')
query_font(WindowHandle,Fontlist)
set_color(WindowHandle,'white')
for i=0 to |Fontlist|-1 by 1
    set_font(WindowHandle,Fontlist[i])
    write_string(WindowHandle,Fontlist[i])
    new_line(WindowHandle)
endfor

```

Result

[query_font](#) returns 2 (H_MSG_TRUE).

Parallelization Information

[query_font](#) is *reentrant*, *local*, and processed without parallelization.

Possible Predecessors

[open_window](#), [open_textwindow](#)

Possible Successors

[set_font](#), [write_string](#), [read_string](#), [read_char](#)

See also

[set_font](#), [write_string](#), [read_string](#), [read_char](#), [new_line](#)

Module

System

query_tshape (: : WindowHandle : TextCursor)

Query all shapes available for text cursors.

[query_tshape](#) queries the available shapes of text cursors for the output window. The retrieved shapes can be used by the operator [set_tshape](#).

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ *integer*
Window identifier.
- ▷ **TextCursor** (output_control) string-array $\leadsto$ *string*
Names of the available text cursors.

Result

[query_tshape](#) returns 2 (H_MSG_TRUE).

Parallelization Information

[query_tshape](#) is *reentrant*, *local*, and processed without parallelization.

Possible Predecessors

[open_window](#), [open_textwindow](#)

Possible Successors

[set_tshape](#), [write_string](#), [read_string](#)

See also

[set_tshape](#), [get_shape](#), [set_tposition](#), [write_string](#), [read_string](#)

Module

System

read_char (: : WindowHandle : Char, Code)

Read a character from a text window.

read_char reads a character from the keyboard in the input window (= output window). If the character is printable it is returned in **Char**. If a control key has been pressed, this will be indicated by the value of **Code**. Some important keys are recognizable by this value. Possible values are:

'character': printable character
'left': cursor left
'right': cursor right
'up': cursor up
'down': cursor down
'insert': insert
'none': none of these keys

Attention

The window has to be a text window.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window identifier.
- ▷ **Char** (output_control) string $\leadsto$ string
Input character (if it is not a control character).
- ▷ **Code** (output_control) string $\leadsto$ string
Code for input character.

Result

read_char returns 2 (H_MSG_TRUE) if the text window is valid. Otherwise an exception handling is raised.

Parallelization Information

read_char is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

open_textwindow, **set_font**

Alternatives

read_string, **fread_char**, **fread_string**

See also

write_string, **set_font**

Module

System

read_string (: : WindowHandle, InString, Length : OutString)

Read a string in a text window.

read_string reads a string with a predetermined maximum size (**Length**) from the keyboard in the input window (= output window). The string is read from the current position of the text cursor using the current font. The maximum size has to be small enough to keep the string within the right window boundary. A default string which can be edited or simply accepted by the user may be provided. After text input the text cursor is positioned at the end of the edited string. Commands for editing:

RETURN finish input

BACKSPACE delete the character on the left side of the cursor and move the cursor to this position.

Attention

The window has to be a text window.

Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window identifier.
▷ InString (input_control)	string $\leadsto$ string Default string (visible before input). Default Value : ”
▷ Length (input_control)	integer $\leadsto$ integer Maximum number of characters. Default Value : 32 Restriction : Length > 0
▷ OutString (output_control)	string $\leadsto$ string Read string.
Result	
read_string returns 2 (H_MSG.TRUE) if the text window is valid and a string of maximal length fits within the right window boundary. Otherwise an exception handling is raised.	
Parallelization Information	
read_string is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.	
Possible Predecessors	
open_textwindow , set_font	
Alternatives	
read_char , fread_string , fread_char	
See also	
set_tposition , new_line , open_textwindow , set_font , set_color	
Module	
System	

set_font (: : WindowHandle, Font :)

Set the font used for text output.

set_font sets the font for the output window. The font is used by the operators **write_string**, **read_string** etc. Text windows as well as windows for image display use fonts. A default font is assigned when a window is opened. This font can be changed with **set_font**. On UNIX systems all available fonts can be queried with **query_font**. The default font can be modified with **set_system** (**'default_font'**, Fontname). Fonts are not used for file operations.

The syntax for the specification of a font (in **Font**) differs for UNIX and Windows NT environments: In Windows NT a string with the following components is used:

-FontName-Height-Width-Italic-Underlined-Strikeout-[Bold-][CharSet-]

where “Italic”, “Underlined”, “Strikeout” and “Bold” can take the values 1 and 0 to activate or deactivate the corresponding feature. “CharSet” can be used to select the character set, if it differs from the default one. You can use the names of the defines or the integer value.

All parameters beside “FontName” und “Height” are optional. But at the beginn and end a Minus is suspected. To use the default setting, a * can be used instead. -Arial-10-*-1-*-1-

Please refer to the Windows NT documentation for a detailed discussion.

Attention
For different machines the available fonts may differ a lot. Therefore it is suggested to use wildcards, tables of fonts and/or the operator **query_font**.

Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window identifier.
▷ Font (input_control)	string $\leadsto$ string Name of new font.

Example

```
// UNIX
set_font(WindowHandle, '-*-courier-*-*-*-18-*-*-*-**')
// Windows NT
set_font(WindowHandle, '-Arial-18-*-*-*-')
write_string(WindowHandle, 'Text with Font size of 18 pixels')
new_line(WindowHandle)
set_font(WindowHandle, '-Arial-18-*-*-*-1-')
write_string(WindowHandle, 'Text with Font size of 18 pixels and bold')
new_line(WindowHandle)
```

Result

[set_font](#) returns 2 (H_MSG_TRUE) if the font name is correct. Otherwise an exception handling is raised.

Parallelization Information

[set_font](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[open_window](#), [open_textwindow](#)

Possible Successors

[query_font](#)

See also

[get_font](#), [query_font](#), [open_textwindow](#), [open_window](#)

Module

System

set_tposition (: : WindowHandle, Row, Column :)

Set the position of the text cursor.

[set_tposition](#) sets the position of the text cursor in the output window. The reference position is the upper left corner of an upper case character.

The position is measured in the coordinate system of the window and has to be valid for the window size and font. The position of the text cursor can be marked e.g. by an underscore. The next text output in this window starts at the cursor position. The left end of the baseline for writing the following text string (not considering descenders) is placed on this position.

The position is changed by the output or input of text ([write_string](#), [read_string](#)) or by an explicit change of position by ([set_tposition](#), [new_line](#)). In order to stop the display of the cursor, the operator [set_tshape](#) with the parameter `''invisible''` can be used.

Attention

If a string starting at the given position does not fit into the window an exception handling will be raised. This exception handling can be avoided by [set_check\('~text'\)](#).

Parameter

- ▷ **WindowHandle** (input_control) window $\rightsquigarrow$ *integer*
Window identifier.
- ▷ **Row** (input_control) point.y $\rightsquigarrow$ *integer*
Row index of text cursor position.
Default Value : 24
- ▷ **Column** (input_control) point.x $\rightsquigarrow$ *integer*
Column index of text cursor position.
Default Value : 12

Result

[set_tposition](#) returns 2 (H_MSG_TRUE) if the window is valid and the values of the parameters are valid. Otherwise an exception handling is raised.

set_tposition	Parallelization Information
<i>set_tposition</i> is <i>reentrant</i> , <i>local</i> , and processed <i>without parallelization</i> .	
open_window , open_textwindow	Possible Predecessors
set_tshape , write_string , read_string	Possible Successors
new_line	Alternatives
read_string , set_tshape , write_string	See also
System	Module

set_tshape (: : WindowHandle, TextCursor :)

Set the shape of the text cursor.

set_tshape sets the shape and the display mode of the text cursor of the output window.

A text cursor marks the current position for text output. It is different from the mouse cursor (although both will be called 'cursor', if the context makes misconceptions impossible). The available shapes for the text cursor can be queried with **query_tshape**.

	Parameter
▷ WindowHandle (input_control)	window $\leadsto$ integer Window identifier.
▷ TextCursor (input_control)	string $\leadsto$ string Name of cursor shape. Default Value : 'invisible'

	Result
get_tshape returns 2 (H.MSG_TRUE) if the window is valid and the given cursor shape is defined for this window. Otherwise an exception handling is raised.	

set_tshape	Parallelization Information
<i>set_tshape</i> is <i>reentrant</i> , <i>local</i> , and processed <i>without parallelization</i> .	
open_window , open_textwindow , query_tshape , get_tshape	Possible Predecessors
write_string , read_string	Possible Successors
get_tshape , query_tshape , write_string , read_string	See also
System	Module

write_string (: : WindowHandle, String :)

Print text in a window.

write_string prints **String** in the output window starting at the current cursor position. The output text has to fit within the right window boundary (the width of the string can be queried by **get_string_extents**).

The font currently assigned to the window will be used. The text cursor is positioned at the end of the text.

write_string can output all three types of data used in HALCON. The conversion to a string is guided by the following rules:

- strings are not converted.
- integer numbers are converted without any spaces before or after the number.
- floating numbers are printed (if possible) with a floating point and without an exponent.
- the resulting strings are concatenated without spaces.

For buffering of texts see [set_system](#) with the flag 'flush_graphic'.

Attention

If clipping at the window boundary is desired, exceptions can be switched off by [set_check\('~text'\)](#).

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window identifier.
- ▷ **String** (input_control) string(-array) $\leadsto$ string / integer / real
Tuple of output values (all types).
- Default Value** : 'hello'

Result

[write_string](#) returns 2 (H_MSG_TRUE) if the window is valid and the output text fits within the current line (see [set_check](#)). Otherwise an exception handling is raised.

Parallelization Information

[write_string](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[open_window](#), [open_textwindow](#), [set_font](#), [get_string_extents](#)

Alternatives

[fwrite_string](#)

See also

[set_tposition](#), [get_string_extents](#), [open_textwindow](#), [set_font](#), [set_system](#),
[set_check](#)

Module

System

6.8 Window

clear_rectangle (: : WindowHandle, Row1, Column1, Row2, Column2 :)

Delete a rectangle on the output window.

[clear_rectangle](#) deletes all entries in the rectangle which is defined through the upper left corner ([Row1](#),[Column1](#)) and the lower right corner ([Row2](#),[Column2](#)). Deletion signifies that the specified rectangle is set to the background color (see [open_window](#), [open_textwindow](#)).

If you want to delete more than one rectangle, you may pass several rectangles, i.e., the parameters [Row1](#), [Column1](#), [Row2](#) and [Column2](#) are tuple.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window identifier.
- ▷ **Row1** (input_control) rectangle.origin.y(-array) $\leadsto$ integer
Line index of upper left corner.
Default Value : 10
Typical range of values : $0 \leq \text{Row1} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Column1** (input_control) rectangle.origin.x(-array) $\leadsto$ integer
Column index of upper left corner.
Default Value : 10
Typical range of values : $0 \leq \text{Column1} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1

- ▷ **Row2** (input_control) rectangle.corner.y(-array) $\leadsto$ *integer*
 Row index of lower right corner.
Default Value : 118
Typical range of values : $0 \leq \text{Row2} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
Restriction : $\text{Row2} > \text{Row1}$
- ▷ **Column2** (input_control) rectangle.corner.x(-array) $\leadsto$ *integer*
 Column index of lower right corner.
Default Value : 118
Typical range of values : $0 \leq \text{Column2} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
Restriction : $\text{Column2} \geq \text{Column1}$

Example

```
/* "Interaktives" Loeschen eines Rechtecks im Ausgabefenster: */
draw_rectangle1(WindowHandle, L1, C1, L2, C2)
clear_rectangle(WindowHandle, L1, C1, L2, C2).
```

Result

If an output window exists and the specified parameters are correct `clear_rectangle` returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Parallelization Information

`clear_rectangle` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`open_window`, `set_draw`, `set_color`, `set_colored`, `set_line_width`, `set_rgb`, `set_hsi`,
`draw_rectangle1`

Alternatives

`clear_window`, `disp_rectangle1`

See also

`open_window`, `open_textwindow`

Module

System

clear_window (: : WindowHandle :)

Delete an output window.

`clear_window` deletes all entries in the output window. The window (background and edge) is reset to its original state. Parameters assigned to this window (e.g., with `set_color`, `set_paint`, etc.) remain unmodified.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ *integer*
 Window identifier.

Example

```
clear_window(WindowHandle).
```

Result

If the output window is valid `clear_window` returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Parallelization Information

`clear_window` is *reentrant*, *local*, and processed *without* parallelization.

	Possible Predecessors
open_window , open_textwindow	
	Alternatives
clear_rectangle , disp_rectangle1	
	See also
open_window , open_textwindow	
	Module
System	

close_window (: : WindowHandle :)

Close an output window.

[close_window](#) closes a window which have been opened by [open_window](#) or by [open_textwindow](#). Afterwards the output device or the window area, respectively, is ready to accept new calls of [open_window](#) or [open_textwindow](#).

	Parameter
▷ WindowHandle (input_control) window $\leadsto$ integer Window identifier.	
	Result
If the output window is valid close_window returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.	

	Parallelization Information
close_window is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.	
	Possible Predecessors
open_window , open_textwindow	
	See also
open_window , open_textwindow	
	Module
System	

copy_rectangle (: : WindowHandleSource, WindowHandleDestination, Row1, Column1, Row2, Column2, DestRow, DestColumn :)

Copy all pixels within rectangles between output windows.

[copy_rectangle](#) copies all pixels from the specified window with the logical window number [WindowHandleSource](#) in the specified window with the logical window number [WindowHandleDestination](#). It copies pixels which reside inside a rectangle which is specified by parameters [Row1](#), [Column1](#), [Row2](#) and [Column2](#). The target position is specified through the upper left corner of the rectangle ([DestRow](#), [DestColumn](#)).

If you want to move more than one rectangle, you may pass them at once (in form of the tuple mode).

You may use [copy_rectangle](#) to copy edited graphics from an "invisible" window in a visible window. Therefore a window with the option 'buffer' is opened. The graphics is then displayed in this window and is copied in a visible window afterwards. The advantage of this strategy is, that [copy_rectangle](#) is much more rapid than output procedures as, e.g., [disp_channel](#). This means a particular advantage while using demo programs. You could even realise short "clips": you have to create for every image of a sequence a window of a 'buffer' type and pass the data into it. Output is the image sequence whereat all buffers are copied one after another in a visible window.

	Attention
Both windows have to reside on the same computer.	

Parameter

- ▷ **WindowHandleSource** (input_control) window $\leadsto$ integer
Number of the source window.
- ▷ **WindowHandleDestination** (input_control) window $\leadsto$ integer
Number of the destination window.
- ▷ **Row1** (input_control) rectangle.origin.y(-array) $\leadsto$ integer
Row index of upper left corner in the source window.
Default Value : 0
Typical range of values : $0 \leq \text{Row1} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Column1** (input_control) rectangle.origin.x(-array) $\leadsto$ integer
Column index of upper left corner in the source window.
Default Value : 0
Typical range of values : $0 \leq \text{Column1} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Row2** (input_control) rectangle.corner.y(-array) $\leadsto$ integer
Row index of lower right corner in the source window.
Default Value : 128
Typical range of values : $0 \leq \text{Row2} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
Restriction : $\text{Row2} \geq \text{Row1}$
- ▷ **Column2** (input_control) rectangle.corner.x(-array) $\leadsto$ integer
Column index of lower right corner in the source window.
Default Value : 128
Typical range of values : $0 \leq \text{Column2} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
Restriction : $\text{Column2} \geq \text{Column1}$
- ▷ **DestRow** (input_control) point.y(-array) $\leadsto$ integer
Row index of upper left corner in the target window.
Default Value : 0
Typical range of values : $0 \leq \text{DestRow} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **DestColumn** (input_control) point.x(-array) $\leadsto$ integer
Column index of upper left corner in the target window.
Default Value : 0
Typical range of values : $0 \leq \text{DestColumn} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1

Example

```

read_image(Image, 'affe')
open_window(0, 0, -1, -1, 'root', 'buffer', '', WindowHandle)
disp_image(Image, WindowHandle)
open_window(0, 0, -1, -1, 'root', 'visible', '', WindowHandleDestination)
repeat()
get_mbutton(WindowHandleDestination, Row, Column, Button)
copy_rectangle(BufferID, WindowHandleDestination, 20, 90, 120, 390, Row, Column)
until(Button = 1)
close_window(WindowHandleDestination)
close_window(WindowHandle)
clear(Image).

```

<i>Result</i>
If the output window is valid and if the specified parameters are correct <code>close_window</code> returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.
<i>Parallelization Information</i>
<code>copy_rectangle</code> is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.
<i>Possible Predecessors</i>
<code>open_window</code> , <code>open_textwindow</code>
<i>Possible Successors</i>
<code>close_window</code>
<i>Alternatives</i>
<code>move_rectangle</code> , <code>slide_image</code>
<i>See also</i>
<code>open_window</code> , <code>open_textwindow</code>
<i>Module</i>
System

dump_window (: : WindowHandle, Device, FileName :)

Write the window content to a file.

`dump_window` writes the content of the window to a file. You may continue to process this file by convenient printers or other programs. The content of a display is prepared for each special device (*Device*), i.e., it is formatted in a manner, that you may print this file directly or it can be processed furthermore by a graphical program.

To transform gray values the current color table of the window is used, i.e., the values of `set_lut_style` remain unconsidered.

The gray values of the display have in general a poorer resolution than the original gray values of the output images. This results in a reduced resolution (default setting are, e.g., 200 gray levels with 8 bit planes) for the representation of gray values (see also `disp_channel`). You may request the amount of actual disposable gray levels by `get_system`. To modify them before opening the first window you may use `set_system`.

Possible values for *Device*

'postscript': PostScript - file. Filetermination: `<FileName>.ps`

'postscript',Width,Height: PostScript - file with specification of the output size. *Width* and *Height* refer to the size. In this case a tuple with three values as *Device* is passed. Filetermination: `<FileName>.ps`

'tiff': TIFF - file, 1 byte per pixel incl. current color table or 3 bytes per sample (dependent on VGA card), uncompressed. Filetermination: `<FileName>.tiff`

'bmp': Windows-BMP format, RGB image, 3 bytes per pixel. The color resolution depends on the VGA card. File extension: `'.bmp'`

'jpeg': JPEG format, with loss of information; together with the format string the quality value determining the compression rate can be provided: e.g., `'jpeg 30'`.

'png': PNG format (lossless compression); together with the format string, a compression level between 0 and 9 can be specified, where 0 corresponds to no compression and 9 to the best possible compression. Alternatively, the compression can be selected with the following strings: `'best'`, `'fastest'`, and `'none'`. Hence, examples for correct parameters are `'png'`, `'png 7'`, and `'png none'`. Images of type `byte` and `uint2` can be stored in PNG files. If an image with a reduced domain is written, the region is stored as the alpha channel, where the points within the domain are stored as the maximum gray value of the image type and the points outside the domain are stored as the gray value 0. If an image with a full domain is written, no alpha channel is stored.

<i>Attention</i>

Under X Windows, the HALCON window must be completely visible on the root window, because otherwise the contents of the window cannot be read due to limitations in X Windows. If larger graphical displays are to be written to a file, the window type `'pixmap'` can be used.

<i>Parameter</i>	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window identifier.
▷ Device (input_control)	string(-array) $\leadsto$ string / integer Name of the target device or of the graphic format. Default Value : 'postscript' List of values : Device $\in$ { 'postscript', 'tiff', 'bmp', 'jpeg', 'png', 'jpeg 100', 'jpeg 80', 'jpeg 60', 'jpeg 40', 'jpeg 20', 'png best', 'png fastest', 'png none' }
▷ FileName (input_control)	filename $\leadsto$ string File name (without extension). Default Value : 'halcon_dump'

Example

```
/* PostScript - Dump von Bild und eingeblendeten Regionen: */
disp_image(Image,WindowHandle)
set_colored(WindowHandle,12)
disp_region(Regions,WindowHandle)
dump_window(WindowHandle,'postscript','/tmp/halcon_dump')
system_call('lp -d ps /tmp/halcon_dump.ps').
```

Result

If the appropriate window is valid and the specified parameters are correct **dump_window** returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Parallelization Information

dump_window is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

open_window, **set_draw**, **set_color**, **set_colored**, **set_line_width**, **open_textwindow**, **disp_region**

Possible Successors

system_call

See also

open_window, **open_textwindow**, **set_system**

Module

System

get_os_window_handle (: : WindowHandle : OSWindowHandle,
OSDisplayHandle)

Get the operating system window handle.

get_os_window_handle returns the operating system window handle of the HALCON window **WindowHandle** in **OSWindowHandle**. Under UNIX, additionally the operating system display handle is returned in **OSDisplayHandle**. The operating system window handle can be used to access the window using functions from the operating system, e.g., to draw in a user-defined manner into the window. Under Windows, **OSWindowHandle** can be cast to a variable of type HWND. Under UNIX systems, **OSWindowHandle** can be cast into a variable of type Window, while **OSDisplayHandle** can be cast into a variable of type Display.

<i>Parameter</i>	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window identifier.
▷ OSWindowHandle (output_control)	integer $\leadsto$ integer Operating system window handle.
▷ OSDisplayHandle (output_control)	integer $\leadsto$ integer Operating system display handle (under UNIX only).

Example

```

/* Draw a line into a HALCON window under UNIX using X11 calls. */
#include "HalconC.h"
#include <X11/X.h>
#include <X11/Xlib.h>

int main(int argc, char **argv)
{
    long hwin, win, disp;
    Display *display;
    Window window;
    GC gc;
    XGCValues values;
    static char dashes[] = { 20, 20 };

    open_window(0, 0, 500, 500, 0, "visible", "", &hwin);
    get_os_window_handle(hwin, &win, &disp);
    display = (Display *)disp;
    window = (Window)win;
    gc = XCreateGC(display, window, 0, &values);
    XSetFunction(display, gc, GXset);
    XSetLineAttributes(display, gc, 10, LineOnOffDash, CapRound, JoinRound);
    XSetDashes(display, gc, 0, dashes, 2);
    XSetForeground(display, gc, WhitePixel(display, DefaultScreen(display)));
    XSetBackground(display, gc, BlackPixel(display, DefaultScreen(display)));
    XDrawLine(display, win, gc, 20, 20, 480, 480);
    XFlush(display);
    XFreeGC(display, gc);
    wait_seconds(5);
    return 0;
}

/* Draw a line into a HALCON window under Windows using GDI calls. */
#include "HalconC.h"
#include "windows.h"

int main(int argc, char **argv)
{
    long hwin, win, disp;
    HDC hdc;
    HPEN hpen;
    HPEN *hpen_old;
    LOGBRUSH logbrush;
    POINT point;
    static DWORD dashes[] = { 20, 20 };

    open_window(0, 0, 500, 500, 0, "visible", "", &hwin);
    get_os_window_handle(hwin, &win, &disp);
    logbrush.lbColor = RGB(255,255,255);
    logbrush.lbStyle = BS_SOLID;
    hpen = ExtCreatePen(PS_USERSTYLE|PS_GEOMETRIC, 10, &logbrush, 2, dashes);
    hdc = GetDC((HWND)win);
    hpen_old = (HPEN *)SelectObject(hdc, hpen);
    MoveToEx(hdc, 20, 20, &point);
    LineTo(hdc, 480, 480);
    DeleteObject(SelectObject(hdc, hpen_old));
    ReleaseDC((HWND)win, hdc);
}

```

```

    wait_seconds(5);
    return 0;
}

```

Result

If the window is valid `get_os_window_handle` returns 2 (H_MSG_TRUE). Otherwise, an exception handling is raised.

Parallelization Information

`get_os_window_handle` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`open_window`, `open_textwindow`

Module

System

get_window_attr (: : AttributeName : AttributeValue)

Get window characteristics.

The operator `get_window_attr` can be used to read characteristics of graphics windows that were set using `set_window_attr`. The following parameters of a window may be queried:

'border_width' Width of the window border in pixels.

'border_color' Color of the window border.

'background_color' Background color of the window.

'window_title' Name of the window in the titlebar.

Parameter

- ▷ **AttributeName** (input_control) string $\leadsto$ *string*
Name of the attribute that should be returned.
List of values : AttributeName $\in$ { 'border_width', 'border_color', 'background_color', 'window_title' }
- ▷ **AttributeValue** (output_control) string $\leadsto$ *string* / integer
Attribute value.

Result

If the parameters are correct `get_window_attr` returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Parallelization Information

`get_window_attr` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`open_window`, `set_draw`, `set_color`, `set_colored`, `set_line_width`, `open_textwindow`

See also

`open_window`, `set_window_attr`

Module

System

get_window_extents (: : WindowHandle : Row, Column, Width, Height)

Information about a window's size and position.

`get_window_extents` returns the position of the upper left corner, as well as width and height of the output window.

Attention

Size and position of a window may be modified by the window manager, without explicit instruction in the program. Therefore the values which are returned by `get_window_extents` may change cause of side effects.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window identifier.
- ▷ **Row** (output_control) rectangle.origin.y $\leadsto$ integer
Row index of upper left corner of the window.
- ▷ **Column** (output_control) rectangle.origin.x $\leadsto$ integer
Column index of upper left corner of the window.
- ▷ **Width** (output_control) rectangle.extent.x $\leadsto$ integer
Window width.
- ▷ **Height** (output_control) rectangle.extent.y $\leadsto$ integer
Window height.

Example

```

open_window(100,100,200,200,'root','visible','',WindowHandle)
fwrite_string('Move the window with the mouse!')
fnew_line()
repeat()
get_mbutton(WindowHandle,_,_,Button)
get_window_extents(WindowHandle,Row,Column,Width,Height)
fwrite(['('Row,',',',Column,')'])
fnew_line()
until(Button = 4).

```

Result

If the window is valid `get_window_extents` returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Parallelization Information

`get_window_extents` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`open_window`, `set_draw`, `set_color`, `set_colored`, `set_line_width`, `open_textwindow`

See also

`set_window_extents`, `open_window`, `open_textwindow`

Module

System

get_window_pointer3 (: : WindowHandle : ImageRed, ImageGreen, ImageBlue, Width, Height)

Access to a window's pixel data.

`get_window_pointer3` enables (in some window systems) the direct access to the bitmap. Result values are the three pointers on the color extracts of a 24-bit window (`ImageRed`, `ImageGreen`, `ImageBlue`), as well as the window size (`Width`, `Height`). In the language C the type of the image points is unsigned char.

Attention

`get_window_pointer3` is usable only for window type 'pixmap'.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window identifier.
- ▷ **ImageRed** (output_control) integer $\leadsto$ integer
Pointer on red channel of pixel data.
- ▷ **ImageGreen** (output_control) integer $\leadsto$ integer
Pointer on green channel of pixel data.
- ▷ **ImageBlue** (output_control) integer $\leadsto$ integer
Pointer on blue channel of pixel data.

- ▷ **Width** (output_control) extent.x $\leadsto$ *integer*
Length of an image line.
- ▷ **Height** (output_control) extent.y $\leadsto$ *integer*
Number of image lines.

Result

If a window of type 'pixmap' exists and it is valid `get_window_pointer3` returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Parallelization Information

`get_window_pointer3` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`open_window`, `open_textwindow`

See also

`open_window`, `set_window_type`

Module

System

get_window_type (: : WindowHandle : WindowType)

Get the window type.

`get_window_type` determines the type or the graphical software, respectively, of the output device for the window. You may query the available types of output devices with procedure `query_window_type`. A reasonable use for `get_window_type` might be in the field of the development of machine independent software. Possible values are:

'X-Window' X-Window Version 11.

'pixmap' Windows are not shown, but managed in memory. By this means HALCON programs can be ported on computers without a graphical display.

'PostScript' Objects are output to a PostScript File.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ *integer*
Window identifier.
- ▷ **WindowType** (output_control) string $\leadsto$ *string*
Window type

Example

```
open_window(100,100,200,200,'root','visible','','WindowHandle)
get_window_type(WindowHandle,WindowType)
fwrite_string(['Window type: ',WindowType])
fnew_line().
```

Result

If the window is valid `get_window_type` returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Parallelization Information

`get_window_type` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`open_window`, `open_textwindow`

See also

`query_window_type`, `set_window_type`, `get_window_pointer3`, `open_window`, `open_textwindow`

Module

System

move_rectangle (: : WindowHandle, Row1, Column1, Row2, Column2, DestRow, DestColumn :)

Copy inside an output window.

move_rectangle copies all entries in the rectangle (**Row1**,**Column1**), (**Row2**,**Column2**) of the output window to a new position inside the same window. This position is determined by the upper left corner (**DestRow**, **DestColumn**). Regions of the window, which are "uncovered" through moving the rectangle, are set to the color of the background.

If you want to move several rectangles at once, you may pass parameters in form of tuples.

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ *integer*
Window identifier.
- ▷ **Row1** (input_control) rectangle.origin.y(-array) $\leadsto$ *integer*
Row index of upper left corner of the source rectangle.
Default Value : 0
Typical range of values : $0 \leq \text{Row1} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Column1** (input_control) rectangle.origin.x(-array) $\leadsto$ *integer*
Column index of upper left corner of the source rectangle.
Default Value : 0
Typical range of values : $0 \leq \text{Column1} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Row2** (input_control) rectangle.corner.y(-array) $\leadsto$ *integer*
Row index of lower right corner of the source rectangle.
Default Value : 64
Typical range of values : $0 \leq \text{Row2} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Column2** (input_control) rectangle.corner.x(-array) $\leadsto$ *integer*
Column index of lower right corner of the source rectangle.
Default Value : 64
Typical range of values : $0 \leq \text{Column2} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **DestRow** (input_control) point.y(-array) $\leadsto$ *integer*
Row index of upper left corner of the target position.
Default Value : 64
Typical range of values : $0 \leq \text{DestRow} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **DestColumn** (input_control) point.x(-array) $\leadsto$ *integer*
Column index of upper left corner of the target position.
Default Value : 64
Typical range of values : $0 \leq \text{DestColumn} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1

Example

```
/* "Interaktives" Kopieren eines Rechtecks im Ausgabefenster: */
draw_rectangle1(WindowHandle,L1,C1,L2,C2)
get_mbutton(WindowHandle, LN, CN, Button)
move_rectangle(WindowHandle, L1, C1, L2, C2, LN, CN) .
```

Result

If the window is valid and the specified parameters are correct **move_rectangle** returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Parallelization Information

[move_rectangle](#) is *reentrant*, *local*, and processed without parallelization.

Possible Predecessors

[open_window](#), [open_textwindow](#)

Alternatives

[copy_rectangle](#)

See also

[open_window](#), [open_textwindow](#)

Module

System

new_extern_window (: : WINHWnd, WINHDC, Row, Column, Width, Height : WindowHandle)

Create a virtual graphics window under Windows NT.

[new_extern_window](#) opens a new virtual window. Virtual means that a new window will not be created, but the window whose Windows NT handle is given in the parameter [WINHWnd](#) is used to perform output of gray value data, regions, graphics as well as to perform textual output. Visualization parameters for the output of data can be done either using HALCON commands or by the appropriate Windows NT commands.

Example: setting of the drawing color:

HALCON:

```
set\_color(WindowHandle, "green");
disp\_region(WindowHandle, region);
```

Windows NT:

```
HPEN* penold;
HPEN penGreen = CreatePen(PS\_SOLID, 1, RGB(0, 255, 0));
pen = (HPEN*)SelectObject(WINHDC, penGreen);
disp\_region(WindowHandle, region);
```

Interactive operators, for example [draw_region](#), [draw_circle](#) or [get_mbutton](#) cannot be used in this window. The following operators can be used:

- Output of gray values: [set_paint](#), [set_comprise](#), ([set_lut](#) and [set_lut_style](#) after output)
- Regions: [set_color](#), [set_rgb](#), [set_hsi](#), [set_gray](#), [set_pixel](#), [set_shape](#), [set_line_width](#), [set_insert](#), [set_line_style](#), [set_draw](#)
- Image part: [set_part](#)
- Text: [set_font](#)

You may query current set values by calling procedures like [get_shape](#). As some parameters are specified through the hardware (Resolution/Colors), you may query current available resources by calling operators like [query_color](#).

The parameter [WINHWnd](#) is used to pass the window handle of the Windows NT window, in which output should be done. The parameter [WINHDC](#) is used to pass the device context of the window [WINHWnd](#). This device context is used in the output routines of HALCON.

The origin of the coordinate system of the window resides in the upper left corner (coordinates: (0,0)). The row index grows downward (maximum: [Height](#)-1), the column index grows to the right (maximal: [Width](#)-1).

You may use the value -1 for parameters [Width](#) and [Height](#). This means, that the corresponding value is chosen automatically. In particular, this is important if the aspect ratio of the pixels is not 1.0 (see [set_system](#)). If one of the two parameters is set to -1, it will be chosen through the size which results out of the aspect ratio of the pixels. If both parameters are set to -1, they will be set to the current image format.

The position and size of a window may change during runtime of a program. This may be achieved by calling `set_window_extents`, but also through external influences (window manager). For the latter case the procedure `set_window_extents` is provided.

Opening a window causes the assignment of a default font. It is used in connection with procedures like `write_string` and you may change it by performing `set_font` after calling `open_window`. On the other hand, you have the possibility to specify a default font by calling `set_system (::'default_font', <Fontname>:)` before opening a window (and all following windows; see also `query_font`).

You may set the color of graphics and font, which is used for output procedures like `disp_region` or `disp_circle`, by calling `set_rgb`, `set_hsi`, `set_gray` or `set_pixel`. Calling `set_insert` specifies how graphics is combined with the content of the image repeat memory. Thereto you may achieve by calling, e.g., `set_insert (::'not')` to eliminate the font after writing text twice at the same position.

The content of the window is not saved, if other windows overlap the window. This must be done in the program code that handles the Windows NT window in the calling program.

For graphical output (`disp_image`, `disp_region`, etc.) you may adjust the window by calling procedure `set_part` in order to represent a logical clipping of the image format. In particular this implies that only this part (appropriately scaled) of images and regions is displayed. Before you close your window, you have to close the HALCON-window.

Steps to use `new_extern_window`:

Creation: • Create a Windows-window.

- Call `new_extern_window` with the `WINHWnd` of the above created window.

Use: • Before drawing in the window you have to call the method `set_window_dc`. This ensures that the halcon drawing routines use the right DC. After drawing call again `set_window_dc`, but this time with the address of a long set to zero, this ensures that HALCON can delete the created graphic objects.

Destroy: • Call `close_window`.

Attention

Note that parameters as `Row`, `Column`, `Width` and `Height` are constrained through the output device, i.e., the size of the Windows NT desktop.

Parameter

- ▷ **WINHWnd** (input_control) integer $\rightsquigarrow$ integer
Windows windowhandle of a previously created window.
Restriction : `WINHWnd` $\neq$ 0
- ▷ **WINHDC** (input_control) integer $\rightsquigarrow$ integer
Device context of `WINHWnd`.
Restriction : `WINHDC` $\neq$ 0
- ▷ **Row** (input_control) rectangle.origin.y $\rightsquigarrow$ integer
Row coordinate of upper left corner.
Default Value : 0
Restriction : `Row` $\geq$ 0
- ▷ **Column** (input_control) rectangle.origin.x $\rightsquigarrow$ integer
Column coordinate of upper left corner.
Default Value : 0
Restriction : `Column` $\geq$ 0
- ▷ **Width** (input_control) rectangle.extent.x $\rightsquigarrow$ integer
Width of the window.
Default Value : 512
Restriction : (`Width` $>$ 0) $\vee$ (`Width` = -1)
- ▷ **Height** (input_control) rectangle.extent.y $\rightsquigarrow$ integer
Height of the window.
Default Value : 512
Restriction : (`Height` $>$ 0) $\vee$ (`Height` = -1)
- ▷ **WindowHandle** (output_control) window $\rightsquigarrow$ integer
Window identifier.

Example

```

HTuple m_tHalconWindow ;
Hobject m_objImage ;

WM_CREATE:
/* here you should create your extern halcon window*/
HTuple tWnd, tDC ;
::set_check("~father") ;
tWnd = (INT)((INT*)&m_hWnd) ;
tDC = (INT)(INT*)GetWindowDC() ;
::new_extern_window(tWnd, tDC, 0, 0, sizeTotal.cx, sizeTotal.cy, &m_tHalconWindow) ;
::set_check("father") ;

WM_PAINT:
/* here you can draw halcon objects */
long l = 0 ;
if (m_thWindow != -1) {
/* don't forget to set the dc !! */
HTuple tDC((INT)(INT*)&pDC->m_hDC) ;
HTuple tDCNull((INT)(INT*)&l) ;
::set_window_dc(m_tHalconWindow, tDC) ;
::disp_obj(pDoc->m_objImage, m_tHalconWindow) ;
/* release the graphic objects */
::set_window_dc(m_tHalconWindow, tDCNull) ;
}

WM_CLOSE:
/* close the halcon window */
if (m_tHalconWindow != -1) {
::close_window(m_tHalconWindow) ;
}

```

Result

If the values of the specified parameters are correct [new_extern_window](#) returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Parallelization Information

[new_extern_window](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[reset_obj_db](#)

Possible Successors

[set_color](#), [query_window_type](#), [get_window_type](#), [set_window_type](#), [get_mposition](#), [set_tposition](#), [set_tshape](#), [set_window_extents](#), [get_window_extents](#), [query_color](#), [set_check](#), [set_system](#)

Alternatives

[open_window](#), [open_textwindow](#)

See also

[open_window](#), [disp_region](#), [disp_image](#), [disp_color](#), [set_lut](#), [query_color](#), [set_color](#), [set_rgb](#), [set_hsi](#), [set_pixel](#), [set_gray](#), [set_part](#), [set_part_style](#), [query_window_type](#), [get_window_type](#), [set_window_type](#), [get_mposition](#), [set_tposition](#), [set_window_extents](#), [get_window_extents](#), [set_window_attr](#), [set_check](#), [set_system](#)

Module

System

```
open_textwindow ( : : Row, Column, Width, Height, BorderWidth,
  BorderColor, BackgroundColor, FatherWindow, Mode,
  Machine : WindowHandle )
```

Open a textual window.

`open_textwindow` opens a new textual window, which can be used to perform textual input and output, as well as to perform output of images. All output (`write_string`, `read_string`, `disp_region`, etc.) is redirected to this window, if the same logical window number `WindowHandle` is used.

Besides the mouse cursor textual windows possess also a textual cursor which indicates the current writing position (more exactly: the lower left corner of the output string without consideration of descenders). Its position is indicated through an underscore or another type (the indication of this position may also be disabled (= default setting); cf. `set_tshape`). You may set or query the position by calling the procedures `set_tposition` or `get_tposition`.

After you opened a textual window the position of the cursor is set to (H,0). Whereby H signifies the height of the default font less the descenders. But the cursor is not shown. Hence the output starts for writing in the upper left corner of the window.

You may query the colors of the background and the image edges by calling `query_color`. In the same way you may use `query_color` in a window of type 'invisible'. During output (`write_string`) you may set the clipping of text out of the window edges by calling `set_check(::'~text')`. This disables the creation of error messages, if text passes over the edge of the window.

The origin of the coordinate system of the window resides in the upper left corner (coordinates: (0,0)). The row index grows downward (maximal: `Height-1`), the column index grows to the right (maximal: `Width-1`).

The parameter `Machine` indicates the name of the computer, which has to open the window. In case of a X-window, TCP-IP only sets the name, DEC-Net sets in addition a colon behind the name. The "server" or the "screen", respectively, are not specified. If the empty string is passed the environment variable `DISPLAY` is used. It indicates the target computer. At this the name is indicated in common syntax

```
<Host>:0.0
```

.

Position and size of a window may change during runtime of a program. This may be achieved by calling `set_window_extents`, but also through external interferences (window manager). In the latter case the procedure `set_window_extents` is provided.

Opening a window causes the assignment of a called default font. It is used in connection with procedures like `write_string` and you may overwrite it by performing `set_font` after calling `open_textwindow`. On the other hand you have the possibility to specify a default font by calling `set_system (::'default_font', <Fontname>:)` before opening a window (and all following windows; see also `query_font`).

You may set the color of the font (`write_string`, `read_string`) by calling `set_color`, `set_rgb`, `set_hsi`, `set_gray` or `set_pixel`. Calling `set_insert` specifies how the text or the graphics, respectively, is combined with the content of the image repeat memory. So you may achieve by calling, e.g., `set_insert (::'not')` to eliminate the font after writing text twice at the same position.

Normally every output (e.g., `write_string`, `disp_region`, `disp_circle`, etc.) in a window is terminated by a "flush". This causes the data to be fully visible on the display after termination of the output procedure. But this is not necessary in all cases, in particular if there are permanently output tasks or there is a mouse procedure active. Therefore it is more favorable (i.e., more rapid) to store the data until sufficient data is available. You may stop this behavior by calling `set_system (::'flush_graphic', 'false':)`.

The content of windows is saved (in case it is supported by special driver software); i.e., it is preserved, also if the window is hidden by other windows. But this is not necessary in all cases: If you use a textual window, e.g., as a parent window for other windows, you may suppress the security mechanism for it and save the necessary memory at the same moment. You achieve this before opening the window by calling `set_system (::'backing_store', 'false':)`.

Difference: graphical window - textual window

- In contrast to graphical windows (`open_window`) you may specify more parameters (color, edge) for a textual window while opening it.

- You may use textual windows only for input of user data (`read_string`).
- Using textual windows, the output of images, regions and graphics is "clipped" at the edges. Whereas during the use of graphical windows the edges are "zoomed".
- The coordinate system (e.g., with `get_mbutton` or `get_mposition`) consists of display coordinates independently of image size. The maximum coordinates are equal to the size of the window minus 1. In contrast to this, graphical windows (`open_window`) use always a coordinate system, which corresponds to the image format.

The parameter **Mode** specifies the mode of the window. It can have following values:

- 'visible'**: Normal mode for textual windows: The window is created according to the parameters and all inputs and outputs are possible.
- 'invisible'**: Invisible windows are not displayed in the display. Parameters like **Row**, **Column**, **BorderWidth**, **BorderColor**, **BackgroundColor** and **FatherWindow** do not have any meaning. Output to these windows has no effect. Input (`read_string`, mouse, etc.) is not possible. You may use these windows to query representation parameter for an output device without opening a (visible) window. General queries are, e.g., `query_color` and `get_string_extents`.
- 'transparent'**: These windows are transparent: the window itself is not visible (edge and background), but all the other operations are possible and all output is displayed. Parameters like **BorderColor** and **BackgroundColor** do not have any meaning. A common use for this mode is the creation of mouse sensitive regions.
- 'buffer'**: These are also not visible windows. The output of images, regions and graphics is not visible on the display, but is stored in memory. Parameters like **Row**, **Column**, **BorderWidth**, **BorderColor**, **BackgroundColor** and **FatherWindow** do not have any meaning. You may use buffer windows, if you prepare output (in the background) and copy it finally with `copy_rectangle` in a visible window. Another usage might be the rapid processing of image regions during interactive manipulations. Textual input and mouse interaction are not possible in this mode.

Attention

You have to keep in mind that parameters like **Row**, **Column**, **Width** and **Height** are restricted by the output device. Is a father window (`FatherWindow` $\neq$ 'root') specified, then the coordinates are relative to this window.

Parameter

- ▷ **Row** (input_control) `rectangle.origin.y` $\leadsto$ *integer*
 Row index of upper left corner.
Default Value : 0
Typical range of values : $0 \leq \text{Row} (\text{lin})$
Minimum Increment : 1
Recommended Increment : 1
Restriction : $\text{Row} \geq 0$
- ▷ **Column** (input_control) `rectangle.origin.x` $\leadsto$ *integer*
 Column index of upper left corner.
Default Value : 0
Typical range of values : $0 \leq \text{Column} (\text{lin})$
Minimum Increment : 1
Recommended Increment : 1
Restriction : $\text{Column} \geq 0$
- ▷ **Width** (input_control) `rectangle.extent.x` $\leadsto$ *integer*
 Window's width.
Default Value : 256
Typical range of values : $0 \leq \text{Width} (\text{lin})$
Minimum Increment : 1
Recommended Increment : 1
Restriction : $\text{Width} > 0$

- ▷ **Height** (input_control) rectangle.extent.y $\leadsto$ *integer*
Window's height.
Default Value : 256
Typical range of values : $0 \leq \text{Height (lin)}$
Minimum Increment : 1
Recommended Increment : 1
Restriction : Height > 0
- ▷ **BorderWidth** (input_control) integer $\leadsto$ *integer*
Window border's width.
Default Value : 2
Typical range of values : $0 \leq \text{BorderWidth (lin)}$
Minimum Increment : 1
Recommended Increment : 1
Restriction : BorderWidth $\geq$ 0
- ▷ **BorderColor** (input_control) string $\leadsto$ *string*
Window border's color.
Default Value : 'white'
- ▷ **BackgroundColor** (input_control) string $\leadsto$ *string*
Background color.
Default Value : 'black'
- ▷ **FatherWindow** (input_control) integer $\leadsto$ *integer / string*
Logical number of the father window. For the display as father you may specify 'root' or 0.
Default Value : 0
Restriction : FatherWindow $\geq$ 0
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Window mode.
Default Value : 'visible'
List of values : Mode $\in$ {'visible', 'invisible', 'transparent', 'buffer'}
- ▷ **Machine** (input_control) string $\leadsto$ *string*
Computer name, where the window has to be opened or empty string.
Default Value : ''
- ▷ **WindowHandle** (output_control) window $\leadsto$ *integer*
Window identifier.

Example

```

open_textwindow(0,0,900,600,1,'black','slate blue','root','visible',
                'WindowHandle)
open_textwindow(10,10,300,580,3,'red','blue',Father,'visible',
                'WindowHandle)
open_window(10,320,570,580,Father,'visible','WindowHandle)
set_color(WindowHandle,'red')
read_image(Image,'affe')
disp_image(Image,WindowHandle)
repeat()
get_mposition(WindowHandle,Row,Column,Button)
get_grayval(Image,Row,Column,1,Gray)
write_string(WindowHandle,[' Position (',Row,',',',',Column,',') '])
write_string(WindowHandle,['Gray value (',Gray,',') '])
new_line(WindowHandle)
until(Button = 4)
close_window(WindowHandle)
clear_obj(Image).

```

Result

If the values of the specified parameters are correct `open_textwindow` returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Parallelization Information

`open_textwindow` is *reentrant*, *local*, and processed *without* parallelization.

	Possible Predecessors	
reset_obj_db		
	Possible Successors	
set_color , query_window_type , get_window_type , set_window_type , get_mposition , set_tposition , set_tshape , set_window_extents , get_window_extents , query_color , set_check , set_system		
	Alternatives	
open_window		
	See also	
write_string , read_string , newline , get_string_extents , get_tposition , set_color , query_window_type , get_window_type , set_window_type , get_mposition , set_tposition , set_tshape , set_window_extents , get_window_extents , query_color , set_check , set_system		
	Module	
System		

open_window (: : Row, Column, Width, Height, FatherWindow, Mode, Machine : WindowHandle)

Open a graphics window.

[open_window](#) opens a new window, which can be used to perform output of gray value data, regions, graphics as well as to perform textual output. All output ([disp_region](#), [disp_image](#), etc.) is redirected to this window, if the same logical window number [WindowHandle](#) is used.

The background of the created window is set to black in advance and it has a white border, which is 2 pixels wide (see also [set_window_attr](#)(::'border_width', <Breite>:)).

Certain parameters used for the editing of output data are assigned to a window. These parameters are considered during the output itself (e.g., with [disp_image](#) or [disp_region](#)). They are not specified by an output procedure, but by "configuration procedures". If you want to set, e.g., the color red for the output of regions, you have to call [set_color](#)(::WindowHandle, 'red':) before calling [disp_region](#). These parameters are always set for the window with the logical window number [WindowHandle](#) and remain assigned to a window as long as they will be overwritten. You may use the following configuration procedures:

- Output of gray values: [set_paint](#), [set_comprise](#), ([set_lut](#) and [set_lut_style](#) after output)
- Regions: [set_color](#), [set_rgb](#), [set_hsi](#), [set_gray](#), [set_pixel](#), [set_shape](#), [set_line_width](#), [set_insert](#), [set_line_style](#), [set_draw](#)
- Image clipping: [set_part](#)
- Text: [set_font](#)

You may query current set values by calling procedures like [get_shape](#). As some parameters are specified through the hardware (Resolution/Colors), you may query current available resources by calling [query_color](#).

The origin of the coordinate system of the window resides in the upper left corner (coordinates: (0,0)). The row index grows downward (maximal: [Height](#)-1), the column index grows to the right (maximal: [Width](#)-1). You have to keep in mind, that the range of the coordinate system is independent of the window size. It is specified only through the image format (see [reset_obj_db](#)).

The parameter [Machine](#) indicates the name of the computer, which has to open the window. In case of a X-window, TCP-IP only sets the name, DEC-Net sets in addition a colon behind the name. The "server" resp. the "screen" are not specified. If the empty string is passed the environment variable DISPLAY is used. It indicates the target computer. At this the name is indicated in common syntax

```
<Host>:0.0
```

.

You may use the value "-1" for parameters [Width](#) and [Height](#). This means, that the according value has to be specified automatically. In particular this is of importance, if the proportion of pixels is not 1.0 (see [set_system](#)):

Is one of the two parameters set to `"-1"`, it will be specified through the size which results out of the proportion of pixels. Are both parameters set to `"-1"`, they will be set to the maximum image format, which is currently used (further information about the currently used maximum image format can be found in the description of `get_system` using `"width"` or `"height"`).

Position and size of a window may change during runtime of a program. This may be achieved by calling `set_window_extents`, but also through external interferences (window manager). In the latter case the procedure `set_window_extents` is provided.

Opening a window causes the assignment of a called default font. It is used in connection with procedures like `write_string` and you may overwrite it by performing `set_font` after calling `open_window`. On the other hand you have the possibility to specify a default font by calling `set_system (:: 'default_font', <Fontname>:)` before opening a window (and all following windows; see also `query_font`).

You may set the color of graphics and font, which is used for output procedures like `disp_region` or `disp_circle`, by calling `set_rgb`, `set_hsi`, `set_gray` or `set_pixel`. Calling `set_insert` specifies how graphics is combined with the content of the image repeat memory. Thereto you may achieve by calling, e.g., `set_insert (:: 'not':)` to eliminate the font after writing text twice at the same position.

Normally every output (e.g., `disp_image`, `disp_region`, `disp_circle`, etc.) in a window is terminated by a called `"flush"`. This causes the data to be fully visible on the display after termination of the output procedure. But this is not necessary in all cases, in particular if there are permanently output tasks or if there is a mouse procedure active. Therefore it is more favorable (i.e., more rapid) to store the data until sufficient data is available. You may stop this behavior by calling `set_system (:: 'flush_graphic', 'false':)`.

The content of windows is saved (in case it is supported by special driver software); i.e., it is preserved, also if the window is hidden by other windows. But this is not necessary in all cases: If the content of a window is built up permanently new (`copy_rectangle`), you may suppress the security mechanism for that and hence you can save the necessary memory. This is done by calling `set_system (:: 'backing_store', 'false':)` before opening a window. In doing so you save not only memory but also time to compute. This is significant for the output of video clips (see `copy_rectangle`).

For graphical output (`disp_image`, `disp_region`, etc.) you may adjust the window by calling procedure `set_part` in order to represent a logical clipping of the image format. In particular this implicates that you obtain this clipping (with appropriate enlargement) of images and regions only.

Difference: graphical window - textual window

- Using graphical windows the layout is not as variable as concerned to textual windows.
- You may use textual windows for the input of user data only (`read_string`).
- During the output of images, regions and graphics a `"zooming"` is performed using graphical windows: Independent on size and side ratio of the window images are transformed in that way, that they are displayed in the window by filling it completely. On the opposite side using textual windows the output does not care about the size of the window (only if clipping is necessary).
- Using graphical windows the coordinate system of the window corresponds to the coordinate system of the image format. Using textual windows, its coordinate system is always equal to the display coordinates independent on image size.

The parameter `Mode` determines the mode of the window. It may have following values:

'visible': Normal mode for graphical windows: The window is created according to the parameters and all input and output are possible.

'invisible': Invisible windows are not displayed in the display. Parameters like `Row`, `Column` and `FatherWindow` do not have any meaning. Output to these windows has no effect. Input (`read_string`, mouse, etc.) is not possible. You may use these windows to query representation parameter for an output device without opening a (visible) window. Common queries are, e.g., `query_color` and `get_string_extents`.

'transparent': These windows are transparent: the window itself is not visible (edge and background), but all the other operations are possible and all output is displayed. A common use for this mode is the creation of mouse sensitive regions.

'buffer': These are also not visible windows. The output of images, regions and graphics is not visible on the display, but is stored in memory. Parameters like **Row**, **Column** and **FatherWindow** do not have any meaning. You may use buffer windows, if you prepare output (in the background) and copy it finally with **copy_rectangle** in a visible window. Another usage might be the rapid processing of image regions during interactive manipulations. Textual input and mouse interaction are not possible in this mode.

Attention

You may keep in mind that parameters as **Row**, **Column**, **Width** and **Height** are constrained by the output device. If you specify a father window (**FatherWindow** <> 'root') the coordinates are relative to this window.

Parameter

- ▷ **Row** (input_control) rectangle.origin.y $\leadsto$ *integer*
Row index of upper left corner.
Default Value : 0
Typical range of values : $0 \leq \text{Row} \text{ (lin)}$
Minimum Increment : 1
Recommended Increment : 1
Restriction : $\text{Row} \geq 0$
- ▷ **Column** (input_control) rectangle.origin.x $\leadsto$ *integer*
Column index of upper left corner.
Default Value : 0
Typical range of values : $0 \leq \text{Column} \text{ (lin)}$
Minimum Increment : 1
Recommended Increment : 1
Restriction : $\text{Column} \geq 0$
- ▷ **Width** (input_control) rectangle.extent.x $\leadsto$ *integer*
Width of the window.
Default Value : 256
Typical range of values : $0 \leq \text{Width} \text{ (lin)}$
Minimum Increment : 1
Recommended Increment : 1
Restriction : $(\text{Width} > 0) \vee (\text{Width} = -1)$
- ▷ **Height** (input_control) rectangle.extent.y $\leadsto$ *integer*
Height of the window.
Default Value : 256
Typical range of values : $0 \leq \text{Height} \text{ (lin)}$
Minimum Increment : 1
Recommended Increment : 1
Restriction : $(\text{Height} > 0) \vee (\text{Height} = -1)$
- ▷ **FatherWindow** (input_control) *integer* $\leadsto$ *integer* / *string*
Logical number of the father window. To specify the display as father you may enter 'root' or 0.
Default Value : 0
Restriction : $\text{FatherWindow} \geq 0$
- ▷ **Mode** (input_control) *string* $\leadsto$ *string*
Window mode.
Default Value : 'visible'
List of values : $\text{Mode} \in \{\text{'visible'}, \text{'invisible'}, \text{'transparent'}, \text{'buffer'}\}$
- ▷ **Machine** (input_control) *string* $\leadsto$ *string*
Name of the computer on which you want to open the window. Otherwise the empty string.
Default Value : ''
- ▷ **WindowHandle** (output_control) *window* $\leadsto$ *integer*
Window identifier.

Example

```
open_window(0,0,400,-1,'root','visible','',WindowHandle)
read_image(Image,'fabrik')
disp_image(Image,WindowHandle)
write_string(WindowHandle,'File, fabrik.ima')
```

```

new_line(WindowHandle)
get_mbutton(WindowHandle,_,_,_)
set_lut(WindowHandle,'temperature')
set_color(WindowHandle,'blue')
write_string(WindowHandle,'temperature')
new_line(WindowHandle)
write_string(WindowHandle,'Draw Rectangle')
new_line(WindowHandle)
draw_rectangle1(WindowHandle,Row1,Column1,Row2,Column2)
set_part(Row1,Column1,Row2,Column2)
disp_image(Image,WindowHandle)
new_line(WindowHandle).

```

Result

If the values of the specified parameters are correct `open_window` returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Parallelization Information

`open_window` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`reset_obj_db`

Possible Successors

`set_color`, `query_window_type`, `get_window_type`, `set_window_type`, `get_mposition`, `set_tposition`, `set_tshape`, `set_window_extents`, `get_window_extents`, `query_color`, `set_check`, `set_system`

Alternatives

`open_textwindow`

See also

`disp_region`, `disp_image`, `disp_color`, `set_lut`, `query_color`, `set_color`, `set_rgb`, `set_hsi`, `set_pixel`, `set_gray`, `set_part`, `set_part_style`, `query_window_type`, `get_window_type`, `set_window_type`, `get_mposition`, `set_tposition`, `set_window_extents`, `get_window_extents`, `set_window_attr`, `set_check`, `set_system`

Module

System

query_window_type (: : : WindowTypes)

Query all available window types.

`query_window_type` returns a tuple which contains all devices or software systems, respectively, which are used to display image objects. You may use `query_window_type` usefully while developing machine independent programs. Possible values are:

'X-Window' X-Window Version 11.

'pixmap' Windows are not displayed, but managed in memory. In this manner it is possible to port HALCON programs to computers without graphical display.

'PostScript' Objects are output to a PostScript File.

Parameter

- ▷ **WindowTypes** (output_control) string-array $\leadsto$ string
Names of available window types.

Result

`query_window_type` always returns 2 (H_MSG_TRUE).

Parallelization Information

`query_window_type` is *reentrant*, *local*, and processed *without* parallelization.

	Possible Predecessors
reset_obj_db	
	Module
System	

set_window_attr (: : AttributeName, AttributeValue :)

Set window characteristics.

You may use [set_window_attr](#) to set specific characteristics of graphics windows. With it you may modify the following default parameters of a window:

'border_width' Width of the window border in pixels.

'border_color' Color of the window border.

'background_color' Background color of the window.

'window_title' Name of the window in the titlebar.

	Attention
You have to call set_window_attr before calling open_window .	
	Parameter

- ▷ **AttributeName** (input_control) string $\leadsto$ string
Name of the attribute that should be modified.
List of values : AttributeName $\in$ { 'border_width', 'border_color', 'background_color', 'window_title' }
- ▷ **AttributeValue** (input_control) string $\leadsto$ string / integer
Value of the attribute that should be set.
List of values : AttributeValue $\in$ { 0, 1, 2, 'white', 'black', 'MyName', 'default' }

	Result
If the parameters are correct set_window_attr returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.	

	Parallelization Information
set_window_attr is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.	

	Possible Predecessors
open_window , set_draw , set_color , set_colored , set_line_width , open_textwindow	
	See also
open_window , get_window_attr	
	Module
System	

set_window_dc (: : WindowHandle, WINHDC :)

Set the device context of a virtual graphics window (Windows NT).

[set_window_dc](#) sets the device context of a window previously opened with [new_extern_window](#). All output ([disp_region](#), [disp_image](#), etc.) is done in the window with this device context.

The parameter **WINHDC** contains the device context of the window in which HALCON should output its data. This device context is used in all output routines of HALCON.

	Attention
The window WindowHandle has to be created with new_extern_window beforehand.	

	Parameter	
▷ WindowHandle (input_control)	window	↷ integer
Window identifier.		
▷ WINHDC (input_control)	integer	↷ integer
devicecontext of WINHWnd.		
Restriction : WINHDC ≠ 0		
	Example	

```

hwnd = createWINDOW(...)
new_extern_window(hwnd, hdc, 0,0,400,-1,WindowHandle)
set_device_context(WindowHandle, hdc)
read_image(Image,'fabrik')
disp_image(Image,WindowHandle)
write_string(WindowHandle,'File, fabrik.ima')
new_line(WindowHandle)
get_mbutton(WindowHandle,_,_,_)
set_lut(WindowHandle,'temperature')
set_color(WindowHandle,'blue')
write_string(WindowHandle,'temperature')
new_line(WindowHandle)
write_string(WindowHandle,'Draw Rectangle')
new_line(WindowHandle)
draw_rectangle1(WindowHandle,Row1,Column1,Row2,Column2)
set_part(Row1,Column1,Row2,Column2)
disp_image(Image,WindowHandle)
new_line(WindowHandle).

```

Result

If the values of the specified parameters are correct, [set_window_dc](#) returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Parallelization Information

[set_window_dc](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[new_extern_window](#)

Possible Successors

[disp_image](#), [disp_region](#)

See also

[new_extern_window](#), [disp_region](#), [disp_image](#), [disp_color](#), [set_lut](#), [query_color](#), [set_color](#), [set_rgb](#), [set_hsi](#), [set_pixel](#), [set_gray](#), [set_part](#), [set_part_style](#), [query_window_type](#), [get_window_type](#), [set_window_type](#), [get_mposition](#), [set_tposition](#), [set_window_extents](#), [get_window_extents](#), [set_window_attr](#), [set_check](#), [set_system](#)

Module

System

set_window_extents (: : WindowHandle, Row, Column, Width, Height :)

Modify position and size of a window.

[set_window_extents](#) positions the upper left corner of the output window at ([Row](#),[Column](#)) and changes the size of the window to [Width](#) and [Height](#) at the same time.

Attention

If you modify the size of a window an adaptation of the displayed data to the new format is not processed automatically. This has to be done by the program in performing another output of these data.

Parameter	
▷ WindowHandle (input_control)	window $\leadsto$ integer Window identifier.
▷ Row (input_control)	rectangle.origin.y $\leadsto$ integer Row index of upper left corner in target position. Default Value : 0 Typical range of values : $0 \leq \text{Row} (\text{lin})$ Minimum Increment : 1 Recommended Increment : 1
▷ Column (input_control)	rectangle.origin.x $\leadsto$ integer Column index of upper left corner in target position. Default Value : 0 Typical range of values : $0 \leq \text{Column} (\text{lin})$ Minimum Increment : 1 Recommended Increment : 1
▷ Width (input_control)	rectangle.extent.x $\leadsto$ integer Width of the window. Default Value : 512 Typical range of values : $0 \leq \text{Width} (\text{lin})$ Minimum Increment : 1 Recommended Increment : 1
▷ Height (input_control)	rectangle.extent.y $\leadsto$ integer Height of the window. Default Value : 512 Typical range of values : $0 \leq \text{Height} (\text{lin})$ Minimum Increment : 1 Recommended Increment : 1

Result
If the window is valid and the parameters are correct [set_window_extents](#) returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Parallelization Information
[set_window_extents](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors
[open_window](#), [open_textwindow](#)

See also
[get_window_extents](#), [open_window](#), [open_textwindow](#)

Module
System

set_window_type (: : WindowType :)

Specify a window type.

[set_window_type](#) determines on which type of output device the output is going to be displayed. This specification is going to be used by procedure [open_window](#) while opening the windows. You may open different windows on different types of output devices. Therefore you have to specify the wanted type before opening. You may request the available types of output devices by calling procedure [query_window_type](#). Possible values are:

'X-Window' X-Window Version 11.

'WIN32-Window' Microsoft Windows.

'pixmap' Windows are not displayed, but managed in memory only. In this manner you may port HALCON programs to computers without graphical display.

'PostScript' Objects are output to a PostScript File.

A useful usage of `set_window_type` could be the development of machine independent programs.

Parameter

- ▷ **WindowType** (input_control) string $\leadsto$ string
 Name of the window type which has to be set.
Default Value : 'X-Window'
List of values : WindowType $\in$ {'X-Window', 'WIN32-Window', 'pixmap', 'PostScript'}

Result

If the type of the output device is available, then `set_window_type` returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Parallelization Information

`set_window_type` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`open_window`, `open_textwindow`

See also

`open_window`, `open_textwindow`, `query_window_type`, `get_window_type`

Module

System

slide_image (: : WindowHandleSource1, WindowHandleSource2,
 WindowHandle :)

Interactive output from two window buffers.

`slide_image` divides the window horizontal in two logical areas dependent of the mouse position. The content of the first indicated window is copied in the upper area, the content of the second window is copied in the lower area. If you press the left mouse button you may scroll the delimitation between the two areas (you may move the mouse outside the window, too. In doing so the position of the mouse relative to the window determines the borderline).

Pressing the right mouse button in the window terminates the procedure `slide_image`.

A useful application of procedure `slide_image` might be the visualisation of the effect of a filtering operation for an image. The output is directed to the current set window (`WindowHandle`).

Attention

The three windows must have the same size and have to reside on the same computer.

Parameter

- ▷ **WindowHandleSource1** (input_control) window $\leadsto$ integer
 Logical window number of the "upper window".
 ▷ **WindowHandleSource2** (input_control) window $\leadsto$ integer
 Logical window number of the "lower window".
 ▷ **WindowHandle** (input_control) window $\leadsto$ integer
 Window identifier.

Example

```
read_image(Image, 'fabrik')
sobel_amp(Image, Amp, 'sum_abs', 3)
open_window(0, 0, -1, -1, 'root', 'buffer', '', WindowHandle)
disp_image(Amp, WindowHandle)
sobel_dir(Image, Dir, 'sum_abs', 3)
open_window(0, 0, -1, -1, 'root', 'buffer', '', WindowHandle)
disp_image(Dir, WindowHandle)
open_window(0, 0, -1, -1, 'root', 'visible', '', WindowHandle)
slide_image(Puffer1, Puffer2, WindowHandle).
```

Result

If the both windows exist and one of these windows is valid [slide_image](#) returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Parallelization Information

[slide_image](#) is *reentrant*, *local*, and processed without parallelization.

Possible Predecessors

[open_window](#), [open_textwindow](#)

Alternatives

[copy_rectangle](#), [get_mposition](#)

See also

[open_window](#), [open_textwindow](#), [move_rectangle](#)

Module

System

Chapter 7

Image

7.1 Access

get_grayval (Image : : Row, Column : Grayval)

Access the gray values of an image object.

The operator **Grayval** is a tuple of floating point numbers, integer respectively, which returns the gray values of several pixels of **Image**. The line coordinates of the pixels are in the tuple **Row**, the columns in **Column**.

Attention

The type of the values of **Grayval** depends on the type of the gray values.

Gray values which do not belong to the image can also be accessed. The state of these gray values is not ascertained.

The operator **get_grayval** involves a lot of work. It is not suitable for programming image processing operations such as filters. In this case it is more useful to use the procedure **get_image_pointer1** or to directly use the C interface for integrating own procedures.

Parameter

- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex
Image whose gray value is to be accessed.
- ▷ **Row** (input_control) point.y(-array) $\leadsto$ integer
Line numbers of pixels to be viewed.
Default Value : 0
Suggested values : Row $\in$ {0, 64, 128, 256, 512, 1024}
Typical range of values : $0 \leq \text{Row} \leq 32768$ (lin)
Minimum Increment : 1
Recommended Increment : 1
Restriction : $(0 \leq \text{Row}) \wedge (\text{Row} < \text{height}(\text{Image}))$
- ▷ **Column** (input_control) point.x(-array) $\leadsto$ integer
Column numbers of pixels to be viewed.
Default Value : 0
Suggested values : Column $\in$ {0, 64, 128, 256, 512, 1024}
Typical range of values : $0 \leq \text{Column} \leq 32768$ (lin)
Minimum Increment : 1
Recommended Increment : 1
Number of elements : Column = Row
Restriction : $(0 \leq \text{Column}) \wedge (\text{Column} < \text{width}(\text{Image}))$
- ▷ **Grayval** (output_control) grayval(-array) $\leadsto$ real / integer
Gray values of indicated pixels.
Number of elements : Grayval = Row

Result

If the state of the parameters is correct the operator **get_grayval** returns the value 2 (H_MSG_TRUE).

The behavior in case of empty input (no input images available) is set via the operator `set_system('no_object_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`get_grayval` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`read_image`

Alternatives

`get_image_pointer1`

See also

`set_grayval`

Module

Image / region / XLD management

get_image_pointer1 (Image : : : Pointer, Type, Width, Height)

Access the pointer of a channel.

The operator `get_image_pointer1` returns a C pointer to the first channel of the image `Image`. Additionally the image type (`Type` = 'byte', 'integer', 'float' etc.) and the image size (width and height) are returned. Consequently a direct access to the image data in the HALCON databank from the HALCON host language via the pointer is possible. An image is stored in HALCON as a vector of image lines.

Attention

The operator `get_image_pointer1` should only be used for entry into newly created images, since otherwise the gray values of other images might be overwritten (see relational structure).

Parameter

- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Input image.
- ▷ **Pointer** (output_control) pointer $\leadsto$ *integer*
Pointer to the image data in the HALCON databank.
- ▷ **Type** (output_control) string $\leadsto$ *string*
Type of image.
List of values : `Type` $\in$ { 'int1', 'int2', 'uint2', 'int4', 'byte', 'real', 'direction', 'cyclic', 'complex', 'dvf', 'lut' }
- ▷ **Width** (output_control) extent.x $\leadsto$ *integer*
Width of image.
- ▷ **Height** (output_control) extent.y $\leadsto$ *integer*
Height of image.

Example

```
Hobject  Bild;
char     typ[128];
long     width,height;
unsigned char *ptr;

read_image(&Bild,"fabrik");
get_image_pointer1(Bild,(long*)&ptr,typ,&width,&height);
```

Result

The operator `get_image_pointer1` returns the value 2 (H_MSG_TRUE) if exactly one image was passed. The behavior in case of empty input (no input images available) is set via the operator `set_system('no_object_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`get_image_pointer1` is *reentrant* and processed *without* parallelization.

	Possible Predecessors
read_image	
	Alternatives
set_grayval , get_grayval , get_image_pointer3	
	See also
paint_region , paint_gray	
	Module
Image / region / XLD management	

get_image_pointer1_rect (Image : : : PixelPointer, Width, Height, VerticalPitch, HorizontalBitPitch, BitsPerPixel)

Access to the image data pointer and the image data inside the smallest rectangle of the domain of the input image.

The operator [get_image_pointer1_rect](#) returns the pointer [PixelPointer](#) which points to the beginning of the image data inside the smallest rectangle of the domain of [Image](#). [VerticalPitch](#) corresponds to the width of the input image [Image](#) multiplied with the number of bytes per pixel ([HorizontalBitPitch](#) / 8). [Width](#) and [Height](#) correspond to the size of the smallest rectangle of the input region. [HorizontalBitPitch](#) is the horizontal distance (in bits) between two neighbouring pixels. [BitsPerPixel](#) is the number of used bits per pixel. [get_image_pointer1_rect](#) is symmetrical to [gen_image1_rect](#).

Attention

The operator [get_image_pointer1_rect](#) should only be used for entry into newly created images, since otherwise the gray values of other images might be overwritten (see relational structure).

	Parameter
▷ Image (input_object)	image $\leadsto$ <i>Hobject</i> : byte / uint2 / int4 Input image (Himage).
▷ PixelPointer (output_control)	pointer $\leadsto$ <i>integer</i> Pointer to the image data.
▷ Width (output_control)	extent.x $\leadsto$ <i>integer</i> Width of the output image.
▷ Height (output_control)	extent.y $\leadsto$ <i>integer</i> Height of the output image.
▷ VerticalPitch (output_control)	integer $\leadsto$ <i>integer</i> Width(input image)*(HorizontalBitPitch/8).
▷ HorizontalBitPitch (output_control)	integer $\leadsto$ <i>integer</i> Distance between two neighbouring pixels in bits . Default Value : 8 List of values : HorizontalBitPitch $\in$ {8, 16, 32}
▷ BitsPerPixel (output_control)	integer $\leadsto$ <i>integer</i> Number of used bits per pixel. Default Value : 8 List of values : BitsPerPixel $\in$ {8, 16, 32}

Example

```
Hobject image, reg, imagereduced;
char typ[128];
long width, height, vert_pitch, hori_bit_pitch, bits_per_pix, winID;
unsigned char *ptr;

open_window(0, 0, 512, 512, "black", winID);
read_image(&image, "monkey");
draw_region(&reg, winID);
reduce_domain(image, reg, &imagereduced);
get_image_pointer1_rect(imagereduced, (long*)&ptr, &width, &height,
                        &vert_pitch, &hori_bit_pitch, &bits_per_pix);
```

Result

The operator `get_image_pointer1_rect` returns the value 2 (H_MSG_TRUE) if exactly one image was passed. The behavior in case of empty input (no input images available) is set via the operator `set_system ('no_object_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`get_image_pointer1_rect` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`read_image`, `gen_image1_rect`

Alternatives

`set_grayval`, `get_grayval`, `get_image_pointer3`, `get_image_pointer1`

See also

`paint_region`, `paint_gray`, `gen_image1_rect`

Module

Image / region / XLD management

get_image_pointer3 (ImageRGB : : : PointerRed, PointerGreen, PointerBlue, Type, Width, Height)

Access the pointers of a colored image.

The operator `get_image_pointer3` returns a C pointer to the three channels of a colored image (**ImageRGB**). Additionally the image type (**Type** = 'byte', 'int2', 'float' etc.) and the image size (**Width** and **Height**) are returned. Consequently a direct access to the image data in the HALCON databank from the HALCON host language via the pointer is possible. An image is stored in HALCON as a vector of image lines. The three channels must have the same pixel type and the same size.

Attention

Only one image can be passed. The operator `get_image_pointer3` should only be used for entry into newly created images, since otherwise the gray values of other images might be overwritten (see relational structure).

Parameter

- ▷ **ImageRGB** (input_object) image $\leadsto$ Hobject : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Input image.
- ▷ **PointerRed** (output_control) pointer $\leadsto$ integer
Pointer to the pixels of the first channel.
- ▷ **PointerGreen** (output_control) pointer $\leadsto$ integer
Pointer to the pixels of the second channel.
- ▷ **PointerBlue** (output_control) pointer $\leadsto$ integer
Pointer to the pixels of the third channel.
- ▷ **Type** (output_control) string $\leadsto$ string
Type of image.
List of values : Type $\in$ { 'int1', 'int2', 'uint2', 'int4', 'byte', 'real', 'direction', 'cyclic', 'complex', 'dvf', 'lut' }
- ▷ **Width** (output_control) extent.x $\leadsto$ integer
Width of image.
- ▷ **Height** (output_control) extent.y $\leadsto$ integer
Height of image.

Result

The operator `get_image_pointer3` returns the value 2 (H_MSG_TRUE) if exactly one image is passed. The behavior in case of empty input (no input images available) is set via the operator `set_system ('no_object_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`get_image_pointer3` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`read_image`

<i>Alternatives</i>
set_grayval , get_grayval , get_image_pointer1
<i>See also</i>
paint_region , paint_gray
<i>Module</i>
Image / region / XLD management

get_image_time (Image : : : MSecond, Second, Minute, Hour, Day, YDay, Month, Year)

Request time at which the image was created.

The operator [get_image_time](#) returns the time at which the image was created.

<i>Parameter</i>
▷ Image (input_object) image $\leadsto$ Hobject : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Input image. ▷ MSecond (output_control) integer $\leadsto$ integer Milliseconds (0..999). ▷ Second (output_control) integer $\leadsto$ integer Seconds (0..59). ▷ Minute (output_control) integer $\leadsto$ integer Minutes (0..59). ▷ Hour (output_control) integer $\leadsto$ integer Hours (0..11). ▷ Day (output_control) integer $\leadsto$ integer Day of the month (1..31). ▷ YDay (output_control) integer $\leadsto$ integer Day of the year (1..365). ▷ Month (output_control) integer $\leadsto$ integer Month (1..12). ▷ Year (output_control) integer $\leadsto$ integer Year (xxxx).

<i>Result</i>
The operator get_image_time returns the value 2 (H_MSG_TRUE) if exactly one image was passed. The behavior in case of empty input (no input images available) is set via the operator set_system ('no_object_result', <Result>). If necessary an exception handling is raised.

<i>Parallelization Information</i>
get_image_time is <i>reentrant</i>, <i>local</i>, and processed <i>without</i> parallelization.

<i>Possible Predecessors</i>
read_image, grab_image

<i>See also</i>
count_seconds

<i>Module</i>
Image / region / XLD management

7.2 Channel

access_channel (MultiChannelImage : Image : Channel :)

Access a channel of a multichannel image.

The operator [access_channel](#) accesses a channel of the (multichannel) input image. The result is a one-channel image. The definition domain of the input is adopted. The channels are numbered from 1 to n. The number of channels can be determined via the operator [count_channels](#).

Parameter

- ▷ **MultiChannelImage** (input_object) multichannel-image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
 Multichannel image.
- ▷ **Image** (output_object) singlechannel-image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
 One channel of MultiChannelImage.
- ▷ **Channel** (input_control) channel $\leadsto$ *integer*
 Index of channel to be accessed.
Default Value : 1
Suggested values : Channel $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12\}$
Typical range of values : $1 \leq \text{Channel}$

Example

```
read_image(&Color,"patras"); /* Farbbild einlesen */
access_channel(Color,&Red,1); /* Rotkanal extrahieren */
disp_image(Red,windowHandle);
```

Parallelization Information

[access_channel](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[count_channels](#)

Possible Successors

[disp_image](#)

Alternatives

[decompose2](#), [decompose3](#), [decompose4](#), [decompose5](#)

See also

[count_channels](#)

Module

Image / region / XLD management

append_channel (MultiChannelImage, Image : ImageExtended : :)

Append additional matrices (channels) to the image.

The operator [append_channel](#) appends the matrices of the image [Image](#) to the matrices of [MultiChannelImage](#). The result is an image containing as many matrices (channels) as [MultiChannelImage](#) and [Image](#) combined. The definition domain of the output image is calculated as the average of the definition domains of both input images.

Parameter

- ▷ **MultiChannelImage** (input_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
 Multichannel image.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
 Image to be appended.
- ▷ **ImageExtended** (output_object) multichannel-image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
 Image appended by [Image](#).

Parallelization Information

`append_channel` is *reentrant* and processed *without* parallelization.

Possible Successors

`disp_image`

Alternatives

`compose2`, `compose3`, `compose4`, `compose5`

Module

Image / region / XLD management

channels_to_image (Images : MultiChannelImage : :)

Convert one-channel images into a multichannel image

The operator `channels_to_image` converts several one-channel images into a multichannel image. The new definition domain is the average of the definition domains of the input images.

Parameter

- ▷ **Images** (input_object) singlechannel-image-array $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

One-channel images to be combined into a one-channel image.

- ▷ **MultiChannelImage** (output_object) multichannel-image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

Multichannel image.

Parallelization Information

`channels_to_image` is *reentrant*, *local*, and processed *without* parallelization.

Possible Successors

`count_channels`, `disp_image`

Module

Image / region / XLD management

compose2 (Image1, Image2 : MultiChannelImage : :)

Convert two images into a two-channel image.

The operator `compose2` converts 2 one-channel images into a 2-channel image. The definition domain is calculated as the intersection of the definition domains of the input images.

Parameter

- ▷ **Image1** (input_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

Input image 1.

- ▷ **Image2** (input_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

Input image 2.

- ▷ **MultiChannelImage** (output_object) multichannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

Multichannel image.

Parallelization Information

`compose2` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

`disp_image`

	Alternatives
append_channel	
	See also
decompose2	
	Module
Image / region / XLD management	

compose3 (Image1, Image2, Image3 : MultiChannelImage : :)

Convert 3 images into a three-channel image.

The operator [compose3](#) converts 3 one-channel images into a 3-channel image. The definition domain is calculated as the intersection of the definition domains of the input images.

	Parameter
▷ Image1 (input_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Input image 1.	
▷ Image2 (input_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Input image 2.	
▷ Image3 (input_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Input image 3.	
▷ MultiChannelImage (output_object) multichannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Multichannel image.	

	Parallelization Information
compose3 is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).	

	Possible Successors
disp_image	
	Alternatives
append_channel	
	See also
decompose3	
	Module
Image / region / XLD management	

compose4 (Image1, Image2, Image3, Image4 : MultiChannelImage : :)

Convert 4 images into a four-channel image.

The operator [compose4](#) converts 4 one-channel images into a 4-channel image. The definition domain is calculated as the intersection of the definition domains of the input images.

	Parameter
▷ Image1 (input_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Input image 1.	
▷ Image2 (input_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Input image 2.	

- ▷ **Image3** (input_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

Input image 3.

- ▷ **Image4** (input_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

Input image 4.

- ▷ **MultiChannelImage** (output_object) multichannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

Multichannel image.

Parallelization Information

[compose4](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

[disp_image](#)

Alternatives

[append_channel](#)

See also

[decompose4](#)

Module

Image / region / XLD management

compose5 (Image1, Image2, Image3, Image4,
Image5 : MultiChannelImage : :)

Convert 5 images into a five-channel image.

The operator [compose5](#) converts 5 one-channel images into a 5-channel image. The definition domain is calculated as the intersection of the definition domains of the input images.

Parameter

- ▷ **Image1** (input_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

Input image 1.

- ▷ **Image2** (input_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

Input image 2.

- ▷ **Image3** (input_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

Input image 3.

- ▷ **Image4** (input_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

Input image 4.

- ▷ **Image5** (input_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

Input image 5.

- ▷ **MultiChannelImage** (output_object) multichannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

Multichannel image.

Parallelization Information

[compose5](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

[disp_image](#)

Alternatives

[append_channel](#)

See also

[decompose5](#)

Module

Image / region / XLD management

compose6 (Image1, Image2, Image3, Image4, Image5,
Image6 : MultiChannelImage : :)

Convert 6 images into a six-channel image.

The operator [compose6](#) converts 6 one-channel images into a 6-channel image. The definition domain is calculated as the intersection of the definition domains of the input images.

Parameter

- ▷ **Image1** (input_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 /
uint2 / int4 / real / complex / dvf
Input image 1.
- ▷ **Image2** (input_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 /
uint2 / int4 / real / complex / dvf
Input image 2.
- ▷ **Image3** (input_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 /
uint2 / int4 / real / complex / dvf
Input image 3.
- ▷ **Image4** (input_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 /
uint2 / int4 / real / complex / dvf
Input image 4.
- ▷ **Image5** (input_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 /
uint2 / int4 / real / complex / dvf
Input image 5.
- ▷ **Image6** (input_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 /
uint2 / int4 / real / complex / dvf
Input image 6.
- ▷ **MultiChannelImage** (output_object) multichannel-image(-array) $\leadsto$ *Hobject* : byte / direction /
cyclic / int1 / int2 / uint2 /
int4 / real / complex / dvf
Multichannel image.

Parallelization Information

[compose6](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

[disp_image](#)

Alternatives

[append_channel](#)

See also

[decompose6](#)

Module

Image / region / XLD management

compose7 (Image1, Image2, Image3, Image4, Image5, Image6,
Image7 : MultiChannelImage : :)

Convert 7 images into a seven-channel image.

The operator [compose7](#) converts 7 one-channel images into a 7-channel image. The definition domain is calculated as the intersection of the definition domains of the input images.

Parameter	
▷ Image1 (input_object) singlechannel-image(-array) $\leadsto$	<i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Input image 1.	
▷ Image2 (input_object) singlechannel-image(-array) $\leadsto$	<i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Input image 2.	
▷ Image3 (input_object) singlechannel-image(-array) $\leadsto$	<i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Input image 3.	
▷ Image4 (input_object) singlechannel-image(-array) $\leadsto$	<i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Input image 4.	
▷ Image5 (input_object) singlechannel-image(-array) $\leadsto$	<i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Input image 5.	
▷ Image6 (input_object) singlechannel-image(-array) $\leadsto$	<i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Input image 6.	
▷ Image7 (input_object) singlechannel-image(-array) $\leadsto$	<i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Input image 7.	
▷ MultiChannelImage (output_object) multichannel-image(-array) $\leadsto$	<i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Multichannel image.	

Parallelization Information

[compose7](#) is *reentrant* and automatically *parallelized* (on *tuple* level).

Possible Successors

[disp_image](#)

Alternatives

[append_channel](#)

See also

[decompose7](#)

Module

Image / region / XLD management

count_channels (MultiChannelImage : : : Channels)

Count channels of image.

The operator [count_channels](#) counts the number of channels of all input images.

Parameter	
▷ MultiChannelImage (input_object) (multichannel-)image(-array) $\leadsto$	<i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
One- or multichannel image.	
▷ Channels (output_control) integer(-array) $\leadsto$	<i>integer</i>
Number of channels.	

Example

```
read_image(&Color,"patras");
```

```

count_channels(Color,&num_channels);
for (i=1; i<=num_channels; i++)
{
    access_channel(Color,&Channel,i);
    disp_image(Channel,windowHandle);
    clear_obj(Channel);
}

```

Parallelization Information

[count_channels](#) is *reentrant* and processed *without parallelization*.

Possible Successors

[access_channel](#), [append_channel](#), [disp_image](#)

See also

[append_channel](#), [access_channel](#)

Module

Image / region / XLD management

decompose2 (MultiChannelImage : Image1, Image2 : :)

Convert a two-channel image into two images.

The operator [decompose2](#) converts a 2-channel image into two one-channel images with the same definition domain.

Parameter

▷ **MultiChannelImage** (input_object) multichannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

 Multichannel image.

▷ **Image1** (output_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

 Output image 1.

▷ **Image2** (output_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf

 Output image 2.

Parallelization Information

[decompose2](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[count_channels](#)

Possible Successors

[disp_image](#)

Alternatives

[access_channel](#), [image_to_channels](#)

See also

[compose2](#)

Module

Image / region / XLD management

decompose3 (MultiChannelImage : Image1, Image2, Image3 : :)

Convert a three-channel image into three images.

The operator [decompose3](#) converts a 3-channel image into three one-channel images with the same definition domain.

Parameter
▷ MultiChannelImage (input_object) multichannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Multichannel image. ▷ Image1 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Output image 1. ▷ Image2 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Output image 2. ▷ Image3 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Output image 3.
Parallelization Information
decompose3 is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>).
Possible Predecessors
count_channels
Possible Successors
disp_image
Alternatives
access_channel , image_to_channels
See also
compose3
Module
Image / region / XLD management

decompose4 (MultiChannelImage : Image1, Image2, Image3, Image4 : :)

Convert a four-channel image into four images.

The operator [decompose4](#) converts a 4-channel image into four one-channel images with the same definition domain.

Parameter
▷ MultiChannelImage (input_object) multichannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Multichannel image. ▷ Image1 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Output image 1. ▷ Image2 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Output image 2. ▷ Image3 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Output image 3. ▷ Image4 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Output image 4.
Parallelization Information
decompose4 is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>).

count_channels	Possible Predecessors
disp_image	Possible Successors
access_channel, image_to_channels	Alternatives
compose4	See also
Image / region / XLD management	Module

decompose5 (MultiChannelImage : Image1, Image2, Image3, Image4, Image5 : :)

Convert a five-channel image into five images.

The operator [decompose5](#) converts a 5-channel image into five one-channel images with the same definition domain.

	Parameter
▷ MultiChannelImage (input_object) multichannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Multichannel image.	
▷ Image1 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Output image 1.	
▷ Image2 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Output image 2.	
▷ Image3 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Output image 3.	
▷ Image4 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Output image 4.	
▷ Image5 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Output image 5.	

decompose5 is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>).	Parallelization Information
count_channels	Possible Predecessors
disp_image	Possible Successors
access_channel, image_to_channels	Alternatives
compose5	See also
Image / region / XLD management	Module

decompose6 (MultiChannelImage : Image1, Image2, Image3, Image4, Image5, Image6 : :)

Convert a six-channel image into six images.

The operator [decompose6](#) converts a 6-channel image into six one-channel images with the same definition domain.

Parameter
▷ MultiChannelImage (input_object) multichannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Multichannel image.
▷ Image1 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Output image 1.
▷ Image2 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Output image 2.
▷ Image3 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Output image 3.
▷ Image4 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Output image 4.
▷ Image5 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Output image 5.
▷ Image6 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Output image 6.
Parallelization Information
decompose6 is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).
Possible Predecessors
count_channels
Possible Successors
disp_image
Alternatives
access_channel , image_to_channels
See also
compose6
Module
Image / region / XLD management

decompose7 (MultiChannelImage : Image1, Image2, Image3, Image4, Image5, Image6, Image7 : :)

Convert a seven-channel image into seven images.

The operator [decompose7](#) converts a 7-channel image into seven one-channel images with the same definition domain.

Parameter	
▷ MultiChannelImage (input_object) multichannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Multichannel image.	
▷ Image1 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Output image 1.	
▷ Image2 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Output image 2.	
▷ Image3 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Output image 3.	
▷ Image4 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Output image 4.	
▷ Image5 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Output image 5.	
▷ Image6 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Output image 6.	
▷ Image7 (output_object) singlechannel-image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Output image 7.	
Parallelization Information	
decompose7 is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>).	
Possible Predecessors	
count_channels	
Possible Successors	
disp_image	
Alternatives	
access_channel , image_to_channels	
See also	
compose7	
Module	
Image / region / XLD management	

image_to_channels (MultiChannelImage : Images : :)

Convert a multichannel image into One-channel images

The operator [image_to_channels](#) generates a one-channel image for each channel of the multichannel image in [MultiChannelImage](#). The definition domains are adopted from the input image. As many images are created as [MultiChannelImage](#) has channels.

Parameter	
▷ MultiChannelImage (input_object) multichannel-image $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Multichannel image to be decomposed.	
▷ Images (output_object) singlechannel-image-array $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf	
Generated one-channel images.	

<i>Parallelization Information</i>
image_to_channels is <i>reentrant</i> and processed <i>without parallelization</i> .
<i>Possible Predecessors</i>
count_channels
<i>Possible Successors</i>
disp_image
<i>Alternatives</i>
access_channel , decompose2 , decompose3 , decompose4 , decompose5
<i>Module</i>
Image / region / XLD management

7.3 Creation

copy_image (Image : DupImage : :)

Copy an image and allocate new memory for it.

[copy_image](#) copies the first channel of the input image into a new (single-channel) image with the same domain as the input image. In contrast to HALCON operators such as [copy_obj](#), the image is copied into a newly allocated memory segment. This can be used to modify the gray values of the new image, for example (see [get_image_pointer1](#)).

<i>Parameter</i>
▷ Image (input_object) image $\rightsquigarrow$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Image to be copied.
▷ DupImage (output_object) image $\rightsquigarrow$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Copied image.

<i>Parallelization Information</i>
copy_image is <i>reentrant</i> and processed <i>without parallelization</i> .
<i>Possible Predecessors</i>
read_image , gen_image_const
<i>Possible Successors</i>
set_grayval , get_image_pointer1
<i>Alternatives</i>
set_grayval , paint_gray , gen_image_const , gen_image_proto
<i>See also</i>
get_image_pointer1
<i>Module</i>
Basic operators

gen_image1 (: Image : Type, Width, Height, PixelPointer :)

Create an image from a pointer to the pixels.

The operator [gen_image1](#) creates an image of the size [Width](#) $\times$ [Height](#). The pixels in [PixelPointer](#) are stored line-sequentially. The type of the given pixels ([PixelPointer](#)) must correspond to [Type](#). The storage for the new image is newly created by HALCON. Thus, the storage on the [PixelPointer](#) can be released after the call. Since the type of the parameter [PixelPointer](#) is generic (long) a cast has to be used for the call.

Parameter	
▷ Image (output_object)	image $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real Created image with new image matrix.
▷ Type (input_control)	string $\leadsto$ <i>string</i> Pixel type. Default Value : 'byte' List of values : Type $\in$ {'byte', 'direction', 'cyclic', 'int1', 'int2', 'uint2', 'int4', 'real'}
▷ Width (input_control)	extent.x $\leadsto$ <i>integer</i> Width of image. Default Value : 512 Suggested values : Width $\in$ {128, 256, 512, 1024} Typical range of values : $1 \leq \text{Width} \leq 512$ (lin) Minimum Increment : 1 Recommended Increment : 10 Restriction : Width ≥ 1
▷ Height (input_control)	extent.y $\leadsto$ <i>integer</i> Height of image. Default Value : 512 Suggested values : Height $\in$ {128, 256, 512, 1024} Typical range of values : $1 \leq \text{Height} \leq 512$ (lin) Minimum Increment : 1 Recommended Increment : 10 Restriction : Height ≥ 1
▷ PixelPointer (input_control)	integer $\leadsto$ <i>integer</i> Pointer to first gray value.

Example

```
void NewImage(Hobject *new)
{
    unsigned char  image[768*525];
    int            r, c;
    for (r=0; r<525; r++)
        for (c=0; c<768; c++)
            image[r*768+c] = c % 255;
    gen_image1(new, "byte", 768, 525, (long)image);
}
```

Result

If the parameter values are correct, the operator [gen_image1](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

[gen_image1](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_image_const](#), [get_image_pointer1](#)

Alternatives

[gen_image3](#), [gen_image_const](#), [get_image_pointer1](#)

See also

[reduce_domain](#), [paint_gray](#), [paint_region](#), [set_grayval](#)

Module

Image / region / XLD management

gen_image1_extern (: Image : Type, Width, Height, PixelPointer, ClearProc :)

Create an image from a pointer on the pixels with storage management.

The operator `gen_image1_extern` creates an image of the size `Width` $\times$ `Height`. The pixels in `PixelPointer` are stored line-sequentially. The type of the given pixels (`PixelPointer`) must correspond to `Type`. Since the type of the parameter `PixelPointer` is generic (long) a cast must be used for the call.

The memory for the new image is not newly allocated by HALCON, contrary to `gen_image1`, and thus is not copied either. This means that the memory space that `PixelPointer` points to must be released by deleting the object `Image`. This is done by the procedure `ClearProc` provided by the caller. This procedure must have the following signature:

```
void ClearProc(void* ptr);
```

It is called when deleting `Image`. If the memory shall not be released (in the case of frame grabbers or static memory) a procedure “without trunk” or the NULL-Pointer can be passed. Analogous to the parameter `PixelPointer` the pointer has to be passed to the procedure by casting it to long.

Parameter

- ▷ **Image** (output_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Created HALCON image.
- ▷ **Type** (input_control) string $\leadsto$ *string*
Pixel type.
Default Value : 'byte'
List of values : Type $\in$ { 'int1', 'int2', 'uint2', 'int4', 'byte', 'real', 'direction', 'cyclic' }
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer*
Width of image.
Default Value : 512
Suggested values : Width $\in$ { 128, 256, 512, 1024 }
Typical range of values : $1 \leq \text{Width} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : Width ≥ 1
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer*
Height of image.
Default Value : 512
Suggested values : Height $\in$ { 128, 256, 512, 1024 }
Typical range of values : $1 \leq \text{Height} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : Height ≥ 1
- ▷ **PixelPointer** (input_control) integer $\leadsto$ *integer*
Pointer to the first gray value.
- ▷ **ClearProc** (input_control) integer $\leadsto$ *integer*
Pointer to the procedure re-releasing the memory of the image when deleting the object.
Default Value : 0

Example

```
void NewImage(Hobject *new)
{
    unsigned char *image;
    int r, c;
    image = malloc(640*480);
    for (r=0; r<480; r++)
        for (c=0; c<640; c++)
            image[r*640+c] = c % 255;
    gen_image1_extern(new, "byte", 640, 480, (long)image, (long)free);
}
```

Result

The operator `gen_image1_extern` returns the value 2 (H_MSG_TRUE) if the parameter values are correct. Otherwise an exception handling is raised.

Parallelization Information

`gen_image1_extern` is *reentrant* and processed *without* parallelization.

Alternatives

`gen_image1`, `gen_image_const`, `get_image_pointer1`

See also

`reduce_domain`, `paint_gray`, `paint_region`, `set_grayval`

Module

Image / region / XLD management

gen_image1_rect (: Image : PixelPointer, Width, Height, VerticalPitch, HorizontalBitPitch, BitsPerPixel, DoCopy, ClearProc :)

Create an image with a rectangular domain from a pointer on the pixels (with storage management).

The operator `gen_image1_rect` creates an image of size $(\text{VerticalPitch} / (\text{HorizontalBitPitch} / 8)) * \text{Height}$. The pixels pointed to by `PixelPointer` are stored line by line. Since the type of the parameter `PixelPointer` is generic (long) a cast must be used for the call. `VerticalPitch` determines the distance (in bytes) between pixel m in row n and pixel m in row $n+1$ inside of memory. All rows of the 'input image' have the same vertical pitch. The width of the output image equals $\text{VerticalPitch} / (\text{HorizontalBitPitch} / 8)$. The height of input and output image are equal. The domain of the output image `Image` is a rectangle of the size $\text{Width} * \text{Height}$. The parameter `HorizontalBitPitch` is the horizontal distance (in bits) between two neighbouring pixels. `BitsPerPixel` is the number of used bits per pixel.

If `DoCopy` is set 'true', the image data pointed to by `PixelPointer` is copied and memory for the new image is newly allocated by HALCON. Else the image data is not duplicated and the memory space that `PixelPointer` points to must be released when deleting the object `Image`. This is done by the procedure `ClearProc` provided by the caller. This procedure must have the following signature:

```
void ClearProc(void* ptr);
```

It is called when deleting `Image`. If the memory shall not be released (in the case of frame grabbers or static memory) a procedure "without trunk" or the NULL-pointer can be passed. Analogously to the parameter `PixelPointer` the pointer has to be passed to the procedure by casting it to long. If `DoCopy` is 'true' then `ClearProc` is irrelevant. The operator `gen_image1_rect` is symmetrical to `get_image_pointer1_rect`.

Parameter

- ▷ **Image** (output_object) image $\leadsto$ *Hobject* : byte / uint2 / int4
Created HALCON image.
- ▷ **PixelPointer** (input_control) integer $\leadsto$ *integer*
Pointer to the first pixel.
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer*
Width of the image.
Default Value : 512
Suggested values : $\text{Width} \in \{128, 256, 512, 1024\}$
Typical range of values : $1 \leq \text{Width} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : $\text{Width} \geq 1$
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer*
Height of the image.
Default Value : 512
Suggested values : $\text{Height} \in \{128, 256, 512, 1024\}$
Typical range of values : $1 \leq \text{Height} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : $\text{Height} \geq 1$
- ▷ **VerticalPitch** (input_control) integer $\leadsto$ *integer*
Distance (in bytes) between pixel m in row n and pixel m in row $n+1$ of the 'input image'.
Restriction : $\text{VerticalPitch} \geq (\text{Width} \cdot (\text{HorizontalBitPitch} / 8))$

- ▷ **HorizontalBitPitch** (input_control) integer $\leadsto$ integer
Distance between two neighbouring pixels in bits.
Default Value : 8
List of values : HorizontalBitPitch $\in \{8, 16, 32\}$
- ▷ **BitsPerPixel** (input_control) integer $\leadsto$ integer
Number of used bits per pixel.
Default Value : 8
List of values : BitsPerPixel $\in \{8, 9, 10, 11, 12, 13, 14, 15, 16, 32\}$
Restriction : BitsPerPixel $\leq$ HorizontalBitPitch
- ▷ **DoCopy** (input_control) string $\leadsto$ string
Copy image data.
Default Value : 'false'
Suggested values : DoCopy $\in \{'true', 'false'\}$
- ▷ **ClearProc** (input_control) integer $\leadsto$ integer
Pointer to the procedure releasing the memory of the image when deleting the object.
Default Value : 0

Example

```
void NewImage(Hobject *new)
{
    unsigned char    *image;
    int              r, c;

    image = malloc(640*480);
    for (r=0; r<480; r++)
        for (c=0; c<640; c++)
            image[r*640+c] = c % 255;
    gen_image1_rect(new, (long)image, 400, 480, 640, 8, 8, 'false', (long)free);
}
```

Result

The operator [gen_image1_rect](#) returns the value 2 (H_MSG_TRUE) if the parameter values are correct. Otherwise an exception handling is raised.

Parallelization Information

[gen_image1_rect](#) is *reentrant* and processed *without parallelization*.

Possible Successors

[get_image_pointer1_rect](#)

Alternatives

[gen_image1](#), [gen_image1_extern](#)

See also

[get_image_pointer1_rect](#)

Module

Image / region / XLD management

gen_image3 (: ImageRGB : Type, Width, Height, PixelPointerRed, PixelPointerGreen, PixelPointerBlue :)

Create an image from three pointers to the pixels (red/green/blue).

The operator [gen_image3](#) creates a three-channel image of the size [Width](#) $\times$ [Height](#). The pixels in [PixelPointerRed](#), [PixelPointerGreen](#) and [PixelPointerBlue](#) are stored line-sequentially. The type of the given pixels ([PixelPointerRed](#) etc.) must correspond to the name of the pixels ([Type](#)). The storage for the new image is newly created by HALCON. Thus, it can be released after the call. Since the type of the parameters ([PixelPointerRed](#) etc.) is generic (long) a “cast” must be used for the call.

Parameter

- ▷ **ImageRGB** (output_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Created image with new image matrix.
- ▷ **Type** (input_control) string $\leadsto$ *string*
Pixel type.
Default Value : 'byte'
List of values : Type $\in$ {'byte', 'direction', 'cyclic', 'int1', 'int2', 'uint2', 'int4', 'real'}
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer*
Width of image.
Default Value : 512
Suggested values : Width $\in$ {128, 256, 512, 1024}
Typical range of values : $1 \leq \text{Width} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer*
Height of image.
Default Value : 512
Suggested values : Height $\in$ {128, 256, 512, 1024}
Typical range of values : $1 \leq \text{Height} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10
- ▷ **PixelPointerRed** (input_control) integer $\leadsto$ *integer*
Pointer to first red value (channel 1).
- ▷ **PixelPointerGreen** (input_control) integer $\leadsto$ *integer*
Pointer to first green value (channel 2).
- ▷ **PixelPointerBlue** (input_control) integer $\leadsto$ *integer*
Pointer to first blue value (channel 3).

Example

```
void NewRGBImage(Hobject *new)
{
    unsigned char  red[768*525];
    unsigned char  green[768*525];
    unsigned char  blue[768*525];
    int            r,c;
    for (r=0; r<525; r++)
        for (c=0; c<768; c++)
        {
            red[r*768+c]   = c % 255;
            green[r*768+c] = (767 - c) % 255;
            blue[r*768+c]  = r % 255;
        }
    gen_image3(new, "byte", 768, 525, (long)red, (long)green, (long)blue);
}

main()
{
    Hobject  rgb;
    open_window(0,0,768,525,0,"","",&windowHandle);
    NewRGBImage(&rgb);
    disp_color(rgb,windowHandle);
    clear_obj(rgb);
}
```

Result

If the parameter values are correct, the operator **gen_image3** returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

<i>Parallelization Information</i>
<code>gen_image3</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
<i>Possible Predecessors</i>
<code>gen_image_const</code> , <code>get_image_pointer1</code>
<i>Possible Successors</i>
<code>disp_color</code>
<i>Alternatives</i>
<code>gen_image1</code> , <code>compose3</code> , <code>gen_image_const</code>
<i>See also</i>
<code>reduce_domain</code> , <code>paint_gray</code> , <code>paint_region</code> , <code>set_grayval</code> , <code>get_image_pointer1</code> , <code>decompose3</code>
<i>Module</i>
Image / region / XLD management

gen_image_const (: Image : Type, Width, Height :)

Create an image with constant gray value.

The operator `gen_image_const` creates an image of the indicated size. The height and width of the image are determined by `Height` and `Width`. HALCON supports the following image types:

'byte' 1 byte per pixel (0..255)
'int1' 1 byte per pixel (-127..127)
'int2' 2 bytes per pixel (-32767..32767)
'uint2' 2 bytes per pixel (0..65535)
'int4' 4 bytes per pixel (-2147483647..2147483647)
'real' 4 bytes per pixel, floating point
'complex' two matrices of the type *real*
'dvf' two matrices of the type *int1*
'dir' 1 byte per pixel (0..180)
'cyclic' 1 byte per pixel; cyclic arithmetics (0..255).

The default value 0 is set via the operator `set_system('init_new_image', '<'true'/'false'>)`.

<i>Parameter</i>
▷ Image (output_object) image $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Created image with new image matrix. ▷ Type (input_control) string $\leadsto$ <i>string</i> Pixel type. Default Value : 'byte' List of values : Type $\in$ {'byte', 'direction', 'cyclic', 'int1', 'int2', 'uint2', 'int4', 'real', 'complex', 'dvf'} ▷ Width (input_control) extent.x $\leadsto$ <i>integer</i> Width of image. Default Value : 512 Suggested values : Width $\in$ {128, 256, 512, 1024} Typical range of values : $1 \leq \text{Width} \leq 512$ (lin) Minimum Increment : 1 Recommended Increment : 10 Restriction : Width ≥ 1

- ▷ **Height** (input_control) extent.y $\leadsto$ integer
 Height of image.
Default Value : 512
Suggested values : Height $\in \{128, 256, 512, 1024\}$
Typical range of values : $1 \leq \text{Height} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : Height ≥ 1

Example

```
gen_image_const(&New, "byte", width, height);
get_image_pointer1(New, (long*)&pointer, type, &width, &height);
for (row=0; row<height-1; row++)
  for (col=0; col<width-1; col++)
    pointer[row*width+col] = (row + col) % 256;
```

Result

If the parameter values are correct, the operator `gen_image_const` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

`gen_image_const` is *reentrant* and processed *without parallelization*.

Possible Successors

`paint_region`, `reduce_domain`, `get_image_pointer1`, `copy_obj`

Alternatives

`gen_image1`, `gen_image3`

See also

`reduce_domain`, `paint_gray`, `paint_region`, `set_grayval`, `get_image_pointer1`

Module

Image / region / XLD management

gen_image_gray_ramp (: ImageGrayRamp : Alpha, Beta, Mean, Row, Column, Width, Height :)

Create a gray value ramp.

The operator `gen_image_gray_ramp` creates a gray value ramp according to the following equation:

$$\text{ImageGrayRamp}'(r, c) = \text{Alpha}(r - \text{Row}) + \text{Beta}(c - \text{Column}) + \text{Mean}$$

The size of the image is determined by `Width` and `Height`. The gray values are of the type *byte*. Gray values outside the valid area are clipped.

Parameter

- ▷ **ImageGrayRamp** (output_object) image $\leadsto$ *Hobject* : byte
 Created image with new image matrix.
- ▷ **Alpha** (input_control) number $\leadsto$ *real*
 Gradient in line direction.
Default Value : 1.0
Suggested values : Alpha $\in \{-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0\}$
Minimum Increment : 0.000001
Recommended Increment : -0.005
- ▷ **Beta** (input_control) number $\leadsto$ *real*
 Gradient in column direction.
Default Value : 1.0
Suggested values : Beta $\in \{-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0\}$
Minimum Increment : 0.000001
Recommended Increment : -0.005

- ▷ **Mean** (input_control) number $\leadsto$ *real*
Mean gray value.
Default Value : 128
Suggested values : Mean $\in \{0, 20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 220, 255\}$
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Row** (input_control) point.y $\leadsto$ *integer*
Line index of reference point.
Default Value : 256
Suggested values : Row $\in \{128, 256, 512, 1024\}$
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Column** (input_control) point.x $\leadsto$ *integer*
Column index of reference point.
Default Value : 256
Suggested values : Column $\in \{128, 256, 512, 1024\}$
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer*
Width of image.
Default Value : 512
Suggested values : Width $\in \{128, 256, 512, 1024\}$
Typical range of values : $1 \leq \text{Width} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : Width ≥ 1
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer*
Height of image.
Default Value : 512
Suggested values : Height $\in \{128, 256, 512, 1024\}$
Typical range of values : $1 \leq \text{Height} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : Height ≥ 1

Result

If the parameter values are correct [gen_image_gray_ramp](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

[gen_image_gray_ramp](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[moments_gray_plane](#)

Possible Successors

[paint_region](#), [reduce_domain](#), [get_image_pointer1](#), [copy_obj](#)

Alternatives

[gen_image1](#)

See also

[reduce_domain](#), [paint_gray](#)

Module

Image / region / XLD management

gen_image_proto (Image : ImageCleared : Grayval :)

Set the gray values of an image to a specified value.

`gen_image_proto` sets the gray values of the image `ImageCleared` to the value `Grayval`. The input image is not modified. It is merely used to determine the size of the output image.

Parameter

- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Image, whose gray values are to be cleared.
- ▷ **ImageCleared** (output_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Image with constant gray value.
- ▷ **Grayval** (input_control) number $\leadsto$ *real* / integer
Gray value to be used for the output image.
Default Value : 0
Suggested values : `Grayval` $\in$ {0, 1, 2, 5, 10, 16, 32, 64, 128, 253, 254, 255}

Result

`gen_image_proto` returns 2 (`H_MSG_TRUE`) if all parameters are correct. If necessary, an exception is raised.

Parallelization Information

`gen_image_proto` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`test_obj_def`

Alternatives

`set_grayval`, `paint_gray`, `gen_image_const`, `copy_image`

See also

`get_image_pointer1`

Module

Basic operators

gen_image_surface_first_order (: ImageSurface : Type, Alpha, Beta, Gamma, Row, Col, Width, Height :)

Create a curved gray surface with first order polynomial.

The operator `gen_image_surface_second_order` creates a curved gray value surface according to the following equation:

$$\text{ImageSurface}(r, c) = \text{Alpha}(r - \text{Row}) + \text{Beta}(c - \text{Col}) + \text{Gamma}$$

The size of the image is determined by `Width` and `Height`. The gray values are of the type `Type`. Gray values outside the valid area are clipped.

Parameter

- ▷ **ImageSurface** (output_object) image $\leadsto$ *Hobject* : byte / uint2 / real
Created image with new image matrix.
- ▷ **Type** (input_control) string $\leadsto$ *string*
Pixel type.
Default Value : 'byte'
List of values : `Type` $\in$ {'byte', 'uint2', 'real'}
- ▷ **Alpha** (input_control) number $\leadsto$ *real*
First order coefficient in vertical direction.
Default Value : 1.0
Suggested values : `Alpha` $\in$ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0}
Minimum Increment : 0.000001
Recommended Increment : -0.005

- ▷ **Beta** (input_control) number $\leadsto$ *real*
First order coefficient in horizontal direction.
Default Value : 1.0
Suggested values : Beta $\in \{-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0\}$
Minimum Increment : 0.000001
Recommended Increment : -0.005
- ▷ **Gamma** (input_control) number $\leadsto$ *real*
Zero order coefficient
Default Value : 1.0
Suggested values : Gamma $\in \{-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0\}$
Minimum Increment : 0.000001
Recommended Increment : -0.005
- ▷ **Row** (input_control) number $\leadsto$ *real*
line coordinate of the apex of the surface
Default Value : 256.0
Suggested values : Row $\in \{0.0, 128.0, 256.0, 512.0\}$
Minimum Increment : 0.000001
Recommended Increment : -0.005
- ▷ **Col** (input_control) number $\leadsto$ *real*
Column coordinate of the apex of the surface
Default Value : 256.0
Suggested values : Col $\in \{0.0, 128.0, 256.0, 512.0\}$
Minimum Increment : 0.000001
Recommended Increment : -0.005
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer*
Width of image.
Default Value : 512
Suggested values : Width $\in \{128, 256, 512, 1024\}$
Typical range of values : $1 \leq \text{Width} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : Width ≥ 1
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer*
Height of image.
Default Value : 512
Suggested values : Height $\in \{128, 256, 512, 1024\}$
Typical range of values : $1 \leq \text{Height} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : Height ≥ 1

Result

If the parameter values are correct [gen_image_surface_first_order](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

[gen_image_surface_first_order](#) is *reentrant*, *local*, and processed *without* parallelization.

See also

[gen_image_gray_ramp](#), [gen_image_surface_second_order](#)

Module

Image / region / XLD management

gen_image_surface_second_order (: ImageSurface : Type, Alpha, Beta, Gamma, Delta, Epsilon, Zeta, Row, Col, Width, Height :)

Create a curved gray surface with second order polynomial.

The operator [gen_image_surface_second_order](#) creates a curved gray value surface according to the following equation:

$$\text{ImageSurface}(r, c) = \text{Alpha}(r - \text{Row})^{**2} + \text{Beta}(c - \text{Col})^{**2} + \text{Gamma}(r - \text{Row}) * (c - \text{Col}) + \text{Delta}(r - \text{Row}) + \text{Epsilon}(c - \text{Col}) + \text{Zeta}$$

The size of the image is determined by **Width** and **Height**. The gray values are of the type **Type**. Gray values outside the valid area are clipped.

	Parameter
▷ ImageSurface (output_object)	image $\leadsto$ <i>Hobject</i> : byte / uint2 / real Created image with new image matrix.
▷ Type (input_control)	string $\leadsto$ <i>string</i> Pixel type. Default Value : 'byte' List of values : Type $\in$ {'byte', 'uint2', 'real'}
▷ Alpha (input_control)	number $\leadsto$ <i>real</i> Second order coefficient in vertical direction. Default Value : 1.0 Suggested values : Alpha $\in$ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0} Minimum Increment : 0.000001 Recommended Increment : -0.005
▷ Beta (input_control)	number $\leadsto$ <i>real</i> Second order coefficient in horizontal direction. Default Value : 1.0 Suggested values : Beta $\in$ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0} Minimum Increment : 0.000001 Recommended Increment : -0.005
▷ Gamma (input_control)	number $\leadsto$ <i>real</i> Mixed second order coefficient. Default Value : 1.0 Suggested values : Gamma $\in$ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0} Minimum Increment : 0.000001 Recommended Increment : -0.005
▷ Delta (input_control)	number $\leadsto$ <i>real</i> First order coefficient in vertical direction. Default Value : 1.0 Suggested values : Delta $\in$ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0} Minimum Increment : 0.000001 Recommended Increment : -0.005
▷ Epsilon (input_control)	number $\leadsto$ <i>real</i> First order coefficient in horizontal direction. Default Value : 1.0 Suggested values : Epsilon $\in$ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0} Minimum Increment : 0.000001 Recommended Increment : -0.005
▷ Zeta (input_control)	number $\leadsto$ <i>real</i> Zero order coefficient Default Value : 1.0 Suggested values : Zeta $\in$ {-2.0, -1.0, -0.5, -0.0, 0.5, 1.0, 2.0} Minimum Increment : 0.000001 Recommended Increment : -0.005
▷ Row (input_control)	number $\leadsto$ <i>real</i> line coordinate of the apex of the surface Default Value : 256.0 Suggested values : Row $\in$ {0.0, 128.0, 256.0, 512.0} Minimum Increment : 0.000001 Recommended Increment : -0.005

- ▷ **Col** (input_control) number $\leadsto$ *real*
 Column coordinate of the apex of the surface
Default Value : 256.0
Suggested values : Col $\in \{0.0, 128.0, 256.0, 512.0\}$
Minimum Increment : 0.000001
Recommended Increment : -0.005
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer*
 Width of image.
Default Value : 512
Suggested values : Width $\in \{128, 256, 512, 1024\}$
Typical range of values : $1 \leq \text{Width} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : Width ≥ 1
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer*
 Height of image.
Default Value : 512
Suggested values : Height $\in \{128, 256, 512, 1024\}$
Typical range of values : $1 \leq \text{Height} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : Height ≥ 1

Result

If the parameter values are correct [gen_image_surface_second_order](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

[gen_image_surface_second_order](#) is *reentrant*, *local*, and processed *without* parallelization.

See also

[gen_image_gray_ramp](#), [gen_image_surface_first_order](#)

Module

Image / region / XLD management

region_to_bin (Region : BinImage : ForegroundGray, BackgroundGray,
 Width, Height :)

Convert a region into a binary byte-image.

[region_to_bin](#) converts the input region given in [Region](#) into a *byte-image* and assigns a gray value of [ForegroundGray](#) to all pixels in the region. If the input region is larger than the generated image, it is clipped at the image borders. The background is set to [BackgroundGray](#).

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
 Regions to be converted.
- ▷ **BinImage** (output_object) image $\leadsto$ *Hobject* : byte
 Result image of dimension Width $\times$ Height containing the converted regions.
- ▷ **ForegroundGray** (input_control) integer $\leadsto$ *integer*
 Gray value in which the regions are displayed.
Default Value : 255
Suggested values : ForegroundGray $\in \{0, 1, 50, 100, 128, 150, 200, 254, 255\}$
Typical range of values : $0 \leq \text{ForegroundGray} \leq 255$ (lin)
Recommended Increment : 1

- ▷ **BackgroundGray** (input_control) integer $\leadsto$ integer
 Gray value in which the background is displayed.
Default Value : 0
Suggested values : BackgroundGray $\in \{0, 1, 50, 100, 128, 150, 200, 254, 255\}$
Typical range of values : $0 \leq \text{BackgroundGray} \leq 255$ (lin)
Recommended Increment : 1
- ▷ **Width** (input_control) extent.x $\leadsto$ integer
 Width of the image to be generated.
Default Value : 512
Suggested values : Width $\in \{256, 512, 1024\}$
Typical range of values : $1 \leq \text{Width} \leq 1024$ (lin)
Minimum Increment : 1
Recommended Increment : 16
Restriction : Width ≥ 1
- ▷ **Height** (input_control) extent.y $\leadsto$ integer
 Height of the image to be generated.
Default Value : 512
Suggested values : Height $\in \{256, 512, 1024\}$
Typical range of values : $1 \leq \text{Height} \leq 1024$ (lin)
Minimum Increment : 1
Recommended Increment : 16
Restriction : Height ≥ 1

Complexity

$O(2 * \text{Height} * \text{Width})$.

Result

`region_to_bin` always returns 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via `set_system('no-object-result', <Result>)` and the behavior in case of an empty input region via `set_system('empty-region-result', <Result>)`. If necessary, an exception handling is raised.

Parallelization Information

`region_to_bin` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `connection`, `regiongrowing`, `pouring`

Possible Successors

`get_grayval`

Alternatives

`region_to_label`, `paint_region`, `set_grayval`

See also

`gen_image_proto`, `paint_gray`

Module

Image / region / XLD management

region_to_label (Region : ImageLabel : Type, Width, Height :)

Convert regions to a label image.

`region_to_label` converts the input regions into a label image according to their index (1..n), i.e., the first region is painted with the gray value 1, the second the gray value 2, etc. Only positive gray values are used. For *byte-images* the index is entered modulo 256.

Regions larger than the generated image are clipped appropriately. If regions overlap the regions with the higher image are entered (i.e., they are painted in the order in which they are contained in the input regions). If so desired, the regions can be made non-overlapping by calling `expand_region`.

The background, i.e., the area not covered by any regions, is set to 0. This can be used to test in which image range no region is present.

Parameter	
▷ Region (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Regions to be converted.
▷ ImageLabel (output_object)	image $\leadsto$ <i>Hobject</i> : byte / int2 / int4 Result image of dimension Width $\times$ Height containing the converted regions.
▷ Type (input_control)	string $\leadsto$ <i>string</i> Pixel type of the result image. Default Value : 'int2' List of values : Type $\in$ {'byte', 'int2', 'int4'}
▷ Width (input_control)	extent.y $\leadsto$ <i>integer</i> Width of the image to be generated. Default Value : 512 Suggested values : Width $\in$ {64, 128, 256, 512, 1024} Typical range of values : $1 \leq \text{Width} \leq 1024$ (lin) Minimum Increment : 1 Recommended Increment : 16 Restriction : Width ≥ 1
▷ Height (input_control)	extent.x $\leadsto$ <i>integer</i> Height of the image to be generated. Default Value : 512 Suggested values : Height $\in$ {64, 128, 256, 512, 1024} Typical range of values : $1 \leq \text{Height} \leq 1024$ (lin) Minimum Increment : 1 Recommended Increment : 16 Restriction : Height ≥ 1

Complexity

$O(2 * \text{Height} * \text{Width})$.

Result

`region_to_label` always returns 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via `set_system('no_object_result', <Result>)` and the behavior in case of an empty input region via `set_system('empty_region_result', <Result>)`. If necessary, an exception handling is raised.

Parallelization Information

`region_to_label` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `expand_region`

Possible Successors

`get_grayval`, `get_image_pointer1`

Alternatives

`region_to_bin`, `paint_region`

See also

`label_to_region`

Module

Image / region / XLD management

region_to_mean (Regions, Image : ImageMean : :)

Paint regions with their average gray value.

`region_to_mean` returns an image in which the regions `Regions` are painted with their average gray value based on the image `Image`. This operator is mainly intended to visualize segmentation results.

Parameter
▷ Regions (input_object) region(-array) $\leadsto$ <i>Hobject</i> Input regions. ▷ Image (input_object) image $\leadsto$ <i>Hobject</i> : byte original gray-value image. ▷ ImageMean (output_object) image $\leadsto$ <i>Hobject</i> : byte Result image with painted regions.
Example
<pre>read_image(Image, 'fabrik') region_growing(Image, Regions, 3, 3, 6, 100) region_to_mean(Regions, Image, Disp) disp_image(Disp, WindowHandle) set_draw(WindowHandle, 'margin') set_color(WindowHandle, 'black') disp_region(Regions, WindowHandle).</pre>
Result
region_to_mean returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via set_system('no_object_result', <Result>) . If necessary, an exception is raised.
Parallelization Information
region_to_mean is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i> , <i>channel level</i>).
Possible Predecessors
regiongrowing , connection
Possible Successors
disp_image
Alternatives
paint_region , intensity
Module
Image filters

7.4 Domain

add_channels (Regions, Image : GrayRegions : :)

Add gray values to regions.

The operator **add_channels** adds the gray values from **Image** to the regions in **Regions**. All channels of **Image** are adopted. The definition domain is calculated as the average of the definition domain of the image with the region. Thus the new definition domain can be a subset of the input region. The size of the matrix is not changed.

Parameter
▷ Regions (input_object) region(-array) $\leadsto$ <i>Hobject</i> Input regions (without gray values). ▷ Image (input_object) (multichannel-)image $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Gray image for regions. ▷ GrayRegions (output_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Regions with gray values (also gray images). Number of elements : Regions = GrayRegions
Parallelization Information
add_channels is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>).

	<i>Possible Predecessors</i>	
threshold , regiongrowing , gen_circle , draw_region		
	<i>Possible Successors</i>	
threshold , regiongrowing , get_domain		
	<i>Alternatives</i>	
change_domain , reduce_domain		
	<i>See also</i>	
full_domain , get_domain , intersection		
	<i>Module</i>	
Image / region / XLD management		

change_domain (Image, NewDomain : ImageNew : :)

Change definition domain of an image.

The operator [change_domain](#) uses the indicated region as new definition domain. Unlike the operator [reduce_domain](#) it does not form the intersection of the previous definition domain. This can lead to errors particularly when the region is larger than the image matrix. The size of the matrix is not changed.

	<i>Attention</i>	
Due to running time the transferred region is not checked for consistency (i.e., whether it fits with the image matrix). Incorrect regions lead to system hang-ups during subsequent operations.		

	<i>Parameter</i>	
▷ Image (input_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf		
Input image.		
▷ NewDomain (input_object) region $\leadsto$ <i>Hobject</i>		
New definition domain.		
▷ ImageNew (output_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf		
Image with new definition domain.		

	<i>Parallelization Information</i>	
change_domain is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>).		

	<i>Possible Predecessors</i>	
get_domain		
	<i>Alternatives</i>	
reduce_domain		
	<i>See also</i>	
full_domain , get_domain , intersection		
	<i>Module</i>	
Image / region / XLD management		

full_domain (Image : ImageFull : :)

Expand the domain of an image to maximum.

The operator [full_domain](#) enters a rectangle with the edge length of the image as new definition domain. This means that all pixels of the matrix are included in further operations. Thus the same definition domain is obtained as by reading or generating an image. The size of the matrix is not changed.

Parameter
▷ Image (input_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Input image. ▷ ImageFull (output_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Image with maximum definition domain.
Parallelization Information
full_domain is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>).
Possible Predecessors
get_domain
Alternatives
change_domain , reduce_domain
See also
get_domain , gen_rectangle1
Module
Image / region / XLD management

get_domain (Image : Domain : :)

Get the domain of an image.

The operator **get_domain** returns the definition domains of all input images as a region.

Parameter
▷ Image (input_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Input images. ▷ Domain (output_object) region(-array) $\leadsto$ <i>Hobject</i> Definition domains of input images.
Parallelization Information
get_domain is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>).
Possible Successors
change_domain , reduce_domain , full_domain
See also
get_domain , change_domain , reduce_domain , full_domain
Module
Image / region / XLD management

rectangle1_domain (Image : ImageReduced : Row1, Column1, Row2, Column2 :)

Reduce the domain of an image to a rectangle.

The operator **rectangle1_domain** reduces the definition domain of the given image to the specified rectangle. The old domain of the input image is ignored. The size of the matrix is not changed.

Parameter
▷ Image (input_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Input image. ▷ ImageReduced (output_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Image with reduced definition domain.

- ▷ **Row1** (input_control) rectangle.origin.y $\leadsto$ integer
Line index of upper left corner of image area.
Default Value : 100
Suggested values : Row1 $\in$ {10, 20, 50, 100, 200, 300, 500}
Typical range of values : $0 \leq \text{Row1} \leq 1024$
- ▷ **Column1** (input_control) rectangle.origin.x $\leadsto$ integer
Column index of upper left corner of image area.
Default Value : 100
Suggested values : Column1 $\in$ {10, 20, 50, 100, 200, 300, 500}
Typical range of values : $0 \leq \text{Column1} \leq 1024$
- ▷ **Row2** (input_control) rectangle.origin.y $\leadsto$ integer
Line index of lower right corner of image area.
Default Value : 200
Suggested values : Row2 $\in$ {10, 20, 50, 100, 200, 300, 500}
Typical range of values : $0 \leq \text{Row2} \leq 1024$
- ▷ **Column2** (input_control) rectangle.origin.x $\leadsto$ integer
Column index of lower right corner of image area.
Default Value : 200
Suggested values : Column2 $\in$ {10, 20, 50, 100, 200, 300, 500}
Typical range of values : $0 \leq \text{Column2} \leq 1024$

Parallelization Information

[rectangle1_domain](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[get_domain](#)

Alternatives

[change_domain](#), [reduce_domain](#), [add_channels](#)

See also

[full_domain](#), [get_domain](#), [intersection](#)

Module

Image / region / XLD management

reduce_domain (Image, Region : ImageReduced : :)

Reduce the domain of an image.

The operator [reduce_domain](#) reduces the definition domain of the given image to the indicated region. The new definition domain is calculated as the intersection of the old definition domain with the region. Thus, the new definition domain can be a subset of the region. The size of the matrix is not changed.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Input image.
- ▷ **Region** (input_object) region $\leadsto$ *Hobject*
New definition domain.
- ▷ **ImageReduced** (output_object) image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Image with reduced definition domain.

Parallelization Information

[reduce_domain](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[get_domain](#)

Alternatives

[change_domain](#), [rectangle1_domain](#), [add_channels](#)

	See also
full_domain , get_domain , intersection	
	Module
Image / region / XLD management	

7.5 Features

area_center_gray (Regions, Image : : : Area, Row, Column)

Compute the area and center of gravity of a region in a gray value image.

[area_center_gray](#) computes the area and center of gravity of the regions [Regions](#) that have gray values which are defined by the image [Image](#). This operator is similar to [area_center](#), but in contrast to that operator, the gray values of the image are taken into account while computing the area and center of gravity.

The area A of a region R in the image with the gray values $g(r, c)$ is defined as

$$A = \sum_{(r,c) \in R} g(r, c).$$

This means that the area is defined by the volume of the gray value function $g(r, c)$. The center of gravity is defined by the first two normalized moments of the gray values $g(r, c)$, i.e., by $(m_{1,0}, m_{0,1})$, where

$$m_{p,q} = \frac{1}{A} \sum_{(r,c) \in R} r^p c^q g(r, c).$$

	Parameter
▷ Regions (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Region(s) to be examined.
▷ Image (input_object)	singlechannel-image $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real Gray value image.
▷ Area (output_control)	real(-array) $\leadsto$ <i>real</i> Gray value volume of the region.
▷ Row (output_control)	point.y(-array) $\leadsto$ <i>real</i> Row coordinate of the gray value center of gravity.
▷ Column (output_control)	point.x(-array) $\leadsto$ <i>real</i> Column coordinate of the gray value center of gravity.

	Result
area_center_gray returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during execution. If the input is empty the behavior can be set via set_system (::'no_object_result', <Result>:) . If necessary, an exception handling is raised.	

	Parallelization Information
area_center_gray is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>).	

	Possible Predecessors
threshold , regiongrowing , connection	

	Alternatives
area_center	

	See also
area_center_xld , elliptic_axis_gray	

	Module
Image filters	

cooc_feature_image (Regions, Image : : LdGray, Direction : Energy, Correlation, Homogeneity, Contrast)

Calculate a co-occurrence matrix and derive gray value features thereof.

The call of `cooc_feature_image` corresponds to the consecutive execution of the operators `gen_cooc_matrix` and `cooc_feature_matrix`. If several direction matrices of the co-occurrence matrix are to be evaluated consecutively, it is more efficient to generate the matrix via `gen_cooc_matrix` and then call the operator `cooc_feature_matrix` for the resulting matrix. The parameter `Direction` transfers the direction of the neighborhood in angle or 'mean'. In the case of 'mean' the mean value is calculated in all four directions.

Parameter	
▷ Regions (input_object)	region(-array) $\leadsto$ Hobject Region to be examined.
▷ Image (input_object)	image $\leadsto$ Hobject : byte Corresponding gray values.
▷ LdGray (input_control)	integer $\leadsto$ integer Number of gray values to be distinguished (2 ^{LdGray}).
Default Value : 6	
List of values : LdGray $\in$ {1, 2, 3, 4, 5, 6, 7, 8}	
▷ Direction (input_control)	integer $\leadsto$ integer / string Direction in which the matrix is to be calculated.
Default Value : 0	
List of values : Direction $\in$ {0, 45, 90, 135, 'mean'}	
▷ Energy (output_control)	real(-array) $\leadsto$ real Gray value energy.
▷ Correlation (output_control)	real(-array) $\leadsto$ real Correlation of gray values.
▷ Homogeneity (output_control)	real(-array) $\leadsto$ real Local homogeneity of gray values.
▷ Contrast (output_control)	real(-array) $\leadsto$ real Gray value contrast.

The operator `cooc_feature_image` returns the value 2 (H_MSG_TRUE) if an image with defined gray values (byte) is entered and the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator `set_system( : : 'no_object_result', <Result> : )`, the behavior in case of empty region is set via `set_system( : : 'empty_region_result', <Result> : )`. If necessary an exception handling is raised.

Parallelization Information
`cooc_feature_image` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors
`gen_cooc_matrix`

Alternatives
`cooc_feature_matrix`

See also
`intensity`, `min_max_gray`, `entropy_gray`, `select_gray`

Module
Image filters

cooc_feature_matrix (CoocMatrix : : : Energy, Correlation, Homogeneity, Contrast)

Calculate gray value features from a co-occurrence matrix.

The procedure calculates from a co-occurrence matrix (**CoocMatrix**) the energy (**Energy**), correlation (**Correlation**), local homogeneity (**Homogeneity**) and contrast (**Contrast**).

The operator **cooc_feature_matrix** calculates the gray value features from the part of the input matrix generated by **gen_cooc_matrix** corresponding to the direction matrix indicated by the parameters *LdGray* and *Direction* according to the following formulae:

Energy:

$$Energy = \sum_{i,j=0}^{width} c_{ij}^2$$

(Measure for image homogeneity)

Correlation:

$$Correlation = \frac{\sum_{i,j=0}^{width} (i - u_x)(j - u_y)c_{ij}}{s_x s_y}$$

(Measure for gray value dependencies)

Local homogeneity:

$$Homogeneity = \sum_{i,j=0}^{width} \frac{1}{1 + (i - j)^2} c_{ij}$$

Contrast:

$$Contrast = \sum_{i,j=0}^{width} (i - j)^2 c_{ij}$$

(Measure for the size of the intensity differences)

where

$$\begin{aligned} width &= \text{Width of } \mathbf{CoocMatrix} \\ c_{ij} &= \text{Entry of co-occurrence matrix} \\ u_x &= \sum_{i,j=0}^{width} i * c_{ij} \\ u_y &= \sum_{i,j=0}^{width} j * c_{ij} \\ s_x^2 &= \sum_{i,j=0}^{width} (i - u_x)^2 * c_{ij} \\ s_y^2 &= \sum_{i,j=0}^{width} (j - u_y)^2 * c_{ij} \end{aligned}$$

Attention

The region of the input image is disregarded.

Parameter

- ▷ **CoocMatrix** (input_object) image $\leadsto$ *Hobject*: real
Co-occurrence matrix.
- ▷ **Energy** (output_control) real $\leadsto$ *real*
Homogeneity of the gray values.
- ▷ **Correlation** (output_control) real $\leadsto$ *real*
Correlation of gray values.
- ▷ **Homogeneity** (output_control) real $\leadsto$ *real*
Local homogeneity of gray values.
- ▷ **Contrast** (output_control) real $\leadsto$ *real*
Gray value contrast.

Result

The operator **cooc_feature_matrix** returns the value 2 (H_MSG_TRUE) if an image with defined gray values is passed and the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator **set_system(::'no_object_result', <Result>:)**. If necessary an exception handling is raised.

Parallelization Information

cooc_feature_matrix is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

gen_cooc_matrix

	Alternatives
cooc_feature_image	
	See also
intensity , min_max_gray , entropy_gray , select_gray	
	Module
Image filters	

elliptic_axis_gray (Regions, Image : : : Ra, Rb, Phi)

Compute the orientation and major axes of a region in a gray value image.

The operator [elliptic_axis_gray](#) calculates the length of the axes and the orientation of the ellipse having the “same orientation” and the “aspect ratio” as the input region. Several input regions can be passed in [Regions](#) as tuples. The length of the major axis [Ra](#) and the minor axis [Rb](#) as well as the orientation of the major axis with regard to the x-axis ([Phi](#)) are determined. The angle is returned in radians. The calculation is done analogously to [elliptic_axis](#). The only difference is that in [elliptic_axis_gray](#) the gray value moments are used instead of the region moments. The gray value moments are derived from the input image [Image](#). For the definition of the gray value moments, see [area_center_gray](#).

	Parameter
▷ Regions (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Region(s) to be examined.
▷ Image (input_object)	singlechannel-image $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real Gray value image.
▷ Ra (output_control)	real(-array) $\leadsto$ <i>real</i> Major axis of the region.
▷ Rb (output_control)	real(-array) $\leadsto$ <i>real</i> Minor axis of the region.
▷ Phi (output_control)	angle.rad(-array) $\leadsto$ <i>real</i> Angle enclosed by the major axis and the x-axis.

	Result
elliptic_axis_gray returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during execution. If the input is empty the behavior can be set via set_system (::'no_object_result', <Result>:) . If necessary, an exception handling is raised.	

	Parallelization Information
elliptic_axis_gray is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).	

	Possible Predecessors
threshold , regiongrowing , connection	

	Possible Successors
gen_ellipse	

	Alternatives
elliptic_axis	

	See also
area_center_gray	

	Module
Image filters	

entropy_gray (Regions, Image : : : Entropy, Anisotropy)

Determine the entropy and anisotropy of images.

The operator `entropy_gray` creates the histogram of relative frequencies of the gray values in the input image and calculates from these frequencies the entropy and the anisotropy coefficient for each region from `Regions` according to the following formulae:

Entropy:

$$Entropy = - \sum_0^{255} rel[i] * \log_2(rel[i])$$

Anisotropy coefficient:

$$Anisotropy = \frac{\sum_0^k rel[i] * \log_2(rel[i])}{Entropy}$$

where

$rel[i]$ = Histogram of relative gray value frequencies

i = Gray value of input image ($0 \dots 255$)

k = Smallest possible gray value with $\sum_0^k rel[i] \geq 0.5$

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ *Hobject*
Regions where the features are to be determined.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte
Gray value image.
- ▷ **Entropy** (output_control) real(-array) $\leadsto$ *real*
Information content (entropy) of the gray values.
Assertion : $(0 \leq Entropy) \wedge (Entropy \leq 8)$
- ▷ **Anisotropy** (output_control) real(-array) $\leadsto$ *real*
Measure of the symmetry of gray value distribution.

Complexity

If F is the area of the region the runtime complexity is $O(F + 255)$.

Result

The operator `entropy_gray` returns the value 2 (H_MSG.TRUE) if an image with defined gray values is entered and the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator `set_system(::'no_object_result', <Result>:)`, the behavior in case of empty region is set via `set_system(::'empty_region_result', <Result>:)`. If necessary an exception handling is raised.

Parallelization Information

`entropy_gray` is *reentrant* and automatically *parallelized* (on *tuple level*).

Alternatives

`select_gray`

See also

`entropy_image`, `gray_histo`, `gray_histo_abs`, `fuzzy_entropy`, `fuzzy_perimeter`

Module

Image filters

fit_surface_first_order (Regions, Image : : Algorithm, Iterations, ClippingFactor : Alpha, Beta, Gamma)

Calculate gray value moments and approximation by a first order surface (plane).

The operator `fit_surface_first_order` calculates the gray value moments and the parameters of the approximation of the gray values by a first order surface. The calculation is done by minimizing the distance between the gray values and the surface. A first order surface is described by the following formula:

$$Image'(r, c) = Alpha(r - r\_center) + Beta(c - c\_center) + Gamma$$

`r_center` and `c_center` are the coordinates of the center of the input region. By the minimization process the parameters from `Alpha` to `Gamma` is calculated.

The algorithm used for the fitting can be selected via `Algorithm`:

`'regression'` Standard 'least squares' line fitting.

`'huber'` Weighted 'least squares' fitting, where the impact of outliers is decreased based on the approach of Huber.

`'tukey'` Weighted 'least squares' fitting, where the impact of outliers is decreased based on the approach of Tukey.

The parameter `ClippingFactor` (a scaling factor for the standard deviation) controls the amount of damping outliers: The smaller the value chosen for `ClippingFactor` the more outliers are detected. The detection of outliers is repeated. The parameter `Iterations` specifies the number of iterations. In the modus `'regression'` this value is ignored.

<i>Parameter</i>	
▷ Regions (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Regions to be checked.
▷ Image (input_object)	image $\leadsto$ <i>Hobject</i> : byte / uint2 / direction / cyclic / real Corresponding gray values.
▷ Algorithm (input_control)	string $\leadsto$ <i>string</i> Algorithm for the fitting. Default Value : 'regression' List of values : Algorithm $\in$ {'regression', 'huber', 'tukey'}
▷ Iterations (input_control)	integer $\leadsto$ <i>integer</i> Maximum number of iterations (unused for 'regression'). Default Value : 5 Restriction : Iterations ≥ 0
▷ ClippingFactor (input_control)	real $\leadsto$ <i>real</i> Clipping factor for the elimination of outliers. Default Value : 2.0 List of values : ClippingFactor $\in$ {1.0, 1.5, 2.0, 2.5, 3.0} Restriction : ClippingFactor > 0
▷ Alpha (output_control)	real(-array) $\leadsto$ <i>real</i> Parameter Alpha of the approximating surface.
▷ Beta (output_control)	real(-array) $\leadsto$ <i>real</i> Parameter Beta of the approximating surface.
▷ Gamma (output_control)	real(-array) $\leadsto$ <i>real</i> Parameter Gamma of the approximating surface.

Result

The operator `fit_surface_second_order` returns the value 2 (`H_MSG_TRUE`) if an image with the defined gray values (*byte*) is entered and the parameters are correct. If necessary an exception handling is raised.

Parallelization Information

`fit_surface_first_order` is *reentrant* and automatically *parallelized* (on *tuple level*).

See also

`moments_gray_plane`, `fit_surface_second_order`

Module

Image filters

fit_surface_second_order (Regions, Image : : Algorithm, Iterations, ClippingFactor : Alpha, Beta, Gamma, Delta, Epsilon, Zeta)

Calculate gray value moments and approximation by a second order surface.

The operator `fit_surface_second_order` calculates the gray value moments and the parameters of the approximation of the gray values by a second order surface. The calculation is done by minimizing the distance between the gray values and the surface. A second order surface is described by the following formula:

$$\text{Image}(r, c) = \text{Alpha}(r - r\_center)^2 + \text{Beta}(c - c\_center)^2 + \text{Gamma}(r - r\_center) * (c - c\_center) + \text{Delta}(r - r\_center) +$$

r_center and c_center are the coordinates of the center of the input region. By the minimization process the parameters from **Alpha** to **Zeta** is calculated.

The algorithm used for the fitting can be selected via **Algorithm**:

'regression' Standard 'least squares' fitting.

'huber' Weighted 'least squares' fitting, where the impact of outliers is decreased based on the approach of Huber.

'tukey' Weighted 'least squares' fitting, where the impact of outliers is decreased based on the approach of Tukey.

The parameter **ClippingFactor** (a scaling factor for the standard deviation) controls the amount of damping outliers: The smaller the value chosen for **ClippingFactor** the more outliers are detected. The detection of outliers is repeated. The parameter **Iterations** specifies the number of iterations. In the modus 'regression' this value is ignored.

Parameter	
▷ Regions (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Regions to be checked.
▷ Image (input_object)	image $\leadsto$ <i>Hobject</i> : byte / uint2 / direction / cyclic / real Corresponding gray values.
▷ Algorithm (input_control)	string $\leadsto$ <i>string</i> Algorithm for the fitting. Default Value : 'regression' List of values : $\text{Algorithm} \in \{\text{'regression'}, \text{'tukey'}, \text{'huber'}\}$
▷ Iterations (input_control)	integer $\leadsto$ <i>integer</i> Maximum number of iterations (unused for 'regression'). Default Value : 5 Restriction : $\text{Iterations} \geq 0$
▷ ClippingFactor (input_control)	real $\leadsto$ <i>real</i> Clipping factor for the elimination of outliers. Default Value : 2.0 List of values : $\text{ClippingFactor} \in \{1.0, 1.5, 2.0, 2.5, 3.0\}$ Restriction : $\text{ClippingFactor} > 0$
▷ Alpha (output_control)	real(-array) $\leadsto$ <i>real</i> Parameter Alpha of the approximating surface.
▷ Beta (output_control)	real(-array) $\leadsto$ <i>real</i> Parameter Beta of the approximating surface.
▷ Gamma (output_control)	real(-array) $\leadsto$ <i>real</i> Parameter Gamma of the approximating surface.
▷ Delta (output_control)	real(-array) $\leadsto$ <i>real</i> Parameter Delta of the approximating surface.
▷ Epsilon (output_control)	real(-array) $\leadsto$ <i>real</i> Parameter Epsilon of the approximating surface.
▷ Zeta (output_control)	real(-array) $\leadsto$ <i>real</i> Parameter Zeta of the approximating surface.

Result
The operator **fit_surface_second_order** returns the value 2 (H_MSG_TRUE) if an image with the defined gray values (byte) is entered and the parameters are correct. If necessary an exception handling is raised.

Parallelization Information
fit_surface_second_order is reentrant and automatically parallelized (on tuple level).

See also
moments_gray_plane, **fit_surface_first_order**

Module
Image filters

fuzzy_entropy (Regions, Image : : Apar, Cpar : Entropy)

Determine the fuzzy entropy of regions.

fuzzy_entropy calculates the fuzzy entropy of a fuzzy set. To do so, the image is regarded as a fuzzy set. The entropy then is a measure of how well the image approximates a white or black image. It is defined as follows:

$$H(X) = \frac{1}{MN \ln 2} \sum_l T_e(l) h(l)$$

where $M \times N$ is the size of the image, and $h(l)$ is the histogram of the image. Furthermore,

$$T_e(l) = -\mu(l) \ln \mu(l) - (1 - \mu(l)) \ln(1 - \mu(l))$$

Here, $u(x(m, n))$ is a fuzzy membership function defining the fuzzy set (see **fuzzy_perimeter**). The same restrictions hold as in **fuzzy_perimeter**.

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ *Hobject*
Regions for which the fuzzy entropy is to be calculated.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte
Input image containing the fuzzy membership values.
- ▷ **Apar** (input_control) integer $\leadsto$ *integer*
Start of the fuzzy function.
Default Value : 0
Suggested values : Apar $\in \{0, 5, 10, 20, 50, 100\}$
Typical range of values : $0 \leq \text{Apar} \leq 255$ (lin)
Minimum Increment : 1
Recommended Increment : 5
- ▷ **Cpar** (input_control) integer $\leadsto$ *integer*
End of the fuzzy function.
Default Value : 255
Suggested values : Cpar $\in \{50, 100, 150, 200, 220, 255\}$
Typical range of values : $0 \leq \text{Cpar} \leq 255$ (lin)
Minimum Increment : 1
Recommended Increment : 5
Restriction : Apar $\leq$ Cpar
- ▷ **Entropy** (output_control) real(-array) $\leadsto$ *real*
Fuzzy entropy of a region.

Example

```
/* To find a Fuzzy Entropy from an Image */
read_image(Image, 'affe')
fuzzy_entropy(Trans, Trans, 0, 255, Entro).
```

Result

The operator **fuzzy_entropy** returns the value 2 (H_MSG_TRUE) if the parameters are correct. Otherwise an exception is raised.

Parallelization Information

fuzzy_entropy is *reentrant* and automatically *parallelized* (on *tuple level*).

See also

fuzzy_perimeter

References

M.K. Kundu, S.K. Pal: "Automatic selection of object enhancement operator with quantitative justification based on fuzzy set theoretic measures"; Pattern Recognition Letters 11; 1990; pp. 811-829.

Module

Image filters

fuzzy_perimeter (Regions, Image : : Apar, Cpar : Perimeter)

Calculate the fuzzy perimeter of a region.

The operator **fuzzy_perimeter** is used to determine the differences of fuzzy membership between an image point and its neighbor points. The right and lower neighbor are taken into account. The fuzzy perimeter is then defined as follows:

$$p(X) = \sum_{m=1}^{M-1} \sum_{n=1}^{N-1} |\mu_X(x_{m,n}) - \mu_X(x_{m,n+1})| + \sum_{m=1}^{M-1} \sum_{n=1}^{N-1} |\mu_X(x_{m,n}) - \mu_X(x_{m+1,n})|$$

where $M \times N$ is the size of the image, and $u(x(m, n))$ is the fuzzy membership function (i.e., the input image). This implementation uses Zadeh's Standard-S function, which is defined as follows:

$$\mu_X(x) = \begin{cases} 0, & x \leq a \\ 2 \left(\frac{x-a}{c-a} \right)^2, & a < x \leq b \\ 1 - 2 \left(\frac{x-a}{c-a} \right)^2, & b < x \leq c \\ 1, & c \leq x \end{cases}$$

The parameters a , b and c obey the following restrictions: $b = \frac{a+c}{2}$ is the inflection point of the function, $\Delta b = b - a = c - b$ is the bandwidth, and for $x = b$ $\mu(x) = 0.5$ holds. In **fuzzy_perimeter**, the parameters **Apar** and **Cpar** are defined as follows: b is $\frac{\text{Apar} + \text{Cpar}}{2}$.

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ *Hobject*
Regions for which the fuzzy perimeter is to be calculated.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte
Input image containing the fuzzy membership values.
- ▷ **Apar** (input_control) integer $\leadsto$ *integer*
Start of the fuzzy function.
Default Value : 0
Suggested values : Apar $\in \{0, 5, 10, 20, 50, 100\}$
Typical range of values : $0 \leq \text{Apar} \leq 255$ (lin)
Minimum Increment : 1
Recommended Increment : 5
- ▷ **Cpar** (input_control) integer $\leadsto$ *integer*
End of the fuzzy function.
Default Value : 255
Suggested values : Cpar $\in \{50, 100, 150, 200, 220, 255\}$
Typical range of values : $0 \leq \text{Cpar} \leq 255$ (lin)
Minimum Increment : 1
Recommended Increment : 5
Restriction : Apar $\leq$ Cpar
- ▷ **Perimeter** (output_control) real(-array) $\leadsto$ *real*
Fuzzy perimeter of a region.

Example

```
/* To find a Fuzzy Entropy from an Image */
read_image(Image, 'affe')
fuzzy_perimeter(Trans, Trans, 0, 255, Per).
```

Result

The operator **fuzzy_perimeter** returns the value 2 (H_MSG_TRUE) if the parameters are correct. Otherwise an exception is raised.

Parallelization Information

fuzzy_perimeter is *reentrant* and automatically *parallelized* (on *tuple level*).

See also

[fuzzy_entropy](#)

References

M.K. Kundu, S.K. Pal: "Automatic selection of object enhancement operator with quantitative justification based on fuzzy set theoretic measures"; Pattern Recognition Letters 11; 1990; pp. 811-829.

Module

Image filters

gen_cooc_matrix (Regions, Image : Matrix : LdGray, Direction :)

Calculate the co-occurrence matrix of a region in an image.

The operator [gen_cooc_matrix](#) determines from the input regions how often the gray values i and j are located next to each other in a certain direction (0, 45, 90, 135 degrees), stores this number in the co-occurrence matrix at the locations (i, j) and (j, i) (the matrix is symmetrical), and finally scale the matrix with the number of entries. [LdGray](#) indicates the number of gray values to be distinguished (namely 2^{LdGray}).

Example ([LdGray](#) = 2, i.e. 4 gray values are distinguished):

Input image with gray values:	Co-occurrence matrix (not scaled)	
0 0 3	2 0 0 1	0 1 1 0
1 1 2	0 2 2 0	1 0 1 1
1 2 3	0 2 0 1	1 1 0 0
	1 0 1 0	0 1 0 0
	0 2 0 0	0 1 0 0
	2 2 1 0	1 2 0 1
	0 1 0 2	0 0 2 0
	0 0 2 0	0 1 0 0

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ Hobject
Region to be checked.
- ▷ **Image** (input_object) image $\leadsto$ Hobject : byte
Image providing the gray values.
- ▷ **Matrix** (output_object) image(-array) $\leadsto$ Hobject : real
Co-occurrence matrix (matrices).
- ▷ **LdGray** (input_control) integer $\leadsto$ integer
Number of gray values to be distinguished (2^{LdGray}).
Default Value : 6
List of values : $\text{LdGray} \in \{1, 2, 3, 4, 5, 6, 7, 8\}$
Typical range of values : $1 \leq \text{LdGray} \leq 256$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Direction** (input_control) integer $\leadsto$ integer
Direction of neighbor relation.
Default Value : 0
List of values : $\text{Direction} \in \{0, 45, 90, 135\}$

Result

The operator [gen_cooc_matrix](#) returns the value 2 (H_MSG_TRUE) if an image with defined gray values is entered and the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator [set_system\(::'no_object_result', <Result>:\)](#), the behavior in case of empty region is set via [set_system\(::'empty_region_result', <Result>:\)](#). If necessary an exception handling is raised.

Parallelization Information

[gen_cooc_matrix](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[draw_region](#), [gen_circle](#), [gen_ellipse](#), [gen_rectangle1](#), [gen_rectangle2](#), [threshold](#), [erosion_circle](#), [gauss_image](#), [smooth_image](#), [sub_image](#)

Alternatives

[cooc_feature_image](#)

See also

[cooc_feature_matrix](#)

Module

Image filters

gray_histo (Regions , Image : : : AbsoluteHisto , RelativeHisto)

Calculate the gray value distribution.

The operator [gray_histo](#) calculates for the image ([Image](#)) within [Regions](#) the absolute ([AbsoluteHisto](#)) and relative ([RelativeHisto](#)) histogram of the gray values.

Both histograms are tupels of 256 values, which — beginning at 0 — contain the frequencies of the individual gray values of the image.

[AbsoluteHisto](#) indicates the absolute frequencies of the gray values in integers, and [RelativeHisto](#) indicates the relative, i.e. the absolute frequencies divided by the area of the image as floating point numbers.

real-, *int2*-, *uint2*-, and *int4*-images are transformed into *byte*-images (first the largest and smallest gray value in the image are determined, and then the original gray values are mapped linearly into the area 0..255) and then processed as mentioned above. The histogram can also be returned directly as a graphic via the operators [set_paint\(::WindowHandle, 'histogram'\)](#) and [disp_image](#).

Attention

Real, int2, uint2, and int4 images are reduced to 256 gray values.

Parameter

- ▷ **Regions** (input_object) [region](#)(-array) $\leadsto$ *Hobject*
Region in which the histogram is to be calculated.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : *byte* / *cyclic* / *direction* / *int1* / *int2* / *uint2* / *int4* / *real*
Image the gray value distribution of which is to be calculated.
- ▷ **AbsoluteHisto** (output_control) histogram-array $\leadsto$ *integer*
Absolute frequencies of the gray values.
- ▷ **RelativeHisto** (output_control) histogram-array $\leadsto$ *real*
Frequencies, normalized to the area of the region.

Complexity

If F is the area of the region the runtime complexity is $O(F + 255)$.

Result

The operator [gray_histo](#) returns the value 2 ([H_MSG_TRUE](#)) if the image has defined gray values and the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator [set_system\(::'no_object_result', <Result>:\)](#), the behavior in case of empty region is set via [set_system\(::'empty_region_result', <Result>:\)](#). If necessary an exception handling is raised.

Parallelization Information

[gray_histo](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[histo_to_thresh](#), [gen_region_histo](#)

Alternatives

[min_max_gray](#), [intensity](#), [gray_histo_abs](#)

See also

[set_paint](#), [disp_image](#), [histo_2dim](#), [scale_image_max](#), [entropy_gray](#)

Module

Image filters

gray_histo_abs (Regions, Image : : Quantization : AbsoluteHisto)

Calculate the gray value distribution.

The operator `gray_histo_abs` calculates for the image (`Image`) within `Regions` the absolute (`AbsoluteHisto`) histogram of the gray values.

The parameter `Quantization` defines, how many frequencies of neighbored gray values are added for one frequency value. The resulting histogram `AbsoluteHisto` is a tuple, whose indices are mapped on the gray values of the input image `Image` and whose elements contain the frequencies of the gray values. The indices i of the frequency value are calculated from the gray values g and the quantisation q as follows:

$$i = \left\lceil \frac{g + 0.5}{q} \right\rceil \quad \text{for unsigned image types,}$$

$$i = \left\lceil \frac{g - (MIN - 0.5)}{q} \right\rceil \quad \text{for signed image types,}$$

whereas MIN denotes the minimal gray value, e.g., -128 for an `int1` image type. Therefore, the size of the tuple results from the ratio of the full domain of gray values and the quantisation, e.g. for images of `int2` in $\lceil \frac{65536}{3.0} \rceil = 21846$. The origin gray value of the signed image types `int1` resp. `int2` is mapped on the index 128 resp. 32768, negative resp. positive gray values have smaller resp. greater indices.

The histogram can also be returned directly as a graphic via the operators `set_paint (::WindowHandle, 'histogram')` and `disp_image`.

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ *Hobject*
Region in which the histogram is to be calculated.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte / cyclic / direction / int1 / int2 / uint2
Image the gray value distribution of which is to be calculated.
- ▷ **Quantization** (input_control) histogram $\leadsto$ *real* / integer
Quantization of the gray values.
Default Value : 1.0
List of values : Quantization $\in \{1.0, 2.0, 3.0, 5.0, 10.0\}$
Restriction : Quantization ≥ 1.0
- ▷ **AbsoluteHisto** (output_control) histogram-array $\leadsto$ *integer*
Absolute frequencies of the gray values.

Result

The operator `gray_histo_abs` returns the value 2 (H_MSG.TRUE) if the image has defined gray values and the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator `set_system (::'no_object_result', <Result>:)`, the behavior in case of empty region is set via `set_system (::'empty_region_result', <Result>:)`. If necessary an exception handling is raised.

Parallelization Information

`gray_histo_abs` is *reentrant* and processed *without* parallelization.

Possible Successors

`histo_to_thresh`, `gen_region_histo`

Alternatives

`min_max_gray`, `intensity`, `gray_histo`

See also

`set_paint`, `disp_image`, `histo_2dim`, `scale_image_max`, `entropy_gray`

Module

Image filters

gray_projections (Region, Image : : Mode : HorProjection, VertProjection)

Calculate horizontal and vertical gray-value projections.

gray_projections calculates the horizontal and vertical gray-value projections, i.e., the mean values in the horizontal and vertical direction of the gray values of the input image **Image** within the input region **Region**.

If **Mode** = 'simple' is selected the projection is performed in the direction of the coordinate axes of the image, i.e.:

$$\begin{aligned}\text{HorProjection}(r) &= \sum_{(r+r', c+c') \in \text{Region}} \text{Image}(r+r', c+c') \\ \text{VertProjection}(c) &= \sum_{(r+r', c+c') \in \text{Region}} \text{Image}(r+r', c+c')\end{aligned}$$

Here, (r', c') denotes the upper left corner of the smallest enclosing axis-parallel rectangle of the input region (see **smallest_rectangle1**). Hence, the horizontal projection returns a one-dimensional function that reflects the vertical gray value changes. Likewise, the vertical projection returns a function that reflects the horizontal gray value changes.

If **Mode** = 'rectangle' is selected the projection is performed in the direction of the major axes of the smallest enclosing rectangle of arbitrary orientation of the input region (see **smallest_rectangle2**). Here, the horizontal projection direction corresponds to the smaller axis, while the vertical direction corresponds to the larger axis. In this mode, all gray values within the smallest enclosing rectangle of arbitrary orientation of the input region are used to compute the projections.

Parameter	
▷ Region (input_object)	region $\leadsto$ <i>Hobject</i> Region to be processed.
▷ Image (input_object)	image $\leadsto$ <i>Hobject</i> : byte / int2 / uint2 Grayvalues for projections.
▷ Mode (input_control)	string $\leadsto$ <i>string</i> Method to compute the projections. Default Value : 'simple' List of values : Mode $\in$ {'simple', 'rectangle'}
▷ HorProjection (output_control)	real-array $\leadsto$ <i>real</i> Horizontal projection.
▷ VertProjection (output_control)	real-array $\leadsto$ <i>real</i> Vertical projection.

Parallelization Information

gray_projections is *reentrant* and processed *without* parallelization.

Module
Image filters

histo_2dim (Regions, ImageCol, ImageRow : Histo2Dim : :)

Calculate the histogram of two-channel gray value images.

The operator **histo_2dim** calculates the 2-dimensional histogram of two images within **Regions**. The gray values of channel 1 (**ImageCol**) are interpreted as row index, those of channel 2 (**ImageRow**) as column index. The gray value at one point $P(g_1, g_2)$ in the output image **Histo2Dim** indicates the frequency of the gray value combination (g_1, g_2) with g_1 indicating the line index and g_2 the column index.

Parameter	
▷ Regions (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Region in which the histogram is to be calculated.
▷ ImageCol (input_object)	image $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 Channel 1.
▷ ImageRow (input_object)	image $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 Channel 2.
▷ Histo2Dim (output_object)	image $\leadsto$ <i>Hobject</i> : int4 Histogram to be calculated.

Example

```
read_image(Image, 'affe')
texture_laws(Image, Texture, 'el', 1, 5)
draw_region(Region, WindowHandle)
histo_2dim(Region, Texture, Image, Histo2Dim)
disp_image(Histo2Dim, WindowHandle).
```

Complexity

If F is the plane of the region, the runtime complexity is $O(F + 256^2)$.

Result

The operator `histo_2dim` returns the value 2 (H_MSG_TRUE) if both images have defined gray values. The behavior in case of empty input (no input images available) is set via the operator `set_system (::'no_object_result', <Result>:)`, the behavior in case of empty region is set via `set_system (::'empty_region_result', <Result>:)`. If necessary an exception handling is raised.

Parallelization Information

`histo_2dim` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`decompose3`, `decompose2`, `draw_region`

Possible Successors

`threshold`, `class_2dim_sup`, `pouring`, `local_max`, `gray_skeleton`

Alternatives

`gray_histo`, `gray_histo_abs`

See also

`get_grayval`

Module

Image filters

intensity (Regions, Image : : : Mean, Deviation)

Calculate the mean and deviation of gray values.

The operator `intensity` calculates the mean and the deviation of the gray values in the input image within **Regions**. If R is a region, p a pixel from R with the gray value $g(p)$ and F the plane ($F = |R|$), the features are defined by:

$$\text{Mean} := \frac{\sum_{p \in R} g(p)}{F}$$

$$\text{Deviation} := \sqrt{\frac{\sum_{p \in R} (g(p) - \text{Mean})^2}{F}}$$

Attention

The calculation of **Deviation** does not follow the usual definition if the region of the image contains only one pixel. In this case 0.0 is returned.

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ *Hobject*
Regions the features of which are to be calculated.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Gray value image.
- ▷ **Mean** (output_control) real(-array) $\leadsto$ *real*
Mean gray value of a region.
- ▷ **Deviation** (output_control) real(-array) $\leadsto$ *real*
Deviation of gray values within a region.

Complexity

If F is the area of the region, the runtime complexity is $O(F)$.

Result

The operator `intensity` returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator `set_system(::'no_object_result', <Result>:)`, the behavior in case of empty region is set via `set_system(::'empty_region_result', <Result>:)`. If necessary an exception handling is raised.

Parallelization Information

`intensity` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

`threshold`

Alternatives

`select_gray`, `min_max_gray`

See also

`mean_image`, `mean_image`, `gray_histo`, `gray_histo_abs`

Module

Image filters

min_max_gray (Regions, Image : : Percent : Min, Max, Range)

Determine the minimum and maximum gray values within regions.

The operator `min_max_gray` creates the histogram of the absolute frequencies of the gray values within **Regions** in the input image **Image** (see `gray_histo`) and calculates the number of pixels **Percent** corresponding to the area of the input image. Then it goes inwards on both sides of the histogram by this number of pixels and determines the smallest and the largest gray value:

e.g.:

Area = 60, percent = 5, i.e. 3 pixels

histogram = [2,8,0,7,13,0,0,...,0,10,10,5,3,1,1]

⇒ Maximum = 255, Minimum = 0, Range = 255

`min_max_gray` returns: Maximum = 253, Minimum = 1, Range = 252

If **Percent** is set at 50, **Min** = **Max** = Median. If **Percent** is 0 no histogram is calculated in order to enhance the runtime.

Attention

In case of int2, uint2, int4- and real-images **Percent** has to be 0.

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ *Hobject*
Regions, the features of which are to be calculated.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Gray value image.
- ▷ **Percent** (input_control) number $\leadsto$ *real* / integer
Percentage below (above) the absolute maximum (minimum).
Default Value : 0
Suggested values : Percent $\in \{0, 1, 2, 5, 7, 10, 15, 20, 30, 40, 50\}$
Restriction : $(0 \leq \text{Percent}) \wedge (\text{Percent} \leq 50)$
- ▷ **Min** (output_control) real(-array) $\leadsto$ *real*
“Minimum” gray value.
- ▷ **Max** (output_control) real(-array) $\leadsto$ *real*
“Maximum” gray value.
Assertion : $\text{Max} \geq \text{Min}$
- ▷ **Range** (output_control) real(-array) $\leadsto$ *real*
Difference between Max and Min.
Assertion : $\text{Range} \geq 0$

Example

```
/* Threshold segmentation with training region: */
read_image(Image, 'fabrik')
draw_region(Region, WindowHandle)
min_max_gray(Region, Image, 5, Min, Max, _)
threshold(Bild, Seg, Min, Max)
disp_region(Seg, WindowHandle).
```

Complexity

If F is the area of the region the runtime complexity is $O(F)$ if **Percent** = 0, $O(F + 255)$ otherwise.

Result

The operator **min_max_gray** returns the value 2 (H_MSG_TRUE) if the input image has the defined gray values and the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator **set_system(::'no_object_result', <Result>:)**. The behaviour in case of an empty region is set via the operator **set_system(::'empty_region_result', <Result>:)**. If necessary an exception handling is raised.

Parallelization Information

min_max_gray is *reentrant* and processed *without* parallelization.

Possible Predecessors

draw_region, **gen_circle**, **gen_ellipse**, **gen_rectangle1**, **threshold**, **regiongrowing**

Possible Successors

threshold

Alternatives

select_gray, **intensity**

See also

gray_histo, **scale_image**, **scale_image_max**, **learn_ndim_norm**

Module

Image filters

moments_gray_plane (**Regions**, **Image** : : : **MRow**, **MCol**, **Alpha**, **Beta**, **Mean**)

Calculate gray value moments and approximation by a plane.

The operator **moments_gray_plane** calculates the gray value moments and the parameters of the approximation of the gray values by a plane. The calculation is carried out according to the following formula:

$$\mathbf{MRow} = \frac{1}{F^2} \sum_{(r,c) \in \mathbf{Regions}} (r - \bar{r})(\mathbf{Image}(r, c) - \mathbf{Mean}) \quad \mathbf{MCol} = \frac{1}{F^2} \sum_{(r,c) \in \mathbf{Regions}} (c - \bar{c})(\mathbf{Image}(r, c) - \mathbf{Mean})$$

$$\mathbf{Alpha} = \frac{\mathbf{MRow} F m_{02} - m_{11} \mathbf{MCol}}{F} m_{20} m_{02} - m_{11}^2 \quad \mathbf{Beta} = \frac{m_{20} \mathbf{MCol} F - \mathbf{MRow} F m_{11}}{m_{20} m_{02} - m_{11}^2}$$

where F is the plane, $\bar{r}$, $\bar{c}$ the center, and m_{11} , m_{20} , and m_{02} the scaled moments of **Regions**.

The parameters **Alpha**, **Beta** and **Mean** describe a plane above the region:

$$\mathbf{Image}'(r, c) = \mathbf{Alpha}(r - \bar{r}) + \mathbf{Beta}(c - \bar{c}) + \mathbf{Mean}$$

Thus **Alpha** indicates the gradient in the direction of the line axis ("down"), **Beta** the gradient in the direction of the column axis (to the "right").

Parameter	
▷ Regions (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Regions to be checked.
▷ Image (input_object)	image $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / uint2 / real Corresponding gray values.
▷ MRow (output_control)	real(-array) $\leadsto$ <i>real</i> Mixed moments along a line.
▷ MCol (output_control)	real(-array) $\leadsto$ <i>real</i> Mixed moments along a column.
▷ Alpha (output_control)	real(-array) $\leadsto$ <i>real</i> Parameter Alpha of the approximating plane.
▷ Beta (output_control)	real(-array) $\leadsto$ <i>real</i> Parameter Beta of the approximating plane.
▷ Mean (output_control)	real(-array) $\leadsto$ <i>real</i> Mean gray value.

Result

The operator `moments_gray_plane` returns the value 2 (H_MSG_TRUE) if an image with the defined gray values (*byte*) is entered and the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator `set_system(:, 'no_object_result', <Result>:)`, the behavior in case of empty region is set via `set_system(:, 'empty_region_result', <Result>:)`. If necessary an exception handling is raised.

Parallelization Information

`moments_gray_plane` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`draw_region`, `gen_circle`, `gen_ellipse`, `gen_rectangle1`, `gen_rectangle2`, `threshold`, `regiongrowing`

See also

`intensity`, `moments_region_2nd`

References

R. Haralick, L. Shapiro; "Computer and Robot Vision"; Addison-Wesley, 1992, pp 75-76

Module

Image filters

plane_deviation (Regions, Image : : : Deviation)

Calculate the deviation of the gray values from the approximating image plane.

The operator `plane_deviation` calculates the deviation of the gray values in `Image` from the approximation of the gray values through a plane. Contrary to the standard deviation in case of `intensity` slanted gray value planes also receive the value zero. The gray value plane is calculated according to `gen_image_gray_ramp`.

If F is the plane, α, β, μ the parameters of the image plane and (r', c') the center, `Deviation` is defined by:

$$\text{Deviation} = \sqrt{\frac{\sum_{(r,c) \in \text{Regions}} ((\alpha(r - r') + \beta(c - c') + \mu) - \text{Image}(r, c))^2}{F}}$$

Attention

It should be noted that the calculation of `Deviation` does not follow the usual definition. It is defined to return the value 0.0 for an image with only one pixel.

Parameter	
▷ Regions (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Regions, of which the plane deviation is to be calculated.
▷ Image (input_object)	image $\leadsto$ <i>Hobject</i> : byte / cyclic Gray value image.

- ▷ **Deviation** (output_control) real(-array) $\leadsto$ real
Deviation of the gray values within a region.

Complexity

If F is the area of the region the runtime complexity amounts to $O(F)$.

Result

The operator `plane_deviation` returns the value 2 (H_MSG_TRUE) if `Image` is of the type `byte`. The behavior in case of empty input (no input images available) is set via the operator `set_system (::'no_object_result', <Result>:)`, the behavior in case of empty region is set via `set_system (::'empty_region_result', <Result>:)`. If necessary an exception handling is raised.

Parallelization Information

`plane_deviation` is *reentrant* and automatically *parallelized* (on *tuple level*).

Alternatives

`intensity`, `gen_image_gray_ramp`, `sub_image`

See also

`moments_gray_plane`

Module

Image filters

select_gray (Regions, Image : SelectedRegions : Features, Operation, Min, Max :)

Select regions based on gray value features.

The operator `select_gray` has a number of regions (`Regions`) as input. For each of these regions the features (`Features`) are calculated. If each (`Operation` = 'and') or at least one (`Operation` = 'or') of the calculated features is within the limits determined by the parameter, the region is transferred (duplicated) into the output. The parameter `Image` contains an image which returns the gray values for calculating the features.

Condition:

$$\text{Min}_i \leq \text{Features}_i(\text{Regions}, \text{Image}) \leq \text{Max}_i$$

Possible values for `Features`:

'area'	Gray value volume of region (see <code>area_center_gray</code>)
'row'	Row index of the center of gravity (see <code>area_center_gray</code>)
'column'	Column index of the center of gravity (see <code>area_center_gray</code>)
'ra'	Major axis of equivalent ellipse (see <code>elliptic_axis_gray</code>)
'rb'	Minor axis of equivalent ellipse (see <code>elliptic_axis_gray</code>)
'phi'	Orientation of equivalent ellipse (see <code>elliptic_axis_gray</code>)
'min'	Minimum gray value (see <code>min_max_gray</code>)
'max'	Maximum gray value (see <code>min_max_gray</code>)
'mean'	Mean gray value (see <code>intensity</code>)
'deviation'	Deviation of gray values (see <code>intensity</code>)
'plane_deviation'	Deviation from the approximating plane (see <code>plane_deviation</code>)
'anisotropy'	Anisotropy (see <code>entropy_gray</code>)
'entropy'	Entropy (see <code>entropy_gray</code>)
'fuzzy_entropy'	Fuzzy entropy of region (see <code>fuzzy_entropy</code> , with a fuzzy function from Apar=0 to Cpar=255)
'fuzzy_perimeter'	Fuzzy perimeter of region (see <code>fuzzy_perimeter</code> , with a fuzzy function from Apar=0 to Cpar=255)
'moments_row'	Mixed moments along a row (see <code>moments_gray_plane</code>)
'moments_column'	Mixed moments along a column (see <code>moments_gray_plane</code>)
'alpha'	Approximating plane, parameter Alpha (see <code>moments_gray_plane</code>)
'beta'	Approximating plane, parameter Beta (see <code>moments_gray_plane</code>)

Attention

If only one feature is used the value of **Operation** is meaningless. Several features are processed in the order in which they are entered.

Parameter

- ▷ **Regions** (input_object) region-array $\leadsto$ *Hobject*
Regions to be examined.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Gray value image.
- ▷ **SelectedRegions** (output_object) region-array $\leadsto$ *Hobject*
Regions having features within the limits.
- ▷ **Features** (input_control) string(-array) $\leadsto$ *string*
Names of the features.
Default Value : 'mean'
List of values : Features $\in$ {'area', 'row', 'column', 'ra', 'rb', 'phi', 'min', 'max', 'mean', 'deviation', 'plane_deviation', 'anisotropy', 'entropy', 'fuzzy_entropy', 'fuzzy_perimeter', 'moments_row', 'moments_column', 'alpha', 'beta'}
- ▷ **Operation** (input_control) string $\leadsto$ *string*
Logical connection of features.
Default Value : 'and'
List of values : Operation $\in$ {'and', 'or'}
- ▷ **Min** (input_control) number(-array) $\leadsto$ *real* / integer
Lower limit(s) of features.
Default Value : 128.0
Suggested values : Min $\in$ {0.5, 1.0, 10.0, 20.0, 50.0, 128.0, 255.0, 1000.0}
- ▷ **Max** (input_control) number(-array) $\leadsto$ *real* / integer
Upper limit(s) of features.
Default Value : 255.0
Suggested values : Max $\in$ {0.5, 1.0, 10.0, 20.0, 50.0, 128.0, 255.0, 1000.0}

Complexity

If F is the area of the region and N the number of features the runtime complexity is $O(F * N)$.

Result

The operator **select_gray** returns the value 2 (H_MSG_TRUE) if the input image has the defined gray values and the parameters are correct. The behavior in case of empty input (no input images available) is set via the operator **set_system(::'no_object_result', <Result>:)**, the behavior in case of empty region is set via **set_system(::'empty_region_result', <Result>:)**. If necessary an exception handling is raised.

Parallelization Information

select_gray is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

connection, **mean_image**, **entropy_image**, **sobel_amp**, **median_separate**

Possible Successors

select_shape, **select_gray**, **shape_trans**, **reduce_domain**, **count_obj**

See also

deviation_image, **entropy_gray**, **intensity**, **mean_image**, **min_max_gray**, **select_obj**

Module

Image filters

shape_histo_all (Region, Image : : Feature : AbsoluteHisto,
RelativeHisto)

Determine a histogram of features along all threshold values.

The operator **shape_histo_all** carries out 255 threshold operations within **Region** with the gray values of **Image**. The entry i in the histogram corresponds to the number of connected components/holes of this image

segmented with the threshold i (**Feature** = 'connected_components', 'holes') or the mean value of the feature values of the regions segmented in this way (**Feature** = 'convexity', 'compactness', 'anisometry'), respectively. The histogram can also be displayed directly as a graphic via the operators `set_paint (::WindowHandle, 'component_histogram')` and `disp_image`.

Attention

The operator `shape_histo_all` expects a region and exactly one gray value image as input. Because of the power of this operator the runtime of `shape_histo_all` is relatively large!

Parameter

- ▷ **Region** (input_object) region $\leadsto$ *Hobject*
Region in which the features are to be examined.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject*: byte
Gray value image.
- ▷ **Feature** (input_control) string $\leadsto$ *string*
Feature to be examined.
Default Value: 'connected_components'
List of values: **Feature** $\in$ {'connected_components', 'convexity', 'compactness', 'anisometry', 'holes'}
- ▷ **AbsoluteHisto** (output_control) histogram-array $\leadsto$ *real* / integer
Absolute distribution of the feature.
- ▷ **RelativeHisto** (output_control) histogram-array $\leadsto$ *real*
Relative distribution of the feature.

Example

```
/* Simulation von shape_histo_all mit Merkmal 'connected_components': */
my_shape_histo_all(Region, Image, AbsHisto, RelHisto):
  reduce_domain(Region, Image, RegionGray)
  for(0, 255, i)
    threshold(RegionGray, Seg, i, 255)
    connect_and_holes(Seg, AbsHisto[i], _)
    clear_obj([Seg, H])
  end_for()
  eval(0, Sum)
  for(0, 255, i)
    eval(Sum+AbsHisto[i], Sum)
  end_for()
  for(0, 255, i)
    eval(AbsHisto[i]/Sum, RelHisto[i])
  end_for()
```

Complexity

If F is the area of the input region and N the mean number of connected components the runtime complexity is $O(255(F + \sqrt{F}\sqrt{N}))$.

Result

The operator `shape_histo_all` returns the value 2 (H_MSG_TRUE) if an image with the defined gray values is entered. The behavior in case of empty input (no input images) is set via the operator `set_system (::'no_object_result', <Result>:)`, the behavior in case of empty region is set via `set_system (::'empty_region_result', <Result>:)`. If necessary an exception handling is raised.

Parallelization Information

`shape_histo_all` is *reentrant* and processed *without* parallelization.

Possible Successors

`histo_to_thresh`, `threshold`, `gen_region_histo`

Alternatives

`shape_histo_point`

See also

`connection`, `convexity`, `compactness`, `connect_and_holes`, `entropy_gray`, `gray_histo`, `set_paint`, `count_obj`

Module

Image filters

shape_histo_point (Region, Image : : Feature, Row,
Column : AbsoluteHisto, RelativeHisto)

Determine a histogram of features along all threshold values.

Like [shape_histo_all](#) the operator [shape_histo_point](#) carries out 255 threshold value operations within [Region](#) with the gray values of [Image](#). Contrary to [shape_histo_all](#) only the segmented region containing the pixel ([Row](#), [Column](#)) is taken into account here. The entry i in the histogram then corresponds to the number of holes of this region segmented with the threshold i ([Feature](#) = 'holes') or the feature value of the region ([Feature](#) = 'convexity', 'compactness', 'anisometry'), respectively.

The histogram can also be displayed directly as a graphic via the operators [set_paint \(::WindowHandle, 'component_histogram'\)](#) and [disp_image](#).

Parameter

- ▷ **Region** (input_object) region $\leadsto$ *Hobject*
Region in which the features are to be examined.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte
Gray value image.
- ▷ **Feature** (input_control) string $\leadsto$ *string*
Feature to be examined.
Default Value : 'convexity'
List of values : Feature $\in$ { 'convexity', 'compactness', 'anisometry', 'holes' }
- ▷ **Row** (input_control) point.y $\leadsto$ *integer*
Row of the pixel which the region must contain.
Default Value : 256
Suggested values : Row $\in$ { 10, 50, 100, 200, 300, 400 }
- ▷ **Column** (input_control) point.x $\leadsto$ *integer*
Column of the pixel which the region must contain.
Default Value : 256
Suggested values : Column $\in$ { 10, 50, 100, 200, 300, 400 }
- ▷ **AbsoluteHisto** (output_control) histogram-array $\leadsto$ *real* / *integer*
Absolute distribution of the feature.
- ▷ **RelativeHisto** (output_control) histogram-array $\leadsto$ *real*
Relative distribution of the feature.

Result

The operator [shape_histo_point](#) returns the value 2 (H_MSG_TRUE) if an image with defined gray values is entered. The behavior in case of empty input (no input images available) is set via the operator [set_system \(::'no_object_result', <Result>:\)](#), the behavior in case of empty region is set via [set_system \(::'empty_region_result', <Result>:\)](#). If necessary an exception handling is raised.

Parallelization Information

[shape_histo_point](#) is *reentrant* and processed *without parallelization*.

Possible Predecessors

[get_mbutton](#), [area_center](#)

Possible Successors

[histo_to_thresh](#), [threshold](#), [gen_region_histo](#)

Alternatives

[shape_histo_all](#)

See also

[connection](#), [connect_and_holes](#), [convexity](#), [compactness](#), [set_paint](#)

Module

Image filters

7.6 Format

change_format (Image : ImagePart : Width, Height :)

Change image size.

The operator [change_format](#) increases or decreases the size of the input images to the indicated height or width, respectively. If the image is reduced, parts are cut off at the “right” or “lower” edge of the image, respectively. If the image is enlarged, the additional areas are set to 0. The definition domain of the new image is equal to the domain of the input image, clipped to the size of the new image. No zooming is carried out.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Input image.
- ▷ **ImagePart** (output_object) image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Image with new format.
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer*
Width of new image.
Default Value : 512
Suggested values : Width $\in \{32, 64, 128, 256, 512, 768, 1024\}$
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer*
Height of new image.
Default Value : 512
Suggested values : Height $\in \{32, 64, 128, 256, 512, 525, 1024\}$

Parallelization Information

[change_format](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

[disp_image](#)

Alternatives

[crop_part](#)

See also

[zoom_image_size](#), [zoom_image_factor](#)

Module

Image / region / XLD management

crop_domain (Image : ImagePart : :)

Cut out of defined gray values.

The operator [crop_domain](#) cuts a rectangular area from the input images. This rectangle is the smallest surrounding rectangle of the domain of the input image. The new definition domain includes all pixels of the new image. The new image matrix has the size of the rectangle.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Input image.
- ▷ **ImagePart** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Image area.

Parallelization Information

[crop_domain](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

[disp_image](#)

<i>Alternatives</i>
crop_part , crop_rectangle1 , change_format , reduce_domain
<i>See also</i>
zoom_image_size , zoom_image_factor
<i>Module</i>
Image / region / XLD management

crop_domain_rel (Image : ImagePart : Top, Left, Bottom, Right :)

Cut out an image area relative to the domain.

crop_domain_rel cuts a rectangular area from the input images. The area is determined by the surrounding rectangle of the domain of the input image. The rectangle can be influenced by the control parameters to modify at the top (**Top**), at the left (**Left**), at the bottom (**Bottom**), and at the right (**Right**). Positive values results in a smaller, negative values in a larger size. If all parameters are set to zero, the region remains unchanged.

<i>Parameter</i>
▷ Image (input_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real Input image.
▷ ImagePart (output_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real Image area.
▷ Top (input_control) integer $\leadsto$ integer Number of rows clipped at the top. Default Value : -1 Suggested values : Top $\in \{-20, -10, -5, -3, -2, -1, 0, 1, 2, 3, 4, 5, 10, 20\}$
▷ Left (input_control) integer $\leadsto$ integer Number of columns clipped at the left. Default Value : -1 Suggested values : Left $\in \{-20, -10, -5, -3, -2, -1, 0, 1, 2, 3, 4, 5, 10, 20\}$
▷ Bottom (input_control) integer $\leadsto$ integer Number of rows clipped at the bottom. Default Value : -1 Suggested values : Bottom $\in \{-20, -10, -5, -3, -2, -1, 0, 1, 2, 3, 4, 5, 10, 20\}$
▷ Right (input_control) integer $\leadsto$ integer Number of columns clipped at the right. Default Value : -1 Suggested values : Right $\in \{-20, -10, -5, -3, -2, -1, 0, 1, 2, 3, 4, 5, 10, 20\}$

<i>Result</i>
crop_domain_rel returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via set_system('no_object_result', <Result>) and the behavior in case of an empty input region via set_system('empty_region_result', <Result>) . If necessary, an exception handling is raised.

<i>Parallelization Information</i>
crop_domain_rel is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i> , <i>channel level</i>).
<i>Possible Predecessors</i>
reduce_domain , threshold , connection , regiongrowing , pouring
<i>Alternatives</i>
crop_domain , crop_rectangle1
<i>See also</i>
smallest_rectangle1 , intersection , gen_rectangle1 , clip_region
<i>Module</i>
Image / region / XLD management

crop_part (Image : ImagePart : Row, Column, Width, Height :)

Cut out a rectangular image area.

The operator **crop_part** cuts a rectangular area from the input images. The area is indicated by a rectangle (upper left corner and size). The area must be within the image. The definition domain includes all pixels of the new image. The new image matrix has the size of a rectangle.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Input image.
- ▷ **ImagePart** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Image area.
- ▷ **Row** (input_control) rectangle.origin.y $\leadsto$ *integer*
Line index of upper left corner of image area.
Default Value : 100
Suggested values : Row $\in$ {10, 20, 50, 100, 200, 300, 500}
Typical range of values : $0 \leq \text{Row} \leq 1024$
- ▷ **Column** (input_control) rectangle.origin.x $\leadsto$ *integer*
Column index of upper left corner of image area.
Default Value : 100
Suggested values : Column $\in$ {10, 20, 50, 100, 200, 300, 500}
Typical range of values : $0 \leq \text{Column} \leq 1024$
- ▷ **Width** (input_control) rectangle.extent.x $\leadsto$ *integer*
Width of new image.
Default Value : 128
Suggested values : Width $\in$ {32, 64, 128, 256, 512, 768}
Typical range of values : $0 \leq \text{Width} \leq 1024$
- ▷ **Height** (input_control) rectangle.extent.y $\leadsto$ *integer*
Height of new image.
Default Value : 128
Suggested values : Height $\in$ {32, 64, 128, 256, 512, 525}
Typical range of values : $0 \leq \text{Height} \leq 1024$

Parallelization Information

crop_part is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

disp_image

Alternatives

crop_rectangle1, **crop_domain**, **change_format**, **reduce_domain**

See also

zoom_image_size, **zoom_image_factor**

Module

Image / region / XLD management

crop_rectangle1 (Image : ImagePart : Row1, Column1, Row2, Column2 :)

Cut out a rectangular image area.

The operator **crop_rectangle1** cuts a rectangular area from the input images. The area is indicated by a rectangle (upper left and lower right corner). The area must be within the image. The definition domain includes all pixels of the new image. The new image matrix has the size of a rectangle.

Parameter	
▷ Image (input_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real	Input image.
▷ ImagePart (output_object) (multichannel-)image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real	Image area.
▷ Row1 (input_control) rectangle.origin.y $\leadsto$ <i>integer</i>	Line index of upper left corner of image area. Default Value : 100 Suggested values : Row1 $\in \{10, 20, 50, 100, 200, 300, 500\}$ Typical range of values : $0 \leq \text{Row1} \leq 1024$
▷ Column1 (input_control) rectangle.origin.x $\leadsto$ <i>integer</i>	Column index of upper left corner of image area. Default Value : 100 Suggested values : Column1 $\in \{10, 20, 50, 100, 200, 300, 500\}$ Typical range of values : $0 \leq \text{Column1} \leq 1024$
▷ Row2 (input_control) rectangle.corner.y $\leadsto$ <i>integer</i>	Line index of lower right corner of image area. Default Value : 200 Suggested values : Row2 $\in \{10, 20, 50, 100, 200, 300, 500\}$ Typical range of values : $0 \leq \text{Row2} \leq 1024$
▷ Column2 (input_control) rectangle.corner.x $\leadsto$ <i>integer</i>	Column index of lower right corner of image area. Default Value : 200 Suggested values : Column2 $\in \{10, 20, 50, 100, 200, 300, 500\}$ Typical range of values : $0 \leq \text{Column2} \leq 1024$
Parallelization Information	
crop_rectangle1 is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).	
Possible Successors	
disp_image	
Alternatives	
crop_part , crop_domain , change_format , reduce_domain	
See also	
zoom_image_size , zoom_image_factor	
Module	
Image / region / XLD management	

tile_channels (Image : TiledImage : NumColumns, TileOrder :)

Tile multiple images into a large image.

[tile_channels](#) tiles an image consisting of multiple channels into a large single-channel image. The input image [Image](#) contains *Num* images of the same size, which are stored in the individual channels. The output image [TiledImage](#) contains a single channel image, where the *Num* input channels have been tiled into [NumColumns](#) columns. In particular, this means that [tile_channels](#) cannot tile color images. For this purpose, [tile_images](#) can be used. The parameter [TileOrder](#) determines the order in which the images are copied into the output in the cases in which this is not already determined by [NumColumns](#) (i.e., if [NumColumns](#) $\neq 1$ and [NumColumns](#) $\neq \text{Num}$). If [TileOrder](#) = 'horizontal' the images are copied in the horizontal direction, i.e., the second channel of [Image](#) will be to the right of the first channel. If [TileOrder](#) = 'vertical' the images are copied in the vertical direction, i.e., the second channel of [Image](#) will be below the first channel. The domain of [TiledImage](#) is obtained by copying the domain of [Image](#) to the corresponding locations in the output image. If *Num* is not a multiple of [NumColumns](#) the output image will have undefined gray values in the lower right corner of the image. The output domain will reflect this.

Parameter

- ▷ **Image** (input_object) multichannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Input image.
- ▷ **TiledImage** (output_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Tiled output image.
- ▷ **NumColumns** (input_control) integer $\leadsto$ *integer*
Number of columns to use for the output image.
Default Value : 1
Suggested values : NumColumns $\in \{1, 2, 3, 4, 5, 6, 7\}$
Restriction : NumColumns ≥ 1
- ▷ **TileOrder** (input_control) string $\leadsto$ *string*
Order of the input images in the output image.
Default Value : 'vertical'
List of values : TileOrder $\in \{'horizontal', 'vertical'\}$

Example

```
/* Grab 5 single-channel images and stack them vertically. */
gen_rectangle1 (Image, 0, 0, Height-1, Width-1)
for I := 1 to 5 by 1
    grab_image_async (ImageGrabbed, FGHandle, -1)
    append_channel (Image, ImageGrabbed, Image)
endfor
tile_channels (Image, TiledImage, 1, 'vertical')
```

Result

tile_channels returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during execution. If the input is empty the behavior can be set via **set_system**(::'no_object_result', <Result>:). If necessary, an exception handling is raised.

Parallelization Information

tile_channels is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

append_channel

Alternatives

tile_images, **tile_images_offset**

See also

change_format, **crop_part**, **crop_rectangle1**

Module

Image / region / XLD management

tile_images (Images : TiledImage : NumColumns, TileOrder :)

Tile multiple image objects into a large image.

tile_images tiles multiple input image objects, which must contain the same number of channels, into a large image. The input image object **Images** contains *Num* images, which may be of different size. The output image **TiledImage** contains as many channels as the input images. In the output image the *Num* input images have been tiled into **NumColumns** columns. Each tile has the same size, which is determined by the maximum width and height of all input images. If an input image is smaller than the tile size it is copied to the center of the respective tile. The parameter **TileOrder** determines the order in which the images are copied into the output in the cases in which this is not already determined by **NumColumns** (i.e., if **NumColumns** $\neq 1$ and **NumColumns** $\neq$ *Num*). If **TileOrder** = 'horizontal' the images are copied in the horizontal direction, i.e., the second image of **Images** will be to the right of the first image. If **TileOrder** = 'vertical' the images are copied in the vertical direction, i.e., the second image of **Images** will be below the first image. The domain of **TiledImage** is obtained by

copying the domains of [Images](#) to the corresponding locations in the output image. If *Num* is not a multiple of [NumColumns](#) the output image will have undefined gray values in the lower right corner of the image. The output domain will reflect this.

Parameter	
▷ Images (input_object) (multichannel-)image-array $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real	Input images.
▷ TiledImage (output_object) (multichannel-)image $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real	Tiled output image.
▷ NumColumns (input_control) integer $\leadsto$ <i>integer</i>	Number of columns to use for the output image. Default Value : 1 Suggested values : NumColumns $\in \{1, 2, 3, 4, 5, 6, 7\}$ Restriction : NumColumns ≥ 1
▷ TileOrder (input_control) string $\leadsto$ <i>string</i>	Order of the input images in the output image. Default Value : 'vertical' List of values : TileOrder $\in \{'horizontal', 'vertical'\}$
Example	

```
/* Grab 5 (multi-channel) images and stack them vertically. */
gen_empty_obj (Images)
for I := 1 to 5 by 1
    grab_image_async (ImageGrabbed, FGHandle, -1)
    concat_obj (Images, ImageGrabbed, Images)
endfor
tile_images (Images, TiledImage, 1, 'vertical')
```

Result

[tile_images](#) returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during execution. If the input is empty the behavior can be set via [set_system \(:: 'no_object_result', <Result> :\)](#). If necessary, an exception handling is raised.

Parallelization Information

[tile_images](#) is *reentrant* and automatically *parallelized* (on channel level).

Possible Predecessors

[append_channel](#)

Alternatives

[tile_channels](#), [tile_images_offset](#)

See also

[change_format](#), [crop_part](#), [crop_rectangle1](#)

Module

Image / region / XLD management

tile_images_offset (Images : TiledImage : OffsetRow, OffsetCol, Row1, Col1, Row2, Col2, Width, Height :)

Tile multiple image objects into a large image with explicit positioning information.

[tile_images_offset](#) tiles multiple input image objects, which must contain the same number of channels, into a large image. The input image object [Images](#) contains *Num* images, which may be of different size. The output image [TiledImage](#) contains as many channels as the input images. The size of the output image is determined by the parameters [Width](#) and [Height](#). The position of the upper left corner of the input images in the output images is determined by the parameters [OffsetRow](#) and [OffsetCol](#). Both parameters must contain exactly *Num* values. Optionally, each input image can be cropped to an arbitrary rectangle that is smaller than the

input image. To do so, the parameters **Row1**, **Col1**, **Row2**, and **Col2** must be set accordingly. If any of these four parameters is set to **-1**, the corresponding input image is not cropped. In any case, all four parameters must contain *Num* values. If the input images are cropped the position parameters **OffsetRow** and **OffsetCol** refer to the upper left corner of the cropped image. If the input images overlap each other in the output image (while taking into account their respective domains), the image with the higher index in **Images** overwrites the image data of the image with the lower index. The domain of **TiledImage** is obtained by copying the domains of **Images** to the corresponding locations in the output image.

Attention

If the input images all have the same size and tile the output image exactly, the operator **tile_images** usually will be slightly faster.

Parameter

- ▷ **Images** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Input images.
- ▷ **TiledImage** (output_object) (multichannel-)image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Tiled output image.
- ▷ **OffsetRow** (input_control) point.y(-array) $\leadsto$ *integer*
Row coordinate of the upper left corner of the input images in the output image.
Default Value : 0
Suggested values : **OffsetRow** $\in$ {0, 50, 100, 150, 200, 250}
- ▷ **OffsetCol** (input_control) point.x(-array) $\leadsto$ *integer*
Column coordinate of the upper left corner of the input images in the output image.
Default Value : 0
Suggested values : **OffsetCol** $\in$ {0, 50, 100, 150, 200, 250}
- ▷ **Row1** (input_control) rectangle.origin.y(-array) $\leadsto$ *integer*
Row coordinate of the upper left corner of the copied part of the respective input image.
Default Value : -1
Suggested values : **Row1** $\in$ {-1, 0, 10, 20, 50, 100, 200, 300, 500}
- ▷ **Col1** (input_control) rectangle.origin.x(-array) $\leadsto$ *integer*
Column coordinate of the upper left corner of the copied part of the respective input image.
Default Value : -1
Suggested values : **Col1** $\in$ {-1, 0, 10, 20, 50, 100, 200, 300, 500}
- ▷ **Row2** (input_control) rectangle.corner.y(-array) $\leadsto$ *integer*
Row coordinate of the lower right corner of the copied part of the respective input image.
Default Value : -1
Suggested values : **Row2** $\in$ {-1, 0, 10, 20, 50, 100, 200, 300, 500}
- ▷ **Col2** (input_control) rectangle.corner.x(-array) $\leadsto$ *integer*
Column coordinate of the lower right corner of the copied part of the respective input image.
Default Value : -1
Suggested values : **Col2** $\in$ {-1, 0, 10, 20, 50, 100, 200, 300, 500}
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer*
Width of the output image.
Default Value : 512
Suggested values : **Width** $\in$ {32, 64, 128, 256, 512, 768, 1024, 2048, 4096}
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer*
Height of the output image.
Default Value : 512
Suggested values : **Height** $\in$ {32, 64, 128, 256, 512, 525, 1024, 2048, 4096}

Example

```
/* Grab 2 (multi-channel) NTSC images, crop the bottom 5 lines off */
/* of each image, the right 5 columns off of the first image, and */
/* the left five lines off of the second image, and put the cropped */
/* images side-by-side. */
gen_empty_obj (Images)
```

```

for I := 1 to 2 by 1
    grab_image_async (ImageGrabbed, FGHandle, -1)
    concat_obj (Images, ImageGrabbed, Images)
endfor
tile_images_offset (Images, TiledImage, [0,635], [0,0], [0,0],
                    [0,5], [474,474], [634,639])

```

Result

[tile_images_offset](#) returns 2 (H_MSG_TRUE) if all parameters are correct and no error occurs during execution. If the input is empty the behavior can be set via [set_system \(::'no_object_result', <Result>:\)](#). If necessary, an exception handling is raised.

Parallelization Information

[tile_images_offset](#) is *reentrant* and automatically *parallelized* (on *channel level*).

Possible Predecessors

[append_channel](#)

Alternatives

[tile_channels](#), [tile_images](#)

See also

[change_format](#), [crop_part](#), [crop_rectangle1](#)

Module

Image / region / XLD management

7.7 Framegrabber

[close_all_framegrabbers](#) (: : :)

Close all frame grabbers.

The operator [close_all_framegrabbers](#) closes all frame grabbers. It is used to cope with deadlocks resulting from damaged frame grabber handles (in that case the use of [close_framegrabber](#) is impossible).

Attention

Since all frame grabbers are closed by [close_all_framegrabbers](#) all frame grabber handles become invalid.

Result

If it is possible to close the frame grabbers the operator [close_all_framegrabbers](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

[close_all_framegrabbers](#) is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[grab_image](#)

See also

[open_framegrabber](#)

Module

Image / region / XLD management

[close_framegrabber](#) (: : FGHandle :)

Close specified frame grabber.

The operator [close_framegrabber](#) closes the frame grabber specified by [FGHandle](#). In particular possible storage space for data buffers is released and the frame grabber is made available for other processes.

Parameter
▷ FGHandle (input_control) framegrabber $\leadsto$ integer Handle of the frame grabber to be closed.
Result
If the frame grabber is open the operator close_framegrabber returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.
Parallelization Information
close_framegrabber is <i>local</i> and processed <i>completely exclusively</i> without parallelization.
Possible Predecessors
grab_image
See also
open_framegrabber
Module
Image / region / XLD management

get_framegrabber_lut (: : FGHandle : ImageRed, ImageGreen, ImageBlue)

Query frame grabber lut.

The operator [get_framegrabber_lut](#) queries the lut of the frame grabber specified by [FGHandle](#). This operation is not supported for all kinds of frame grabbers.

Parameter
▷ FGHandle (input_control) framegrabber $\leadsto$ integer Handle of the desired frame grabber.
▷ ImageRed (output_control) integer-array $\leadsto$ integer Red level of the lut entries.
▷ ImageGreen (output_control) integer-array $\leadsto$ integer Green level of the lut entries.
▷ ImageBlue (output_control) integer-array $\leadsto$ integer Blue level of the lut entries.
Result
The operator get_framegrabber_lut returns the value 2 (H_MSG_TRUE) if the frame grabber is open.
Parallelization Information
get_framegrabber_lut is <i>reentrant</i> and processed <i>without</i> parallelization.
Possible Predecessors
open_framegrabber
Possible Successors
set_framegrabber_lut
See also
set_framegrabber_lut , open_framegrabber
Module
Image / region / XLD management

get_framegrabber_param (: : FGHandle, Param : Value)

Get specific parameters for a frame grabber.

The operator [get_framegrabber_param](#) returns specific parameter values for the frame grabber specified by [FGHandle](#). The standard parameters listed below are available for all frame grabbers. Additional parameters may be supported by specific frame grabbers. A list of those can be obtained with the query 'parameter' via [info_framegrabber](#).

Standard values for [Param](#), see [open_framegrabber](#):

'name' Name of the frame grabber interface.
 'horizontal_resolution' Horizontal resolution of the frame grabber.
 'vertical_resolution' Vertical resolution of the frame grabber.
 'image_width' Width of the desired image part.
 'image_height' Height of the desired image part.
 'start_row' Row coordinate of upper left corner of desired image part.
 'start_column' Column coordinate of upper left corner of desired image part.
 'field' Selected video field or full frame.
 'bits_per_channel' Number of transferred bits per pixel and image channel.
 'color_space' Color space of resulting image.
 'gain' Amplification factor for video amplifier.
 'external_trigger' External triggering ('true' / 'false').
 'camera_type' Type of used camera (frame grabber specific).
 'device' Device the frame grabber is linked to.
 'port' Port of the frame grabber the video input is linked to.
 'line_in' Camera input line of multiplexer (optional).

Parameter

- ▷ **FGHandle** (input_control) framegrabber $\leadsto$ integer
Handle of the frame grabber to be used.
- ▷ **Param** (input_control) string(-array) $\leadsto$ string
Parameter of interest.
Default Value : 'revision'
Suggested values : $\text{Param} \in \{\text{'name'}, \text{'horizontal\_resolution'}, \text{'vertical\_resolution'}, \text{'image\_width'}, \text{'image\_height'}, \text{'start\_row'}, \text{'start\_column'}, \text{'field'}, \text{'bits\_per\_channel'}, \text{'color\_space'}, \text{'gain'}, \text{'external\_trigger'}, \text{'camera\_type'}, \text{'device'}, \text{'port'}, \text{'line\_in'}, \text{'grab\_timeout'}, \text{'volatile'}, \text{'revision'}\}$
- ▷ **Value** (output_control) string(-array) $\leadsto$ string / real / integer
Parameter value.

Result

If the frame grabber is open and the specified parameter is supported by the frame grabber, the operator [get_framegrabber_param](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

[get_framegrabber_param](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[open_framegrabber](#)

Possible Successors

[grab_image](#), [grab_region](#), [grab_image_start](#), [grab_image_async](#), [grab_region_async](#),
[close_framegrabber](#)

See also

[open_framegrabber](#), [info_framegrabber](#), [set_framegrabber_param](#)

Module

Image / region / XLD management

grab_image (: Image : FGHandle :)

Grab an image from the specified frame grabber.

The operator [grab_image](#) grabs an image via the frame grabber specified by [FGHandle](#). The desired operational mode of the frame grabber as well as a suitable image area can be adjusted via the operator [open_framegrabber](#). Additional (frame grabber specific) settings might be possible via [set_framegrabber_param](#).

Parameter	
▷ Image (output_object)	image $\leadsto$ Hobject : byte / int2 Grabbed image.
▷ FGHandle (input_control)	framegrabber $\leadsto$ integer Handle of the frame grabber to be used.
Example	
<pre>// Select a suitable frame grabber FgName info_framegrabber(FgName,'ports',Information,Val) // Choose the port P and the input line L your camera is connected to open_framegrabber(FgName,1,1,0,0,0,'default',-1,'default',-1.0, 'default','default','default',P,L,FgHandle) grab_image(Img,FgHandle) close_framegrabber(FgHandle)</pre>	
Result	
If the frame grabber is open the operator grab_image returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.	
Parallelization Information	
grab_image is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Predecessors	
open_framegrabber , grab_image_start	
Possible Successors	
grab_image , grab_image_start , grab_image_async , close_framegrabber	
See also	
open_framegrabber , info_framegrabber , set_framegrabber_param	
Module	
Image / region / XLD management	

grab_image_async (: Image : FGHandle, MaxDelay :)

Grab of an image from the specified frame grabber and start of the asynchronous grab of the next image.

The operator **grab_image_async** grabs an image via the frame grabber specified by **FGHandle** and starts the asynchronous grab of the next image. The desired operational mode of the frame grabber as well as a suitable image area can be adjusted via the operator **open_framegrabber**. Additional (frame grabber specific) settings might be possible via **set_framegrabber_param**.

The grab of the next image is finished by calling **grab_image_async** or **grab_region_async**. If more than **MaxDelay** ms have passed since the asynchronous grab was started, a new image is grabbed (the asynchronously grabbed image is considered as too old). If a negative value is assigned to **MaxDelay** this control mechanism is deactivated.

Please note that if you call the operators **grab_image** or **grab_region** after **grab_image_async**, the asynchronous grab started by **grab_image_async** is aborted and a new image is grabbed (and waited for).

Parameter	
▷ Image (output_object)	image $\leadsto$ Hobject : byte / int2 Grabbed image.
▷ FGHandle (input_control)	framegrabber $\leadsto$ integer Handle of the frame grabber to be used.
▷ MaxDelay (input_control)	number $\leadsto$ real Maximum tolerated delay between the start of the asynchronous grab and the delivery of the image [ms]. Default Value : -1.0 Suggested values : MaxDelay $\in$ {-1.0, 20.0, 33.3, 40.0, 66.6, 80.0, 99.9}

Example

```
// Select a suitable frame grabber FgName
open_framegrabber(FgName,1,1,0,0,0,0,'default',-1,'default',-1.0,
                  'default','default','default',-1,-1,FgHandle)
// grab image + start next grab
grab_image_async(Img1,FgHandle,-1.0)
// Process Img1 ...
// Finish asynchronous grab + start next grab
grab_image_async(Img2,FgHandle,-1.0)
close_framegrabber(FgHandle)
```

Result

If the frame grabber is open and supports asynchronous grabbing the operator `grab_image_async` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

`grab_image_async` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`open_framegrabber`

Possible Successors

`grab_image_async`, `grab_region_async`, `close_framegrabber`

See also

`grab_image_start`, `open_framegrabber`, `info_framegrabber`, `set_framegrabber_param`

Module

Image / region / XLD management

<code>grab_image_start</code> (: : FGHandle, MaxDelay :)

Start an asynchronous grab of an image from the specified frame grabber.

The operator `grab_image_start` starts an asynchronous grab of an image via the frame grabber specified by `FGHandle`. The desired operational mode of the frame grabber as well as a suitable image part can be adjusted via the operator `open_framegrabber`. Additional (frame grabber specific) settings might be possible via `set_framegrabber_param`.

The grab is finished via `grab_image_async` or `grab_region_async`. If one of those operators is called more than `MaxDelay` ms later, a new image is grabbed (the asynchronously grabbed image is considered as to old). If a negative value is assigned to `MaxDelay` this control mechanism is deactivated.

Please note that the operator `grab_image_start` makes sense only when used together with `grab_image_async` or `grab_region_async`. If you call the operators `grab_image` or `grab_region` instead, the asynchronous grab started by `grab_image_start` is aborted and a new image is grabbed (and waited for).

Parameter

- ▷ **FGHandle** (input_control) framegrabber $\leadsto$ *integer*
Handle of the frame grabber to be used.
- ▷ **MaxDelay** (input_control) number $\leadsto$ *real*
Maximum tolerated delay between the start of the asynchronous grab and the delivery of the image [ms].
Default Value : -1.0
Suggested values : `MaxDelay` $\in$ {20.0, 33.3, 40.0, 66.6, 80.0, 99.9}

Example

```
// Select a suitable frame grabber FgName
open_framegrabber(FgName,1,1,0,0,0,0,'default',-1,'default',-1.0,
                  'default','default','default',-1,-1,FgHandle)
```

```

grab_image(Img1,FgHandle)
// Start next grab
grab_image_start(FgHandle,-1.0)
// Process Img1 ...
// Finish asynchronous grab
grab_image_async(Img2,FgHandle,-1.0)
close_framegrabber(FgHandle)

```

Result

If the frame grabber is open and supports asynchronous grabbing the operator `grab_image_start` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

`grab_image_start` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`open_framegrabber`

Possible Successors

`grab_image_async`, `grab_region_async`, `close_framegrabber`

See also

`open_framegrabber`, `info_framegrabber`, `set_framegrabber_param`

Module

Image / region / XLD management

grab_region (: Region : FGHandle :)

Grab and segment an image from the specified frame grabber.

The operator `grab_region` grabs an image via the frame grabber specified by `FGHandle` and segments it. The desired operational mode of the frame grabber as well as a suitable image area can be adjusted via the operator `open_framegrabber`. Additional (frame grabber specific) settings might be possible via `set_framegrabber_param`.

Parameter

- ▷ **Region** (output_object) region(-array) $\leadsto$ *Hobject*
Grabbed and segmented image: Region(s).
- ▷ **FGHandle** (input_control) framegrabber $\leadsto$ *integer*
Handle of the frame grabber to be used.

Example

```

// Select a suitable frame grabber FgName
info_framegrabber(FgName,'ports',Information,Val)
// Choose the port P and the input line L your camera is connected to
open_framegrabber(FgName,1,1,0,0,0,0,'default',-1,'default',-1.0,
                  'default','default','default',P,L,FgHandle)
// grab and segment image
grab_region(Region,FgHandle)
// Process Region ...
close_framegrabber(FgHandle)

```

Result

If the frame grabber is open the operator `grab_region` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

`grab_region` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`open_framegrabber`, `grab_image_start`

Possible Successors

[grab_region](#), [grab_region_async](#), [grab_image_start](#), [grab_image](#), [grab_image_async](#), [close_framegrabber](#)

See also

[open_framegrabber](#), [info_framegrabber](#), [set_framegrabber_param](#)

Module

Image / region / XLD management

grab_region_async (: Region : FGHandle, MaxDelay :)

Grab and segment an image from the specified frame grabber and start the asynchronous grab of the next image.

The operator [grab_region_async](#) grabs an image via the frame grabber specified by [FGHandle](#), segments it, and starts the asynchronous grab of the next image. The desired operational mode of the frame grabber as well as a suitable image area can be adjusted via the operator [open_framegrabber](#). Additional (frame grabber specific) settings might be possible via [set_framegrabber_param](#).

The grab of the next image is finished via [grab_region_async](#) or via [grab_region_async](#). If more than [MaxDelay](#) ms have passed since the asynchronous grab was started, a new image is grabbed (the asynchronously grabbed image is considered as too old). If a negative value is assigned to [MaxDelay](#) this control mechanism is deactivated.

Please note that if you call the operators [grab_image](#) or [grab_region](#) after [grab_region_async](#), the asynchronous grab started by [grab_region_async](#) is aborted and a new image is grabbed (and waited for).

Parameter

- ▷ **Region** (output_object) region(-array) $\leadsto$ *Hobject*
Grabbed and segmented image: Region(s).
- ▷ **FGHandle** (input_control) framegrabber $\leadsto$ *integer*
Handle of the frame grabber to be used.
- ▷ **MaxDelay** (input_control) number $\leadsto$ *real*
Maximum tolerated delay between the start of the asynchronous grab and the delivery of the image [ms].
Default Value : -1.0
Suggested values : $\text{MaxDelay} \in \{-1.0, 20.0, 33.3, 40.0, 66.6, 80.0, 99.9\}$

Example

```
// Select a suitable frame grabber FgName
open_framegrabber(FgName,1,1,0,0,0,0,'default',-1,'default',-1.0,
                  'default','default','default',-1,-1,FgHandle)
// grab image, segment it, and start next grab
grab_region_async(Region1,FgHandle,-1.0)
// Process Region1 ...
// Finish asynchronous grab, segment this image, and start next grab
grab_region_async(Region2,FgHandle,-1.0)
close_framegrabber(FgHandle)
```

Result

If the frame grabber is open the operator [grab_region_async](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

[grab_region_async](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[open_framegrabber](#), [grab_image_start](#)

Possible Successors

[grab_region_async](#), [grab_image_async](#), [close_framegrabber](#)

See also

[open_framegrabber](#), [info_framegrabber](#), [set_framegrabber_param](#)

Module

Image / region / XLD management

info_framegrabber (:: Name, Query : Information, ValueList)

Information about the specified frame grabber.

The operator `info_framegrabber` returns information about the frame grabber `Name`. The desired information is specified via `Query`. A textual description according to the selected topic is returned in `Information`. If applicable, `ValueList` contains a list of supported values. Up to now, the following queries are possible:

'camera_types': Description of the frame grabber specific parameter 'CameraType', see `open_framegrabber`.

'defaults': Framegrabber specific default values in `ValueList`, see `open_framegrabber`.

'general': General information (in `Information`).

'info_boards': Information about currently installed boards, connected ports, etc. This data is especially useful for the auto-detect functionality of ActivVisionTools.

'parameters': List of all frame grabber specific parameters which are accessible via `set_framegrabber_param` and `get_framegrabber_param`.

'ports': Description of the ports (signal, connectors etc.) in `Information` and the port numbers in `ValueList`.

'revision': Version number of the frame grabber interface.

Please check also the directory `doc/html/manuals` for files describing selected frame grabbers.

Parameter

- ▷ **Name** (input_control) string $\leadsto$ string
Frame grabber of interest.
Default Value : 'File'
Suggested values : Name $\in$ { 'Barracuda', 'Bcam1394', 'BitFlow', 'CCi4', 'DFG-BW', 'DFG-LC', 'DirectShow', 'DqVil', 'DT315x', 'DT3162', 'File', 'Fire-i', 'FirePackage', 'FlashBus', 'FlashBusMX', 'Ginga', 'Ginga++', 'IDS', 'Inspecta', 'MatrixVision', 'MultiCam', 'Opteon', 'p3i2', 'p3i4', 'PCEye', 'PicPort', 'PicPortPro', 'PicProdigy', 'PX', 'PXC', 'PXD', 'PXR', 'Ramses1', 'TWIN', 'uEye' }
- ▷ **Query** (input_control) string $\leadsto$ string
Name of the chosen query.
Default Value : 'info_boards'
List of values : Query $\in$ { 'camera_types', 'defaults', 'general', 'info_boards', 'parameters', 'ports', 'revision' }
- ▷ **Information** (output_control) string $\leadsto$ string
Textual information (according to `Query`).
- ▷ **ValueList** (output_control) string(-array) $\leadsto$ string / integer / real
If applicable, a list of values (according to `Query`).

Example

```
// Select a suitable frame grabber FgName
info_framegrabber(FgName, 'ports', Information, Val)
// Choose the port P and the input line L your camera is connected to
open_framegrabber(FgName, 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1.0,
                  'default', 'default', 'default', P, L, FgHandle)
grab_image(Img, FgHandle)
close_framegrabber(FgHandle)
```

Result

If the parameter values are correct and the desired frame grabber is available at call time, `info_framegrabber` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information
info_framegrabber is local and processed completely exclusively without parallelization.
Possible Predecessors
open_framegrabber
Possible Successors
open_framegrabber
See also
open_framegrabber
Module
Image / region / XLD management

```
open_framegrabber ( :: Name, HorizontalResolution,
VerticalResolution, ImageWidth, ImageHeight, StartRow, StartColumn, Field,
BitsPerChannel, ColorSpace, Gain, ExternalTrigger, CameraType, Device,
Port, LineIn : FGHandle )
```

Open and configure a frame grabber.

The operator [open_framegrabber](#) opens and configures the chosen frame grabber. During this process the connection to the frame grabber is tested, the frame grabber is (normally) locked for other processes, and if necessary memory is reserved as data buffer. The image is actually grabbed via the operators [grab_image](#), [grab_region](#), [grab_image_async](#), or [grab_region_async](#). If the frame grabber is not needed anymore, it should be closed via the operator [close_framegrabber](#), releasing it for other processes. This automatically happens when a frame grabber is reopened. Some frame grabbers allow to open several instances of the same frame grabber class (up to 5).

For some parameters frame grabber specific default values can be chosen explicitly (see the parameter description below). Additional information for a specific frame grabber is available via [info_framegrabber](#). Please check also the directory *doc/html/manuals* for files describing selected frame grabbers.

The meaning of the particular parameters is as follows:

HorizontalResolution, VerticalResolution Desired resolution of the frame grabber.

ImageWidth, ImageHeight Size of the image area to be returned by [grab_image](#) etc.

StartRow, StartColumn StartRow, StartColumn upper left corner of the desired image area.

Field Desired half image ('first', 'second', or 'next') or selection of a full image.

BitsPerChannel Number of bits, which are transferred from the frame grabber board per pixel and image channel (typically 5, 8, 10, 12, or 16 Bits).

ColorSpace Specify to grab single-channel ('gray') or three-channel images ('rgb', 'yuv', ...).

Gain Amplification factor for the video amplifier (if available).

ExternalTrigger Activation of external triggering (if available).

CameraType Frame grabber specific parameter (string type) to specify the type of the used camera, which can be inquired via [info_framegrabber](#).

Device Device name of the frame grabber card.

Port Port of the frame grabber the video signal is connected to.

LineIn Camera input line of multiplexer (if available).

The operator [open_framegrabber](#) returns a handle ([FGHandle](#)) to the opened frame grabber.

Attention
Due to the multitude of supported frame grabbers a large number of parameters is necessary for the operator [open_framegrabber](#). However, not all parameters are needed for a specific frame grabber.

Parameter

- ▷ **Name** (input_control) string $\leadsto$ string
HALCON frame grabber interface, i.e. name of the corresponding DLL (Windows) or shared library (UNIX).
Default Value : 'File'
Suggested values : Name $\in$ {'Barracuda', 'Bcam1394', 'BitFlow', 'CCi4', 'DFG-BW', 'DFG-LC', 'DirectShow', 'DqVll', 'DT315x', 'DT3162', 'File', 'Fire-i', 'FirePackage', 'FlashBus', 'FlashBusMX', 'Ginga', 'Ginga++', 'IDS', 'Inspecta', 'MatrixVision', 'MultiCam', 'Opteon', 'p3i2', 'p3i4', 'PCEye', 'PicPort', 'PicPortPro', 'PicProdigy', 'PX', 'PXC', 'PXD', 'PXR', 'Ramses1', 'TWIN', 'uEye'}
- ▷ **HorizontalResolution** (input_control) extent.x $\leadsto$ integer
Desired horizontal resolution of frame grabber (1: full resolution, 2: half resolution, 4: quarter resolution).
Default Value : 1
Suggested values : HorizontalResolution $\in$ {1, 2, 4, 768, 720, 640, 384, 320, 192, 160}
- ▷ **VerticalResolution** (input_control) extent.y $\leadsto$ integer
Desired vertical resolution of frame grabber (1: full resolution, 2: half resolution, 4: quarter resolution).
Default Value : 1
Suggested values : VerticalResolution $\in$ {1, 2, 4, 576, 480, 288, 240, 144, 120}
- ▷ **ImageWidth** (input_control) rectangle.extent.x $\leadsto$ integer
Width of desired image part (0: [HorizontalResolution](#) - 2*[StartColumn](#)).
Default Value : 0
Suggested values : ImageWidth $\in$ {-1, 0}
- ▷ **ImageHeight** (input_control) rectangle.extent.y $\leadsto$ integer
Height of desired image part (0: [VerticalResolution](#) - 2*[StartRow](#)).
Default Value : 0
Suggested values : ImageHeight $\in$ {-1, 0}
- ▷ **StartRow** (input_control) rectangle.origin.y $\leadsto$ integer
Line number of upper left corner of desired image part, for [ImageHeight](#) = 0: Height of a border.
Default Value : 0
Suggested values : StartRow $\in$ {-1, 0}
- ▷ **StartColumn** (input_control) rectangle.origin.x $\leadsto$ integer
Column number of upper left corner of desired image part, for [ImageWidth](#) = 0: Width of a border.
Default Value : 0
Suggested values : StartColumn $\in$ {-1, 0}
- ▷ **Field** (input_control) string $\leadsto$ string
Desired half image or for full image.
Default Value : 'default'
Suggested values : Field $\in$ {'first', 'second', 'next', 'interlaced', 'progressive', 'default'}
- ▷ **BitsPerChannel** (input_control) integer(-array) $\leadsto$ integer
Number of transferred bits per pixel and image channel (-1: frame grabber specific default value).
Default Value : -1
Suggested values : BitsPerChannel $\in$ {5, 8, 10, 12, 16, -1}
- ▷ **ColorSpace** (input_control) string(-array) $\leadsto$ string
Specify to grab single-channel ('gray') or three-channel ('rgb', 'yuv', ...) images ('default': frame grabber specific default value, mostly 'rgb').
Default Value : 'default'
Suggested values : ColorSpace $\in$ {'gray', 'rgb', 'yuv', 'default'}
- ▷ **Gain** (input_control) real $\leadsto$ real
Amplification factor for video amplifier (-1.0: frame grabber specific default value).
Default Value : -1.0
Suggested values : Gain $\in$ {0.25, 0.5, 0.75, 1.0, -1.0}
- ▷ **ExternalTrigger** (input_control) string $\leadsto$ string
External triggering.
Default Value : 'default'
List of values : ExternalTrigger $\in$ {'true', 'false', 'default'}
- ▷ **CameraType** (input_control) string(-array) $\leadsto$ string
Type of used camera (frame grabber specific) ('default': frame grabber specific default value).
Default Value : 'default'
Suggested values : CameraType $\in$ {'ntsc', 'pal', 'auto', 'default'}

- ▷ **Device** (input_control) string(-array) $\leadsto$ *string*
Device the frame grabber is linked to ('default': frame grabber specific default value).
Default Value : 'default'
Suggested values : Device $\in \{-1, '0', '1', '2', '3', 'default'\}$
- ▷ **Port** (input_control) integer(-array) $\leadsto$ *integer*
Port of the frame grabber the video signal is linked to (-1: frame grabber specific default value).
Default Value : -1
Suggested values : Port $\in \{0, 1, 2, 3, -1\}$
- ▷ **LineIn** (input_control) integer(-array) $\leadsto$ *integer*
Camera input line of multiplexer.
Default Value : -1
Suggested values : LineIn $\in \{1, 2, 3, 4, -1\}$
- ▷ **FGHandle** (output_control) framegrabber $\leadsto$ *integer*
Handle of the opened frame grabber.

Example

```
// Select a suitable frame grabber FgName
info_framegrabber(FgName, 'ports', Information, Val)
// Choose the port P and the input line L your camera is connected to
open_framegrabber(FgName, 1, 1, 0, 0, 0, 0, 'default', -1, 'default', -1.0,
                  'default', 'default', 'default', P, L, FgHandle)
grab_image(Img, FgHandle)
close_framegrabber(FgHandle)
```

Result

If the parameter values are correct and the desired frame grabber is available at call time, [open_framegrabber](#) returns the value 2 (H.MSG.TRUE). Otherwise an exception handling is raised.

Parallelization Information

[open_framegrabber](#) is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[info_framegrabber](#)

Possible Successors

[grab_image](#), [grab_region](#), [grab_image_start](#), [grab_image_async](#), [grab_region_async](#),
[set_framegrabber_param](#)

See also

[info_framegrabber](#), [close_framegrabber](#), [grab_image](#)

Module

Image / region / XLD management

set_framegrabber_lut (: : FGHandle, ImageRed, ImageGreen,
ImageBlue :)

Set frame grabber lut.

The operator [set_framegrabber_lut](#) sets the lut of the frame grabbers spezified by [FGHandle](#). This operation is not supported for all kinds of frame grabbers.

Parameter

- ▷ **FGHandle** (input_control) framegrabber $\leadsto$ *integer*
Handle of the desired frame grabber.
- ▷ **ImageRed** (input_control) integer-array $\leadsto$ *integer*
Red level of the lut entries.
- ▷ **ImageGreen** (input_control) integer-array $\leadsto$ *integer*
Green level of the lut entries.
- ▷ **ImageBlue** (input_control) integer-array $\leadsto$ *integer*
Blue level of the lut entries.

Result

The operator `set_framegrabber_lut` returns the value 2 (H_MSG_TRUE) if the transferred lut is correct and the frame grabber is open.

Parallelization Information

`set_framegrabber_lut` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`open_framegrabber`, `get_framegrabber_lut`

Possible Successors

`grab_image`, `grab_region`, `grab_image_start`, `grab_image_async`, `grab_region_async`

See also

`get_framegrabber_lut`, `open_framegrabber`

Module

Image / region / XLD management

set_framegrabber_param (: : FGHandle, Param, Value :)

Set specific parameters for a frame grabber.

The operator `set_framegrabber_param` sets specific parameters for the frame grabber specified by `FGHandle`.

Parameter

- ▷ **FGHandle** (input_control) framegrabber $\leadsto$ integer
Handle of the frame grabber to be used.
- ▷ **Param** (input_control) string(-array) $\leadsto$ string
Parameter to be set.
Suggested values : Param $\in$ { 'continuous_grabbing', 'external_trigger', 'gain', 'image_height',
'image_width', 'port', 'start_column', 'start_row', 'trigger_signal', 'volatile' }
- ▷ **Value** (input_control) string(-array) $\leadsto$ string / real / integer
Parameter value.

Result

If the frame grabber is open and the specified parameter / parameter value is supported by the frame grabber, the operator `set_framegrabber_param` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

`set_framegrabber_param` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`open_framegrabber`

Possible Successors

`grab_image`, `grab_region`, `grab_image_start`, `grab_image_async`, `grab_region_async`,
`close_framegrabber`

See also

`open_framegrabber`, `info_framegrabber`, `get_framegrabber_param`

Module

Image / region / XLD management

7.8 Manipulation

paint_gray (ImageSource, ImageDestination : MixedImage : :)

Paint the gray values of an image into another image.

paint_gray paints the gray values of the image given in **ImageSource** into the image in **ImageDestination** and returns the resulting image in **MixedImage**. Only the gray values of the domain of **ImageSource** are copied (see **reduce_domain**).

Parameter

- ▷ **ImageSource** (input_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Input image containing the desired gray values.
- ▷ **ImageDestination** (input_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Input image to be painted over.
- ▷ **MixedImage** (output_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf
Result image.

Example

```
/* Copy a circular part of the image 'monkey' into a new image (New1): */

read_image(Image, 'monkey')
gen_circle(Circle, 200, 200, 150)
reduce_domain(Image, Circle, Mask)
/* New image with black (0) background */
gen_image_proto(Image, New1, 0.0)
/* Copy a part of the image 'monkey' into New1 */
paint_gray(Mask, New1, New1).
```

Result

paint_gray returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Parallelization Information

paint_gray is *reentrant* and processed *without* parallelization.

Possible Predecessors

read_image, **gen_image_const**, **gen_image_proto**

Alternatives

get_image_pointer1, **set_grayval**, **copy_image**

See also

paint_region

Module

Basic operators

paint_region (Region, Image : ImageResult : Grayval, Type :)

Paint regions into an image.

paint_region paints the regions given in **Region** with a constant gray value into the image given in **Image** and returns the result in **ImageResult**. These gray values can either be specified for each channel once, valid for all regions, or for each region separately. To define the latter, group the channel gray values *g* of each region and concatenate them to a tuple according to the regions' order, e.g., for a three channel image:

$$[g(\text{channel1}, \text{region1}), g(\text{channel2}, \text{region1}), g(\text{channel3}, \text{region1}), g(\text{channel1}, \text{region2}), \dots]$$

The parameter **Type** determines whether the region should be painted filled (*'fill'*) or whether only its boundary should be painted (*'margin'*).

Parameter	
▷ Region (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Regions to be painted into the input image.
▷ Image (input_object)	image $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex Image in which the regions are to be painted.
▷ ImageResult (output_object)	image $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex Image containing the result.
▷ Grayval (input_control)	number-array $\leadsto$ <i>real</i> / integer Desired gray values of the regions. Default Value : 255.0 Suggested values : $\text{Grayval} \in \{0.0, 1.0, 2.0, 5.0, 10.0, 16.0, 32.0, 64.0, 128.0, 253.0, 254.0, 255.0\}$
▷ Type (input_control)	string $\leadsto$ <i>string</i> Paint regions filled or as boundaries. Default Value : 'fill' List of values : $\text{Type} \in \{'fill', 'margin'\}$
Example	

```
/* Paint a rectangle into a new image (New1) */
gen_rectangle1(Rectangle,100.0,100.0,300.0,300.0)
/* generate a black image */
gen_image_const(New1,"byte", 768, 576)
/* paint a white rectangle */
paint_region(Rectangle,New1,New1,255.0,'fill',).
```

Result
paint_region returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via **set_system**(::'no_object_result',<Result>:). If necessary, an exception is raised.

Parallelization Information
paint_region is *reentrant* and processed *without* parallelization.

Possible Predecessors
read_image, **gen_image_const**, **gen_image_proto**, **reduce_domain**

Alternatives
set_grayval, **paint_gray**, **paint_xld**

See also
reduce_domain, **set_draw**, **gen_image_const**

Module
Basic operators

paint_xld (XLD, Image : ImageResult : Grayval :)

Paint XLD objects into an image.

paint_xld paints the XLD objects **XLD** of type contour or polygon with the constant gray values **Grayval** into each channel of the background image given in **Image** and returns the result in **ImageResult**. Open contours of XLD objects are closed and their enclosed regions are filled up. The rim of the subpixel XLD objects is painted onto the background image using anti-aliasing. Note that only objects without crossings or touching segments are painted correctly.

Grayval contains the gray values for painting the XLD objects. These gray values can either be specified for each channel once, valid for all XLD objects, or for each XLD object separately. To define the latter, group the channel gray values *g* of each XLD object and concatenate them to a tuple according to the order of the XLD objects, e.g., for a three channel image:

$$[g(channel1,xld1), g(channel2,xld1), g(channel3,xld1), g(channel1,xld2), \dots]$$

Parameter

- ▷ **XLD** (input_object) xld(-array) $\leadsto$ *Hobject*
XLD objects to be painted into the input image.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex
Image in which the xld objects are to be painted.
- ▷ **ImageResult** (output_object) image $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex
Image containing the result.
- ▷ **Grayval** (input_control) number-array $\leadsto$ *real* / integer
Desired gray value of the xld object.
Default Value : 255.0
Suggested values : $\text{Grayval} \in \{0.0, 1.0, 2.0, 5.0, 10.0, 16.0, 32.0, 64.0, 128.0, 253.0, 254.0, 255.0\}$

Example

```
/* Paint colored xld objects into a gray image */

/* read and copy image to generate a three channel image */
read_image(Image1, 'green-dot')
copy_image(Image1, Image2)
copy_image(Image1, Image3)
compose3(Image1, Image2, Image3, Image)
/* extract subpixel border */
threshold_sub_pix(Image1, Border, 128)
/* select the circle and the arrows */
select_obj(Border, circle, 14)
select_obj(Border, arrows, 16)
concat_obj(circle, arrows, green_dot)
/* paint a green circle and white arrows (to paint all
 * objects e.g. blue, pass [0,0,255] tuple for GrayVal) */
paint_xld(green_dot, Image, ImageResult, [0, 255, 0, 255, 255, 255])
```

Result

paint_xld returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via **set_system**(::'no_object_result', <Result>:). If necessary, an exception is raised.

Parallelization Information

paint_xld is *reentrant* and processed *without* parallelization.

Possible Predecessors

read_image, **gen_image_const**, **gen_image_proto**, **gen_contour_polygon_xld**,
threshold_sub_pix

Alternatives

set_grayval, **paint_gray**, **paint_region**

See also

gen_image_const

Module

Basic operators

set_grayval (Image :: Row, Column, Grayval :)

Set single gray values in an image.

set_grayval sets the gray values of the input image **Image** at the positions (**Row**,**Column**) to the values specified by **Grayval**. The number of values in **Grayval** must match the number of points passed to the operator.

Parameter	
▷ Image (input_object)	image $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real Image to be modified.
▷ Row (input_control)	point.y(-array) $\leadsto$ <i>integer</i> Row coordinates of the pixels to be modified. Default Value : 0 Suggested values : Row $\in$ {0, 10, 50, 127, 255, 511} Typical range of values : $0 \leq \text{Row}$ Restriction : $(0 \leq \text{Row}) \wedge (\text{Row} < \text{height}(\text{Image}))$
▷ Column (input_control)	point.x(-array) $\leadsto$ <i>integer</i> Column coordinates of the pixels to be modified. Default Value : 0 Suggested values : Column $\in$ {0, 10, 50, 127, 255, 511} Typical range of values : $0 \leq \text{Column}$ Restriction : $(0 \leq \text{Column}) \wedge (\text{Column} < \text{width}(\text{Image}))$
▷ Grayval (input_control)	grayval(-array) $\leadsto$ <i>real</i> / <i>integer</i> Gray values to be used. Default Value : 255.0 Suggested values : Grayval $\in$ {0.0, 1.0, 10.0, 128.0, 255.0}

set_grayval returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via **set_system**(::'no_object_result', <Result>:). If necessary, an exception is raised.

set_grayval is *reentrant* and processed *without* parallelization.

read_image, **get_image_pointer1**, **gen_image_proto**, **gen_image1**

get_image_pointer1, **paint_gray**, **paint_region**

get_grayval, **gen_image_const**, **gen_image1**, **gen_image_proto**

Basic operators

7.9 Type-Conversion

complex_to_real (ImageComplex : ImageReal, ImageImaginary : :)

Convert a complex image into two real images.

complex_to_real converts a complex image **ImageComplex** into two real images **ImageReal** and **ImageImaginary**, which contain the real and imaginary part of the complex image.

Parameter	
▷ ImageComplex (input_object)	image(-array) $\leadsto$ <i>Hobject</i> : complex Complex image.
▷ ImageReal (output_object)	image(-array) $\leadsto$ <i>Hobject</i> : real Real part.
▷ ImageImaginary (output_object)	image(-array) $\leadsto$ <i>Hobject</i> : real Imaginary part.

complex_to_real is *reentrant* and automatically *parallelized* (on tuple level).

See also

[real_to_complex](#)

Module

Image / region / XLD management

convert_image_type (Image : ImageConverted : NewType :)

Convert the type of an image.

[convert_image_type](#) converts images of an arbitrary type into an arbitrary new image type. If the conversion is done from a larger to a smaller gray value range (e.g., from 'int4' to 'byte'), too large or too small values are simply “clipped.” It is therefore advisable to adapt the range of gray values by calling [scale_image](#) before calling this operator.

Attention

If the source and destination image type are identical, no new image matrix is allocated.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 /
uint2 / int4 / real
Image whose image type is to be changed.
- ▷ **ImageConverted** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / direction /
cyclic / int1 / int2 / uint2 / int4 /
real / complex
Converted image.
- ▷ **NewType** (input_control) string $\leadsto$ *string*
Desired image type (i.e., type of the gray values).
Default Value : 'byte'
List of values : NewType $\in$ { 'int1', 'int2', 'uint2', 'int4', 'byte', 'real', 'direction', 'cyclic', 'complex' }

Result

[convert_image_type](#) returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via [set_system\('no_object_result', <Result>\)](#). If necessary, an exception is raised.

Parallelization Information

[convert_image_type](#) is *reentrant* and automatically *parallelized* (on tuple level, channel level, domain level).

Possible Predecessors

[scale_image](#)

See also

[scale_image](#), [abs_image](#)

Module

Image / region / XLD management

dvf_to_int1 (VectorField : Row, Col : :)

Convert a displacement vector field into two int1-images.

[dvf_to_int1](#) converts the displacement vector field **VectorField** into two int1-images **Row** and **Col**. The output images contain the displacements in *x*- and *y*-direction, respectively.

Parameter

- ▷ **VectorField** (input_object) image(-array) $\leadsto$ *Hobject* : dvf
Displacement vector field.
- ▷ **Row** (output_object) image(-array) $\leadsto$ *Hobject* : int1
Displacement along *x*.
- ▷ **Col** (output_object) image(-array) $\leadsto$ *Hobject* : int1
Displacement along *y*.

- ▷ **Col** (input_object) image(-array) $\leadsto$ Hobject : int1
Displacement along x.

Parallelization Information

dvf_to_int1 is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

mean_image, **optical_flow_match**, **fill_dvf**

See also

optical_flow_match

Module

Image / region / XLD management

int1_to_dvf (Row, Col : VectorField : :)

Convert two int1-images into a displacement vector field.

int1_to_dvf converts two int1-images **Row** and **Col** into a displacement vector field **VectorField**. The input images contain the displacements in *x*- and *y*-direction, respectively.

Parameter

- ▷ **Row** (input_object) image(-array) $\leadsto$ Hobject : int1
Displacement along y.
- ▷ **Col** (input_object) image(-array) $\leadsto$ Hobject : int1
Displacement along x.
- ▷ **VectorField** (output_object) image(-array) $\leadsto$ Hobject : dvf
Displacement vector field.

Parallelization Information

int1_to_dvf is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

dvf_to_int1

Possible Successors

optical_flow_match, **fill_dvf**

Module

Image / region / XLD management

real_to_complex (ImageReal, ImageImaginary : ImageComplex : :)

Convert two real images into a complex image.

real_to_complex converts two real images **ImageReal** and **ImageImaginary**, which contain the real and imaginary part of a complex image, into a complex image **ImageComplex**.

Parameter

- ▷ **ImageReal** (input_object) image(-array) $\leadsto$ Hobject : real
Real part.
- ▷ **ImageImaginary** (input_object) image(-array) $\leadsto$ Hobject : real
Imaginary part.
- ▷ **ImageComplex** (output_object) image(-array) $\leadsto$ Hobject : complex
Complex image.

Parallelization Information

real_to_complex is *reentrant* and automatically *parallelized* (on tuple level).

See also

complex_to_real

Module

Image / region / XLD management

Chapter 8

Lines

8.1 Access

```
approx_chain ( : : Row, Column, MinWidthCoord, MaxWidthCoord,
ThreshStart, ThreshEnd, ThreshStep, MinWidthSmooth, MaxWidthSmooth,
MinWidthCurve, MaxWidthCurve, Weight1, Weight2, Weight3 : ArcCenterRow,
ArcCenterCol, ArcAngle, ArcBeginRow, ArcBeginCol, LineBeginRow,
LineBeginCol, LineEndRow, LineEndCol, Order )
```

Approximate a contour by arcs and lines.

The coordinates of a curve are approximated by a row of lines and arcs. The procedure tries values from a user-definable range for certain parameters. The limits of these ranges are explicitly stated in the parameter list of the function (MinWidthCoord ... MaxWidthCoord, ThreshStart ... ThreshEnd, MinWidthSmooth ... MaxWidthSmooth, MinWidthCurve ... MaxWidthCurve). Additionally, the step width for the parameter area of the threshold value for pointed corners has to be indicated (ThreshStep). By narrowing the covered areas the runtime of the calculation can be shortened, but the result may deteriorate.

The parameters Weight1, Weight2 and Weight3 indicate whether the desired weighting is placed more on precision of the approximation, obtaining as much large segments as possible or as few small segments as possible. Thus, for (Weight1,Weight2,Weight3) (1,0,0) creates a very precise approximation and (0,1,1) an approximation with as few large segments as possible.

The result of the procedure is returned separately as arcs and lines. If one is interested in the sequence of the segments the individual resulting elements can be read successively from the resulting tuples; the sequence can be taken from the return parameter order (0: next element is next line segment, 1: next element is next arc segment).

Attention

Contours which can possibly consist of only one segment should also be examined with a threshold maximum (ThreshEnd) > 1.0, because otherwise at least one “corner point” is determined in any case.

Parameter

- ▷ **Row** (input_control) point.y $\rightsquigarrow$ *integer*
Row of the contour.
Default Value : 32
- ▷ **Column** (input_control) point.x $\rightsquigarrow$ *integer*
Column of the contour.
Default Value : 32
- ▷ **MinWidthCoord** (input_control) real $\rightsquigarrow$ *real*
Minimum width of Gauss operator for coordinate smoothing (> 0.4).
Default Value : 0.5
Suggested values : MinWidthCoord $\in$ {0.5, 0.7, 1.0, 1.2, 1.5, 1.7}
Typical range of values : $0.4 \leq \text{MinWidthCoord} \leq 3.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1

- ▷ **MaxWidthCoord** (input_control) real $\leadsto$ real
 Maximum width of Gauss operator for coordinate smoothing (> 0.4).
Default Value : 2.4
Suggested values : $\text{MaxWidthCoord} \in \{0.5, 0.7, 1.0, 1.2, 1.5, 1.7\}$
Typical range of values : $0.4 \leq \text{MaxWidthCoord} \leq 3.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
- ▷ **ThreshStart** (input_control) real $\leadsto$ real
 Minimum threshold value of the curvature for accepting a corner (relative to the largest curvature present).
Default Value : 0.3
Suggested values : $\text{ThreshStart} \in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8\}$
Typical range of values : $0.1 \leq \text{ThreshStart} \leq 0.9$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
- ▷ **ThreshEnd** (input_control) real $\leadsto$ real
 Maximum threshold value of the curvature for accepting a corner (relative to the largest curvature present).
Default Value : 0.9
Suggested values : $\text{ThreshEnd} \in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8\}$
Typical range of values : $0.1 \leq \text{ThreshEnd} \leq 0.9$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
- ▷ **ThreshStep** (input_control) real $\leadsto$ real
 Step width for threshold increase.
Default Value : 0.2
Suggested values : $\text{ThreshStep} \in \{0.3, 0.4, 0.5\}$
Typical range of values : $0.1 \leq \text{ThreshStep} \leq 0.9$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
- ▷ **MinWidthSmooth** (input_control) real $\leadsto$ real
 Minimum width of Gauss operator for smoothing the curvature function (> 0.4).
Default Value : 0.5
Suggested values : $\text{MinWidthSmooth} \in \{0.5, 0.7, 1.0, 1.2, 1.5, 1.7\}$
Typical range of values : $0.4 \leq \text{MinWidthSmooth} \leq 3.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
- ▷ **MaxWidthSmooth** (input_control) real $\leadsto$ real
 Maximum width of Gauss operator for smoothing the curvature function.
Default Value : 2.4
Suggested values : $\text{MaxWidthSmooth} \in \{0.5, 0.7, 1.0, 1.2, 1.5, 1.7\}$
Typical range of values : $0.4 \leq \text{MaxWidthSmooth} \leq 3.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
- ▷ **MinWidthCurve** (input_control) integer $\leadsto$ integer
 Minimum width of curve area for curvature determination (> 0.4).
Default Value : 2
Suggested values : $\text{MinWidthCurve} \in \{2, 5, 7\}$
Typical range of values : $1 \leq \text{MinWidthCurve} \leq 12$ (lin)
Minimum Increment : 1
Recommended Increment : 2
- ▷ **MaxWidthCurve** (input_control) integer $\leadsto$ integer
 Maximum width of curve area for curvature determination.
Default Value : 12
Suggested values : $\text{MaxWidthCurve} \in \{2, 5, 7\}$
Typical range of values : $1 \leq \text{MaxWidthCurve} \leq 20$ (lin)
Minimum Increment : 1
Recommended Increment : 2

- ▷ **Weight1** (input_control) real $\leadsto$ real
Weighting factor for approximation precision.
Default Value : 1.0
Suggested values : $\text{Weight1} \in \{0.0, 0.5, 1.0\}$
Typical range of values : $0.0 \leq \text{Weight1} \leq 1.0$ (lin)
Minimum Increment : 0.1
Recommended Increment : 0.5
- ▷ **Weight2** (input_control) real $\leadsto$ real
Weighting factor for large segments.
Default Value : 1.0
Suggested values : $\text{Weight2} \in \{0.0, 0.5, 1.0\}$
Typical range of values : $0.0 \leq \text{Weight2} \leq 1.0$ (lin)
Minimum Increment : 0.1
Recommended Increment : 0.5
- ▷ **Weight3** (input_control) real $\leadsto$ real
Weighting factor for small segments.
Default Value : 1.0
Suggested values : $\text{Weight3} \in \{0.0, 0.5, 1.0\}$
Typical range of values : $0.0 \leq \text{Weight3} \leq 1.0$ (lin)
Minimum Increment : 0.1
Recommended Increment : 0.5
- ▷ **ArcCenterRow** (output_control) arc.center.y-array $\leadsto$ integer
Row of the center of an arc.
- ▷ **ArcCenterCol** (output_control) arc.center.x-array $\leadsto$ integer
Column of the center of an arc.
- ▷ **ArcAngle** (output_control) arc.angle.rad-array $\leadsto$ real
Angle of an arc.
- ▷ **ArcBeginRow** (output_control) arc.begin.y-array $\leadsto$ integer
Row of the starting point of an arc.
- ▷ **ArcBeginCol** (output_control) arc.begin.x-array $\leadsto$ integer
Column of the starting point of an arc.
- ▷ **LineBeginRow** (output_control) line.begin.y-array $\leadsto$ integer
Row of the starting point of a line segment.
- ▷ **LineBeginCol** (output_control) line.begin.x-array $\leadsto$ integer
Column of the starting point of a line segment.
- ▷ **LineEndRow** (output_control) line.end.y-array $\leadsto$ integer
Row of the ending point of a line segment.
- ▷ **LineEndCol** (output_control) line.end.x-array $\leadsto$ integer
Column of the ending point of a line segment.
- ▷ **Order** (output_control) integer-array $\leadsto$ integer
Sequence of line (value 0) and arc segments (value 1).

Example

```

/* read edge image */
read_image(&Image, "fig1_kan");
/* construct edge region */
hysteresis_threshold(Image, &RK1, 64, 255, 40, 1);
connection(RK1, &Rand);
/* fetch chain code */
T_get_region_contour(Rand, &Rows, &Columns);
firstline = get_i(Tline, 0);
firstcol = get_i(Tcol, 0);
/* approximation with lines and circular arcs */
set_d(t1, 0.4, 0);
set_d(t2, 2.4, 0);

set_d(t3, 0.3, 0);

```

```

set_d(t4,0.9,0);

set_d(t5,0.2,0);

set_d(t6,0.4,0);
set_d(t7,2.4,0);

set_i(t8,2,0);
set_i(t9,12,0);

set_d(t10,1.0,0);
set_d(t11,1.0,0);
set_d(t12,1.0,0);

T_approx_chain(Rows,Columns,t1,t2,t3,t4,t5,t6,t7,t8,t9,t10,t11,t12,
               &Bzl,&Bzc,&Br,&Bwl,&Bwc,&Ll0,&Lc0,&Ll1,&Lc1,&order);
nob = length_tuple(Bzl);
nol = length_tuple(Ll0);
/* draw lines and arcs */
set_i(WindowHandleTuple,WindowHandle,0) ;
set_line_width(WindowHandle,4);
if (nob>0) T_disp_arc(Bzl,Bzc,Br,Bwl,Bwc);
set_line_width(WindowHandle,1);
if (nol>0) T_disp_line(WindowHandleTuple,Ll0,Lc0,Ll1,Lc1);

```

Result

The operator [approx_chain](#) returns the value 2 (H_MSG_TRUE) if the parameters are correct. Otherwise an exception is raised.

Parallelization Information

[approx_chain](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[sobel_lamp](#), [edges_image](#), [get_region_contour](#), [threshold](#), [hysteresis.threshold](#)

Possible Successors

[set_line_width](#), [disp_arc](#), [disp_line](#)

Alternatives

[get_region_polygon](#), [approx_chain_simple](#)

See also

[get_region_chain](#), [smallest_circle](#), [disp_circle](#), [disp_line](#)

Module

Region processing

approx_chain_simple (:: Row, Column : ArcCenterRow, ArcCenterCol, ArcAngle, ArcBeginRow, ArcBeginCol, LineBeginRow, LineBeginCol, LineEndRow, LineEndCol, Order)

Approximate a contour by arcs and lines.

The contour of a curve is approximated by a sequence of lines and arcs.

The result of the procedure is returned separately as arcs and lines. If one is interested in the sequence of the segments the individual resulting elements can be read successively from the resulting tuples. The sequence can be taken from the return parameter order (0: next element is next line segment, 1: next element is next arc segment).

The operator [approx_chain_simple](#) behaves similarly as [approx_chain](#) except that in the case of [approx_chain_simple](#) the missing parameters are internally allocated as follows: MinWidthCoord = 1.0, MaxWidthCoord = 3.0, ThreshStart = 0.5, ThreshEnd = 0.9, ThreshStep = 0.3, MinWidthSmooth = 1.0, MaxWidthSmooth = 3.0, MinWidthCurve = 2, MaxWidthCurve = 9, Weight1 = 1.0, Weight2 = 1.0, Weight3 = 1.0.

<i>Parameter</i>	
▷ Row (input_control)	point.y $\leadsto$ <i>integer</i> Row of the contour. Default Value : 32
▷ Column (input_control)	point.x $\leadsto$ <i>integer</i> Column of the contour. Default Value : 32
▷ ArcCenterRow (output_control)	arc.center.y-array $\leadsto$ <i>integer</i> Row of the center of an arc.
▷ ArcCenterCol (output_control)	arc.center.x-array $\leadsto$ <i>integer</i> Column of the center of an arc.
▷ ArcAngle (output_control)	arc.angle.rad-array $\leadsto$ <i>real</i> Angle of an arc.
▷ ArcBeginRow (output_control)	arc.begin.y-array $\leadsto$ <i>integer</i> Row of the starting point of an arc.
▷ ArcBeginCol (output_control)	arc.begin.x-array $\leadsto$ <i>integer</i> Column of the starting point of an arc.
▷ LineBeginRow (output_control)	line.begin.y-array $\leadsto$ <i>integer</i> Row of the starting point of a line segment.
▷ LineBeginCol (output_control)	line.begin.x-array $\leadsto$ <i>integer</i> Column of the starting point of a line segment.
▷ LineEndRow (output_control)	line.end.y-array $\leadsto$ <i>integer</i> Row of the ending point of a line segment.
▷ LineEndCol (output_control)	line.end.x-array $\leadsto$ <i>integer</i> Column of the ending point of a line segment.
▷ Order (output_control)	integer-array $\leadsto$ <i>integer</i> Sequence of line (value 0) and arc segments (value 1).
<i>Example</i>	

```

/* read edge image */
read_image(&Image,"fig1_kan");
/* construct edge region */
hysteresis_threshold(Image,&RK1,64,255,40,1);
connection(RK1,&Rand);
/* fetch chain code */
T_get_region_contour(Rand,&Rows,&Columns);
firstline = get_i(Tline,0);
firstcol = get_i(Tcol,0);
/* approximation with lines and circular arcs */
T_approx_chain_simple(Rows,Columns,
                    &Bzl,&Bzc,&Br,&Bwl,&Bwc,&Ll0,&Lc0,&Ll1,&Lc1,&order);
nob = length_tuple(Bzl);
nol = length_tuple(Ll0);
/* draw lines and arcs */
set_i(WindowHandleTuple,WindowHandle,0) ;
set_line_width(WindowHandle,4);
if (nob>0) T_disp_arc(Bzl,Bzc,Br,Bwl,Bwc);
set_line_width(WindowHandle,1);
if (nol>0) T_disp_line(WindowHandleTuple,Ll0,Lc0,Ll1,Lc1);

```

Result

The operator `approx_chain_simple` returns the value 2 (H_MSG_TRUE) if the parameters are correct. Otherwise an exception is raised.

Parallelization Information

`approx_chain_simple` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`sobel_amp`, `edges_image`, `get_region_contour`, `threshold`, `hysteresis_threshold`

Possible Successors
set_line_width , disp_arc , disp_line
Alternatives
get_region_polygon , approx_chain
See also
get_region_chain , smallest_circle , disp_circle , disp_line
Module
Region processing

8.2 Features

line_orientation (: : RowBegin, ColBegin, RowEnd, ColEnd : Phi)

Calculate the orientation of lines.

The operator [line_orientation](#) returns the orientation ($-\pi/2 < \text{Phi} \leq \pi/2$) of the given lines. If more than one line is to be treated the line and column indices can be passed as tuples. In this case [Phi](#) is, of course, also a tuple and contains the corresponding orientations.

The procedure is typically applied to model lines in order to select parallel image lines, which were found, e.g., by [detect_edge_segments](#), via the operator [select_lines](#).

Parameter
▷ RowBegin (input_control) line.begin.y(-array) $\leadsto$ real / integer Row coordinates of the starting points of the input lines. ▷ ColBegin (input_control) line.begin.x(-array) $\leadsto$ real / integer Column coordinates of the starting points of the input lines. ▷ RowEnd (input_control) line.end.y(-array) $\leadsto$ real / integer Row coordinates of the ending points of the input lines. ▷ ColEnd (input_control) line.end.x(-array) $\leadsto$ real / integer Column coordinates of the ending points of the input lines. ▷ Phi (output_control) angle.rad(-array) $\leadsto$ real Orientation of the input lines.

Result
line_orientation always returns the value 2 (H.MSG.TRUE).

Parallelization Information
line_orientation is <i>reentrant</i> and processed <i>without</i> parallelization.

Possible Predecessors
sobel_amp , edges_image , threshold , hysteresis_threshold , split_skeleton_region , split_skeleton_lines

Possible Successors
set_line_width , disp_line

Alternatives
line_position , select_lines , partition_lines

See also
line_position , select_lines , partition_lines , detect_edge_segments

Module
Region processing

line_position (: : RowBegin, ColBegin, RowEnd, ColEnd : RowCenter, ColCenter, Length, Phi)

Calculate the center of gravity, length, and orientation of a line.

The operator `line_position` returns the center (`RowCenter`, `ColCenter`), the (Euclidean) length (`Length`) and the orientation ($-\pi/2 < \text{Phi} \leq \pi/2$) of the given lines. If more than one line is to be treated the line and column indices can be passed as tuples. In this case the output parameters, of course, are also tuples.

The routine is applied, for example, to model lines in order to determine search regions for the edge detection (`detect_edge_segments`).

Parameter	
▷ RowBegin (input_control)	line.begin.y(-array) $\leadsto$ integer Row coordinates of the starting points of the input lines.
▷ ColBegin (input_control)	line.begin.x(-array) $\leadsto$ integer Column coordinates of the starting points of the input lines.
▷ RowEnd (input_control)	line.end.y(-array) $\leadsto$ integer Row coordinates of the ending points of the input lines.
▷ ColEnd (input_control)	line.end.x(-array) $\leadsto$ integer Column coordinates of the ending points of the input lines.
▷ RowCenter (output_control)	point.y(-array) $\leadsto$ real Row coordinates of the centers of gravity of the input lines.
▷ ColCenter (output_control)	point.x(-array) $\leadsto$ real Column coordinates of the centers of gravity of the input lines.
▷ Length (output_control)	real(-array) $\leadsto$ real Euclidean length of the input lines.
▷ Phi (output_control)	angle.rad(-array) $\leadsto$ real Orientation of the input lines.
Result	
<code>line_position</code> always returns the value 2 (H_MSG_TRUE).	
Parallelization Information	
<code>line_position</code> is reentrant and processed without parallelization.	
Possible Predecessors	
<code>sobel_amp</code> , <code>edges_image</code> , <code>threshold</code> , <code>hysteresis_threshold</code> , <code>split_skeleton_region</code> , <code>split_skeleton_lines</code>	
Possible Successors	
<code>set_line_width</code> , <code>disp_line</code>	
Alternatives	
<code>line_orientation</code> , <code>select_lines</code> , <code>partition_lines</code>	
See also	
<code>line_orientation</code> , <code>select_lines</code> , <code>partition_lines</code> , <code>detect_edge_segments</code>	
Module	
Region processing	

partition_lines (: : RowBeginIn, ColBeginIn, RowEndIn, ColEndIn, Feature, Operation, Min, Max : RowBeginOut, ColBeginOut, RowEndOut, ColEndOut, FailRowBOut, FailColBOut, FailRowEOut, FailColEOut)

Partition lines according to various criteria.

The operator `partition_lines` divides lines into two sets according to various criteria. For each input line the indicated features (`Feature`) are calculated. If each (`Operation` = 'and') or at least one (`Operation` = 'or') of the calculated features is within the given limits (`Min`, `Max`) the line is transferred into the first set (parameters `RowBeginOut` to `ColEndOut`), otherwise into the second set (parameters `FailRowBOut` to `FailColEOut`).

Condition: $Min_i \leq Feature_i(Line) \leq Max_i$

Possible values for `Feature`:

'length' (Euclidean) length of the line

'row' Line index of the center

'column' Column index of the center

'phi' Orientation of the line ($-\frac{\pi}{2} < \varphi \leq \frac{\pi}{2}$)

Attention

If only one feature is used the value of **Operation** is meaningless. Several features are processed according to the sequence in which they are passed.

Parameter

- ▷ **RowBeginIn** (input_control) line.begin.y-array $\leadsto$ integer
Row coordinates of the starting points of the input lines.
- ▷ **ColBeginIn** (input_control) line.begin.x-array $\leadsto$ integer
Column coordinates of the starting points of the input lines.
- ▷ **RowEndIn** (input_control) line.end.y-array $\leadsto$ integer
Row coordinates of the ending points of the input lines.
- ▷ **ColEndIn** (input_control) line.end.x-array $\leadsto$ integer
Column coordinates of the ending points of the input lines.
- ▷ **Feature** (input_control) string(-array) $\leadsto$ string
Features to be used for selection.
List of values : Feature $\in \{\text{'length', 'row', 'column', 'phi'}\}$
- ▷ **Operation** (input_control) string $\leadsto$ string
Desired combination of the features.
List of values : Operation $\in \{\text{'and', 'or'}\}$
- ▷ **Min** (input_control) string(-array) $\leadsto$ string / integer / real
Lower limits of the features or 'min'.
Default Value : 'min'
- ▷ **Max** (input_control) string(-array) $\leadsto$ string / integer / real
Upper limits of the features or 'max'.
Default Value : 'max'
- ▷ **RowBeginOut** (output_control) line.begin.y-array $\leadsto$ integer
Row coordinates of the starting points of the lines fulfilling the conditions.
- ▷ **ColBeginOut** (output_control) line.begin.x-array $\leadsto$ integer
Column coordinates of the starting points of the lines fulfilling the conditions.
- ▷ **RowEndOut** (output_control) line.end.y-array $\leadsto$ integer
Row coordinates of the ending points of the lines fulfilling the conditions.
- ▷ **ColEndOut** (output_control) line.end.x-array $\leadsto$ integer
Column coordinates of the ending points of the lines fulfilling the conditions.
- ▷ **FailRowBOut** (output_control) line.begin.y-array $\leadsto$ integer
Row coordinates of the starting points of the lines not fulfilling the conditions.
- ▷ **FailColBOut** (output_control) line.begin.x-array $\leadsto$ integer
Column coordinates of the starting points of the lines not fulfilling the conditions.
- ▷ **FailRowEOut** (output_control) line.end.y-array $\leadsto$ integer
Row coordinates of the ending points of the lines not fulfilling the conditions.
- ▷ **FailColEOut** (output_control) line.end.x-array $\leadsto$ integer
Column coordinates of the ending points of the lines not fulfilling the conditions.

Result

The operator **partition.lines** returns the value 2 (H_MSG_TRUE) if the parameter values are correct. Otherwise an exception is raised.

Parallelization Information

partition.lines is *reentrant* and processed *without* parallelization.

Possible Predecessors

sobel_amp, **edges_image**, **threshold**, **hysteresis_threshold**, **split_skeleton_region**, **split_skeleton_lines**

Possible Successors

set_line.width, **disp_line**

Alternatives

[line_orientation](#), [line_position](#), [select_lines](#), [select_lines_longest](#)

See also

[select_lines](#), [select_lines_longest](#), [detect_edge_segments](#), [select_shape](#)

Module

Region processing

select_lines (: : RowBeginIn, ColBeginIn, RowEndIn, ColEndIn, Feature, Operation, Min, Max : RowBeginOut, ColBeginOut, RowEndOut, ColEndOut)

Select lines according to various criteria.

The operator [select_lines](#) chooses lines according to various criteria. For every input line the indicated features ([Feature](#)) are calculated. If each ([Operation](#) = 'and') or at least one ([Operation](#) = 'or') of the calculated features is within the given limits ([Min](#),[Max](#)) the line is transferred into the output.

Condition: $Min_i \leq Feature_i(Line) \leq Max_i$

Possible values for [Feature](#):

'length' (Euclidean) length of the line

'row' Line index of the center

'column' Column index of the center

'phi' Orientation of the line ($-\frac{\pi}{2} < \varphi \leq \frac{\pi}{2}$)

Attention

If only one feature is used the value of [Operation](#) is meaningless. Several features are processed according to the sequence in which they are passed.

Parameter

- ▷ **RowBeginIn** (input_control) line.begin.y-array $\leadsto$ *integer*
Row coordinates of the starting points of the input lines.
- ▷ **ColBeginIn** (input_control) line.begin.x-array $\leadsto$ *integer*
Column coordinates of the starting points of the input lines.
- ▷ **RowEndIn** (input_control) line.end.y-array $\leadsto$ *integer*
Row coordinates of the ending points of the input lines.
- ▷ **ColEndIn** (input_control) line.end.x-array $\leadsto$ *integer*
Column coordinates of the ending points of the input lines.
- ▷ **Feature** (input_control) string(-array) $\leadsto$ *string*
Features to be used for selection.
Default Value : 'length'
List of values : $Feature \in \{'length', 'row', 'column', 'phi'\}$
- ▷ **Operation** (input_control) string $\leadsto$ *string*
Desired combination of the features.
Default Value : 'and'
List of values : $Operation \in \{'and', 'or'\}$
- ▷ **Min** (input_control) string(-array) $\leadsto$ *string / integer / real*
Lower limits of the features or 'min'.
Default Value : 'min'
- ▷ **Max** (input_control) string(-array) $\leadsto$ *string / integer / real*
Upper limits of the features or 'max'.
Default Value : 'max'
- ▷ **RowBeginOut** (output_control) line.begin.y-array $\leadsto$ *integer*
Row coordinates of the starting points of the output lines.
- ▷ **ColBeginOut** (output_control) line.begin.x-array $\leadsto$ *integer*
Column coordinates of the starting points of the output lines.

- ▷ **RowEndOut** (output_control) line.end.y-array $\leadsto$ *integer*
Row coordinates of the ending points of the output lines.
- ▷ **ColEndOut** (output_control) line.end.x-array $\leadsto$ *integer*
Column coordinates of the ending points of the output lines.

Result

The operator `select_lines` returns the value 2 (H_MSG_TRUE) if the parameter values are correct. Otherwise an exception is raised.

Parallelization Information

`select_lines` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`sobel_amp`, `edges_image`, `threshold`, `hysteresis_threshold`, `split_skeleton_region`,
`split_skeleton_lines`

Possible Successors

`set_line_width`, `disp_line`

Alternatives

`line_orientation`, `line_position`, `partition_lines`

See also

`partition_lines`, `select_lines_longest`, `detect_edge_segments`, `select_shape`

Module

Region processing

select_lines_longest (: : RowBeginIn, ColBeginIn, RowEndIn, ColEndIn,
Num : RowBeginOut, ColBeginOut, RowEndOut, ColEndOut)

Select the longest input lines.

The operator `select_lines_longest` selects the `Num` longest input lines from the input lines described by the tuples `RowBeginIn`, `ColBeginIn`, `RowEndIn` and `ColEndIn`.

Parameter

- ▷ **RowBeginIn** (input_control) line.begin.y-array $\leadsto$ *integer*
Row coordinates of the starting points of the input lines.
- ▷ **ColBeginIn** (input_control) line.begin.x-array $\leadsto$ *integer*
Column coordinates of the starting points of the input lines.
- ▷ **RowEndIn** (input_control) line.end.y-array $\leadsto$ *integer*
Row coordinates of the ending points of the input lines.
- ▷ **ColEndIn** (input_control) line.end.x-array $\leadsto$ *integer*
Column coordinates of the ending points of the input lines.
- ▷ **Num** (input_control) integer $\leadsto$ *integer*
(Maximum) desired number of output lines.
Default Value : 10
- ▷ **RowBeginOut** (output_control) line.begin.y-array $\leadsto$ *integer*
Row coordinates of the starting points of the output lines.
- ▷ **ColBeginOut** (output_control) line.begin.x-array $\leadsto$ *integer*
Column coordinates of the starting points of the output lines.
- ▷ **RowEndOut** (output_control) line.end.y-array $\leadsto$ *integer*
Row coordinates of the ending points of the output lines.
- ▷ **ColEndOut** (output_control) line.end.x-array $\leadsto$ *integer*
Column coordinates of the ending points of the output lines.

Result

The operator `select_lines_longest` returns the value 2 (H_MSG_TRUE) if the parameter values are correct. Otherwise an exception is raised.

Parallelization Information

`select_lines_longest` is *reentrant* and processed *without* parallelization.

<i>Possible Predecessors</i>	
sobel_amp , edges_image , threshold , hysteresis_threshold , split_skeleton_region , split_skeleton_lines	
<i>Possible Successors</i>	
set_line_width , disp_line	
<i>Alternatives</i>	
line_orientation , line_position , select_lines , partition_lines	
<i>See also</i>	
select_lines , partition_lines , detect_edge_segments , select_shape	
<i>Module</i>	
Region processing	

Chapter 9

Matching

9.1 Gray-Value-Based

adapt_template (Image : : TemplateID :)

Adapting a template to the size of an image.

The operator **adapt_template** serves to adapt a template which has been created by **create_template** to the size of an image. The operator **adapt_template** can be called before the template is used with images of another size, or if the image used to create the template had another size. If it is not called explicitly it will be called internally each time another image size is used. The contents of the image is hereby irrelevant; only the width of **Image** will be considered.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ Hobject : byte
Image which determines the size of the later matching.
- ▷ **TemplateID** (input_control) template $\leadsto$ integer
Template number.

Result

If the parameter values are correct, the operator **adapt_template** returns the value 2 (H_MSG.TRUE). If necessary, an exception handling is raised.

Parallelization Information

adapt_template is processed under *mutual exclusion* against itself and without parallelization.

Possible Predecessors

create_template, **create_template_rot**, **read_template**

Possible Successors

set_reference_template, **best_match**, **fast_match**, **fast_match_mg**,
set_offset_template, **best_match_mg**, **best_match_pre_mg**, **best_match_rot**,
best_match_rot_mg

Module

Template matching

best_match (Image : : TemplateID, MaxError, SubPixel : Row, Column, Error)

Searching the best matching of a template and an image.

The operator **best_match** performs a matching of the template of **TemplateID** and **Image**. Hereby the template will be moved over the points of **Image** so that the template will lie always inside **Image**. **best_match** works similar to **fast_match**, with the exception, that each time a better match is found the value of **MaxError** is internally updated to a lower value to reduce runtime.

With regard to the parameter `SubPixel`, the position will be indicated by subpixel accuracy. The matching criterion (“displaced frame difference”) is defined as follows:

$$error[row, col] = \frac{\sum_{u,v} |Image[row - u, col - v] - TemplateID[u, v]|}{area(TemplateID)}$$

The runtime of the operator depends on the size of the domain of `Image`. Therefore it is important to restrict the domain as far as possible, i.e. to apply the operator only in a very confined “region of interest”. The parameter `MaxError` determines the maximal error which the searched position is allowed to have at most. The lower this value is, the faster the operator runs.

`Row` and `Column` return the position of the best match, whereby `Error` indicates the average difference of the grayvalues. If no position with an error below `MaxError` was found the position (0, 0) and a matching result of 255 for `Error` are returned. In this case `MaxError` has to be set larger.

The maximum error of the position (without noise) is 0.1 pixel. The average error is 0.03 pixel.

Parameter	
▷ Image (input_object)	image(-array) $\leadsto$ <i>Hobject</i> : byte Input image inside of which the pattern has to be found.
▷ TemplateID (input_control)	template $\leadsto$ <i>integer</i> Template number.
▷ MaxError (input_control)	real $\leadsto$ <i>real</i> Maximum average difference of the grayvalues. Default Value : 20 Suggested values : <code>MaxError</code> $\in$ {0, 1, 2, 3, 4, 5, 6, 7, 9, 11, 15, 17, 20, 30, 40, 50, 0, 70} Typical range of values : $0 \leq \text{MaxError} \leq 255$ Minimum Increment : 1 Recommended Increment : 3
▷ SubPixel (input_control)	string $\leadsto$ <i>string</i> Subpixel accuracy in case of 'true'. Default Value : 'false' List of values : <code>SubPixel</code> $\in$ {'true', 'false'}
▷ Row (output_control)	point.y(-array) $\leadsto$ <i>real</i> Row position of the best match.
▷ Column (output_control)	point.x(-array) $\leadsto$ <i>real</i> Column position of the best match.
▷ Error (output_control)	real(-array) $\leadsto$ <i>real</i> Average divergence of the grayvalues of the best match.

Result

If the parameter values are correct, the operator `best_match` returns the value 2 (`H_MSG_TRUE`). If the input is empty (no input images are available) the behaviour can be set via `set_system('no_object_result', <Result>)`. If necessary, an exception handling is raised.

Parallelization Information

`best_match` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`create_template`, `read_template`, `set_offset_template`, `set_reference_template`, `adapt_template`, `draw_region`, `draw_rectangle1`, `reduce_domain`

Alternatives

`fast_match`, `fast_match_mg`, `best_match_mg`, `best_match_pre_mg`, `best_match_rot`, `best_match_rot_mg`, `exhaustive_match`, `exhaustive_match_mg`

Module

Template matching

best_match_mg (`Image` : : `TemplateID`, `MaxError`, `SubPixel`, `NumLevels`,
`WhichLevels` : `Row`, `Column`, `Error`)

Searching the best grayvalue matches in a pyramid.

best_match_mg applies gray value matching using an image pyramid. **best_match_mg** works analogously to **best_match**, but it is faster due to the use of a pyramid. Input is an image with an optionally reduced domain. The parameter **MaxError** specifies the maximum error for template matching. Using smaller values results in a reduced runtime but it is possible that the pattern might be missed. The value of **MaxError** has to be set larger compared with **best_match**, because the error is at higher levels of the pyramid often increased.

SubPixel specifies if the result is calculated with sub pixel accuracy or not. A value of 1 for **SubPixel** results in an operator similar to **best_match**, i.e. only the original gray values are used. For values larger than 1, the algorithm starts at the lowest resolution and searches for a position with the lowest matching error. At the next higher resolution this position is refined. This is continued up to the maximum resolution (**WhichLevels** = 'all'). As an alternative Method the mode **WhichLevels** with value 'original' can be used. In this case not only the position with the lowest error but all points below **MaxError** are analysed further on in the next higher resolution. This method is slower but it is more stable and the possibility to miss the correct position is very low. In this case it is often possible to start with a lower resolution (higher level in Pyramid, i.e. larger value for **NumLevels**) which leads to a reduced runtime. Besides the values 'all' and 'original' for **WhichLevels** you can specify the pyramid level explicitly where to switch between a "match all" and the "best match". Here 0 corresponds to 'original' and **NumLevels** - 1 is equivalent to 'all'. A value in between is in most cases a good compromise between speed and a stable detection. A larger value for **WhichLevels** results in a reduced runtime, a smaller value results in a more stable detection. The value of **NumLevels** has to be equal or smaller than the value used to create the template.

The position of the found matching position is returned in **Row** and **Column**. The corresponding error is given in **Error**. If no point below **MaxError** is found a value of 255 for **Error** and 0 for **Row** and **Column** is returned. If the desired object is missed (no object found or wrong position) you have to set **MaxError** higher or **WhichLevels** lower. Check also if the illumination has changed (see **set_offset_template**).

The maximum error of the position (without noise) is 0.1 pixel. The average error is 0.03 pixel.

Parameter	
▷ Image (input_control)	image(-array) $\leadsto$ <i>Hobject</i> : byte Input image inside of which the pattern has to be found.
▷ TemplateID (input_control)	template $\leadsto$ integer Template number.
▷ MaxError (input_control)	real $\leadsto$ real Maximal average difference of the grayvalues. Default Value : 30 Suggested values : $\text{MaxError} \in \{0, 1, 2, 3, 4, 5, 6, 7, 9, 11, 15, 17, 20, 30, 40, 50, 60, 70\}$ Typical range of values : $0 \leq \text{MaxError} \leq 255$ Minimum Increment : 1 Recommended Increment : 3
▷ SubPixel (input_control)	string $\leadsto$ string Exactness in subpixels in case of 'true'. Default Value : 'false' List of values : $\text{SubPixel} \in \{'true', 'false'\}$
▷ NumLevels (input_control)	integer $\leadsto$ integer Number of the used resolution levels. Default Value : 4 List of values : $\text{NumLevels} \in \{1, 2, 3, 4, 5, 6\}$
▷ WhichLevels (input_control)	integer $\leadsto$ integer / string Resolution level up to which the method "best match" is used. Default Value : 2 Suggested values : $\text{WhichLevels} \in \{'all', 'original', 0, 1, 2, 3, 4, 5, 6\}$
▷ Row (output_control)	point.y $\leadsto$ real Row position of the best match.
▷ Column (output_control)	point.x $\leadsto$ real Column position of the best match.
▷ Error (output_control)	real $\leadsto$ real Average divergence of the grayvalues in the best match.

Result

If the parameter values are correct, the operator **best_match_mg** returns the value 2 (H_MSG_TRUE).

If the input is empty (no input images are available) the behaviour can be set via `set_system('no_object_result', <Result>)`. If necessary, an exception handling is raised.

Parallelization Information

`best_match_mg` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`create_template`, `read_template`, `adapt_template`, `draw_region`, `draw_rectangle1`, `reduce_domain`, `set_reference_template`, `set_offset_template`

Alternatives

`fast_match`, `fast_match_mg`, `best_match`, `best_match_pre_mg`, `best_match_rot`, `best_match_rot_mg`, `exhaustive_match`, `exhaustive_match_mg`

Module

Template matching

best_match_pre_mg (ImagePyramid : : TemplateID, MaxError, SubPixel, NumLevels, WhichLevels : Row, Column, Error)

Searching the best grayvalue matches in a pre generated pyramid.

`best_match_pre_mg` applies gray value matching using an image pyramid. `best_match_pre_mg` works analogously to `best_match_mg`, but it makes use of pre calculated pyramid which has to be generated beforehand using `gen_gauss_pyramid`. This reduces runtime if more than one match has to be done or the pyramid has been used otherwise. The pyramid has to be generated using the zooming factor 0.5 and the mode 'constant'.

Parameter

- ▷ **ImagePyramid** (input_object) image-array $\leadsto$ Hobject : byte
Image pyramid inside of which the pattern has to be found.
- ▷ **TemplateID** (input_control) template $\leadsto$ integer
Template number.
- ▷ **MaxError** (input_control) real $\leadsto$ real
Maximal average difference of the grayvalues.
Default Value : 30
Suggested values : MaxError $\in \{0, 1, 2, 3, 4, 5, 6, 7, 9, 11, 15, 17, 20, 30, 40, 50, 60, 70\}$
Typical range of values : $0 \leq \text{MaxError} \leq 255$
Minimum Increment : 1
Recommended Increment : 3
- ▷ **SubPixel** (input_control) string $\leadsto$ string
Exactness in subpixels in case of 'true'.
Default Value : 'false'
List of values : SubPixel $\in \{'true', 'false'\}$
- ▷ **NumLevels** (input_control) integer $\leadsto$ integer
Number of the used resolution levels.
Default Value : 3
List of values : NumLevels $\in \{1, 2, 3, 4, 5, 6\}$
- ▷ **WhichLevels** (input_control) integer $\leadsto$ integer / string
Resolution level up to which the method "best match" is used.
Default Value : 'original'
Suggested values : WhichLevels $\in \{'all', 'original', 0, 1, 2, 3, 4, 5, 6\}$
- ▷ **Row** (output_control) point.y $\leadsto$ real
Row position of the best match.
- ▷ **Column** (output_control) point.x $\leadsto$ real
Column position of the best match.
- ▷ **Error** (output_control) real $\leadsto$ real
Average divergence of the grayvalues in the best match.

Result

If the parameter values are correct, the operator `best_match_pre_mg` returns the value 2 (H_MSG_TRUE).

If the input is empty (no input images are available) the behaviour can be set via `set_system('no_object_result', <Result>)`. If necessary, an exception handling is raised.

Parallelization Information

`best_match_pre_mg` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`gen_gauss_pyramid`, `create_template`, `read_template`, `adapt_template`, `draw_region`, `draw_rectangle1`, `reduce_domain`, `set_reference_template`

Alternatives

`fast_match`, `fast_match_mg`, `exhaustive_match`, `exhaustive_match_mg`

Module

Template matching

best_match_rot (Image : : TemplateID, AngleStart, AngleExtend, MaxError, SubPixel : Row, Column, Angle, Error)

Searching the best matching of a template and an image with rotation.

The operator `best_match_rot` performs a matching of the template of `TemplateID` and `Image`. It works similar to `best_match` with the extension that the pattern can be rotated. The parameters `AngleStart` and `AngleExtend` define the maximum rotation of the pattern: `AngleStart` specifies the maximum counter clockwise rotation and `AngleExtend` the maximum clockwise rotation relative to this angle. Both values have to be smaller or equal to the values used for the creation of the pattern (see `create_template_rot`). In addition to `best_match` `best_match_rot` returns the rotation angle of the pattern in `Angle` (radian). The accuracy of this angle depends on the parameter `AngleStep` of `create_template_rot`. In the case of `SubPixel = 'true'` the position and the angle are calculated with “sub pixel” accuracy.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ Hobject : byte
Input image inside of which the pattern has to be found.
- ▷ **TemplateID** (input_control) template $\leadsto$ integer
Template number.
- ▷ **AngleStart** (input_control) angle.rad $\leadsto$ real
Smallest Rotation auf the pattern.
Default Value : -0.39
Suggested values : `AngleStart` $\in$ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}
- ▷ **AngleExtend** (input_control) angle.rad $\leadsto$ real
Maximum positive Extention of `AngleStart`.
Default Value : 0.79
Suggested values : `AngleExtend` $\in$ {6.28, 3.14, 1.57, 0.79, 0.39}
Restriction : `AngleExtend` > 0
- ▷ **MaxError** (input_control) real $\leadsto$ real
Maximum average difference of the grayvalues.
Default Value : 30
Suggested values : `MaxError` $\in$ {0, 1, 2, 3, 4, 5, 6, 7, 9, 11, 15, 17, 20, 30, 40, 50, 60, 70}
Typical range of values : $0 \leq \text{MaxError} \leq 255$
Minimum Increment : 1
Recommended Increment : 3
- ▷ **SubPixel** (input_control) string $\leadsto$ string
Subpixel accuracy in case of 'true'.
Default Value : 'false'
List of values : `SubPixel` $\in$ {'true', 'false'}
- ▷ **Row** (output_control) point.y(-array) $\leadsto$ real
Row position of the best match.
- ▷ **Column** (output_control) point.x(-array) $\leadsto$ real
Column position of the best match.

- ▷ **Angle** (output_control) angle.rad(-array) $\leadsto$ *real*
Rotation angle of pattern.
- ▷ **Error** (output_control) real(-array) $\leadsto$ *real*
Average divergence of the grayvalues of the best match.

Result

If the parameter values are correct, the operator `best_match_rot` returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behaviour can be set via `set_system('no_object_result', <Result>)`. If necessary, an exception handling is raised.

Parallelization Information

`best_match_rot` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`create_template_rot`, `read_template`, `set_offset_template`, `set_reference_template`, `adapt_template`, `draw_region`, `draw_rectangle1`, `reduce_domain`

Alternatives

`best_match_rot_mg`

See also

`best_match`, `best_match_mg`

Module

Template matching

best_match_rot_mg (Image : : TemplateID, AngleStart, AngleExtend, MaxError, SubPixel, NumLevels : Row, Column, Angle, Error)

Searching the best matching of a template and a pyramid with rotation.

The operator `best_match_rot_mg` performs a matching of the template of `TemplateID` and `Image`. It works similar to `best_match_mg` with the extension that the pattern can be rotated analogously to `best_match_rot`. The parameters `AngleStart` and `AngleExtend` define the maximum rotation of the pattern: `AngleStart` specifies the maximum counter clockwise rotation and `AngleExtend` the maximum clockwise rotation relative to this angle. Both values have to be smaller or equal to the values used for the creation of the pattern (see `create_template_rot`). In addition to `best_match_mg` `best_match_rot_mg` returns the rotation angle of the pattern in `Angle` (radian).

The value of `MaxError` has to be set larger compared with `best_match_rot`, because the error is at higher levels of the pyramid often increased.

In the case of `SubPixel = 'true'` the position and the angle are calculated with “sub pixel” accuracy.

The value of `NumLevels` has to be equal or smaller than the value used to create the template.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte
Input image inside of which the pattern has to be found.
- ▷ **TemplateID** (input_control) template $\leadsto$ *integer*
Template number.
- ▷ **AngleStart** (input_control) angle.rad $\leadsto$ *real*
Smallest Rotation auf the pattern.
Default Value : -0.39
Suggested values : `AngleStart` $\in$ {-3.14, -1.57, -0.79, -0.39, -0.20, 0.0}
- ▷ **AngleExtend** (input_control) angle.rad $\leadsto$ *real*
Maximum positive Extension of `AngleStart`.
Default Value : 0.79
Suggested values : `AngleExtend` $\in$ {6.28, 3.14, 1.57, 0.79, 0.39}
Restriction : `AngleExtend` > 0

- ▷ **MaxError** (input_control) real $\leadsto$ real
Maximum average difference of the grayvalues.
Default Value : 40
Suggested values : MaxError $\in \{0, 1, 2, 3, 4, 5, 6, 7, 9, 11, 15, 17, 20, 30, 40, 50, 60, 70\}$
Typical range of values : $0 \leq \text{MaxError} \leq 255$
Minimum Increment : 1
Recommended Increment : 1
- ▷ **SubPixel** (input_control) string $\leadsto$ string
Subpixel accuracy in case of 'true'.
Default Value : 'false'
List of values : SubPixel $\in \{'true', 'false'\}$
- ▷ **NumLevels** (input_control) integer $\leadsto$ integer
Number of the used resolution levels.
Default Value : 3
List of values : NumLevels $\in \{1, 2, 3, 4, 5, 6\}$
- ▷ **Row** (output_control) point.y(-array) $\leadsto$ real
Row position of the best match.
- ▷ **Column** (output_control) point.x(-array) $\leadsto$ real
Column position of the best match.
- ▷ **Angle** (output_control) angle.rad(-array) $\leadsto$ real
Rotation angle of pattern.
- ▷ **Error** (output_control) real(-array) $\leadsto$ real
Average divergence of the grayvalues of the best match.

Result

If the parameter values are correct, the operator `best_match_rot_mg` returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behaviour can be set via `set_system('no_object_result', <Result>)`. If necessary, an exception handling is raised.

Parallelization Information

`best_match_rot_mg` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`create_template_rot`, `set_reference_template`, `set_offset_template`, `adapt_template`, `draw_region`, `draw_rectangle1`, `reduce_domain`

Alternatives

`best_match_rot`, `best_match_mg`

See also

`fast_match`

Module

Template matching

clear_all_templates (: : :)

Deallocation of the memory of all templates.

The operator `clear_template` deallocates the memory of all template that were created by `create_template` or `create_template_rot`. After calling `clear_all_templates`, no template can be used any longer.

Result

`clear_all_templates` always returns 2 (H_MSG_TRUE).

Parallelization Information

`clear_all_templates` is processed *completely exclusively* without parallelization.

Possible Predecessors

`create_template`, `create_template_rot`, `read_template`, `write_template`

Alternatives

`clear_template`

Module

Template matching

clear_template (: : TemplateID :)

Deallocation of the memory of a template.

The operator `clear_template` deallocates the memory of a template which has been created by `create_template` or `create_template_rot`. After execution of the operator `clear_template` the template can no longer be used. The value of `TemplateID` is not valid. However, the number can be used again by further calls of `create_template` or `create_template_rot`.

Parameter

▷ **TemplateID** (input_control) template $\leadsto$ integer
Template number.

Result

If the number of the template is valid, the operator `clear_template` returns the value 2 (H_MSG_TRUE). If necessary an exception handling will be raised.

Parallelization Information

`clear_template` is processed *completely exclusively* without parallelization.

Possible Predecessors

`create_template`, `create_template_rot`, `read_template`, `write_template`

See also

`clear_all_templates`

Module

Template matching

create_template (Template : : FirstError, NumLevel, Optimize,
GrayValues : TemplateID)

Preparing a pattern for template matching.

The operator `create_template` preprocesses a pattern (`Template`), which is passed as an image, for the template matching. After the transformation, a number (`TemplateID`) is assigned to the template for being used in the further process. The shape and the size of `Template` can be chosen arbitrarily. You have to be aware, that the matching is only applied to that part of an image where `Template` fits completely into the image.

The template has be chosen such that it contains no pixels of the (changing) background. Here you can make use of the arbitrary shape of a template which is not restricted to a rectangle. To create a template you can use segmentation operators like `threshold`. In the case of sub pixel accurate matching `Template` has in addition to be one pixel smaller than the pattern (i.e. one pixel border to the changing background). This can be done e.g. by applying the operator `erosion_circle`.

The parameter `NumLevel` specifies the number of pyramid levels (`NumLevel` = 1 means only original gray values) which can be used for matching. The number of levels used later for matching will be below or equal this value. If the pattern becomes too small due to zooming, the maximum number of pyramid levels is automatically reduced (without error message).

The parameter `GrayValues` defines, whether the original gray values (`'original'`, `'normalized'`) or the edge amplitude (`'gradient'`, `'sobel'`) is used. With `'original'` the sum of the differences is used as feature which is very stable and fast if there is no change in illumination. `'normalized'` is used if the illumination changes. The method is a bit slower and not quite as stable. If there is no change in illumination the mode `'original'` should be used. The edge amplitude is another method to be invariant to changes in illumination. The disadvantage is the increased execution time and the higher sensitivity to changes in the shape of the pattern. The mode `'gradient'` is slightly faster but more sensitive to noise.

The maximum error for matching has typically to be chosen higher when using the edge amplitude. The mode chosen by `GrayValues` leads automatically to calling the appropriate filter during matching — if necessary.

As an alternative to the gradient approach the operator `set_offset_template` can be used, if the change in illumination is known.

The parameter `Optimize` specifies if the pattern has to be optimized for runtime. This optimization results in a longer time to create the template but reduces the time for matching. In addition the optimization leads to a more stable matching, i.e., the possibility to miss good matches is reduced. The optimization process selects the most stable and significant gray values to be tested first during the matching process. Using this technique a wrong match can be eliminated very early.

The reference position for the template is its center of gravity. I.e. if you apply the template to the original image the center of gravity is returned. This default reference can be adapted using the operator `set_reference_template`.

In sub pixel mode a special position correction is calculated which is added after each matching: The template is applied to the original image and the difference between the found position and the center of gravity is used as a correction vector. This is important for patterns in a textured context or for asymmetric pattern. For most templates this correction vector is near null.

If the pattern is no longer used, it has to be freed by the operator `clear_template` in order to deallocate the memory.

Before the use of the template, which is stored independently of the image size, it can be adapted explicitly to the size of a definite image size by using `adapt_template`.

<i>Parameter</i>	
▷ Template (input_object)	image $\leadsto$ <i>Hobject</i> : byte Input image whose domain will be processed for the pattern matching.
▷ FirstError (input_control)	integer $\leadsto$ <i>integer</i> Not jet in use. Default Value : 255 List of values : FirstError $\in$ {255}
▷ NumLevel (input_control)	integer $\leadsto$ <i>integer</i> Maximal number of pyramid levels. Default Value : 4 List of values : NumLevel $\in$ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
▷ Optimize (input_control)	string $\leadsto$ <i>string</i> Kind of optimizing. Default Value : 'sort' List of values : Optimize $\in$ {'none', 'sort'}
▷ GrayValues (input_control)	string $\leadsto$ <i>string</i> Kind of grayvalues. Default Value : 'original' List of values : GrayValues $\in$ {'original', 'normalized', 'gradient', 'sobel'}
▷ TemplateID (output_control)	template $\leadsto$ <i>integer</i> Template number.

Result

If the parameters are valid, the operator `create_template` returns the value 2 (H_MSG_TRUE). If necessary an exception handling will be raised.

Parallelization Information

`create_template` is processed *completely exclusively* without parallelization.

Possible Predecessors

`draw_region`, `reduce_domain`, `threshold`

Possible Successors

`adapt_template`, `set_reference_template`, `clear_template`, `write_template`,
`set_offset_template`, `best_match`, `best_match_mg`, `fast_match`, `fast_match_mg`

Alternatives

`create_template_rot`, `read_template`

Module

Template matching

create_template_rot (Template : : NumLevel, AngleStart, AngleExtend, AngleStep, Optimize, GrayValues : TemplateID)

Preparing a pattern for template matching with rotation.

The operator `create_template_rot` preprocesses a pattern, which is passed as an image, for the template matching. An extension to `create_template` the matching can be applied to rotated patterns. The parameters `AngleStart` and `AngleExtend` define the maximum rotation of the pattern: `AngleStart` specifies the maximum counter clockwise rotation and `AngleExtend` the maximum clockwise rotation relative to this angle. Therefore `AngleExtend` has to be smaller than 2π . With the parameter `AngleStep` the maximum angle resolution (on the highest resolution level) can be specified.

You have to be aware, that all possible rotations are calculated beforehand to reduce runtime during matching. This leads to a higher execution time for `create_template_rot` and high memory requirements for the template. The amount of memory depends on the parameters `AngleExtend` and `AngleStep`. The number of pyramid levels can be neglected. If A is the number of pixels of `Template`, the memory M needed for the template in byte is about:

$$M = \frac{A * 12 * \text{AngleExtend}}{\text{AngleStep}}$$

After the transformation, a number (`TemplateID`) is assigned to the template for being used in the further process.

A description of the other parameters can be found at the operator `create_template`.

Attention

You have to be aware, that depending on the resolution a large number of pre calculated patterns have to be created which might result in a large amount of memory needed.

Parameter

- ▷ **Template** (input_object) image $\leadsto$ *Hobject* : byte
Input image whose domain will be processed for the pattern matching.
- ▷ **NumLevel** (input_control) integer $\leadsto$ *integer*
Maximal number of pyramid levels.
Default Value : 4
List of values : NumLevel $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
- ▷ **AngleStart** (input_control) angle.rad $\leadsto$ *real*
Smallest Rotation of the pattern.
Default Value : -0.39
Suggested values : AngleStart $\in \{-3.14, -1.57, -0.79, -0.39, -0.20, 0.0\}$
- ▷ **AngleExtend** (input_control) angle.rad $\leadsto$ *real*
Maximum positive Extention of `AngleStart`.
Default Value : 0.79
Suggested values : AngleExtend $\in \{6.28, 3.14, 1.57, 0.79, 0.39\}$
Restriction : AngleExtend > 0
- ▷ **AngleStep** (input_control) angle.rad $\leadsto$ *real*
Step rate (angle precision) of matching.
Default Value : 0.0982
Suggested values : AngleStep $\in \{0.3927, 0.1963, 0.0982, 0.0491, 0.0245\}$
Restriction : AngleStep > 0
- ▷ **Optimize** (input_control) string $\leadsto$ *string*
Kind of optimizing.
Default Value : 'sort'
List of values : Optimize $\in \{'none', 'sort'\}$
- ▷ **GrayValues** (input_control) string $\leadsto$ *string*
Kind of grayvalues.
Default Value : 'original'
List of values : GrayValues $\in \{'original', 'normalized', 'gradient', 'sobel'\}$
- ▷ **TemplateID** (output_control) template $\leadsto$ *integer*
Template number.

Result
If the parameters are valid, the operator <code>create_template_rot</code> returns the value 2 (H_MSG_TRUE). If necessary an exception handling will be raised.
Parallelization Information
<code>create_template_rot</code> is processed <i>completely exclusively</i> without parallelization.
Possible Predecessors
<code>draw_region</code> , <code>reduce_domain</code> , <code>threshold</code>
Possible Successors
<code>best_match_rot</code> , <code>best_match_rot_mg</code> , <code>adapt_template</code> , <code>set_reference_template</code> , <code>clear_template</code> , <code>set_offset_template</code> , <code>write_template</code>
Alternatives
<code>create_template</code>
Module
Template matching

fast_match (Image : Matches : TemplateID, MaxError :)

Searching all good matches of a template and an image.

The operator `fast_match` performs a matching of the template of `TemplateID` and `Image`. Hereby the template will be moved over the points of `Image` so that the template always lies completely inside of `Image`. The matching criterion (“displaced frame difference”) is defined as follows:

$$error[row, col] = \frac{\sum_{u,v} |\text{Image}[row - u, col - v] - \text{TemplateID}[u, v]|}{area(\text{TemplateID})}$$

The difference between `fast_match` and `exhaustive_match` is that the matching for one position is stopped if the error is too high. This leads to a reduced runtime but one might miss correct matches. The runtime of the operator depends mainly on the size of the domain of `Image`. Therefore it is important to restrict the domain as far as possible, i.e. to apply the operator only in a very confined “region of interest”. The parameter `MaxError` determines the maximal error which the searched position is allowed to show. The lower this value is, the faster the operator runs.

All points which show a matching error smaller than `MaxError` will be returned in the output region `Matches`. This region can be used for further processing. For example by using a connection and `best_match` to find all the matching objects. If no point has a match error below `MaxError` the empty region (i.e. no points) is returned.

Parameter
▷ Image (input_object) image(-array) $\rightsquigarrow$ Hobject : byte Input image inside of which the pattern has to be found. ▷ Matches (output_object) region(-array) $\rightsquigarrow$ Hobject All points whose error lies below a certain threshold. ▷ TemplateID (input_control) template $\rightsquigarrow$ integer Template number. ▷ MaxError (input_control) real $\rightsquigarrow$ real Maximal average difference of the grayvalues. Default Value : 20 Suggested values : <code>MaxError</code> $\in$ {0, 1, 2, 3, 4, 5, 6, 7, 9, 11, 15, 17, 20, 30} Typical range of values : $0 \leq \text{MaxError} \leq 255$ Minimum Increment : 1 Recommended Increment : 1

Result
If the parameter values are correct, the operator <code>fast_match</code> returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behaviour can be set via <code>set_system('no_object_result', <Result>)</code>. If necessary, an exception handling is raised.

<i>Parallelization Information</i>
<code>fast_match</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>).
<i>Possible Predecessors</i>
<code>create_template</code> , <code>read_template</code> , <code>adapt_template</code> , <code>draw_region</code> , <code>draw_rectangle1</code> , <code>reduce_domain</code>
<i>Possible Successors</i>
<code>connection</code> , <code>best_match</code>
<i>Alternatives</i>
<code>best_match</code> , <code>best_match_mg</code> , <code>fast_match_mg</code> , <code>exhaustive_match</code> , <code>exhaustive_match_mg</code>
<i>Module</i>
Template matching

fast_match_mg (Image : Matches : TemplateID, MaxError, NumLevel :)

Searching all good grayvalue matches in a pyramid.

The operator `fast_match_mg` like the operator `fast_match` performs a matching of the template of `TemplateID` and `Image`. In contrast to `fast_match`, however, the search for good matches starts in scaled down images (pyramid). The number of levels of the pyramid will be determined by `NumLevel`. Hereby the value 1 indicates that no pyramid will be used. In this case the operator `fast_match_mg` is equivalent to the operator `fast_match`. The value 2 triggers the search in the image with half the framesize. The search for all those points showing an error small enough in the scaled down image (error smaller than `MaxError`) will be refined at the corresponding positions in the original image (`Image`).

The runtime of matching depends on the parameter `MaxError`: the larger the value the longer is the processing time, because more points of the pattern have to be tested. If `MaxError` is too low the pattern will not be found. The value has therefore to be optimized for every application.

`NumLevel` indicates the number of levels of the pyramid, including the original image. Optionally a second value can be given. This value specifies the number (0..n) of the lowest level which is used for the matching. The region found up to this level will then be zoomed to the size of the original level. This can be used to increase the runtime in the case that the accuracy does not have to be so high.

<i>Parameter</i>
▷ Image (input_object) image(-array) $\leadsto$ <i>Hobject</i> : byte Input image inside of which the pattern has to be found.
▷ Matches (output_object) region(-array) $\leadsto$ <i>Hobject</i> All points which have an error below a certain threshold.
▷ TemplateID (input_control) template $\leadsto$ <i>integer</i> Template number.
▷ MaxError (input_control) real $\leadsto$ <i>real</i> Maximal average difference of the grayvalues. Default Value : 30 Suggested values : <code>MaxError</code> $\in$ {0, 1, 2, 3, 4, 5, 6, 7, 9, 11, 15, 17, 20, 30, 40, 50, 60, 70} Typical range of values : $0 \leq \text{MaxError} \leq 255$ Minimum Increment : 1 Recommended Increment : 3
▷ NumLevel (input_control) integer(-array) $\leadsto$ <i>integer</i> Number of levels in the pyramid. Default Value : 3 List of values : <code>NumLevel</code> $\in$ {1, 2, 3, 4, 5, 6, 7, 8}

Result

If the parameter values are correct, the operator `fast_match_mg` returns the value 2 (`H_MSG_TRUE`). If the input is empty (no input images are available) the behaviour can be set via `set_system('no_object_result', <Result>)`. If necessary, an exception handling is raised.

<i>Parallelization Information</i>
<code>fast_match_mg</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>).

Possible Predecessors

`create_template`, `read_template`, `adapt_template`, `draw_region`, `draw_rectangle1`, `reduce_domain`

Alternatives

`best_match`, `best_match_mg`, `fast_match`, `exhaustive_match`, `exhaustive_match_mg`

Module

Template matching

<code>read_template</code> (: : <code>FileName</code> : <code>TemplateID</code>)

Reading a template from file.

The operator `read_template` reads a matching template from file which has been written with `write_template`.

Parameter

- ▷ **`FileName`** (input_control) filename $\leadsto$ *string*
file name.
- ▷ **`TemplateID`** (output_control) template $\leadsto$ *integer*
Template number.

Result

If the file name is valid, the operator `read_template` returns the value 2 (`H_MSG_TRUE`). If necessary an exception handling will be raised.

Parallelization Information

`read_template` is processed *completely exclusively* without parallelization.

Possible Successors

`adapt_template`, `set_reference_template`, `set_offset_template`, `best_match`, `fast_match`, `best_match_rot`

Module

Template matching

<code>set_offset_template</code> (: : <code>TemplateID</code> , <code>GrayOffset</code> :)

Gray value offset for template.

`set_offset_template` adds a gray value offset to the template to eliminate gray value changes in the image. The parameter `GrayOffset` specifies a difference relative to the gray values of the pattern when it was created with `create_template` (not relative to the last call of `set_offset_template`). The values of `GrayOffset` has to be chosen according to the gray values of the image: A brighter image results in a positive value, a darker image results in a negative value. `set_offset_template` has to be called each time the gray values of the image changes. The gray values can be measured in a reference area using `intensity` or `min_max_gray`

Parameter

- ▷ **`TemplateID`** (input_control) template $\leadsto$ *integer*
Template number.
- ▷ **`GrayOffset`** (input_control) number $\leadsto$ *integer*
Offset of gray values.
Default Value : 0
Suggested values : `GrayOffset` $\in$ { -10, -5, -2, -1, 0, 1, 2, 5, 10 }
Typical range of values : $-255 \leq \text{GrayOffset} \leq 255$
Minimum Increment : 1
Recommended Increment : 1

Result

If the parameter values are correct, the operator `set_offset_template` returns the value 2 (H_MSG_TRUE). If necessary, an exception handling is raised.

Parallelization Information

`set_offset_template` is processed under *mutual exclusion* against itself and without parallelization.

Possible Predecessors

`create_template`, `adapt_template`, `read_template`

Possible Successors

`best_match`, `best_match_mg`, `best_match_rot`, `fast_match`, `fast_match_mg`

Module

Template matching

set_reference_template (: : TemplateID, Row, Column :)

Define reference position for a matching template.

`set_reference_template` allows to define a new reference position for a template. As default after calling `create_template` or `create_template_rot` the center of gravity of the template is used. Using `set_reference_template` the reference position can be redefined. In the case of the center of gravity as reference the vector (0, 0) is returned after matching for a null translation of the pattern relative to the image.

Parameter

- ▷ **TemplateID** (input_control) template $\leadsto$ integer
Template number.
- ▷ **Row** (input_control) point.y $\leadsto$ real
Reference position of template (row).
- ▷ **Column** (input_control) point.x $\leadsto$ real
Reference position of template (column).

Result

If the parameter values are correct, the operator `set_reference_template` returns the value 2 (H_MSG_TRUE). If necessary, an exception handling is raised.

Parallelization Information

`set_reference_template` is processed under *mutual exclusion* against itself and without parallelization.

Possible Predecessors

`create_template`, `create_template_rot`, `read_template`, `adapt_template`

Possible Successors

`best_match`, `best_match_mg`, `best_match_rot`, `fast_match`, `fast_match_mg`

Module

Template matching

write_template (: : TemplateID, FileName :)

Writing a template to file.

The operator `write_template` writes a matching template to file which can be read again with `read_template`.

Parameter

- ▷ **TemplateID** (input_control) template $\leadsto$ integer
Template number.
- ▷ **FileName** (input_control) filename $\leadsto$ string
file name.

Result

If the file name is valid (permission to write), the operator `write_template` returns the value 2 (H.MSG.TRUE). If necessary an exception handling will be raised.

Parallelization Information

`write_template` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`create_template`, `create_template_rot`

Module

Template matching

9.2 Shape-Based

<code>clear_all_shape_models</code> (: : :)

Free the memory of all shape models.

The operator `clear_all_shape_models` frees the memory of all shape models that were created by `create_shape_model` or `create_scaled_shape_model`. After calling `clear_all_shape_models`, no model can be used any longer.

Result

`clear_all_shape_models` always returns 2 (H.MSG.TRUE).

Parallelization Information

`clear_all_shape_models` is processed *completely exclusively* without parallelization.

Possible Predecessors

`create_shape_model`, `create_scaled_shape_model`, `read_shape_model`,
`write_shape_model`

Alternatives

`clear_shape_model`

Module

Template matching

<code>clear_shape_model</code> (: : ModelID :)

Free the memory of a shape model.

The operator `clear_shape_model` frees the memory of a shape model that was created by `create_shape_model` or `create_scaled_shape_model`. After calling `clear_shape_model`, the model can no longer be used. The handle `ModelID` becomes invalid.

Parameter

- ▷ **ModelID** (input_control) shape_model $\leadsto$ integer
Handle of the model.

Result

If the handle of the model is valid, the operator `clear_shape_model` returns the value 2 (H.MSG.TRUE). If necessary an exception is raised.

Parallelization Information

`clear_shape_model` is processed *completely exclusively* without parallelization.

Possible Predecessors

`create_shape_model`, `create_scaled_shape_model`, `read_shape_model`,
`write_shape_model`

See also

`clear_all_shape_models`


```
create_scaled_shape_model ( Template : : NumLevels, AngleStart,
AngleExtent, AngleStep, ScaleMin, ScaleMax, ScaleStep, Optimization,
Metric, Contrast, MinContrast : ModelID )
```

Prepare a shape model for scale invariant matching.

The operator `create_scaled_shape_model` prepares a template, which is passed in the image `Template`, as a shape model used for scale invariant matching.

The model is generated using multiple image pyramid levels and multiple rotations and scales, and is stored in memory. The output parameter `ModelID` is a handle for this model, which is used in subsequent calls to `find_scaled_shape_model`.

The number of pyramid levels is determined with the parameter `NumLevels`. It should be chosen as large as possible because by this the time necessary to find the object is significantly reduced. On the other hand, `NumLevels` must be chosen such that the model is still recognizable and contains a sufficient number of points (at least four) on the highest pyramid level. This can be checked using the output of `inspect_shape_model`. If not enough model points are generated, the number of pyramid levels is reduced internally until enough model points are found on the highest pyramid level. If this procedure would lead to a model with no pyramid levels, i.e., if the number of model points is already too small on the lowest pyramid level, `create_scaled_shape_model` returns with an error message. If `NumLevels` is set to 0, `create_scaled_shape_model` determines the number of pyramid levels automatically. The automatically computed number of pyramid levels can be queried using `get_shape_model_params`. In rare cases, it might happen that `create_scaled_shape_model` determines a value for the number of pyramid levels that is too large or too small. If the number of pyramid levels is chosen too large, the model may not be recognized in the image or it may be necessary to select very low parameters for `MinScore` or `Greediness` in `find_scaled_shape_model` in order to find the model. If the number of pyramid levels is chosen too small, the time required to find the model in `find_scaled_shape_model` may increase. In these cases, the number of pyramid levels should be selected using the output of `inspect_shape_model`.

The parameters `AngleStart` and `AngleExtent` determine the range of possible rotations, in which the model can occur in the image. Note that the model can only be found in this range of angles by `find_scaled_shape_model`. The parameter `AngleStep` determines the step length within the selected range of angles. Hence, if subpixel accuracy is not specified in `find_scaled_shape_model`, this parameter specifies the accuracy that is achievable for the angles in `find_scaled_shape_model`. `AngleStep` should be chosen based on the size of the object. Smaller models do not have many different discrete rotations in the image, and hence `AngleStep` should be chosen larger for smaller models. If `AngleExtent` is not an integer multiple of `AngleStep`, `AngleStep` is modified accordingly.

The parameters `ScaleMin` and `ScaleMax` determine the range of possible scales (sizes) of the model. A scale of 1 corresponds to the original size of the model. The parameter `ScaleStep` determines the step length within the selected range of scales. Hence, if subpixel accuracy is not specified in `find_scaled_shape_model`, this parameter specifies the accuracy that is achievable for the scales in `find_scaled_shape_model`. Like `AngleStep`, `ScaleStep` should be chosen based on the size of the object. If the range of scales is not an integer multiple of `ScaleStep`, `ScaleStep` is modified accordingly.

The model is pre-generated for the selected angle and scale range and stored in memory. The memory required to store the model is proportional to the number of angle steps, the number of scale steps, and the number of points in the model. Hence, if `AngleStep` or `ScaleStep` are too small or `AngleExtent` or the range of scales are too big, it may happen that the model no longer fits into the (virtual) memory. In this case, either `AngleStep` or `ScaleStep` must be enlarged or `AngleExtent` or the range of scales must be reduced. In any case, it is desirable that the model completely fits into the main memory, because this avoids paging by the operating system, and hence the time to find the object will be much smaller. Since angles can be determined with subpixel resolution by `find_scaled_shape_model`, `AngleStep` $\geq 1^\circ$ and `ScaleStep` ≥ 0.02 can be selected for models of a diameter smaller than about 200 pixels. If `AngleStep` = 0 or `ScaleStep` = 0 is selected, `create_scaled_shape_model` automatically determines a suitable angle or scale step length, respectively, based on the size of the model. The automatically computed angle and scale step lengths can be queried using `get_shape_model_params`.

For particularly large models, it may be useful to reduce the number of model points by setting `Optimization` to a value different from 'none'. If `Optimization` = 'none', all model points are stored. In all other cases,

the number of points is reduced according to the value of `Optimization`. If the number of points is reduced, it may be necessary in `find_scaled_shape_model` to set the parameter `Greediness` to a smaller value, e.g., 0.7 or 0.8. For small models, the reduction of the number of model points does not result in a speed-up of the search because in this case usually significantly more potential instances of the model must be examined.

The parameter `Contrast` determines the contrast the model points must have. The contrast is a measure for local gray value differences between the object and the background and between different parts of the object. `Contrast` should be chosen such that only the significant features of the template are used for the model. `Contrast` can also contain a tuple with two values. In this case, the model is segmented using a method similar to the hysteresis threshold method used in `edges_image`. Here, the first element of the tuple determines the lower threshold, while the second element determines the upper threshold. For more information about the hysteresis threshold method, see `hysteresis_threshold`. Optionally, `Contrast` can contain a third value as the last element of the tuple. This value determines a threshold for the selection of significant model components based on the size of the components, i.e., components that have fewer points than the minimum size thus specified are suppressed. This threshold for the minimum size is divided by two for each successive pyramid level. If small model components should be suppressed, but hysteresis thresholding should not be performed, nevertheless three values must be specified in `Contrast`. In this case, the first two values can simply be set to identical values. The effect of this parameter can be checked in advance with `inspect_shape_model`.

With `MinContrast`, it can be determined which contrast the model must at least have in the recognition performed by `find_scaled_shape_model`. In other words, this parameter separates the model from the noise in the image. Therefore, a good choice is the range of gray value changes caused by the noise in the image. If, for example, the gray values fluctuate within a range of 10 gray levels, `MinContrast` should be set to 10. Obviously, `MinContrast` must be smaller than `Contrast`. If the model should be recognized in very low contrast images, `MinContrast` must be set to a correspondingly small value. If the model should be recognized even if it is severely occluded, `MinContrast` should be slightly larger than the range of gray value fluctuations created by noise in order to ensure that the position and rotation of the model are extracted robustly and accurately by `find_scaled_shape_model`.

The parameter `Metric` determines the conditions under which the model is recognized in the image. If `Metric` = `'use_polarity'`, the object in the image and the model must have the same contrast. If, for example, the model is a bright object on a dark background, the object is found only if it is also brighter than the background. If `Metric` = `'ignore_global_polarity'`, the object is found in the image also if the contrast reverses globally. In the above example, the object hence is also found if it is darker than the background. The runtime of `find_scaled_shape_model` will increase slightly in this case. If `Metric` = `'ignore_local_polarity'`, the model is found even if the contrast changes locally. This mode can, for example, be useful if the object consists of a part with medium gray value, within which either darker or brighter sub-objects lie. Since in this case the runtime of `find_scaled_shape_model` increases significantly, it is usually better to create several models that reflect the possible contrast variations of the object with `create_scaled_shape_model`, and to match them simultaneously with `find_scaled_shape_models`.

The center of gravity of the domain (region) of the model image `Template` is used as the origin (reference point) of the model. `find_scaled_shape_model` returns the coordinates of this point in the search image. A different origin can be set with `set_shape_model_origin`.

Parameter

- ▷ **Template** (input_object) image $\leadsto$ *Hobject* : byte
Input image whose domain will be used to create the model.
- ▷ **NumLevels** (input_control) integer $\leadsto$ *integer*
Maximum number of pyramid levels.
Default Value : 0
List of values : NumLevels $\in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
- ▷ **AngleStart** (input_control) angle.rad $\leadsto$ *real*
Smallest rotation of the pattern.
Default Value : -0.39
Suggested values : AngleStart $\in \{-3.14, -1.57, -0.79, -0.39, -0.20, 0.0\}$
- ▷ **AngleExtent** (input_control) angle.rad $\leadsto$ *real*
Extent of the rotation angles.
Default Value : 0.79
Suggested values : AngleExtent $\in \{6.28, 3.14, 1.57, 0.79, 0.39\}$
Restriction : AngleExtent ≥ 0

- ▷ **AngleStep** (input_control) angle.rad $\leadsto$ *real*
 Step length of the angles (resolution).
Default Value : 0
Suggested values : AngleStep $\in \{0, 0.0175, 0.0349, 0.0524, 0.0698, 0.0873\}$
Restriction : AngleStep ≥ 0
- ▷ **ScaleMin** (input_control) number $\leadsto$ *real*
 Minimum scale of the pattern.
Default Value : 0.9
Suggested values : ScaleMin $\in \{0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
Restriction : ScaleMin > 0
- ▷ **ScaleMax** (input_control) number $\leadsto$ *real*
 Maximum scale of the pattern.
Default Value : 1.1
Suggested values : ScaleMax $\in \{1.0, 1.1, 1.2, 1.3, 1.4, 1.5\}$
Restriction : ScaleMax $\geq$ ScaleMin
- ▷ **ScaleStep** (input_control) number $\leadsto$ *real*
 Scale step length (resolution).
Default Value : 0
Suggested values : ScaleStep $\in \{0.0, 0.01, 0.02, 0.05, 0.1, 0.15, 0.2\}$
Restriction : ScaleStep ≥ 0
- ▷ **Optimization** (input_control) string $\leadsto$ *string*
 Kind of optimization.
Default Value : 'none'
List of values : Optimization $\in \{'none', 'point\_reduction\_low', 'point\_reduction\_medium', 'point\_reduction\_high'\}$
- ▷ **Metric** (input_control) string $\leadsto$ *string*
 Match metric.
Default Value : 'use_polarity'
List of values : Metric $\in \{'use\_polarity', 'ignore\_global\_polarity', 'ignore\_local\_polarity'\}$
- ▷ **Contrast** (input_control) number(-array) $\leadsto$ *integer*
 Threshold or hysteresis thresholds for the contrast of the object in the template image and optionally minimum size of the object parts.
Default Value : 30
Suggested values : Contrast $\in \{10, 20, 30, 40, 60, 80, 100, 120, 140, 160\}$
- ▷ **MinContrast** (input_control) number $\leadsto$ *integer*
 Minimum contrast of the objects in the search images.
Default Value : 5
Suggested values : MinContrast $\in \{1, 2, 3, 5, 7, 10, 20, 20, 40\}$
Restriction : MinContrast $<$ Contrast
- ▷ **ModelID** (output_control) shape_model $\leadsto$ *integer*
 Handle of the model.

Result

If the parameters are valid, the operator `create_scaled_shape_model` returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. If the parameters `NumLevels` and `Contrast` are chosen such that the model contains too few points, the error 8510 is raised.

Parallelization Information

`create_scaled_shape_model` is processed *completely exclusively* without parallelization.

Possible Predecessors

`draw_region`, `reduce_domain`, `threshold`

Possible Successors

`find_scaled_shape_model`, `find_scaled_shape_models`, `get_shape_model_params`,
`clear_shape_model`, `write_shape_model`, `set_shape_model_origin`

Alternatives

`create_shape_model`, `create_template_rot`

Module

Template matching

```
create_shape_model ( Template : : NumLevels, AngleStart, AngleExtent,
AngleStep, Optimization, Metric, Contrast, MinContrast : ModelID )
```

Prepare a shape model for matching.

The operator `create_shape_model` prepares a template, which is passed in the image `Template`, as a shape model used for matching.

The model is generated using multiple image pyramid levels and multiple rotations and is stored in memory. The output parameter `ModelID` is a handle for this model, which is used in subsequent calls to `find_shape_model`.

The number of pyramid levels is determined with the parameter `NumLevels`. It should be chosen as large as possible because by this the time necessary to find the object is significantly reduced. On the other hand, `NumLevels` must be chosen such that the model is still recognizable and contains a sufficient number of points (at least four) on the highest pyramid level. This can be checked using the output of `inspect_shape_model`. If not enough model points are generated, the number of pyramid levels is reduced internally until enough model points are found on the highest pyramid level. If this procedure would lead to a model with no pyramid levels, i.e., if the number of model points is already too small on the lowest pyramid level, `create_shape_model` returns with an error message. If `NumLevels` is set to 0, `create_shape_model` determines the number of pyramid levels automatically. The automatically computed number of pyramid levels can be queried using `get_shape_model_params`. In rare cases, it might happen that `create_shape_model` determines a value for the number of pyramid levels that is too large or too small. If the number of pyramid levels is chosen too large, the model may not be recognized in the image or it may be necessary to select very low parameters for `MinScore` or `Greediness` in `find_shape_model` in order to find the model. If the number of pyramid levels is chosen too small, the time required to find the model in `find_shape_model` may increase. In these cases, the number of pyramid levels should be selected using the output of `inspect_shape_model`.

The parameters `AngleStart` and `AngleExtent` determine the range of possible rotations, in which the model can occur in the image. Note that the model can only be found in this range of angles by `find_shape_model`. The parameter `AngleStep` determines the step length within the selected range of angles. Hence, if subpixel accuracy is not specified in `find_shape_model`, this parameter specifies the accuracy that is achievable for the angles in `find_shape_model`. `AngleStep` should be chosen based on the size of the object. Smaller models do not possess many different discrete rotations in the image, and hence `AngleStep` should be chosen larger for smaller models. If `AngleExtent` is not an integer multiple of `AngleStep`, `AngleStep` is modified accordingly. The model is pre-generated for the selected angle range and stored in memory. The memory required to store the model is proportional to the number of angle steps and the number of points in the model. Hence, if `AngleStep` is too small or `AngleExtent` too big, it may happen that the model no longer fits into the (virtual) memory. In this case, either `AngleStep` must be enlarged or `AngleExtent` must be reduced. In any case, it is desirable that the model completely fits into the main memory, because this avoids paging by the operating system, and hence the time to find the object will be much smaller. Since angles can be determined with subpixel resolution by `find_shape_model`, `AngleStep` ≥ 1 can be selected for models of a diameter smaller than about 200 pixels. If `AngleStep` = 0 is selected, `create_shape_model` automatically determines a suitable angle step length based on the size of the model. The automatically computed angle step length can be queried using `get_shape_model_params`.

For particularly large models, it may be useful to reduce the number of model points by setting `Optimization` to a value different from 'none'. If `Optimization` = 'none', all model points are stored. In all other cases, the number of points is reduced according to the value of `Optimization`. If the number of points is reduced, it may be necessary in `find_shape_model` to set the parameter `Greediness` to a smaller value, e.g., 0.7 or 0.8. For small models, the reduction of the number of model points does not result in a speed-up of the search because in this case usually significantly more potential instances of the model must be examined.

The parameter `Contrast` determines the contrast the model points must have. The contrast is a measure for local gray value differences between the object and the background and between different parts of the object. `Contrast` should be chosen such that only the significant features of the template are used for the model. `Contrast` can also contain a tuple with two values. In this case, the model is segmented using a method similar to the hysteresis threshold method used in `edges_image`. Here, the first element of the tuple determines the lower threshold, while the second element determines the upper threshold. For more information about the hysteresis threshold method, see `hysteresis_threshold`. Optionally, `Contrast` can contain a third value as the last element of the tuple. This value determines a threshold for the selection of significant model components based on the size of the components, i.e., components that have fewer points than the minimum size thus specified are suppressed. This threshold for the minimum size is divided by two for each successive pyramid level. If small model components should be suppressed, but hysteresis thresholding should not be performed, nevertheless three

values must be specified in **Contrast**. In this case, the first two values can simply be set to identical values. The effect of this parameter can be checked in advance with **inspect_shape_model**.

With **MinContrast**, it can be determined which contrast the model must at least have in the recognition performed by **find_shape_model**. In other words, this parameter separates the model from the noise in the image. Therefore, a good choice is the range of gray value changes caused by the noise in the image. If, for example, the gray values fluctuate within a range of 10 gray levels, **MinContrast** should be set to 10. Obviously, **MinContrast** must be smaller than **Contrast**. If the model should be recognized in very low contrast images, **MinContrast** must be set to a correspondingly small value. If the model should be recognized even if it is severely occluded, **MinContrast** should be slightly larger than the range of gray value fluctuations created by noise in order to ensure that the position and rotation of the model are extracted robustly and accurately by **find_shape_model**.

The parameter **Metric** determines the conditions under which the model is recognized in the image. If **Metric** = *'use_polarity'*, the object in the image and the model must have the same contrast. If, for example, the model is a bright object on a dark background, the object is found only if it is also brighter than the background. If **Metric** = *'ignore_global_polarity'*, the object is found in the image also if the contrast reverses globally. In the above example, the object hence is also found if it is darker than the background. The runtime of **find_shape_model** will increase slightly in this case. If **Metric** = *'ignore_local_polarity'*, the model is found even if the contrast changes locally. This mode can, for example, be useful if the object consists of a part with medium gray value, within which either darker or brighter sub-objects lie. Since in this case the runtime of **find_shape_model** increases significantly, it is usually better to create several models that reflect the possible contrast variations of the object with **create_shape_model**, and to match them simultaneously with **find_shape_models**.

The center of gravity of the domain (region) of the model image **Template** is used as the origin (reference point) of the model. **find_shape_model** returns the coordinates of this point in the search image. A different origin can be set with **set_shape_model_origin**.

Parameter

- ▷ **Template** (input_object) image $\leadsto$ *Hobject* : byte
Input image whose domain will be used to create the model.
- ▷ **NumLevels** (input_control) integer $\leadsto$ *integer*
Maximum number of pyramid levels.
Default Value : 0
List of values : NumLevels $\in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
- ▷ **AngleStart** (input_control) angle.rad $\leadsto$ *real*
Smallest rotation of the pattern.
Default Value : -0.39
Suggested values : AngleStart $\in \{-3.14, -1.57, -0.79, -0.39, -0.20, 0.0\}$
- ▷ **AngleExtent** (input_control) angle.rad $\leadsto$ *real*
Extent of the rotation angles.
Default Value : 0.79
Suggested values : AngleExtent $\in \{6.28, 3.14, 1.57, 0.79, 0.39\}$
Restriction : AngleExtent ≥ 0
- ▷ **AngleStep** (input_control) angle.rad $\leadsto$ *real*
Step length of the angles (resolution).
Default Value : 0
Suggested values : AngleStep $\in \{0, 0.0175, 0.0349, 0.0524, 0.0698, 0.0873\}$
Restriction : AngleStep ≥ 0
- ▷ **Optimization** (input_control) string $\leadsto$ *string*
Kind of optimization.
Default Value : 'none'
List of values : Optimization $\in \{'none', 'point\_reduction\_low', 'point\_reduction\_medium', 'point\_reduction\_high'\}$
- ▷ **Metric** (input_control) string $\leadsto$ *string*
Match metric.
Default Value : 'use_polarity'
List of values : Metric $\in \{'use\_polarity', 'ignore\_global\_polarity', 'ignore\_local\_polarity'\}$
- ▷ **Contrast** (input_control) number(-array) $\leadsto$ *integer*
Threshold or hysteresis thresholds for the contrast of the object in the template image and optionally minimum

size of the object parts.

Default Value : 30

Suggested values : Contrast $\in \{10, 20, 30, 40, 60, 80, 100, 120, 140, 160\}$

- ▷ **MinContrast** (input_control) number $\leadsto$ integer
Minimum contrast of the objects in the search images.

Default Value : 5

Suggested values : MinContrast $\in \{1, 2, 3, 5, 7, 10, 20, 40\}$

Restriction : MinContrast < Contrast

- ▷ **ModelID** (output_control) shape_model $\leadsto$ integer
Handle of the model.

Result

If the parameters are valid, the operator `create_shape_model` returns the value 2 (H_MSG_TRUE). If necessary an exception is raised. If the parameters `NumLevels` and `Contrast` are chosen such that the model contains too few points, the error 8510 is raised.

Parallelization Information

`create_shape_model` is processed *completely exclusively* without parallelization.

Possible Predecessors

`draw_region`, `reduce_domain`, `threshold`

Possible Successors

`find_shape_model`, `find_shape_models`, `get_shape_model_params`, `clear_shape_model`, `write_shape_model`, `set_shape_model_origin`

Alternatives

`create_scaled_shape_model`, `create_template_rot`

Module

Template matching

find_scaled_shape_model (Image : : ModelID, AngleStart, AngleExtent, ScaleMin, ScaleMax, MinScore, NumMatches, MaxOverlap, SubPixel, NumLevels, Greediness : Row, Column, Angle, Scale, Score)

Find the best matches of a scale invariant shape model in an image.

The operator `find_scaled_shape_model` finds the best `NumMatches` instances of the scale invariant shape model `ModelID` in the input image `Image`. The model must have been created previously by calling `create_scaled_shape_model` or `read_shape_model`.

The position, rotation, and scale of the found instances of the model are returned in `Row`, `Column`, `Angle`, and `Scale`. The coordinates `Row` and `Column` are the coordinates of the origin of the shape model in the search image. By default, the origin is the center of gravity of the domain (region) of the image that was used to create the shape model with `create_scaled_shape_model`. A different origin can be set with `set_shape_model_origin`. Additionally, the score of each found instance is returned in `Score`. The score is a number between 0 and 1, which is an approximate measure of how much of the model is visible in the image. If, for example, half of the model is occluded, the score cannot exceed 0.5.

The domain of the image `Image` determines the search space for the reference point of the model, i.e., for the center of gravity of the domain (region) of the image that was used to create the shape model with `create_scaled_shape_model`. A different origin set with `set_shape_model_origin` is not taken into account. The model is searched within those points of the domain of the image, in which the model lies completely within the image. This means that the model will not be found if it extends beyond the borders of the image, even if it would achieve a score greater than `MinScore` (see below). The parameters `AngleStart` and `AngleExtent` determine the range of rotations for which the model is searched. The parameters `ScaleMin` and `ScaleMax` determine the range of scales for which the model is searched. If necessary, both ranges are clipped to the range given when the model was created with `create_scaled_shape_model`. In particular, this means that the angle ranges of the model and the search must truly overlap. The angle range in the search is not adapted modulo 2π . To simplify the presentation, all angles in the remainder of the paragraph are given in degrees, whereas they have to be specified in radians in `find_scaled_shape_model`. Hence, if the model, for example, was created with `AngleStart` = -20° and `AngleExtent` = 40° and the angle search space in

`find_scaled_shape_model` is, for example, set to `AngleStart = 350°` and `AngleExtent = 20°`, the model will not be found, even though the angle ranges would overlap if they were regarded modulo 360°. To find the model, in this example it would be necessary to select `AngleStart = -10°`.

The parameter `MinScore` determines what score a potential match must at least have to be regarded as an instance of the model in the image. The larger `MinScore` is chosen, the faster the search is. If the model can be expected never to be occluded in the images, `MinScore` may be set as high as 0.8 or even 0.9.

The maximum number of instances to be found can be determined with `NumMatches`. If more than `NumMatches` instances with a score greater than `MinScore` are found in the image, only the best `NumMatches` instances are returned. If fewer than `NumMatches` are found, only that number is returned, i.e., the parameter `MinScore` takes precedence over `NumMatches`.

If the model exhibits symmetries it may happen that multiple instances with similar positions but different rotations are found in the image. The parameter `MaxOverlap` determines by what fraction (i.e., a number between 0 and 1) two instances may at most overlap in order to consider them as different instances, and hence to be returned separately. If two instances overlap each other by more than `MaxOverlap` only the best instance is returned. The calculation of the overlap is based on the smallest enclosing rectangle of arbitrary orientation (see `smallest_rectangle2`) of the found instances. If `MaxOverlap = 0`, the found instances may not overlap at all, while for `MaxOverlap = 1` all instances are returned.

The parameter `SubPixel` determines whether the instances should be extracted with subpixel accuracy. If `SubPixel` is set to `'none'` (or `'false'` for backwards compatibility) the model's pose is only determined with pixel accuracy and the angle and scale resolution that was specified with `create_scaled_shape_model`. If `SubPixel` is set to `'interpolation'` (or `'true'`) the position as well as the rotation and scale are determined with subpixel accuracy. In this mode, the model's pose is interpolated from the score function. This mode costs almost no computation time and achieves an accuracy that is high enough for most applications. In some applications, however, the accuracy requirements are extremely high. In these cases, the model's pose can be determined through a least-squares adjustment, i.e., by minimizing the distances of the model points to their corresponding image points. In contrast to `'interpolation'`, this mode requires additional computation time. The different modes for least-squares adjustment (`'least_squares'`, `'least_squares_high'`, and `'least_squares_very_high'`) can be used to determine the accuracy with which the minimum distance is being searched. The higher the accuracy is chosen, the longer the subpixel extraction will take, however. Usually, `SubPixel` should be set to `'interpolation'`. If least-squares adjustment is desired, `'least_squares'` should be chosen because this results in the best tradeoff between run time and accuracy.

The number of pyramid levels used during the search is determined with `NumLevels`. If necessary, the number of levels is clipped to the range given when the shape model was created with `create_scaled_shape_model`. If `NumLevels` is set to 0, the number of pyramid levels specified in `create_scaled_shape_model` is used. If `NumLevels` is set to 0, the number of pyramid levels specified in `create_shape_model` is used. Optionally, `NumLevels` can contain a second value that determines the lowest pyramid level to which the found matches are tracked. Hence, a value of `[4,2]` for `NumLevels` means that the matching starts at the fourth pyramid level and tracks the matches to the second lowest pyramid level (the lowest pyramid level is denoted by a value of 1). This mechanism can be used to decrease the runtime of the matching. It should be noted, however, that in general the accuracy of the extracted pose parameters is lower in this mode than in the normal mode, in which the matches are tracked to the lowest pyramid level. Hence, if a high accuracy is desired, `SubPixel` should be set to at least `'least_squares'`. If the lowest pyramid level to use is chosen too large, it may happen that the desired accuracy cannot be achieved, or that wrong instances of the model are found because the model is not specific enough on the higher pyramid levels to facilitate a reliable selection of the correct instance of the model. In this case, the lowest pyramid level to use must be set to a smaller value.

The parameter `Greediness` determines how “greedily” the search should be carried out. If `Greediness = 0`, a safe search heuristic is used, which always finds the model if it is visible in the image. However, the search will be relatively time consuming in this case. If `Greediness = 1`, an unsafe search heuristic is used, which may cause the model not to be found in rare cases, even though it is visible in the image. For `Greediness = 1`, the maximum search speed is achieved. In almost all cases, the shape model will always be found for `Greediness = 0.9`.

Parameter

- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte
Input image in which the model should be found.
- ▷ **ModelID** (input_control) shape_model $\leadsto$ integer
Handle of the model.

- ▷ **AngleStart** (input_control) angle.rad $\leadsto$ real
Smallest rotation of the model.
Default Value : -0.39
Suggested values : AngleStart $\in \{-3.14, -1.57, -0.78, -0.39, -0.20, 0.0\}$
- ▷ **AngleExtent** (input_control) angle.rad $\leadsto$ real
Extent of the rotation angles.
Default Value : 0.78
Suggested values : AngleExtent $\in \{6.28, 3.14, 1.57, 0.78, 0.39, 0.0\}$
Restriction : AngleExtent ≥ 0
- ▷ **ScaleMin** (input_control) number $\leadsto$ real
Minimum scale of the model.
Default Value : 0.9
Suggested values : ScaleMin $\in \{0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
Restriction : ScaleMin > 0
- ▷ **ScaleMax** (input_control) number $\leadsto$ real
Maximum scale of the model.
Default Value : 1.1
Suggested values : ScaleMax $\in \{1.0, 1.1, 1.2, 1.3, 1.4, 1.5\}$
Restriction : ScaleMax $\geq$ ScaleMin
- ▷ **MinScore** (input_control) real $\leadsto$ real
Minimum score of the instances of the model to be found.
Default Value : 0.5
Suggested values : MinScore $\in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
Typical range of values : $0 \leq \text{MinScore} \leq 1$
Minimum Increment : 0.01
Recommended Increment : 0.05
- ▷ **NumMatches** (input_control) integer $\leadsto$ integer
Number of instances of the model to be found.
Default Value : 1
Suggested values : NumMatches $\in \{0, 1, 2, 3, 4, 5, 10, 20\}$
- ▷ **MaxOverlap** (input_control) real $\leadsto$ real
Maximum overlap of the instances of the model to be found.
Default Value : 0.5
Suggested values : MaxOverlap $\in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
Typical range of values : $0 \leq \text{MaxOverlap} \leq 1$
Minimum Increment : 0.01
Recommended Increment : 0.05
- ▷ **SubPixel** (input_control) string $\leadsto$ string
Subpixel accuracy if not equal to 'none'.
Default Value : 'interpolation'
List of values : SubPixel $\in \{'none', 'interpolation', 'least_squares', 'least_squares_high', 'least_squares_very_high'\}$
- ▷ **NumLevels** (input_control) integer(-array) $\leadsto$ integer
Number of pyramid levels used in the matching (and lowest pyramid level to use if |NumLevels| = 2).
Default Value : 0
List of values : NumLevels $\in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
- ▷ **Greediness** (input_control) real $\leadsto$ real
"Greediness" of the search heuristic (0: safe but slow; 1: fast but matches may be missed).
Default Value : 0.9
Suggested values : Greediness $\in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
Typical range of values : $0 \leq \text{Greediness} \leq 1$
Minimum Increment : 0.01
Recommended Increment : 0.05
- ▷ **Row** (output_control) point.y-array $\leadsto$ real
Row coordinate of the found instances of the model.
- ▷ **Column** (output_control) point.x-array $\leadsto$ real
Column coordinate of the found instances of the model.

- ▷ **Angle** (output_control) angle.rad-array $\leadsto$ *real*
Rotation angle of the found instances of the model.
- ▷ **Scale** (output_control) number-array $\leadsto$ *real*
Scale of the found instances of the model.
- ▷ **Score** (output_control) real-array $\leadsto$ *real*
Score of the found instances of the model.

Result

If the parameter values are correct, the operator `find_scaled_shape_model` returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via `set_system('no_object_result', <Result>)`. If necessary, an exception is raised.

Parallelization Information

`find_scaled_shape_model` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`create_scaled_shape_model`, `read_shape_model`, `set_shape_model_origin`

Alternatives

`find_shape_model`, `find_shape_models`, `find_scaled_shape_models`, `best_match_rotmg`

Module

Template matching

```
find_scaled_shape_models ( Image : : ModelIDs, AngleStart,
AngleExtent, ScaleMin, ScaleMax, MinScore, NumMatches, MaxOverlap,
SubPixel, NumLevels, Greediness : Row, Column, Angle, Scale, Score,
Model )
```

Find the best matches of multiple scale invariant shape models.

The operator `find_scaled_shape_models` finds the best `NumMatches` instances of the scale invariant shape models that are passed in `ModelIDs` in the input image `Image`. The models must have been created previously by calling `create_scaled_shape_model` or `read_shape_model`.

Hence, in contrast to `find_scaled_shape_model`, multiple models can be searched in the same image in one call. This changes the semantics of all input parameters to some extent. All input parameters must either contain one element, in which case the parameter is used for all models, or must contain the same number of elements as `ModelIDs`, in which case each parameter element refers to the corresponding element in `ModelIDs`. (`NumLevels` may also contain either two or twice the number of elements as `ModelIDs`; see below.) As usual, the domain of the input image `Image` is used to restrict the search space for the reference point of the models `ModelIDs`. Consistent with the above semantics, the input image `Image` can therefore contain a single image object or an image object tuple containing multiple image objects. If `Image` contains a single image object, its domain is used as the region of interest for all models in `ModelIDs`. If `Image` contains multiple image objects, each domain is used as the region of interest for the corresponding model in `ModelIDs`. In this case, the image matrix of all image objects in the tuple must be identical, i.e., `Image` cannot be constructed in an arbitrary manner using `concat_obj`, but must be created from the same image using `add_channels` or equivalent calls. If this is not the case, an error message is returned. The above semantics also hold for the input control parameters. Hence, for example, `MinScore` can contain a single value or the same number of values as `ModelIDs`. In the first case, the value of `MinScore` is used for all models in `ModelIDs`, while in the second case the respective value of the elements in `MinScore` is used for the corresponding model in `ModelIDs`. An extension to these semantics holds for `NumMatches` and `MaxOverlap`. If `NumMatches` contains one element, `find_scaled_shape_models` returns the best `NumMatches` instances of the model irrespective of the type of the model. If, for example, two models are passed in `ModelIDs` and `NumMatches = 2` is selected, it can happen that two instances of the first model and no instances of the second model, one instance of the first model and one instance of the second model, or no instances of the first model and two instances of the second model are returned. If, on the other hand, `NumMatches` contains multiple values, the number of instances returned of the different models corresponds to the number specified in the respective entry in `NumMatches`. If, for example, `NumMatches = [1, 1]` is selected, one instance of the first model and one instance of the second model is returned. For a detailed description of the semantics of `NumMatches`, see below. A similar extension of the semantics holds for `MaxOverlap`. If a single value is passed for `MaxOverlap`, the overlap is computed for all found instances of the different models, irrespective of the model type, i.e., instances of the same or of different models that overlap too much are

eliminated. If, on the other hand, multiple values are passed in `MaxOverlap`, the overlap is only computed for found instances of the model that have the same model type, i.e., only instances of the same model that overlap too much are eliminated. In this mode, models of different types may overlap completely. For a detailed description of the semantics of `MaxOverlap`, see below. Hence, a call to `find_scaled_shape_models` with multiple values for `ModelIDs`, `NumMatches` and `MaxOverlap` has the same effect as multiple independent calls to `find_scaled_shape_model` with the respective parameters. However, a single call to `find_shape_models` is considerably more efficient.

The type of the found instances of the models is returned in `Model`. The elements of `Model` are indices into the tuple `ModelIDs`, i.e., they can contain values from 0 to $|\text{ModelIDs}| - 1$. Hence, a value of 0 in an element of `Model` corresponds to an instance of the first model in `ModelIDs`.

The position, rotation, and scale of the found instances of the model are returned in `Row`, `Column`, `Angle`, and `Scale`. The coordinates `Row` and `Column` are the coordinates of the origin of the shape model in the search image. By default, the origin is the center of gravity of the domain (region) of the image that was used to create the shape model with `create_scaled_shape_model`. A different origin can be set with `set_shape_model_origin`. Additionally, the score of each found instance is returned in `Score`. The score is a number between 0 and 1, which is an approximate measure of how much of the model is visible in the image. If, for example, half of the model is occluded, the score cannot exceed 0.5.

The domain of the image `Image` determines the search space for the reference point of the model, i.e., for the center of gravity of the domain (region) of the image that was used to create the shape model with `create_scaled_shape_model`. A different origin set with `set_shape_model_origin` is not taken into account. The model is searched within those points of the domain of the image, in which the model lies completely within the image. This means that the model will not be found if it extends beyond the borders of the image, even if it would achieve a score greater than `MinScore` (see below). The parameters `AngleStart` and `AngleExtent` determine the range of rotations for which the model is searched. The parameters `ScaleMin` and `ScaleMax` determine the range of scales for which the model is searched. If necessary, both ranges are clipped to the range given when the model was created with `create_scaled_shape_model`. In particular, this means that the angle ranges of the model and the search must truly overlap. The angle range in the search is not adapted modulo 2π . To simplify the presentation, all angles in the remainder of the paragraph are given in degrees, whereas they have to be specified in radians in `find_scaled_shape_models`. Hence, if the model, for example, was created with `AngleStart` = -20° and `AngleExtent` = 40° and the angle search space in `find_scaled_shape_models` is, for example, set to `AngleStart` = 350° and `AngleExtent` = 20° , the model will not be found, even though the angle ranges would overlap if they were regarded modulo 360° . To find the model, in this example it would be necessary to select `AngleStart` = -10° .

The parameter `MinScore` determines what score a potential match must at least have to be regarded as an instance of the model in the image. The larger `MinScore` is chosen, the faster the search is. If the model can be expected never to be occluded in the images, `MinScore` may be set as high as 0.8 or even 0.9.

The maximum number of instances to be found can be determined with `NumMatches`. If more than `NumMatches` instances with a score greater than `MinScore` are found in the image, only the best `NumMatches` instances are returned. If fewer than `NumMatches` are found, only that number is returned, i.e., the parameter `MinScore` takes precedence over `NumMatches`.

If the model exhibits symmetries it may happen that multiple instances with similar positions but different rotations are found in the image. The parameter `MaxOverlap` determines by what fraction (i.e., a number between 0 and 1) two instances may at most overlap in order to consider them as different instances, and hence to be returned separately. If two instances overlap each other by more than `MaxOverlap` only the best instance is returned. The calculation of the overlap is based on the smallest enclosing rectangle of arbitrary orientation (see `smallest_rectangle2`) of the found instances. If `MaxOverlap` = 0, the found instances may not overlap at all, while for `MaxOverlap` = 1 all instances are returned.

The parameter `SubPixel` determines whether the instances should be extracted with subpixel accuracy. If `SubPixel` is set to `'none'` (or `'false'` for backwards compatibility) the model's pose is only determined with pixel accuracy and the angle and scale resolution that was specified with `create_scaled_shape_model`. If `SubPixel` is set to `'interpolation'` (or `'true'`) the position as well as the rotation and scale are determined with subpixel accuracy. In this mode, the model's pose is interpolated from the score function. This mode costs almost no computation time and achieves an accuracy that is high enough for most applications. In some applications, however, the accuracy requirements are extremely high. In these cases, the model's pose can be determined through a least-squares adjustment, i.e., by minimizing the distances of the model points to their corresponding image points. In contrast to `'interpolation'`, this mode requires additional computation time. The different modes for least-squares adjustment (`'least_squares'`, `'least_squares_high'`, and `'least_squares_very_high'`) can be used to

determine the accuracy with which the minimum distance is being searched. The higher the accuracy is chosen, the longer the subpixel extraction will take, however. Usually, `SubPixel` should be set to `'interpolation'`. If least-squares adjustment is desired, `'least_squares'` should be chosen because this results in the best tradeoff between run time and accuracy.

The number of pyramid levels used during the search is determined with `NumLevels`. If necessary, the number of levels is clipped to the range given when the shape model was created with `create_scaled_shape_model`. If `NumLevels` is set to 0, the number of pyramid levels specified in `create_scaled_shape_model` is used. Optionally, `NumLevels` can contain a second value that determines the lowest pyramid level to which the found matches are tracked. Hence, a value of `[4,2]` for `NumLevels` means that the matching starts at the fourth pyramid level and tracks the matches to the second lowest pyramid level (the lowest pyramid level is denoted by a value of 1). This mechanism can be used to decrease the runtime of the matching. It should be noted, however, that in general the accuracy of the extracted pose parameters is lower in this mode than in the normal mode, in which the matches are tracked to the lowest pyramid level. Hence, if a high accuracy is desired, `SubPixel` should be set to at least `'least_squares'`. If the lowest pyramid level to use is chosen too large, it may happen that the desired accuracy cannot be achieved, or that wrong instances of the model are found because the model is not specific enough on the higher pyramid levels to facilitate a reliable selection of the correct instance of the model. In this case, the lowest pyramid level to use must be set to a smaller value. If the lowest pyramid level is specified separately for each model, `NumLevels` must contain twice the number of elements as `ModelIDs`. In this case, the number of pyramid levels and the lowest pyramid level must be specified interleaved in `NumLevels`. If, for example, two models are specified in `ModelIDs`, the number of pyramid levels is 5 for the first model and 4 for the second model, and the lowest pyramid level is 2 for the first model and 1 for the second model, `NumLevels = [5, 2, 4, 1]` must be selected. If exactly two models are specified in `ModelIDs`, a special case occurs. If in this case the lowest pyramid level is to be specified, the number of pyramid levels and the lowest pyramid level must be specified explicitly for both models, even if they are identical, because specifying two values in `NumLevels` is interpreted as the explicit specification of the number of pyramid levels for the two models.

The parameter `Greediness` determines how “greedily” the search should be carried out. If `Greediness = 0`, a safe search heuristic is used, which always finds the model if it is visible in the image. However, the search will be relatively time consuming in this case. If `Greediness = 1`, an unsafe search heuristic is used, which may cause the model not to be found in rare cases, even though it is visible in the image. For `Greediness = 1`, the maximum search speed is achieved. In almost all cases, the shape model will always be found for `Greediness = 0.9`.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte
Input image in which the models should be found.
- ▷ **ModelIDs** (input_control) shape_model(-array) $\leadsto$ *integer*
Handle of the models.
- ▷ **AngleStart** (input_control) angle.rad(-array) $\leadsto$ *real*
Smallest rotation of the models.
Default Value : -0.39
Suggested values : AngleStart $\in \{-3.14, -1.57, -0.78, -0.39, -0.20, 0.0\}$
- ▷ **AngleExtent** (input_control) angle.rad(-array) $\leadsto$ *real*
Extent of the rotation angles.
Default Value : 0.78
Suggested values : AngleExtent $\in \{6.28, 3.14, 1.57, 0.78, 0.39, 0.0\}$
Restriction : AngleExtent ≥ 0
- ▷ **ScaleMin** (input_control) number(-array) $\leadsto$ *real*
Minimum scale of the models.
Default Value : 0.9
Suggested values : ScaleMin $\in \{0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
Restriction : ScaleMin > 0
- ▷ **ScaleMax** (input_control) number(-array) $\leadsto$ *real*
Maximum scale of the models.
Default Value : 1.1
Suggested values : ScaleMax $\in \{1.0, 1.1, 1.2, 1.3, 1.4, 1.5\}$
Restriction : ScaleMax $\geq$ ScaleMin

- ▷ **MinScore** (input_control) real(-array) $\leadsto$ *real*
Minimum score of the instances of the models to be found.
Default Value : 0.5
Suggested values : MinScore $\in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
Typical range of values : $0 \leq \text{MinScore} \leq 1$
Minimum Increment : 0.01
Recommended Increment : 0.05
- ▷ **NumMatches** (input_control) integer(-array) $\leadsto$ *integer*
Number of instances of the models to be found.
Default Value : 1
Suggested values : NumMatches $\in \{0, 1, 2, 3, 4, 5, 10, 20\}$
- ▷ **MaxOver lap** (input_control) real(-array) $\leadsto$ *real*
Maximum overlap of the instances of the models to be found.
Default Value : 0.5
Suggested values : MaxOver lap $\in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
Typical range of values : $0 \leq \text{MaxOver lap} \leq 1$
Minimum Increment : 0.01
Recommended Increment : 0.05
- ▷ **SubPixel** (input_control) string(-array) $\leadsto$ *string*
Subpixel accuracy if not equal to 'none'.
Default Value : 'interpolation'
List of values : SubPixel $\in \{'none', 'interpolation', 'least\_squares', 'least\_squares\_high', 'least\_squares\_very\_high'\}$
- ▷ **NumLevels** (input_control) integer(-array) $\leadsto$ *integer*
Number of pyramid levels used in the matching (and lowest pyramid level to use if |NumLevels| = 2).
Default Value : 0
List of values : NumLevels $\in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
- ▷ **Greediness** (input_control) real(-array) $\leadsto$ *real*
"Greediness" of the search heuristic (0: safe but slow; 1: fast but matches may be missed).
Default Value : 0.9
Suggested values : Greediness $\in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
Typical range of values : $0 \leq \text{Greediness} \leq 1$
Minimum Increment : 0.01
Recommended Increment : 0.05
- ▷ **Row** (output_control) point.y-array $\leadsto$ *real*
Row coordinate of the found instances of the models.
- ▷ **Column** (output_control) point.x-array $\leadsto$ *real*
Column coordinate of the found instances of the models.
- ▷ **Angle** (output_control) angle.rad-array $\leadsto$ *real*
Rotation angle of the found instances of the models.
- ▷ **Scale** (output_control) number-array $\leadsto$ *real*
Scale of the found instances of the models.
- ▷ **Score** (output_control) real-array $\leadsto$ *real*
Score of the found instances of the models.
- ▷ **Model** (output_control) integer-array $\leadsto$ *integer*
Index of the found instances of the models.

Result

If the parameter values are correct, the operator `find_scaled_shape_models` returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via `set_system('no_object_result', <Result>)`. If necessary, an exception is raised.

Parallelization Information

`find_scaled_shape_models` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`add_channels`, `create_scaled_shape_model`, `read_shape_model`, `set_shape_model_origin`

Alternatives

`find_shape_models`, `find_shape_model`, `find_scaled_shape_model`, `best_match_rot_mg`

Template matching

```
find_shape_model ( Image : : ModelID, AngleStart, AngleExtent,
MinScore, NumMatches, MaxOverlap, SubPixel, NumLevels, Greediness : Row,
Column, Angle, Score )
```

Find the best matches of a shape model in an image.

The operator `find_shape_model` finds the best `NumMatches` instances of the shape model `ModelID` in the input image `Image`. The model must have been created previously by calling `create_shape_model` or `read_shape_model`.

The position and rotation of the found instances of the model is returned in `Row`, `Column` and `Angle`. The coordinates `Row` and `Column` are the coordinates of the origin of the shape model in the search image. By default, the origin is the center of gravity of the domain (region) of the image that was used to create the shape model with `create_shape_model`. A different origin can be set with `set_shape_model_origin`. Additionally, the score of each found instance is returned in `Score`. The score is a number between 0 and 1, which is an approximate measure of how much of the model is visible in the image. If, for example, half of the model is occluded, the score cannot exceed 0.5.

The domain of the image `Image` determines the search space for the reference point of the model, i.e., for the center of gravity of the domain (region) of the image that was used to create the shape model with `create_shape_model`. A different origin set with `set_shape_model_origin` is not taken into account. The model is searched within those points of the domain of the image, in which the model lies completely within the image. This means that the model will not be found if it extends beyond the borders of the image, even if it would achieve a score greater than `MinScore` (see below). The parameters `AngleStart` and `AngleExtent` determine the range of rotations for which the model is searched. If necessary, the range of rotations is clipped to the range given when the model was created with `create_shape_model`. In particular, this means that the angle ranges of the model and the search must truly overlap. The angle range in the search is not adapted modulo 2π . To simplify the presentation, all angles in the remainder of the paragraph are given in degrees, whereas they have to be specified in radians in `find_shape_model`. Hence, if the model, for example, was created with `AngleStart` = -20° and `AngleExtent` = 40° and the angle search space in `find_shape_model` is, for example, set to `AngleStart` = 350° and `AngleExtent` = 20° , the model will not be found, even though the angle ranges would overlap if they were regarded modulo 360° . To find the model, in this example it would be necessary to select `AngleStart` = -10° .

The parameter `MinScore` determines what score a potential match must at least have to be regarded as an instance of the model in the image. The larger `MinScore` is chosen, the faster the search is. If the model can be expected never to be occluded in the images, `MinScore` may be set as high as 0.8 or even 0.9.

The maximum number of instances to be found can be determined with `NumMatches`. If more than `NumMatches` instances with a score greater than `MinScore` are found in the image, only the best `NumMatches` instances are returned. If fewer than `NumMatches` are found, only that number is returned, i.e., the parameter `MinScore` takes precedence over `NumMatches`.

If the model exhibits symmetries it may happen that multiple instances with similar positions but different rotations are found in the image. The parameter `MaxOverlap` determines by what fraction (i.e., a number between 0 and 1) two instances may at most overlap in order to consider them as different instances, and hence to be returned separately. If two instances overlap each other by more than `MaxOverlap` only the best instance is returned. The calculation of the overlap is based on the smallest enclosing rectangle of arbitrary orientation (see `smallest_rectangle2`) of the found instances. If `MaxOverlap` = 0, the found instances may not overlap at all, while for `MaxOverlap` = 1 all instances are returned.

The parameter `SubPixel` determines whether the instances should be extracted with subpixel accuracy. If `SubPixel` is set to `'none'` (or `'false'` for backwards compatibility) the model's pose is only determined with pixel accuracy and the angle resolution that was specified with `create_shape_model`. If `SubPixel` is set to `'interpolation'` (or `'true'`) the position as well as the rotation are determined with subpixel accuracy. In this mode, the model's pose is interpolated from the score function. This mode costs almost no computation time and achieves an accuracy that is high enough for most applications. In some applications, however, the accuracy requirements are extremely high. In these cases, the model's pose can be determined through a least-squares adjustment, i.e., by minimizing the distances of the model points to their corresponding image points. In contrast

to 'interpolation', this mode requires additional computation time. The different modes for least-squares adjustment ('least_squares', 'least_squares_high', and 'least_squares_very_high') can be used to determine the accuracy with which the minimum distance is being searched. The higher the accuracy is chosen, the longer the subpixel extraction will take, however. Usually, `SubPixel` should be set to 'interpolation'. If least-squares adjustment is desired, 'least_squares' should be chosen because this results in the best tradeoff between run time and accuracy.

The number of pyramid levels used during the search is determined with `NumLevels`. If necessary, the number of levels is clipped to the range given when the shape model was created with `create_shape_model`. If `NumLevels` is set to 0, the number of pyramid levels specified in `create_shape_model` is used. Optionally, `NumLevels` can contain a second value that determines the lowest pyramid level to which the found matches are tracked. Hence, a value of [4,2] for `NumLevels` means that the matching starts at the fourth pyramid level and tracks the matches to the second lowest pyramid level (the lowest pyramid level is denoted by a value of 1). This mechanism can be used to decrease the runtime of the matching. It should be noted, however, that in general the accuracy of the extracted pose parameters is lower in this mode than in the normal mode, in which the matches are tracked to the lowest pyramid level. Hence, if a high accuracy is desired, `SubPixel` should be set to at least 'least_squares'. If the lowest pyramid level to use is chosen too large, it may happen that the desired accuracy cannot be achieved, or that wrong instances of the model are found because the model is not specific enough on the higher pyramid levels to facilitate a reliable selection of the correct instance of the model. In this case, the lowest pyramid level to use must be set to a smaller value.

The parameter `Greediness` determines how "greedily" the search should be carried out. If `Greediness` = 0, a safe search heuristic is used, which always finds the model if it is visible in the image. However, the search will be relatively time consuming in this case. If `Greediness` = 1, an unsafe search heuristic is used, which may cause the model not to be found in rare cases, even though it is visible in the image. For `Greediness` = 1, the maximum search speed is achieved. In almost all cases, the shape model will always be found for `Greediness` = 0.9.

Parameter	
▷ Image (input_object)	image $\rightsquigarrow$ <i>Hobject</i> : byte Input image in which the model should be found.
▷ ModelID (input_control)	shape_model $\rightsquigarrow$ integer Handle of the model.
▷ AngleStart (input_control)	angle.rad $\rightsquigarrow$ real Smallest rotation of the model. Default Value : -0.39 Suggested values : AngleStart $\in \{-3.14, -1.57, -0.78, -0.39, -0.20, 0.0\}$
▷ AngleExtent (input_control)	angle.rad $\rightsquigarrow$ real Extent of the rotation angles. Default Value : 0.78 Suggested values : AngleExtent $\in \{6.28, 3.14, 1.57, 0.78, 0.39, 0.0\}$ Restriction : AngleExtent ≥ 0
▷ MinScore (input_control)	real $\rightsquigarrow$ real Minimum score of the instances of the model to be found. Default Value : 0.5 Suggested values : MinScore $\in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$ Typical range of values : $0 \leq \text{MinScore} \leq 1$ Minimum Increment : 0.01 Recommended Increment : 0.05
▷ NumMatches (input_control)	integer $\rightsquigarrow$ integer Number of instances of the model to be found. Default Value : 1 Suggested values : NumMatches $\in \{0, 1, 2, 3, 4, 5, 10, 20\}$
▷ MaxOverlap (input_control)	real $\rightsquigarrow$ real Maximum overlap of the instances of the model to be found. Default Value : 0.5 Suggested values : MaxOverlap $\in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$ Typical range of values : $0 \leq \text{MaxOverlap} \leq 1$ Minimum Increment : 0.01 Recommended Increment : 0.05

- ▷ **SubPixel** (input_control) string $\leadsto$ *string*
Subpixel accuracy if not equal to 'none'.
Default Value : 'interpolation'
List of values : SubPixel $\in$ {'none', 'interpolation', 'least_squares', 'least_squares_high', 'least_squares_very_high'}
- ▷ **NumLevels** (input_control) integer(-array) $\leadsto$ *integer*
Number of pyramid levels used in the matching (and lowest pyramid level to use if |NumLevels| = 2).
Default Value : 0
List of values : NumLevels $\in$ {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
- ▷ **Greediness** (input_control) real $\leadsto$ *real*
"Greediness" of the search heuristic (0: safe but slow; 1: fast but matches may be missed).
Default Value : 0.9
Suggested values : Greediness $\in$ {0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
Typical range of values : $0 \leq \text{Greediness} \leq 1$
Minimum Increment : 0.01
Recommended Increment : 0.05
- ▷ **Row** (output_control) point.y-array $\leadsto$ *real*
Row coordinate of the found instances of the model.
- ▷ **Column** (output_control) point.x-array $\leadsto$ *real*
Column coordinate of the found instances of the model.
- ▷ **Angle** (output_control) angle.rad-array $\leadsto$ *real*
Rotation angle of the found instances of the model.
- ▷ **Score** (output_control) real-array $\leadsto$ *real*
Score of the found instances of the model.

Result

If the parameter values are correct, the operator `find_shape_model` returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via `set_system('no_object_result', <Result>)`. If necessary, an exception is raised.

Parallelization Information

`find_shape_model` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`create_shape_model`, `read_shape_model`, `set_shape_model_origin`

Alternatives

`find_scaled_shape_model`, `find_scaled_shape_models`, `find_shape_models`, `best_match_rotmg`

Module

Template matching

find_shape_models (Image : : ModelIDs, AngleStart, AngleExtent, MinScore, NumMatches, MaxOverlap, SubPixel, NumLevels, Greediness : Row, Column, Angle, Score, Model)

Find the best matches of multiple shape models.

The operator `find_shape_models` finds the best `NumMatches` instances of the shape models that are passed in the tuple `ModelIDs` in the input image `Image`. The models must have been created previously by calling `create_shape_model` or `read_shape_model`.

Hence, in contrast to `find_shape_model`, multiple models can be searched in the same image in one call. This changes the semantics of all input parameters to some extent. All input parameters must either contain one element, in which case the parameter is used for all models, or must contain the same number of elements as `ModelIDs`, in which case each parameter element refers to the corresponding element in `ModelIDs`. (`NumLevels` may also contain either two or twice the number of elements as `ModelIDs`; see below.) As usual, the domain of the input image `Image` is used to restrict the search space for the reference point of the models `ModelIDs`. Consistent with the above semantics, the input image `Image` can therefore contain a single image object or an image object tuple containing multiple image objects. If `Image` contains a single image object, its domain is used as the region

of interest for all models in `ModelIDs`. If `Image` contains multiple image objects, each domain is used as the region of interest for the corresponding model in `ModelIDs`. In this case, the image matrix of all image objects in the tuple must be identical, i.e., `Image` cannot be constructed in an arbitrary manner using `concat_obj`, but must be created from the same image using `add_channels` or equivalent calls. If this is not the case, an error message is returned. The above semantics also hold for the input control parameters. Hence, for example, `MinScore` can contain a single value or the same number of values as `ModelIDs`. In the first case, the value of `MinScore` is used for all models in `ModelIDs`, while in the second case the respective value of the elements in `MinScore` is used for the corresponding model in `ModelIDs`. An extension to these semantics holds for `NumMatches` and `MaxOverlap`. If `NumMatches` contains one element, `find_shape_models` returns the best `NumMatches` instances of the model irrespective of the type of the model. If, for example, two models are passed in `ModelIDs` and `NumMatches = 2` is selected, it can happen that two instances of the first model and no instances of the second model, one instance of the first model and one instance of the second model, or no instances of the first model and two instances of the second model are returned. If, on the other hand, `NumMatches` contains multiple values, the number of instances returned of the different models corresponds to the number specified in the respective entry in `NumMatches`. If, for example, `NumMatches = [1, 1]` is selected, one instance of the first model and one instance of the second model is returned. For a detailed description of the semantics of `NumMatches`, see below. A similar extension of the semantics holds for `MaxOverlap`. If a single value is passed for `MaxOverlap`, the overlap is computed for all found instances of the different models, irrespective of the model type, i.e., instances of the same or of different models that overlap too much are eliminated. If, on the other hand, multiple values are passed in `MaxOverlap`, the overlap is only computed for found instances of the model that have the same model type, i.e., only instances of the same model that overlap too much are eliminated. In this mode, models of different types may overlap completely. For a detailed description of the semantics of `MaxOverlap`, see below. Hence, a call to `find_shape_models` with multiple values for `ModelIDs`, `NumMatches` and `MaxOverlap` has the same effect as multiple independent calls to `find_shape_model` with the respective parameters. However, a single call to `find_shape_models` is considerably more efficient.

The type of the found instances of the models is returned in `Model`. The elements of `Model` are indices into the tuple `ModelIDs`, i.e., they can contain values from 0 to $|\text{ModelIDs}| - 1$. Hence, a value of 0 in an element of `Model` corresponds to an instance of the first model in `ModelIDs`.

The position and rotation of the found instances of the model is returned in `Row`, `Column` and `Angle`. The coordinates `Row` and `Column` are the coordinates of the origin of the shape model in the search image. By default, the origin is the center of gravity of the domain (region) of the image that was used to create the shape model with `create_shape_model`. A different origin can be set with `set_shape_model_origin`. Additionally, the score of each found instance is returned in `Score`. The score is a number between 0 and 1, which is an approximate measure of how much of the model is visible in the image. If, for example, half of the model is occluded, the score cannot exceed 0.5.

The domain of the image `Image` determines the search space for the reference point of the model, i.e., for the center of gravity of the domain (region) of the image that was used to create the shape model with `create_shape_model`. A different origin set with `set_shape_model_origin` is not taken into account. The model is searched within those points of the domain of the image, in which the model lies completely within the image. This means that the model will not be found if it extends beyond the borders of the image, even if it would achieve a score greater than `MinScore` (see below). The parameters `AngleStart` and `AngleExtent` determine the range of rotations for which the model is searched. If necessary, the range of rotations is clipped to the range given when the model was created with `create_shape_model`. In particular, this means that the angle ranges of the model and the search must truly overlap. The angle range in the search is not adapted modulo 2π . To simplify the presentation, all angles in the remainder of the paragraph are given in degrees, whereas they have to be specified in radians in `find_shape_models`. Hence, if the model, for example, was created with `AngleStart = -20°` and `AngleExtent = 40°` and the angle search space in `find_shape_models` is, for example, set to `AngleStart = 350°` and `AngleExtent = 20°`, the model will not be found, even though the angle ranges would overlap if they were regarded modulo $360°$. To find the model, in this example it would be necessary to select `AngleStart = -10°`.

The parameter `MinScore` determines what score a potential match must at least have to be regarded as an instance of the model in the image. The larger `MinScore` is chosen, the faster the search is. If the model can be expected never to be occluded in the images, `MinScore` may be set as high as 0.8 or even 0.9.

The maximum number of instances to be found can be determined with `NumMatches`. If more than `NumMatches` instances with a score greater than `MinScore` are found in the image, only the best `NumMatches` instances are returned. If fewer than `NumMatches` are found, only that number is returned, i.e., the parameter `MinScore` takes precedence over `NumMatches`.

If the model exhibits symmetries it may happen that multiple instances with similar positions but different rotations are found in the image. The parameter `MaxOverlap` determines by what fraction (i.e., a number between 0 and 1) two instances may at most overlap in order to consider them as different instances, and hence to be returned separately. If two instances overlap each other by more than `MaxOverlap` only the best instance is returned. The calculation of the overlap is based on the smallest enclosing rectangle of arbitrary orientation (see `smallest_rectangle2`) of the found instances. If `MaxOverlap` = 0, the found instances may not overlap at all, while for `MaxOverlap` = 1 all instances are returned.

The parameter `SubPixel` determines whether the instances should be extracted with subpixel accuracy. If `SubPixel` is set to 'none' (or 'false' for backwards compatibility) the model's pose is only determined with pixel accuracy and the angle resolution that was specified with `create_shape_model`. If `SubPixel` is set to 'interpolation' (or 'true') the position as well as the rotation are determined with subpixel accuracy. In this mode, the model's pose is interpolated from the score function. This mode costs almost no computation time and achieves an accuracy that is high enough for most applications. In some applications, however, the accuracy requirements are extremely high. In these cases, the model's pose can be determined through a least-squares adjustment, i.e., by minimizing the distances of the model points to their corresponding image points. In contrast to 'interpolation', this mode requires additional computation time. The different modes for least-squares adjustment ('least_squares', 'least_squares_high', and 'least_squares_very_high') can be used to determine the accuracy with which the minimum distance is being searched. The higher the accuracy is chosen, the longer the subpixel extraction will take, however. Usually, `SubPixel` should be set to 'interpolation'. If least-squares adjustment is desired, 'least_squares' should be chosen because this results in the best tradeoff between run time and accuracy.

The number of pyramid levels used during the search is determined with `NumLevels`. If necessary, the number of levels is clipped to the range given when the shape model was created with `create_shape_model`. If `NumLevels` is set to 0, the number of pyramid levels specified in `create_shape_model` is used. Optionally, `NumLevels` can contain a second value that determines the lowest pyramid level to which the found matches are tracked. Hence, a value of [4,2] for `NumLevels` means that the matching starts at the fourth pyramid level and tracks the matches to the second lowest pyramid level (the lowest pyramid level is denoted by a value of 1). This mechanism can be used to decrease the runtime of the matching. It should be noted, however, that in general the accuracy of the extracted pose parameters is lower in this mode than in the normal mode, in which the matches are tracked to the lowest pyramid level. Hence, if a high accuracy is desired, `SubPixel` should be set to at least 'least_squares'. If the lowest pyramid level to use is chosen too large, it may happen that the desired accuracy cannot be achieved, or that wrong instances of the model are found because the model is not specific enough on the higher pyramid levels to facilitate a reliable selection of the correct instance of the model. In this case, the lowest pyramid level to use must be set to a smaller value. If the lowest pyramid level is specified separately for each model, `NumLevels` must contain twice the number of elements as `ModelIDs`. In this case, the number of pyramid levels and the lowest pyramid level must be specified interleaved in `NumLevels`. If, for example, two models are specified in `ModelIDs`, the number of pyramid levels is 5 for the first model and 4 for the second model, and the lowest pyramid level is 2 for the first model and 1 for the second model, `NumLevels` = [5, 2, 4, 1] must be selected. If exactly two models are specified in `ModelIDs`, a special case occurs. If in this case the lowest pyramid level is to be specified, the number of pyramid levels and the lowest pyramid level must be specified explicitly for both models, even if they are identical, because specifying two values in `NumLevels` is interpreted as the explicit specification of the number of pyramid levels for the two models.

The parameter `Greediness` determines how "greedily" the search should be carried out. If `Greediness` = 0, a safe search heuristic is used, which always finds the model if it is visible in the image. However, the search will be relatively time consuming in this case. If `Greediness` = 1, an unsafe search heuristic is used, which may cause the model not to be found in rare cases, even though it is visible in the image. For `Greediness` = 1, the maximum search speed is achieved. In almost all cases, the shape model will always be found for `Greediness` = 0.9.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte
Input image in which the models should be found.
- ▷ **ModelIDs** (input_control) shape_model(-array) $\leadsto$ *integer*
Handle of the models.
- ▷ **AngleStart** (input_control) angle.rad(-array) $\leadsto$ *real*
Smallest rotation of the models.
Default Value : -0.39
Suggested values : AngleStart $\in \{-3.14, -1.57, -0.78, -0.39, -0.20, 0.0\}$

- ▷ **AngleExtent** (input_control) angle.rad(-array) $\leadsto$ *real*
Extent of the rotation angles.
Default Value : 0.78
Suggested values : AngleExtent $\in \{6.28, 3.14, 1.57, 0.78, 0.39, 0.0\}$
Restriction : AngleExtent ≥ 0
- ▷ **MinScore** (input_control) real(-array) $\leadsto$ *real*
Minimum score of the instances of the models to be found.
Default Value : 0.5
Suggested values : MinScore $\in \{0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
Typical range of values : $0 \leq \text{MinScore} \leq 1$
Minimum Increment : 0.01
Recommended Increment : 0.05
- ▷ **NumMatches** (input_control) integer(-array) $\leadsto$ *integer*
Number of instances of the models to be found.
Default Value : 1
Suggested values : NumMatches $\in \{0, 1, 2, 3, 4, 5, 10, 20\}$
- ▷ **MaxOverlap** (input_control) real(-array) $\leadsto$ *real*
Maximum overlap of the instances of the models to be found.
Default Value : 0.5
Suggested values : MaxOverlap $\in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
Typical range of values : $0 \leq \text{MaxOverlap} \leq 1$
Minimum Increment : 0.01
Recommended Increment : 0.05
- ▷ **SubPixel** (input_control) string(-array) $\leadsto$ *string*
Subpixel accuracy if not equal to 'none'.
Default Value : 'interpolation'
List of values : SubPixel $\in \{'none', 'interpolation', 'least_squares', 'least_squares_high', 'least_squares_very_high'\}$
- ▷ **NumLevels** (input_control) integer(-array) $\leadsto$ *integer*
Number of pyramid levels used in the matching (and lowest pyramid level to use if |NumLevels| = 2).
Default Value : 0
List of values : NumLevels $\in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
- ▷ **Greediness** (input_control) real(-array) $\leadsto$ *real*
"Greediness" of the search heuristic (0: safe but slow; 1: fast but matches may be missed).
Default Value : 0.9
Suggested values : Greediness $\in \{0.0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0\}$
Typical range of values : $0 \leq \text{Greediness} \leq 1$
Minimum Increment : 0.01
Recommended Increment : 0.05
- ▷ **Row** (output_control) point.y-array $\leadsto$ *real*
Row coordinate of the found instances of the models.
- ▷ **Column** (output_control) point.x-array $\leadsto$ *real*
Column coordinate of the found instances of the models.
- ▷ **Angle** (output_control) angle.rad-array $\leadsto$ *real*
Rotation angle of the found instances of the models.
- ▷ **Score** (output_control) real-array $\leadsto$ *real*
Score of the found instances of the models.
- ▷ **Model** (output_control) integer-array $\leadsto$ *integer*
Index of the found instances of the models.

Result

If the parameter values are correct, the operator `find_shape_models` returns the value 2 (H_MSG_TRUE). If the input is empty (no input images are available) the behavior can be set via `set_system('no_object_result', <Result>)`. If necessary, an exception is raised.

Parallelization Information

`find_shape_models` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`add_channels`, `create_shape_model`, `read_shape_model`, `set_shape_model_origin`

Alternatives

[find_scaled_shape_models](#), [find_shape_model](#), [find_scaled_shape_model](#),
[best_match_rotmg](#)

Module

Template matching

get_shape_model_origin (: : ModelID : Row, Column)

Return the origin (reference point) of a shape model.

The operator [get_shape_model_origin](#) returns the origin (reference point) of the shape model [ModelID](#). The origin is specified relative to the center of gravity of the domain (region) of the image that was used to create the shape model with [create_shape_model](#) or [create_scaled_shape_model](#). Hence, an origin of (0,0) means that the center of gravity of the domain of the model image is used as the origin. An origin of (-20,-40) means that the origin lies to the upper left of the center of gravity.

Parameter

- ▷ **ModelID** (input_control) shape_model $\leadsto$ integer
Handle of the model.
- ▷ **Row** (output_control) point.y $\leadsto$ real
Row coordinate of the origin of the shape model.
- ▷ **Column** (output_control) point.x $\leadsto$ real
Column coordinate of the origin of the shape model.

Result

If the handle of the model is valid, the operator [get_shape_model_origin](#) returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Parallelization Information

[get_shape_model_origin](#) is processed *completely exclusively* without parallelization.

Possible Predecessors

[create_shape_model](#), [create_scaled_shape_model](#), [read_shape_model](#),
[set_shape_model_origin](#)

Possible Successors

[find_shape_model](#), [find_scaled_shape_model](#), [find_shape_models](#),
[find_scaled_shape_models](#)

See also

[area_center](#)

Module

Template matching

get_shape_model_params (: : ModelID : NumLevels, AngleStart, AngleExtent, AngleStep, ScaleMin, ScaleMax, ScaleStep, Metric, MinContrast)

Return the parameters of a shape model.

The operator [get_shape_model_params](#) returns the parameters of the shape model [ModelID](#) that were used to create it using [create_shape_model](#) or [create_scaled_shape_model](#). In particular, this output can be used to check the parameters [NumLevels](#), [AngleStep](#) and [ScaleStep](#) if they were determined automatically during the model creation with [create_shape_model](#) or [create_scaled_shape_model](#).

Parameter

- ▷ **ModelID** (input_control) shape_model $\leadsto$ integer
Handle of the model.
- ▷ **NumLevels** (output_control) integer $\leadsto$ integer
Number of pyramid levels.

- ▷ **AngleStart** (output_control) angle.rad $\leadsto$ real
Smallest rotation of the pattern.
- ▷ **AngleExtent** (output_control) angle.rad $\leadsto$ real
Extent of the rotation angles.
Assertion : AngleExtent ≥ 0
- ▷ **AngleStep** (output_control) angle.rad $\leadsto$ real
Step length of the angles (resolution).
Assertion : AngleStep ≥ 0
- ▷ **ScaleMin** (output_control) number $\leadsto$ real
Minimum scale of the pattern.
Assertion : ScaleMin > 0
- ▷ **ScaleMax** (output_control) number $\leadsto$ real
Maximum scale of the pattern.
Assertion : ScaleMax $\geq$ ScaleMin
- ▷ **ScaleStep** (output_control) number $\leadsto$ real
Scale step length (resolution).
Assertion : ScaleStep ≥ 0
- ▷ **Metric** (output_control) string $\leadsto$ string
Match metric.
- ▷ **MinContrast** (output_control) number $\leadsto$ integer
Minimum contrast of the objects in the search images.

Result

If the handle of the model is valid, the operator `get_shape_model_params` returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Parallelization Information

`get_shape_model_params` is processed *completely exclusively* without parallelization.

Possible Predecessors

`create_shape_model`, `create_scaled_shape_model`, `read_shape_model`

See also

`find_shape_model`, `find_scaled_shape_model`, `find_shape_models`,
`find_scaled_shape_models`

Module

Template matching

inspect_shape_model (Image : ModelImages, ModelRegions : NumLevels,
Contrast :)

Create the representation of a shape model.

`inspect_shape_model` creates a representation of a shape model. The operator is particularly useful in order to determine the parameters `NumLevels` and `Contrast`, which are used in `create_shape_model`, quickly and conveniently. The representation of the model is created on multiple image pyramid levels, where the number of levels is determined by `NumLevels`. In contrast to `create_shape_model`, the model is only created for the rotation of the object in the input image, i.e., 0°. As output, `inspect_shape_model` creates an image object `ModelImages` containing the images of the individual levels of the image pyramid as well as a region in `ModelRegions` for each pyramid level that represents the model at the respective pyramid level. The individual objects can be accessed with `select_obj`. As described for `create_shape_model`, the number of pyramid levels should be chosen as large as possible, while taking into account that the model must be recognizable on the highest pyramid level and must have enough model points. The parameter `Contrast` should be chosen such that only the significant features of the template object are used for the model. `Contrast` can also contain a tuple with two values. In this case, the model is segmented using a method similar to the hysteresis threshold method used in `edges_image`. Here, the first element of the tuple determines the lower threshold, while the second element determines the upper threshold. For more information about the hysteresis threshold method, see `hysteresis_threshold`. Optionally, `Contrast` can contain a third value as the last element of the tuple. This value determines a threshold for the selection of significant model components based on the size of the components, i.e., components that have fewer points than the minimum size thus specified are suppressed. This

threshold for the minimum size is divided by two for each successive pyramid level. If small model components should be suppressed, but hysteresis thresholding should not be performed, nevertheless three values must be specified in `Contrast`. In this case, the first two values can simply be set to identical values. In its typical use, `inspect_shape_model` is called interactively multiple times with different parameters for `NumLevels` and `Contrast`, until a satisfactory model is obtained. After this, `create_shape_model` is called with the parameters thus obtained.

Parameter	
▷ Image (input_object)	image $\rightsquigarrow$ Hobject : byte Input image.
▷ ModelImages (output_object)	image-array $\rightsquigarrow$ Hobject : byte Image pyramid of the input image
▷ ModelRegions (output_object)	region-array $\rightsquigarrow$ Hobject Model region pyramid
▷ NumLevels (input_control)	integer $\rightsquigarrow$ integer Number of pyramid levels. Default Value : 4 List of values : NumLevels $\in$ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
▷ Contrast (input_control)	number(-array) $\rightsquigarrow$ integer Threshold or hysteresis thresholds for the contrast of the object in the image and optionally minimum size of the object parts. Default Value : 30 Suggested values : Contrast $\in$ {10, 20, 30, 40, 60, 80, 100, 120, 140, 160}
Result	
If the parameters are valid, the operator <code>inspect_shape_model</code> returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.	
Parallelization Information	
<code>inspect_shape_model</code> is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Predecessors	
<code>reduce_domain</code>	
Possible Successors	
<code>select_obj</code>	
See also	
<code>create_shape_model</code>	
Module	
Template matching	

read_shape_model (: : FileName : ModelID)

Read a shape model from a file.

The operator `read_shape_model` reads a shape model, which has been written with `write_shape_model`, from the file `FileName`.

Parameter	
▷ FileName (input_control)	filename $\rightsquigarrow$ string File name.
▷ ModelID (output_control)	shape_model $\rightsquigarrow$ integer Handle of the model.
Result	
If the file name is valid, the operator <code>read_shape_model</code> returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.	
Parallelization Information	
<code>read_shape_model</code> is processed <i>completely exclusively</i> without parallelization.	

Possible Successors

[find_shape_model](#), [find_scaled_shape_model](#), [find_shape_models](#),
[find_scaled_shape_models](#)

See also

[create_shape_model](#), [clear_shape_model](#)

Module

Template matching

set_shape_model_origin (: : ModelID, Row, Column :)

Set the origin (reference point) of a shape model.

The operator [set_shape_model_origin](#) sets the origin (reference point) of the shape model [ModelID](#) to a new value. The origin is specified relative to the center of gravity of the domain (region) of the image that was used to create the shape model with [create_shape_model](#) or [create_scaled_shape_model](#). Hence, an origin of (0,0) means that the center of gravity of the domain of the model image is used as the origin. An origin of (-20,-40) means that the origin lies to the upper left of the center of gravity.

Parameter

- ▷ **ModelID** (input_control) shape_model $\leadsto$ integer
Handle of the model.
- ▷ **Row** (input_control) point.y $\leadsto$ real
Row coordinate of the origin of the shape model.
- ▷ **Column** (input_control) point.x $\leadsto$ real
Column coordinate of the origin of the shape model.

Result

If the handle of the model is valid, the operator [set_shape_model_origin](#) returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Parallelization Information

[set_shape_model_origin](#) is processed *completely exclusively* without parallelization.

Possible Predecessors

[create_shape_model](#), [create_scaled_shape_model](#), [read_shape_model](#)

Possible Successors

[find_shape_model](#), [find_scaled_shape_model](#), [find_shape_models](#),
[find_scaled_shape_models](#), [get_shape_model_origin](#)

See also

[area_center](#)

Module

Template matching

write_shape_model (: : ModelID, FileName :)

Write a shape model to a file.

The operator [write_shape_model](#) writes a shape model to the file [FileName](#). The model can be read again with [read_shape_model](#).

Parameter

- ▷ **ModelID** (input_control) shape_model $\leadsto$ integer
Handle of the model.
- ▷ **FileName** (input_control) filename $\leadsto$ string
File name.

Result

If the file name is valid (write permission), the operator `write_shape_model` returns the value 2 (H_MSG_TRUE). If necessary an exception is raised.

Parallelization Information

`write_shape_model` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`create_shape_model`

Module

Template matching

Chapter 10

Morphology

10.1 Gray-Values

dual_rank (Image : ImageRank : MaskType, Radius, ModePercent, Margin :)

Opening, Median and Closing with circle or rectangle mask.

The operator **dual_rank** carries out a non-linear transformation of the gray values of all input images (**Image**). Circles or squares can be used as structuring elements. The operator **dual_rank** effects two consecutive calls of **rank_image**. At the first call the range gray value is calculated with the indicated range (**ModePercent**). The result of this calculation is the input of a further call of **rank_image**, this time using the range value 100–**ModePercent**.

When filtering different parameters for border treatment (**Margin**) can be chosen:

gray value	Pixels outside of the image edges are assumed to be constant (with the indicated gray value).
'continued'	Continuation of edge pixels.
'cyclic'	Cyclic continuation of image edges.
'mirrored'	Reflection of pixels at the image edges.

A range filtering is calculated according to the following scheme: The indicated mask is put over the image to be filtered in such a way that the center of the mask touches all pixels once. For each of these pixels all neighboring pixels covered by the mask are sorted in an ascending sequence corresponding to their gray values. Each sorted sequence of gray values contains the same number of gray values like the mask has image points. The n-th highest element, (= **ModePercent**, rank values between 0...100 in percent) is selected and set as result gray value in the corresponding result image.

If **ModePercent** is 0, then the operator equals to the gray value opening (**gray_opening**). If **ModePercent** is 50, the operator results in the median filter, which is applied twice (**median_image**). The **ModePercent** 100 in **dual_rank** means that it calculates the gray value closing (**gray_closing**). Choosing parameter values inside this range results in a smooth transformation of these operators.

Parameter

- ▷ **Image** (input_object)(multichannel-)image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4 / real
Image to be filtered.
- ▷ **ImageRank** (output_object) multichannel-image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4 / real
Filtered Image.
- ▷ **MaskType** (input_control) string $\leadsto$ *string*
Shape of the mask.
Default Value : 'circle'
List of values : MaskType $\in$ {'circle', 'rectangle'}

- ▷ **Radius** (input_control) integer $\leadsto$ integer
 Radius of the filter mask.
Default Value : 1
Suggested values : Radius $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 15, 19, 25, 31, 39, 47, 59\}$
Typical range of values : $1 \leq \text{Radius} \leq 101$
Minimum Increment : 1
Recommended Increment : 2
- ▷ **ModePercent** (input_control) integer $\leadsto$ integer
 Filter Mode: 0 corresponds to a gray value opening, 50 corresponds to a median and 100 to a gray values closing.
Default Value : 10
Suggested values : ModePercent $\in \{0, 2, 5, 10, 15, 20, 40, 50, 60, 80, 85, 90, 95, 98, 100\}$
Typical range of values : $0 \leq \text{ModePercent} \leq 100$
Minimum Increment : 1
Recommended Increment : 2
- ▷ **Margin** (input_control) string $\leadsto$ string / integer / real
 Border treatment.
Default Value : 'mirrored'
Suggested values : Margin $\in \{\text{'mirrored'}, \text{'cyclic'}, \text{'continued'}, 0, 30, 60, 90, 120, 150, 180, 210, 240, 255\}$

Example

```
read_image(Image, 'fabrik')
dual_rank(Image, ImageOpening, 'circle', 10, 10, 'mirrored')
disp_image(ImageOpening, WindowHandle).
```

Complexity

For each pixel: $O(\sqrt{F} * 10)$ with F = area of the structuring element.

Result

If the parameter values are correct the operator `dual_rank` returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no input images available) is set via the operator `set_system('no_object_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`dual_rank` is *reentrant* and automatically *parallelized* (on tuple level, channel level, domain level).

Possible Predecessors

`read_image`

Possible Successors

`threshold`, `dyn_threshold`, `sub_image`, `regiongrowing`

Alternatives

`rank_image`, `gray_closing`, `gray_opening`, `median_image`

See also

`gen_circle`, `gen_rectangle1`, `gray_erosion_rect`, `gray_dilation_rect`, `sigma_image`

References

W. Eckstein, O. Munkelt "Extracting Objects from Digital Terrain Model" Remote Sensing and Reconstruction for Threedimensional Objects and Scenes, SPIE Symposium on Optical Science, Engineering, and Instrumentation, July 1995, San Diego

Module

Image filters

gen_disc_se (: SE : Width, Height, Smax :)

Generate ellipsoidal structuring elements for gray morphology.

`gen_disc_se` generates an ellipsoidal structuring element (SE) for gray morphology of images. The parameters `Width` and `Height` determine the length of the two major axes of the ellipse. The value of `Smax` determines

the maximum gray value of the structuring element. For the generation of arbitrary structuring elements, see [read_gray_se](#).

Parameter	
▷ SE (output_object)	image $\rightsquigarrow$ <i>Hobject</i> : byte Generated structuring element.
▷ Width (input_control)	integer $\rightsquigarrow$ <i>integer</i> Width of the structuring element. Default Value : 5 Suggested values : Width $\in \{0, 1, 2, 3, 4, 5, 10, 15, 20\}$ Typical range of values : $0 \leq \text{Width} \leq 511$ (lin) Minimum Increment : 1 Recommended Increment : 1
▷ Height (input_control)	integer $\rightsquigarrow$ <i>integer</i> Height of the structuring element. Default Value : 5 Suggested values : Height $\in \{0, 1, 2, 3, 4, 5, 10, 15, 20\}$ Typical range of values : $0 \leq \text{Height} \leq 511$ (lin) Minimum Increment : 1 Recommended Increment : 1
▷ Smax (input_control)	integer $\rightsquigarrow$ <i>integer</i> Maximum gray value of the structuring element. Default Value : 0 Suggested values : Smax $\in \{0, 1, 2, 5, 10, 20, 30, 40\}$ Typical range of values : $0 \leq \text{Smax} \leq 255$ (lin) Minimum Increment : 1 Recommended Increment : 1

Result
gen_disc_se returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception is raised.

Parallelization Information
gen_disc_se is <i>reentrant</i> and processed <i>without</i> parallelization.

Possible Successors
gray_erosion , gray_dilation , gray_opening , gray_closing , gray_tophat , gray_bothat

Alternatives
read_gray_se

See also
read_image , paint_region , paint_gray , crop_part

Module
Image filters

gray_bothat (Image, SE : ImageBotHat : :)

Perform a gray value bottom hat transformation on an image.

[gray_bothat](#) applies a gray value bottom hat transformation to the input image [Image](#) with the structuring element [SE](#). The gray value bottom hat transformation of an image i with a structuring element s is defined as

$$\text{bothat}(i, s) = (i \bullet s) - i,$$

i.e., the difference of the closing of the image with s and the image (see [gray_closing](#)). For the generation of structuring elements, see [read_gray_se](#).

Parameter
▷ Image (input_object) image(-array) $\leadsto$ Hobject : byte / real Input image. ▷ SE (input_object) image $\leadsto$ Hobject : byte Structuring element. ▷ ImageBotHat (output_object) image(-array) $\leadsto$ Hobject : byte / real Bottom hat image.
Result
gray_bothat returns 2 (H_MSG_TRUE) if the structuring element is not the empty region. Otherwise, an exception is raised.
Parallelization Information
gray_bothat is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).
Possible Predecessors
read_gray_se , gen_disc_se
Possible Successors
threshold
Alternatives
gray_closing
See also
gray_tophat , top_hat , gray_erosion_rect , sub_image
Module
Image filters

gray_closing (Image, SE : ImageClosing : :)

Perform a gray value closing on an image.

[gray_closing](#) applies a gray value closing to the input image [Image](#) with the structuring element [SE](#). The gray value closing of an image i with a structuring element s is defined as

$$i \bullet s = (i \oplus s) \ominus s ,$$

i.e., a dilation of the image with s followed by an erosion with s (see [gray_dilation](#) and [gray_erosion](#)). For the generation of structuring elements, see [read_gray_se](#).

Parameter
▷ Image (input_object) image(-array) $\leadsto$ Hobject : byte / real Input image. ▷ SE (input_object) image $\leadsto$ Hobject : byte Structuring element. ▷ ImageClosing (output_object) image(-array) $\leadsto$ Hobject : byte / real Gray-closed image.
Result
gray_closing returns 2 (H_MSG_TRUE) if the structuring element is not the empty region. Otherwise, an exception is raised.
Parallelization Information
gray_closing is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).
Possible Predecessors
read_gray_se
Alternatives
dual_rank
See also
closing , gray_dilation , gray_erosion

Module

Image filters

```
gray_closing_shape ( Image : ImageClosing : MaskHeight, MaskWidth,
MaskShape : )
```

Perform a grayvalue closing with a selected mask.

[gray_closing_shape](#) applies a gray value closing to the input image [Image](#) with the structuring element of shape [MaskShape](#). The mask's offset values are 0 and its horizontal and vertical size is defined by [MaskHeight](#) and [MaskWidth](#). The resulting image is returned in [ImageClosing](#). If the parameters [MaskHeight](#) or [MaskWidth](#) are even, they are changed to the next larger odd value. In case of the values 'rhombus' and 'octagon' for the [MaskShape](#) control parameter, [MaskHeight](#) and [MaskWidth](#) must be equal. The parameter value 'octagon' for [MaskShape](#) denotes an equilateral octagonal mask which is a suitable approximation for a circular structure. At the border of the image the gray values are mirrored.

The gray value closing of an image i with a structuring element s is defined as

$$i \bullet s = (i \oplus s) \ominus s ,$$

i.e., a dilation of the image with s followed by an erosion with s (see [gray_dilation_shape](#) and [gray_erosion_shape](#)).

Parameter

-
- ▷ **Image** (input_object) image(-array) $\rightsquigarrow$ Hobject : byte
Image for which the minimum gray values are to be calculated.
 - ▷ **ImageClosing** (output_object) image(-array) $\rightsquigarrow$ Hobject : byte
Image containing the minimum gray values.
 - ▷ **MaskHeight** (input_control) extent.y $\rightsquigarrow$ integer
Height of the filter mask.
Default Value : 11
Suggested values : MaskHeight $\in \{3, 5, 7, 9, 11, 13, 15\}$
Typical range of values : $3 \leq \text{MaskHeight} \leq 511$
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskHeight)
 - ▷ **MaskWidth** (input_control) extent.x $\rightsquigarrow$ integer
Width of the filter mask.
Default Value : 11
Suggested values : MaskWidth $\in \{3, 5, 7, 9, 11, 13, 15\}$
Typical range of values : $3 \leq \text{MaskWidth} \leq 511$
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskWidth)
 - ▷ **MaskShape** (input_control) string $\rightsquigarrow$ string
Shape of the mask.
Default Value : 'octagon'
List of values : MaskShape $\in \{\text{'rectangle'}, \text{'rhombus'}, \text{'octagon'}\}$
-

Result

[gray_closing_shape](#) returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

[gray_closing_shape](#) is *reentrant* and automatically *parallelized* (on tuple level, channel level, domain level).

Alternatives

[gray_closing](#)

See also

[gray_dilation_shape](#), [gray_erosion_shape](#), [closing](#)

Module

Image filters

gray_dilation (Image, SE : ImageDilation : :)

Perform a gray value dilation on an image.

gray_dilation applies a gray value dilation to the input image **Image** with the structuring element **SE**. The gray value dilation of an image i with a structuring element s at the pixel position x is defined as:

$$(i \oplus s)(x) = \max\{f(x - z) + s(z) | z \in S\}$$

Here, S is the domain of the structuring element s , i.e., the pixels z where $s(z) > 0$ (see [read_gray_se](#)).

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte / real
Input image.
- ▷ **SE** (input_object) image $\leadsto$ *Hobject* : byte
Structuring element.
- ▷ **ImageDilation** (output_object) image(-array) $\leadsto$ *Hobject* : byte / real
Gray-dilated image.

Result

gray_dilation returns 2 (H_MSG_TRUE) if the structuring element is not the empty region. Otherwise, an exception is raised.

Parallelization Information

gray_dilation is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

[read_gray_se](#)

Possible Successors

[sub_image](#), [gray_erosion](#)

Alternatives

[gray_dilation_rect](#)

See also

[gray_opening](#), [gray_closing](#), [dilation1](#), [gray_skeleton](#)

Module

Image filters

gray_dilation_rect (Image : ImageMax : MaskHeight, MaskWidth :)

Determine the maximum gray value within a rectangle.

gray_dilation_rect calculates the maximum gray value of the input image **Image** within a rectangular mask of size (**MaskHeight**, **MaskWidth**) for each image point. The resulting image is returned in **ImageMax**. If the parameters **MaskHeight** or **MaskWidth** are even, they are changed to the next larger odd value. At the border of the image the gray values are mirrored.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int2 / int4 / real
Image for which the maximum gray values are to be calculated.
- ▷ **ImageMax** (output_object) image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int2 / int4 / real
Image containing the maximum gray values.
- ▷ **MaskHeight** (input_control) extent.y $\leadsto$ *integer*
Height of the filter mask.
Default Value : 11
Suggested values : MaskHeight $\in \{3, 5, 7, 9, 11, 13, 15\}$
Typical range of values : $3 \leq \text{MaskHeight} \leq 511$
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskHeight)

- ▷ **MaskWidth** (input_control) extent.x $\leadsto$ *integer*
Width of the filter mask.
Default Value : 11
Suggested values : MaskWidth $\in \{3, 5, 7, 9, 11, 13, 15\}$
Typical range of values : $3 \leq \text{MaskWidth} \leq 511$
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskWidth)

Result

`gray_dilation_rect` returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via `set_system('no_object_result', <Result>)`. If necessary, an exception is raised.

Parallelization Information

`gray_dilation_rect` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

See also

`gray_skeleton`

Module

Image filters

gray_dilation_shape (Image : ImageMax : MaskHeight, MaskWidth,
MaskShape :)

Determine the maximum gray value within a selected mask.

`gray_dilation_shape` calculates the maximum gray value of the input image `Image` within a mask of shape `MaskShape`, vertical size `MaskHeight` and horizontal size `MaskWidth` for each image point. The resulting image is returned in `ImageMax`. If the parameters `MaskHeight` or `MaskWidth` are even, they are changed to the next larger odd value. In case of the values 'rhombus' und 'octagon' for the `MaskShape` control parameter, `MaskHeight` and `MaskWidth` must be equal. The parameter value 'octagon' for `MaskShape` denotes an equilateral octagonal mask which is a suitable approximation for a circular structure. At the border of the image the gray values are mirrored.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte
Image for which the maximum gray values are to be calculated.
- ▷ **ImageMax** (output_object) image(-array) $\leadsto$ *Hobject* : byte
Image containing the maximum gray values.
- ▷ **MaskHeight** (input_control) extent.y $\leadsto$ *integer*
Height of the filter mask.
Default Value : 11
Suggested values : MaskHeight $\in \{3, 5, 7, 9, 11, 13, 15\}$
Typical range of values : $3 \leq \text{MaskHeight}$
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskHeight)
- ▷ **MaskWidth** (input_control) extent.x $\leadsto$ *integer*
Width of the filter mask.
Default Value : 11
Suggested values : MaskWidth $\in \{3, 5, 7, 9, 11, 13, 15\}$
Typical range of values : $3 \leq \text{MaskWidth}$
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskWidth)
- ▷ **MaskShape** (input_control) string $\leadsto$ *string*
Shape of the mask.
Default Value : 'octagon'
List of values : MaskShape $\in \{\text{'rectangle'}, \text{'rhombus'}, \text{'octagon'}\}$

Result

[gray_dilation_shape](#) returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

[gray_dilation_shape](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Alternatives

[gray_dilation](#), [gray_dilation_rect](#)

See also

[gray_opening_shape](#), [gray_closing_shape](#), [gray_skeleton](#)

Module

Image filters

gray_erosion (Image, SE : ImageErosion : :)

Perform a gray value erosion on an image.

[gray_erosion](#) applies a gray value erosion to the input image [Image](#) with the structuring element [SE](#). The gray value erosion of an image i with a structuring element s at the pixel position x is defined as:

$$(i \ominus s)(x) = \min\{f(x+z) - s(z) | z \in S\}$$

Here, S is the domain of the structuring element s , i.e., the pixels z where $s(z) > 0$ (see [read_gray_se](#)).

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte / real
Input image.
- ▷ **SE** (input_object) image $\leadsto$ *Hobject* : byte
Structuring element.
- ▷ **ImageErosion** (output_object) image(-array) $\leadsto$ *Hobject* : byte / real
Gray-eroded image.

Result

[gray_erosion](#) returns 2 (H_MSG_TRUE) if the structuring element is not the empty region. Otherwise, an exception is raised.

Parallelization Information

[gray_erosion](#) is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

[read_gray_se](#)

Possible Successors

[gray_dilation](#), [sub_image](#)

Alternatives

[gray_erosion_rect](#)

See also

[gray_opening](#), [gray_closing](#), [erosion1](#), [gray_skeleton](#)

Module

Image filters

gray_erosion_rect (Image : ImageMin : MaskHeight, MaskWidth :)

Determine the minimum gray value within a rectangle.

[gray_erosion_rect](#) calculates the minimum gray value of the input image [Image](#) within a rectangular mask of size ([MaskHeight](#), [MaskWidth](#)) for each image point. The resulting image is returned in [ImageMin](#). If the

parameters **MaskHeight** or **MaskWidth** are even, they are changed to the next larger odd value. At the border of the image the gray values are mirrored.

Parameter
▷ Image (input_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int2 / int4 / real Image for which the minimum gray values are to be calculated. ▷ ImageMin (output_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int2 / int4 / real Image containing the minimum gray values. ▷ MaskHeight (input_control) extent.y $\leadsto$ <i>integer</i> Height of the filter mask. Default Value : 11 Suggested values : MaskHeight $\in \{3, 5, 7, 9, 11, 13, 15\}$ Typical range of values : $3 \leq \text{MaskHeight} \leq 511$ (lin) Minimum Increment : 2 Recommended Increment : 2 Restriction : odd(MaskHeight) ▷ MaskWidth (input_control) extent.x $\leadsto$ <i>integer</i> Width of the filter mask. Default Value : 11 Suggested values : MaskWidth $\in \{3, 5, 7, 9, 11, 13, 15\}$ Typical range of values : $3 \leq \text{MaskWidth} \leq 511$ (lin) Minimum Increment : 2 Recommended Increment : 2 Restriction : odd(MaskWidth)

Result
gray_erosion_rect returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via **set_system('no_object_result', <Result>)**. If necessary, an exception is raised.

Parallelization Information
gray_erosion_rect is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

See also
gray_dilation_rect

Module

Image filters

gray_erosion_shape (Image : ImageMin : MaskHeight, MaskWidth, MaskShape :)

Determine the minimum gray value within a selected mask.

gray_dilation_shape calculates the minimum gray value of the input image **Image** within a mask of shape **MaskShape**, vertical size **MaskHeight** and horizontal size **MaskWidth** for each image point. The resulting image is returned in **ImageMin**. If the parameters **MaskHeight** or **MaskWidth** are even, they are changed to the next larger odd value. In case of the values 'rhombus' and 'octagon' for the **MaskShape** control parameter, **MaskHeight** and **MaskWidth** must be equal. The parameter value 'octagon' for **MaskShape** denotes an equilateral octagonal mask which is a suitable approximation for a circular structure. At the border of the image the gray values are mirrored.

Parameter
▷ Image (input_object) image(-array) $\leadsto$ <i>Hobject</i> : byte Image for which the minimum gray values are to be calculated. ▷ ImageMin (output_object) image(-array) $\leadsto$ <i>Hobject</i> : byte Image containing the minimum gray values.

- ▷ **MaskHeight** (input_control) extent.y $\leadsto$ integer
Height of the filter mask.
Default Value : 11
Suggested values : MaskHeight $\in \{3, 5, 7, 9, 11, 13, 15\}$
Typical range of values : $3 \leq \text{MaskHeight}$
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskHeight)
- ▷ **MaskWidth** (input_control) extent.x $\leadsto$ integer
Width of the filter mask.
Default Value : 11
Suggested values : MaskWidth $\in \{3, 5, 7, 9, 11, 13, 15\}$
Typical range of values : $3 \leq \text{MaskWidth}$
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskWidth)
- ▷ **MaskShape** (input_control) string $\leadsto$ string
Shape of the mask.
Default Value : 'octagon'
List of values : MaskShape $\in \{\text{'rectangle'}, \text{'rhombus'}, \text{'octagon'}\}$

Result

[gray_dilation_shape](#) returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

[gray_erosion_shape](#) is *reentrant* and automatically *parallelized* (on tuple level, channel level, domain level).

Alternatives

[gray_erosion](#), [gray_erosion_rect](#)

See also

[gray_opening_shape](#), [gray_closing_shape](#), [gray_skeleton](#)

Module

Image filters

gray_opening (Image, SE : ImageOpening : :)

Perform a gray value opening on an image.

[gray_opening](#) applies a gray value opening to the input image [Image](#) with the structuring element [SE](#). The gray value opening of an image i with a structuring element s is defined as

$$i \circ s = (i \ominus s) \oplus s ,$$

i.e., an erosion of the image with s followed by a dilation with s (see [gray_erosion](#) and [gray_dilation](#)). For the generation of structuring elements, see [read_gray_se](#).

-
- Parameter
-
- ▷ **Image** (input_object) image(-array) $\leadsto$ Hobject : byte / real
Input image.
 - ▷ **SE** (input_object) image $\leadsto$ Hobject : byte
Structuring element.
 - ▷ **ImageOpening** (output_object) image(-array) $\leadsto$ Hobject : byte / real
Gray-opened image.

Result

[gray_opening](#) returns 2 (H_MSG_TRUE) if the structuring element is not the empty region. Otherwise, an exception is raised.

Parallelization Information

[gray_opening](#) is *reentrant* and automatically *parallelized* (on tuple level).

	Possible Predecessors
read_gray_se	
	Alternatives
dual_rank	
	See also
opening , gray_dilation , gray_erosion	
	Module
Image filters	

gray_opening_shape (Image : ImageOpening : MaskHeight, MaskWidth, MaskShape :)

Perform a gray value opening with a selected mask.

[gray_opening_shape](#) applies a gray value opening to the input image [Image](#) with the structuring element of shape [MaskShape](#). The mask's offset values are 0 and its horizontal and vertical size is defined by [MaskHeight](#) and [MaskWidth](#). The resulting image is returned in [ImageOpening](#). If the parameters [MaskHeight](#) or [MaskWidth](#) are even, they are changed to the next larger odd value. In case of the values 'rhombus' and 'octagon' for the [MaskShape](#) control parameter, [MaskHeight](#) and [MaskWidth](#) must be equal. The parameter value 'octagon' for [MaskShape](#) denotes an equilateral octagonal mask which is a suitable approximation for a circular structure. At the border of the image the gray values are mirrored.

The gray value opening of an image i with a structuring element s is defined as

$$i \circ s = (i \ominus s) \oplus s ,$$

i.e., an erosion of the image with s followed by a dilation with s (see [gray_erosion_shape](#) and [gray_dilation_shape](#)).

	Parameter
▷ Image (input_object)	image(-array) $\leadsto$ <i>Hobject</i> : byte Image for which the minimum gray values are to be calculated.
▷ ImageOpening (output_object)	image(-array) $\leadsto$ <i>Hobject</i> : byte Image containing the minimum gray values.
▷ MaskHeight (input_control)	extent.y $\leadsto$ <i>integer</i> Height of the filter mask. Default Value : 11 Suggested values : MaskHeight $\in \{3, 5, 7, 9, 11, 13, 15\}$ Typical range of values : $3 \leq \text{MaskHeight} \leq 511$ Minimum Increment : 2 Recommended Increment : 2 Restriction : odd(MaskHeight)
▷ MaskWidth (input_control)	extent.x $\leadsto$ <i>integer</i> Width of the filter mask. Default Value : 11 Suggested values : MaskWidth $\in \{3, 5, 7, 9, 11, 13, 15\}$ Typical range of values : $3 \leq \text{MaskWidth} \leq 511$ Minimum Increment : 2 Recommended Increment : 2 Restriction : odd(MaskWidth)
▷ MaskShape (input_control)	string $\leadsto$ <i>string</i> Shape of the mask. Default Value : 'octagon' List of values : MaskShape $\in \{\text{'rectangle'}, \text{'rhombus'}, \text{'octagon'}\}$

	Result
gray_opening_shape	returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

[gray_opening_shape](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Alternatives

[gray_opening](#)

See also

[gray_dilation_shape](#), [gray_erosion_shape](#), [opening](#)

Module

Image filters

gray_range_rect (Image : ImageResult : MaskHeight, MaskWidth :)

Determine the gray value range within a rectangle.

[gray_range_rect](#) calculates the gray value range, i.e., the difference (max – min) of the maximum and minimum gray values, of the input image [Image](#) within a rectangular mask of size ([MaskHeight](#), [MaskWidth](#)) for each image point. The resulting image is returned in [ImageResult](#). If the parameters [MaskHeight](#) or [MaskWidth](#) are even, they are changed to the next larger odd value. At the border of the image the gray values are mirrored.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int2 / int4 / real
Image for which the gray value range is to be calculated.
- ▷ **ImageResult** (output_object) image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int2 / int4 / real
Image containing the gray value range.
- ▷ **MaskHeight** (input_control) extent.y $\leadsto$ *integer*
Height of the filter mask.
Default Value : 11
Suggested values : MaskHeight $\in \{3, 5, 7, 9, 11, 13, 15\}$
Typical range of values : $3 \leq \text{MaskHeight} \leq 511$ (lin)
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskHeight)
- ▷ **MaskWidth** (input_control) extent.x $\leadsto$ *integer*
Width of the filter mask.
Default Value : 11
Suggested values : MaskWidth $\in \{3, 5, 7, 9, 11, 13, 15\}$
Typical range of values : $3 \leq \text{MaskWidth} \leq 511$ (lin)
Minimum Increment : 2
Recommended Increment : 2
Restriction : odd(MaskWidth)

Result

[gray_range_rect](#) returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via `set_system('no_object_result', <Result>)`. If necessary, an exception is raised.

Parallelization Information

[gray_range_rect](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*, *domain level*).

Alternatives

[gray_dilation_rect](#), [gray_erosion_rect](#), [sub_image](#)

Module

Image filters

gray_tophat (Image, SE : ImageTopHat : :)

Perform a gray value top hat transformation on an image.

gray_tophat applies a gray value top hat transformation to the input image **Image** with the structuring element **SE**. The gray value top hat transformation of an image i with a structuring element s is defined as

$$\text{tophat}(i, s) = i - (i \circ s),$$

i.e., the difference of the image and its opening with s (see **gray_opening**). For the generation of structuring elements, see **read_gray_se**.

Parameter
▷ Image (input_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / real Input image. ▷ SE (input_object) image $\leadsto$ <i>Hobject</i> : byte Structuring element. ▷ ImageTopHat (output_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / real Top hat image.
Result
gray_tophat returns 2 (H_MSG_TRUE) if the structuring element is not the empty region. Otherwise, an exception is raised.
Parallelization Information
gray_tophat is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).
Possible Predecessors
read_gray_se , gen_disc_se
Possible Successors
threshold
Alternatives
gray_opening
See also
gray_bothat , top_hat , gray_erosion_rect , sub_image
Module
Image filters

read_gray_se (: SE : FileName :)

Load a structuring element for gray morphology.

read_gray_se loads a structuring element for gray morphology from a file. The file names of these structuring elements must end in '.gse' (for gray-scale structuring element). This suffix is automatically appended by **read_gray_se** to the passed file name, and thus must not be passed. The structuring element's data must be contained in the file in the following format: The first two numbers in the file determine the width and height of the structuring element, and determine a rectangle enclosing the structuring element. Both values must be greater than 0. Then, Width*Height integer numbers follow, with the following interpretation: Values smaller than 0 are regarded as not belonging to the region of the structuring element, i.e., they are not considered in morphological operations. This allows the creation of irregularly shaped, not connected structuring elements. All other values are regarded as the corresponding values for gray morphology. Structuring elements are stored internally as byte-images, with negative values being mapped to 0, and all other values increased by 1. Thus, normal byte-images can also be used as structuring elements. However, care should be taken not to use too large images, since the runtime is proportional to the area of the image times the area of the structuring element.

Parameter
▷ SE (output_object) image $\leadsto$ <i>Hobject</i> : byte Generated structuring element. ▷ FileName (input_control) filename $\leadsto$ <i>string</i> Name of the file containing the structuring element.

Result

`read_gray_se` returns 2 (H_MSG_TRUE) if all parameters are correct. If the file cannot be opened, 5 (H_MSG_FAIL) is returned. Otherwise, an exception is raised.

Parallelization Information

`read_gray_se` is *reentrant* and processed *without* parallelization.

Possible Successors

`gray_erosion`, `gray_dilation`, `gray_opening`, `gray_closing`, `gray_tophat`, `gray_bothat`

Alternatives

`gen_disc_se`

See also

`read_image`, `paint_region`, `paint_gray`, `crop_part`

Module

Image filters

10.2 Region

bottom_hat (Region, StructElement : RegionBottomHat : :)

Compute the bottom hat of regions.

`bottom_hat` computes the `closing` of `Region` with `StructElement`. The difference between the result of the closing and the original region is called the bottom hat. In contrast to `closing`, which merges regions under certain circumstances, `bottom_hat` computes the regions generated by such a merge.

The position of `StructElement` is meaningless, since a closing operation is invariant with respect to the choice of the reference point.

Structuring elements (`StructElement`) can be generated with operators such as `gen_circle`, `gen_rectangle1`, `gen_rectangle2`, `gen_ellipse`, `draw_region`, `gen_region_polygon`, `gen_region_points`, etc.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be processed.
- ▷ **StructElement** (input_object) region $\leadsto$ *Hobject*
Structuring element (position independent).
- ▷ **RegionBottomHat** (output_object) region(-array) $\leadsto$ *Hobject*
Result of the bottom hat operator.

Example

```
read_image (Image, '/bilder/name.ext')
threshold (Image, Regions, 128, 255)
gen_circle (Circle, 0, 0, 16)
bottom_hat (Regions, Circle, RegionBottomHat).
```

Result

`bottom_hat` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

`bottom_hat` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`,

[gen_circle](#), [gen_ellipse](#), [gen_rectangle1](#), [gen_rectangle2](#), [draw_region](#),
[gen_region_points](#), [gen_struct_elements](#), [gen_region_polygon_filled](#)

Possible Successors

[reduce_domain](#), [select_shape](#), [area_center](#), [connection](#)

Alternatives

[closing](#), [difference](#)

See also

[top_hat](#), [morph_hat](#), [gray_bothat](#), [opening](#)

Module

Morphology

boundary (Region : RegionBorder : BoundaryType :)

Reduce a region to its boundary.

[boundary](#) computes the boundary of a region by using morphological operations. The parameter [BoundaryType](#) determines the type of boundary to compute:

'inner', 'inner_filled' and 'outer'.

[boundary](#) computes the contour of each input region. The resulting regions consist only of the minimal border of the input regions. If [BoundaryType](#) is set to 'inner', the contour lies within the original region, if it is set to 'outer', it is one pixel outside of the original region. If [BoundaryType](#) is set to 'inner_filled', holes in the interior of the input region are suppressed.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions for which the boundary is to be computed.
- ▷ **RegionBorder** (output_object) region(-array) $\leadsto$ *Hobject*
Resulting boundaries.
- ▷ **BoundaryType** (input_control) string $\leadsto$ *string*
Boundary type.
Default Value : 'inner'
List of values : $\text{BoundaryType} \in \{\text{'inner'}, \text{'outer'}, \text{'inner\_filled'}\}$

Example

```
#include "HalconCpp.h"

main()
{
    HWindow w;
    HRegion circ1 = HRegion::GenCircle (20, 10, 10.5);

    circ1.Display (w);
    w.Click ();

    HRegion marg1 = circ1.Boundary ("inner");
    w.SetColor ("red");
    marg1.Display (w);
    w.Click ();

    return(0);
}
```

Complexity

Let F be the area of the input region. Then the runtime complexity for one region is

$$O(3\sqrt{F}) .$$

Result

boundary returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

boundary is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`dilation_circle`, `erosion_circle`, `difference`

See also

`fill_up`

Module

Morphology

closing (Region, StructElement : RegionClosing : :)

Close a region.

A **closing** operation is defined as a dilation followed by a Minkowski subtraction. By applying **closing** to a region, larger structures remain mostly intact, while small gaps between adjacent regions and holes smaller than **StructElement** are closed, and the regions' boundaries are smoothed. All **closing** variants share the property that separate regions are not merged, but remain separate objects. The position of **StructElement** is meaningless, since a closing operation is invariant with respect to the choice of the reference point.

Structuring elements (**StructElement**) can be generated with operators such as `gen_circle`, `gen_rectangle1`, `gen_rectangle2`, `gen_ellipse`, `draw_region`, `gen_region_polygon`, `gen_region_points`, etc.

Attention

closing is applied to each input region separately. If gaps between different regions are to be closed, `union1` or `union2` has to be called first.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be closed.
- ▷ **StructElement** (input_object) region $\leadsto$ *Hobject*
Structuring element (position-invariant).
- ▷ **RegionClosing** (output_object) region(-array) $\leadsto$ *Hobject*
Closed regions.

Example

```
#include <iostream.h>
#include "HalconCpp.h"

main()
```

```

{
  cout << "Reproduction of 'closing ()' using " << endl;
  cout << "'dilation()' and 'minkowski_sub1()'" << endl;

  HByteImage img("monkey");
  HWindow      w;

  HRegion      circ = HRegion::GenCircle (10, 10, 1.5);
  HRegionArray regs = (img >= 128).Connection();

  HRegionArray dilreg = regs.Dilation1 (circ, 1);
  HRegionArray minsub = dilreg.MinkowskiSub1 (circ, 1);

  img.Display (w);      w.Click ();
  w.SetColor ("red");   regs.Display (w);   w.Click ();
  w.SetColor ("green"); dilreg.Display (w); w.Click ();
  w.SetColor ("blue"); minsub.Display (w);  w.Click ();

  return(0);
}

```

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(2 \cdot \sqrt{F1} \cdot \sqrt{F2}) .$$

Result

closing returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

closing is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`,
`gen_circle`, `gen_ellipse`, `gen_rectangle1`, `gen_rectangle2`, `draw_region`,
`gen_region_points`, `gen_struct_elements`, `gen_region_polygon-filled`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`closing_circle`, `closing_golay`

See also

`dilation1`, `erosion1`, `opening`, `minkowski_sub1`

Module

Morphology

closing_circle (Region : RegionClosing : Radius :)

Close a region with a circular structuring element.

closing_circle behaves analogously to **closing**, i.e., the regions' boundaries are smoothed and holes within a region which are smaller than the circular structuring element of radius **Radius** are closed. The

closing_circle operation is defined as a dilation followed by a Minkowski subtraction, both with the same circular structuring element.

Attention

closing_circle is applied to each input region separately. If gaps between different regions are to be closed, **union1** or **union2** has to be called first.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be closed.
- ▷ **RegionClosing** (output_object) region(-array) $\leadsto$ *Hobject*
Closed regions.
- ▷ **Radius** (input_control) real $\leadsto$ *real* / integer
Radius of the circular structuring element.
Default Value : 3.5
Suggested values : $\text{Radius} \in \{1.5, 2.5, 3.5, 4.5, 5.5, 7.5, 9.5, 12.5, 15.5, 19.5, 25.5, 33.5, 45.5, 60.5, 110.5\}$
Typical range of values : $0.5 \leq \text{Radius} \leq 511.5$ (lin)
Minimum Increment : 1.0
Recommended Increment : 1.0

Example

```
my_closing_circle(Hobject In, double Radius, Hobject *Out)
{
    Hobject tmp, StructElement;
    gen_circle(StructElement, 100.0, 100.0, Radius);
    dilation1(In, StructElement, &tmp, 1);
    minkowski_sub1(tmp, StructElement, Out, 1);
    clear_obj(tmp); clear_obj(StructElement);
}
```

Complexity

Let $F1$ be the area of the input region. Then the runtime complexity for one region is:

$$O(4 \cdot \sqrt{F1} \cdot \text{Radius}) .$$

Result

closing_circle returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: **set_system('no_object_result', <RegionResult>)**
- empty region: **set_system('empty_region_result', <RegionResult>)**

Otherwise, an exception is raised.

Parallelization Information

closing_circle is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

threshold, **regiongrowing**, **connection**, **union1**, **watersheds**, **class_ndim_norm**

Possible Successors

reduce_domain, **select_shape**, **area_center**, **connection**

Alternatives

rank_region, **fill_up**, **closing**, **closing_circle**, **closing_golay**

See also

dilation1, **minkowski_sub1**, **erosion1**, **opening**

Module

Morphology

closing_golay (Region : RegionClosing : GolayElement, Rotation :)

Close a region with an element from the Golay alphabet.

closing_golay is defined as a Minkowski addition followed by a Minkowski subtraction. First the Minkowski addition of the input region (**Region**) with the structuring element from the Golay alphabet defined by **GolayElement** and **Rotation** is computed. Then the Minkowski subtraction of the result and the structuring element rotated by 180° is performed.

The following structuring elements are available:

'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'.

The rotation number **Rotation** determines which rotation of the element should be used, and whether the foreground (even) or background version (odd) of the selected element should be used. The Golay elements, together with all possible rotations, are described with the operator **golay_elements**.

closing_golay serves to close holes smaller than the structuring element, and to smooth regions' boundaries.

Attention

Not all values of **Rotation** are valid for any Golay element. For some of the values of **Rotation**, the resulting regions are identical to the input regions.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be closed.
- ▷ **RegionClosing** (output_object) region(-array) $\leadsto$ *Hobject*
Closed regions.
- ▷ **GolayElement** (input_control) string $\leadsto$ *string*
Structuring element from the Golay alphabet.
Default Value : 'h'
List of values : GolayElement $\in$ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}
- ▷ **Rotation** (input_control) integer $\leadsto$ *integer*
Rotation of the Golay element. Depending on the element, not all rotations are valid.
Default Value : 0
List of values : Rotation $\in$ {0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15}

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(6 \cdot \sqrt{F}) .$$

Result

closing_golay returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: **set_system('no_object_result', <RegionResult>)**
- empty region: **set_system('empty_region_result', <RegionResult>)**

Otherwise, an exception is raised.

Parallelization Information

closing_golay is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

threshold, **regiongrowing**, **connection**, **union1**, **watersheds**, **class_ndim_norm**

Possible Successors

reduce_domain, **select_shape**, **area_center**, **connection**

Alternatives

closing

See also

erosion_golay, **dilation_golay**, **opening_golay**, **hit_or_miss_golay**, **thinning_golay**, **thickening_golay**, **golay_elements**

Module

Morphology

closing_rectangle1 (Region : RegionClosing : Width, Height :)

Close a region with a rectangular structuring element.

closing_rectangle1 performs a **dilation_rectangle1** followed by an **erosion_rectangle1** on the input region **Region**. The size of the rectangular structuring element is determined by the parameters **Width** and **Height**. As is the case for all **closing** variants, regions' boundaries are smoothed and holes within a region which are smaller than the rectangular structuring element are closed.

Attention

closing_rectangle1 is applied to each input region separately. If gaps between different regions are to be closed, **union1** or **union2** has to be called first.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be closed.
- ▷ **RegionClosing** (output_object) region(-array) $\leadsto$ *Hobject*
Closed regions.
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer / real*
Width of the structuring rectangle.
Default Value : 10
Suggested values : $\text{Width} \in \{1, 2, 3, 4, 5, 7, 9, 12, 15, 19, 25, 33, 45, 60, 110, 150, 200\}$
Typical range of values : $1 \leq \text{Width} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer / real*
Height of the structuring rectangle.
Default Value : 10
Suggested values : $\text{Height} \in \{1, 2, 3, 4, 5, 7, 9, 12, 15, 19, 25, 33, 45, 60, 110, 150, 200\}$
Typical range of values : $1 \leq \text{Height} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1

Complexity

Let $F1$ be the area of an input region and H be the height of the rectangle. Then the runtime complexity for one region is:

$$O(2 \cdot \sqrt{F1} \cdot \log_2(H)) .$$

Result

closing_rectangle1 returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: **set_system('no_object_result', <RegionResult>)**
- empty region: **set_system('empty_region_result', <RegionResult>)**

Otherwise, an exception is raised.

Parallelization Information

closing_rectangle1 is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

threshold, **regiongrowing**, **connection**, **union1**, **watersheds**, **class_ndim_norm**

Possible Successors

reduce_domain, **select_shape**, **area_center**, **connection**

Alternatives

closing

See also

dilation_rectangle1, **erosion_rectangle1**, **opening_rectangle1**, **gen_rectangle1**

Module

Morphology

dilation1 (Region, StructElement : RegionDilation : Iterations :)

Dilate a region.

dilation1 dilates the input regions with a structuring element. By applying **dilation1** to a region, its boundary gets smoothed. In the process, the area of the region is enlarged. Furthermore, disconnected regions may be merged. Such regions, however, remain logically distinct region. The dilation is a set-theoretic region operation. It uses the union operation.

Let M (**StructElement**) and R (**Region**) be two regions, where M is the structuring element and R is the region to be processed. Furthermore, let m be a point in M . Then the displacement vector $\vec{v}_m = (dx, dy)$ is defined as the difference of the center of gravity of M and the vector $\vec{m}$. Let $t_{\vec{v}_m}(R)$ denote the translation of a region R by a vector $\vec{v}$. Then

$$\text{dilation1}(R, M) := \bigcup_{m \in M} t_{\vec{v}_m}(R)$$

For each point m in M a translation of the region R is performed. The union of all these translations is the dilation of R with M . **dilation1** is similar to the operator **minkowski_add1**, the difference is that in **dilation1** the structuring element is mirrored at the origin. The position of **StructElement** is meaningless, since the displacement vectors are determined with respect to the center of gravity of M .

The parameter **Iterations** determines the number of iterations which are to be performed with the structuring element. The result of iteration $n - 1$ is used as input for iteration n . From the above definition it follows that an empty region is generated in case of an empty structuring element.

Structuring elements (**StructElement**) can be generated with operators such as **gen_circle**, **gen_rectangle1**, **gen_rectangle2**, **gen_ellipse**, **draw_region**, **gen_region_polygon**, **gen_region_points**, etc.

Attention

A dilation always results in enlarged regions. Closely spaced regions which may touch or overlap as a result of the dilation are still treated as two separate regions. If the desired behavior is to merge them into one region, the operator **union1** has to be called first.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be dilated.
- ▷ **StructElement** (input_object) region $\leadsto$ *Hobject*
Structuring element.
- ▷ **RegionDilation** (output_object) region(-array) $\leadsto$ *Hobject*
Dilated regions.
- ▷ **Iterations** (input_control) integer $\leadsto$ *integer*
Number of iterations.
Default Value : 1
Suggested values : Iterations $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50\}$
Typical range of values : $1 \leq \text{Iterations} \text{ (lin)}$
Minimum Increment : 1
Recommended Increment : 1

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \text{Iterations}) .$$

Result

dilation1 returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: **set_system('no_object_result', <RegionResult>)**
- empty region: **set_system('empty_region_result', <RegionResult>)**

Otherwise, an exception is raised.

Parallelization Information

dilation1 is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

threshold, **regiongrowing**, **connection**, **union1**, **watersheds**, **class_ndim_norm**,
gen_circle, **gen_ellipse**, **gen_rectangle1**, **gen_rectangle2**, **draw_region**,
gen_region_points, **gen_struct_elements**, **gen_region_polygon_filled**

Possible Successors

reduce_domain, **add_channels**, **select_shape**, **area_center**, **connection**

Alternatives

minkowski_add1, **minkowski_add2**, **dilation2**, **dilation_golay**, **dilation_seq**

See also

erosion1, **erosion2**, **opening**, **closing**

Module

Morphology

dilation2 (**Region**, **StructElement** : **RegionDilation** : **Row**, **Column**,
Iterations :)

Dilate a region (using a reference point).

dilation2 dilates the input regions with a structuring element (**StructElement**) having the reference point (**Row,Column**). **dilation2** has a similar effect as **dilation1**, the difference is that the reference point of the structuring element can be chosen arbitrarily. The parameter **Iterations** determines the number of iterations which are to be performed with the structuring element. The result of iteration $n - 1$ is used as input for iteration n .

An empty region is generated in case of an empty structuring element.

Structuring elements (**StructElement**) can be generated with operators such as **gen_circle**, **gen_rectangle1**, **gen_rectangle2**, **gen_ellipse**, **draw_region**, **gen_region_polygon**, **gen_region_points**, etc.

Attention

A dilation always results in enlarged regions. Closely spaced regions which may touch or overlap as a result of the dilation are still treated as two separate regions. If the desired behavior is to merge them into one region, the operator **union1** has to be called first.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be dilated.
- ▷ **StructElement** (input_object) region $\leadsto$ *Hobject*
Structuring element.
- ▷ **RegionDilation** (output_object) region(-array) $\leadsto$ *Hobject*
Dilated regions.
- ▷ **Row** (input_control) point.y $\leadsto$ *integer*
Row coordinate of the reference point.
Default Value : 0
- ▷ **Column** (input_control) point.x $\leadsto$ *integer*
Column coordinate of the reference point.
Default Value : 0
- ▷ **Iterations** (input_control) integer $\leadsto$ *integer*
Number of iterations.
Default Value : 1
Suggested values : Iterations $\in \{1, 2, 3, 4, 5, 7, 11, 17, 25, 32, 64, 128\}$
Typical range of values : $1 \leq \text{Iterations}$ (lin)
Minimum Increment : 1
Recommended Increment : 1

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \text{Iterations}) .$$

Result

dilation2 returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

dilation2 is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`,
`gen_circle`, `gen_ellipse`, `gen_rectangle1`, `gen_rectangle2`, `draw_region`,
`gen_region_points`, `gen_struct_elements`, `gen_region_polygon_filled`

Possible Successors

`reduce_domain`, `add_channels`, `select_shape`, `area_center`, `connection`

Alternatives

`minkowski_add1`, `minkowski_add2`, `dilation1`, `dilation_golay`, `dilation_seq`

See also

`erosion1`, `erosion2`, `opening`, `closing`

Module

Morphology

dilation_circle (Region : RegionDilation : Radius :)

Dilate a region with a circular structuring element.

dilation_circle applies a Minkowski addition with a circular structuring element to the input regions **Region**. Because the circular mask is symmetrical, this is identical to a dilation. The size of the circle used as structuring element is determined by **Radius**.

The operator results in enlarged regions, smoothed region boundaries, and the holes smaller than the circular mask in the interior of the region are closed. It is useful to select only values like 3.5, 5.5, etc. for **Radius** in order to avoid a translation of a region, because integer radii result in the circle having a non-integer center of gravity which is rounded to the next integer.

Attention

dilation_circle is applied to each input region separately. If gaps between different regions are to be closed, **union1** or **union2** has to be called first.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be dilated.
- ▷ **RegionDilation** (output_object) region(-array) $\leadsto$ *Hobject*
Dilated regions.
- ▷ **Radius** (input_control) real $\leadsto$ *real* / integer
Radius of the circular structuring element.
Default Value : 3.5
Suggested values : $\text{Radius} \in \{1.5, 2.5, 3.5, 4.5, 5.5, 7.5, 9.5, 12.5, 15.5, 19.5, 25.5, 33.5, 45.5, 60.5, 110.5\}$
Typical range of values : $0.5 \leq \text{Radius} \leq 511.5$ (lin)
Minimum Increment : 1.0
Recommended Increment : 1.0

Example

```

#include <iostream.h>
#include "HalconCpp.h"

main()
{
    cout << "Reproduction of 'dilation_circle ()'" << endl;
    cout << "First = original image " << endl;
    cout << "Blue = after dilation " << endl;
    cout << "Red = before dilation " << endl;

    HByteImage img("monkey");
    HWindow w;

    HRegion circ = HRegion::GenCircle (10, 10, 1.5);
    HRegionArray regs = (img >= 128).Connection();
    HRegionArray minadd = regs.MinkowskiAdd1 (circ, 1);

    img.Display (w); w.Click ();
    w.SetColor ("blue"); minadd.Display (w); w.Click ();
    w.SetColor ("red"); regs.Display (w); w.Click ();

    return(0);
}

```

Complexity

Let $F1$ be the area of an input region. Then the runtime complexity for one region is:

$$O(2 \cdot \text{Radius} \cdot \sqrt{F1}) .$$

Result

dilation_circle returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

dilation_circle is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`minkowski_add1`, `minkowski_add2`, `expand_region`, `dilation1`, `dilation2`, `dilation_rectangle1`

See also

`gen_circle`, `erosion_circle`, `closing_circle`, `opening_circle`

Module

Morphology

dilation_golay (Region : RegionDilation : GolayElement, Iterations, Rotation :)

Dilate a region with an element from the Golay alphabet.

dilation_golay dilates a region with the selected element **GolayElement** from the Golay alphabet. The following structuring elements are available:

'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'.

The rotation number **Rotation** determines which rotation of the element should be used, and whether the foreground (even) or background version (odd) of the selected element should be used. The Golay elements, together with all possible rotations, are described with the operator **golay_elements**. The operator works by shifting the structuring element over the region to be processed (**Region**). For all positions of the structuring element that intersect the region, the corresponding reference point (relative to the structuring element) is added to the output region. This means that the union of all translations of the structuring element within the region is computed.

The parameter **Iterations** determines the number of iterations which are to be performed with the structuring element. The result of iteration $n - 1$ is used as input for iteration n .

Attention

Not all values of **Rotation** are valid for any Golay element. For some of the values of **Rotation**, the resulting regions are identical to the input regions.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be dilated.
- ▷ **RegionDilation** (output_object) region(-array) $\leadsto$ *Hobject*
Dilated regions.
- ▷ **GolayElement** (input_control) string $\leadsto$ *string*
Structuring element from the Golay alphabet.
Default Value : 'h'
List of values : **GolayElement** $\in$ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}
- ▷ **Iterations** (input_control) integer $\leadsto$ *integer*
Number of iterations.
Default Value : 1
Suggested values : **Iterations** $\in$ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50}
Typical range of values : $1 \leq \text{Iterations} (\text{lin})$
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Rotation** (input_control) integer $\leadsto$ *integer*
Rotation of the Golay element. Depending on the element, not all rotations are valid.
Default Value : 0
List of values : **Rotation** $\in$ {0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15}

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(3 \cdot \sqrt{F}) .$$

Result

dilation_golay returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: **set_system('no_object_result', <RegionResult>)**
- empty region: **set_system('empty_region_result', <RegionResult>)**

Otherwise, an exception is raised.

Parallelization Information

dilation_golay is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

threshold, **regiongrowing**, **connection**, **union1**, **watersheds**, **class_ndim_norm**

Possible Successors

reduce_domain, **select_shape**, **area_center**, **connection**

Alternatives

dilation1, **dilation2**, **dilation_seq**

See also

[erosion_golay](#), [opening_golay](#), [closing_golay](#), [hit_or_miss_golay](#), [thinning_golay](#), [thickening_golay](#), [golay_elements](#)

Module

Morphology

dilation_rectangle1 (Region : RegionDilation : Width, Height :)

Dilate a region with a rectangular structuring element.

dilation_rectangle1 applies a dilation with a rectangular structuring element to the input regions **Region**. The size of the structuring rectangle is **Width** × **Height**. The operator results in enlarged regions, and the holes smaller than the rectangular mask in the interior of the regions are closed.

dilation_rectangle1 is a very fast operation because the height of the rectangle enters only logarithmically into the runtime complexity, while the width does not enter at all. This leads to excellent runtime efficiency, even in the case of very large rectangles (edge length > 100).

Attention

dilation_rectangle1 is applied to each input region separately. If gaps between different regions are to be closed, **union1** or **union2** has to be called first.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be dilated.
- ▷ **RegionDilation** (output_object) region(-array) $\leadsto$ *Hobject*
Dilated regions.
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer*
Width of the structuring rectangle.
Default Value : 10
Suggested values : $\text{Width} \in \{1, 2, 3, 4, 6, 10, 15, 20, 30, 50, 70, 100, 150, 200\}$
Typical range of values : $1 \leq \text{Width} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer*
Height of the structuring rectangle.
Default Value : 10
Suggested values : $\text{Height} \in \{1, 2, 3, 4, 6, 10, 15, 20, 30, 50, 70, 100, 150, 200\}$
Typical range of values : $1 \leq \text{Height} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10

Example

```
#include <iostream.h>
#include "HalconCpp.h"

main()
{
    cout << "Reproduction of 'dilation_rectangle ()'" << endl;
    cout << "First = original image " << endl;
    cout << "Blue = after dilation " << endl;
    cout << "Red = after segmentation " << endl;

    HByteImage img("monkey");
    HWindow w;

    HRegionArray regs = (img >= 220).Connection();
    HRegionArray dilreg = regs.DilationRectangle1 (2, 4);
```



```

        img.Display (w);          w.Click ();
    w.SetColor ("blue"); dilreg.Display (w); w.Click ();
    w.SetColor ("red");  regs.Display (w);  w.Click ();

    return(0);
}

```

Complexity

Let $F1$ be the area of an input region and H be the height of the rectangle. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \lg(H)) .$$

Result

`dilation_rectangle1` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

`dilation_rectangle1` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`minkowski_add1`, `minkowski_add2`, `expand_region`, `dilation1`, `dilation2`, `dilation_circle`

See also

`gen_rectangle1`, `gen_region_polygon_filled`

Module

Morphology

dilation_seq (Region : RegionDilation : GolayElement, Iterations :)

Dilate a region sequentially.

`dilation_seq` computes the sequential dilation of the input region `Region` with the selected structuring element `GolayElement` from the Golay alphabet. This is done by executing the operator `dilation_golay` with all rotations of the structuring element `Iterations` times. The following structuring elements can be selected:

'l', 'd', 'c', 'f', 'h', 'k'.

In order to compute the skeleton of a region, usually the elements 'l' and 'm' are used. Only the “foreground elements” (even rotation numbers) are used. The elements 'i' and 'e' result in unchanged output regions. The elements 'l', 'm' and 'f2' are identical for the foreground. The Golay elements, together with all possible rotations, are described with the operator `golay_elements`.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be dilated.
- ▷ **RegionDilation** (output_object) region(-array) $\leadsto$ *Hobject*
Dilated regions.

- ▷ **GolayElement** (input_control) string $\leadsto$ string
Structuring element from the Golay alphabet.
Default Value : 'h'
List of values : GolayElement $\in \{'l', 'd', 'c', 'f', 'h', 'k'\}$
- ▷ **Iterations** (input_control) integer $\leadsto$ integer
Number of iterations.
Default Value : 1
Suggested values : Iterations $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50\}$
Typical range of values : $1 \leq \text{Iterations}$ (lin)
Minimum Increment : 1
Recommended Increment : 1

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(\text{Iterations} \cdot 20 \cdot \sqrt{F}) .$$

Result

`dilation_seq` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

`dilation_seq` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`dilation1`, `dilation2`, `dilation_golay`

See also

`erosion_seq`, `hit_or_miss_seq`, `thinning_seq`

Module

Morphology

erosion1 (Region, StructElement : RegionErosion : Iterations :)

Erode a region.

`erosion1` erodes the input regions with a structuring element. By applying `erosion1` to a region, its boundary gets smoothed. In the process, the area of the region is reduced. Furthermore, connected regions may be split. Such regions, however, remain logically one region. The erosion is a set-theoretic region operation. It uses the intersection operation.

Let M (**StructElement**) and R (**Region**) be two regions, where M is the structuring element and R is the region to be processed. Furthermore, let m be a point in M . Then the displacement vector $\vec{v}_m = (dx, dy)$ is defined as the difference of the center of gravity of M and the vector $\vec{m}$. Let $t_{\vec{v}_m}(R)$ denote the translation of a region R by a vector $\vec{v}$. Then

$$\text{erosion1}(R, M) := \bigcap_{m \in M} t_{-\vec{v}_m}(R).$$

For each point m in M a translation of the region R is performed. The intersection of all these translations is the erosion of R with M . `erosion1` is similar to the operator `minkowski_sub1`, the difference is that in

erosion1 the structuring element is mirrored at the origin. The position of **StructElement** is meaningless, since the displacement vectors are determined with respect to the center of gravity of M .

The parameter **Iterations** determines the number of iterations which are to be performed with the structuring element. The result of iteration $n - 1$ is used as input for iteration n . From the above definition it follows that the maximum region is generated in case of an empty structuring element.

Structuring elements (**StructElement**) can be generated with operators such as **gen_circle**, **gen_rectangle1**, **gen_rectangle2**, **gen_ellipse**, **draw_region**, **gen_region_polygon**, **gen_region_points**, etc.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ Hobject
Regions to be eroded.
- ▷ **StructElement** (input_object) region $\leadsto$ Hobject
Structuring element.
- ▷ **RegionErosion** (output_object) region(-array) $\leadsto$ Hobject
Eroded regions.
- ▷ **Iterations** (input_control) integer $\leadsto$ integer
Number of iterations.
Default Value : 1
Suggested values : Iterations $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50\}$
Typical range of values : $1 \leq \text{Iterations} \text{ (lin)}$
Minimum Increment : 1
Recommended Increment : 1

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \text{Iterations}) .$$

Result

erosion1 returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: **set_system('no_object_result', <RegionResult>)**
- empty region: **set_system('empty_region_result', <RegionResult>)**

Otherwise, an exception is raised.

Parallelization Information

erosion1 is *reentrant* and automatically *parallelized* (on *tuple* level).

Possible Predecessors

threshold, **regiongrowing**, **watersheds**, **class_ndim_norm**, **gen_circle**, **gen_ellipse**, **gen_rectangle1**, **gen_rectangle2**, **draw_region**, **gen_region_points**, **gen_struct_elements**, **gen_region_polygon_filled**

Possible Successors

connection, **reduce_domain**, **select_shape**, **area_center**

Alternatives

minkowski_sub1, **minkowski_sub2**, **erosion2**, **erosion_golay**, **erosion_seq**

See also

transpose_region

Module

Morphology

erosion2 (Region, StructElement : RegionErosion : Row, Column, Iterations :)

Erode a region (using a reference point).

erosion2 erodes the input regions with a structuring element (**StructElement**) having the reference point (**Row,Column**). **erosion2** has a similar effect as **erosion1**, the difference is that the reference point of the structuring element can be chosen arbitrarily. The parameter **Iterations** determines the number of iterations which are to be performed with the structuring element. The result of iteration $n - 1$ is used as input for iteration n .

A maximum region is generated in case of an empty structuring element.

Structuring elements (**StructElement**) can be generated with operators such as **gen_circle**, **gen_rectangle1**, **gen_rectangle2**, **gen_ellipse**, **draw_region**, **gen_region_polygon**, **gen_region_points**, etc.

Parameter	
▷ Region (input_object)	region(-array) $\leadsto$ Hobject Regions to be eroded.
▷ StructElement (input_object)	region $\leadsto$ Hobject Structuring element.
▷ RegionErosion (output_object)	region(-array) $\leadsto$ Hobject Eroded regions.
▷ Row (input_control)	point.y $\leadsto$ integer Row coordinate of the reference point. Default Value : 0 Typical range of values : $0 \leq \text{Row} \leq 511$ (lin) Minimum Increment : 1 Recommended Increment : 1
▷ Column (input_control)	point.x $\leadsto$ integer Column coordinate of the reference point. Default Value : 0 Typical range of values : $0 \leq \text{Column} \leq 511$ (lin) Minimum Increment : 1 Recommended Increment : 1
▷ Iterations (input_control)	integer $\leadsto$ integer Number of iterations. Default Value : 1 Suggested values : $\text{Iterations} \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50\}$ Typical range of values : $1 \leq \text{Iterations}$ (lin) Minimum Increment : 1 Recommended Increment : 1

Complexity	
Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:	

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \text{Iterations}) .$$

Result	
erosion2 returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:	

- no region: **set_system('no_object_result', <RegionResult>)**
- empty region: **set_system('empty_region_result', <RegionResult>)**

Otherwise, an exception is raised.

Parallelization Information	
erosion2 is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).	

Possible Predecessors	
threshold , regiongrowing , watersheds , class_ndim_norm , gen_circle , gen_ellipse , gen_rectangle1 , gen_rectangle2 , draw_region , gen_region_points , gen_struct_elements , gen_region_polygon_filled	

Possible Successors

[reduce_domain](#), [select_shape](#), [area_center](#), [connection](#)

Alternatives

[minkowski_sub2](#), [minkowski_sub1](#), [erosion1](#), [erosion_golay](#), [erosion_seq](#)

See also

[transpose_region](#), [gen_circle](#), [gen_rectangle2](#), [gen_region_polygon](#)

Module

Morphology

erosion_circle (Region : RegionErosion : Radius :)*Erode a region with a circular structuring element.*

[erosion_circle](#) applies a Minkowski subtraction with a circular structuring element to the input regions [Region](#). Because the circular mask is symmetrical, this is identical to an erosion. The size of the circle used as structuring element is determined by [Radius](#).

The operator results in reduced regions, smoothed region boundaries, and the regions smaller than the circular mask are eliminated. It is useful to select only values like 3.5, 5.5, etc. for [Radius](#) in order to avoid a translation of a region, because integer radii result in a circle having a non-integer center of gravity which is rounded to the next integer.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be eroded.
- ▷ **RegionErosion** (output_object) region(-array) $\leadsto$ *Hobject*
Eroded regions.
- ▷ **Radius** (input_control) real $\leadsto$ *real* / integer
Radius of the circular structuring element.
Default Value : 3.5
Suggested values : Radius $\in \{1.5, 2.5, 3.5, 4.5, 5.5, 7.5, 9.5, 12.5, 15.5, 19.5, 25.5, 33.5, 45.5, 60.5, 110.5\}$
Typical range of values : $0.5 \leq \text{Radius} \leq 511.5$ (lin)
Minimum Increment : 1.0
Recommended Increment : 1.0

Example

```
#include <iostream.h>
#include "HalconCpp.h"

main()
{
    cout << "Simulation of 'erosion_circle ()'" << endl;
    cout << "First = original image " << endl;
    cout << "Red   = after segmentation " << endl;
    cout << "Blue  = after erosion " << endl;

    HByteImage img("monkey");
    HWindow     w;

    HRegion      circ   = HRegion::GenCircle (10, 10, 1.5);
    HRegionArray regs   = (img >= 128).Connection();
    HRegionArray minsub = regs.MinkowskiSub1 (circ, 1);

    img.Display (w);      w.Click ();
    w.SetColor ("red");   regs.Display (w);   w.Click ();
    w.SetColor ("blue");  minsub.Display (w);  w.Click ();
}
```

```

    return(0);
}

```

Complexity

Let $F1$ be the area of an input region. Then the runtime complexity for one region is:

$$O(2 \cdot \text{Radius} \cdot \sqrt{F1}) .$$

Result

`erosion_circle` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

`erosion_circle` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`threshold`, `regiongrowing`, `watersheds`, `class_ndim_norm`

Possible Successors

`connection`, `reduce_domain`, `select_shape`, `area_center`

Alternatives

`minkowski_sub1`

See also

`gen_circle`, `dilation_circle`, `closing_circle`, `opening_circle`

Module

Morphology

erosion_golay (Region : RegionErosion : GolayElement, Iterations, Rotation :)

Erode a region with an element from the Golay alphabet.

`erosion_golay` erodes a region with the selected element `GolayElement` from the Golay alphabet. The following structuring elements are available:

'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'.

The rotation number `Rotation` determines which rotation of the element should be used, and whether the foreground (even) or background version (odd) of the selected element should be used. The Golay elements, together with all possible rotations, are described with the operator `golay_elements`. The operator works by shifting the structuring element over the region to be processed (`Region`). For all positions of the structuring element fully contained in the region, the corresponding reference point (relative to the structuring element) is added to the output region. This means that the intersection of all translations of the structuring element within the region is computed.

The parameter `Iterations` determines the number of iterations which are to be performed with the structuring element. The result of iteration $n - 1$ is used as input for iteration n .

Attention

Not all values of `Rotation` are valid for any Golay element. For some of the values of `Rotation`, the resulting regions are identical to the input regions.

Parameter	
▷ Region (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Regions to be eroded.
▷ RegionErosion (output_object)	region(-array) $\leadsto$ <i>Hobject</i> Eroded regions.
▷ GolayElement (input_control)	string $\leadsto$ <i>string</i> Structuring element from the Golay alphabet. Default Value : 'h' List of values : GolayElement $\in \{ 'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k' \}$
▷ Iterations (input_control)	integer $\leadsto$ <i>integer</i> Number of iterations. Default Value : 1 Suggested values : Iterations $\in \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50 \}$ Typical range of values : $1 \leq \text{Iterations} \text{ (lin)}$ Minimum Increment : 1 Recommended Increment : 1
▷ Rotation (input_control)	integer $\leadsto$ <i>integer</i> Rotation of the Golay element. Depending on the element, not all rotations are valid. Default Value : 0 List of values : Rotation $\in \{ 0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15 \}$

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(3 \cdot \sqrt{F}) .$$

Result

erosion_golay returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

erosion_golay is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`threshold`, `regiongrowing`, `watersheds`, `class_ndim_norm`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`erosion_seq`, `erosion1`, `erosion2`

See also

`dilation_golay`, `opening_golay`, `closing_golay`, `hit_or_miss_golay`, `thinning_golay`, `thickening_golay`, `golay_elements`

Module

Morphology

erosion_rectangle1 (Region : RegionErosion : Width, Height :)

Erode a region with a rectangular structuring element.

erosion_rectangle1 applies an erosion with a rectangular structuring element to the input regions **Region**. The size of the structuring rectangle is **Width** $\times$ **Height**. The operator results in reduced regions, and the areas smaller than the rectangular mask are eliminated.

`erosion_rectangle1` is a very fast operation because the height of the rectangle enters only logarithmically into the runtime complexity, while the width does not enter at all. This leads to excellent runtime efficiency, even in the case of very large rectangles (edge length > 100).

Regions containing small connecting strips between large areas are separated only seemingly. They remain logically one region.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be eroded.
- ▷ **RegionErosion** (output_object) region(-array) $\leadsto$ *Hobject*
Eroded regions.
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer*
Width of the structuring rectangle.
Default Value : 10
Suggested values : Width $\in \{1, 2, 3, 4, 6, 10, 15, 20, 30, 50, 70, 100\}$
Typical range of values : $1 \leq \text{Width} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer*
Height of the structuring rectangle.
Default Value : 10
Suggested values : Height $\in \{1, 2, 3, 4, 6, 10, 15, 20, 30, 50, 70, 100\}$
Typical range of values : $1 \leq \text{Height} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1

Complexity

Let $F1$ be the area of an input region and H be the height of the rectangle. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \text{ld}(H)) .$$

Result

`erosion_rectangle1` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

`erosion_rectangle1` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `regiongrowing`, `watersheds`, `class_ndim_norm`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`erosion1`, `minkowski_sub1`

See also

`gen_rectangle1`

Module

Morphology

erosion_seq (Region : RegionErosion : GolayElement, Iterations :)

Erode a region sequentially.

erosion_seq computes the sequential erosion of the input region **Region** with the selected structuring element **GolayElement** from the Golay alphabet. This is done by executing the operator **erosion_golay** with all rotations of the structuring element **Iterations** times. The following structuring elements can be selected:

'l', 'd', 'c', 'f', 'h', 'k'.

Only the “foreground elements” (even rotation numbers) are used. The elements 'i' and 'e' result in unchanged output regions. The elements 'l', 'm' and 'f2' are identical for the foreground. The Golay elements, together with all possible rotations, are described with the operator **golay_elements**.

Parameter	
▷ Region (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Regions to be eroded.
▷ RegionErosion (output_object)	region(-array) $\leadsto$ <i>Hobject</i> Eroded regions.
▷ GolayElement (input_control)	string $\leadsto$ <i>string</i> Structuring element from the Golay alphabet. Default Value : 'h' List of values : GolayElement $\in$ {'l', 'd', 'c', 'f', 'h', 'k'}
▷ Iterations (input_control)	integer $\leadsto$ <i>integer</i> Number of iterations. Default Value : 1 Suggested values : Iterations $\in$ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50} Typical range of values : $1 \leq \text{Iterations}$ (lin) Minimum Increment : 1 Recommended Increment : 1

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(\text{Iterations} \cdot 20 \cdot \sqrt{F}) .$$

erosion_seq returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: **set_system('no_object_result', <RegionResult>)**
- empty region: **set_system('empty_region_result', <RegionResult>)**

Otherwise, an exception is raised.

erosion_seq is *reentrant* and automatically *parallelized* (on tuple level).

threshold, **regiongrowing**, **watersheds**, **class_ndim_norm**

connection, **reduce_domain**, **select_shape**, **area_center**

erosion_golay, **erosion1**, **erosion2**

dilation_seq, **hit_or_miss_seq**, **thinning_seq**

Morphology

fitting (Region, StructElements : RegionFitted : :)

Perform a closing after an opening with multiple structuring elements.

fitting performs an **opening** and a **closing** successively on the input regions. The eight structuring elements normally used for this operation can be generated with the operator **gen_struct_elements**. However, other user-defined structuring elements can also be used. Let R be the input region(s) and let M_i denote the structuring elements. Furthermore, let P be the result of the opening and Q be the final result. Then the operator can be formalized as follows:

$$P = \bigcup_{i=1}^n (R \circ M_i)$$

$$Q = \bigcap_{i=1}^n (P \bullet M_i)$$

Regions larger than the structuring elements are preserved, while small gaps are closed.

Parameter	
▷ Region (input_object)	region(-array) $\leadsto$ Hobject Regions to be processed.
▷ StructElements (input_object)	region(-array) $\leadsto$ Hobject Structuring elements.
▷ RegionFitted (output_object)	region(-array) $\leadsto$ Hobject Fitted regions.

fitting returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

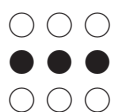
Otherwise, an exception is raised.

Parallelization Information
fitting is <i>reentrant</i> and processed <i>without</i> parallelization.
Possible Predecessors
gen_struct_elements , gen_region_points
Possible Successors
reduce_domain , select_shape , area_center , connection
Alternatives
opening , closing , connection , select_shape
Module
Morphology

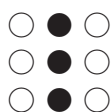
gen_struct_elements (: StructElements : Type, Row, Column :)

Generate standard structuring elements.

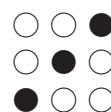
gen_struct_elements serves to generate eight structuring elements normally used in the operator **fitting**. The default value 'noise' of the parameter **Type** generates elements especially suited for the elimination of noise.



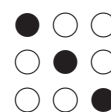
M_1



M_2



M_3



M_4



	Parameter
▷ StructElements (output_object)	region(-array) $\leadsto$ <i>Hobject</i> Generated structuring elements.
▷ Type (input_control)	string $\leadsto$ <i>string</i> Type of structuring element to generate. Default Value : 'noise' List of values : Type $\in$ {'noise'}
▷ Row (input_control)	point.y $\leadsto$ <i>integer</i> Row coordinate of the reference point. Default Value : 1 Suggested values : Row $\in$ {0, 1, 10, 50, 100, 200, 300, 400} Typical range of values : $-\infty \leq \text{Row} \leq \infty$ (lin)
▷ Column (input_control)	point.x $\leadsto$ <i>integer</i> Column coordinate of the reference point. Default Value : 1 Suggested values : Column $\in$ {0, 1, 10, 50, 100, 200, 300, 400} Typical range of values : $-\infty \leq \text{Column} \leq \infty$ (lin)

Result
[gen_struct_elements](#) returns 2 (H_MSG_TRUE) if all parameters are correct. Otherwise, an exception is raised.

Parallelization Information
[gen_struct_elements](#) is *reentrant* and processed *without* parallelization.

Possible Successors
[fitting](#), [hit_or_miss](#), [opening](#), [closing](#), [erosion2](#), [dilation2](#)

See also
[golay_elements](#)

Module
Morphology

golay_elements (: StructElement1, StructElement2 : GolayElement, Rotation, Row, Column :)

Generate the structuring elements of the Golay alphabet.

[golay_elements](#) generates the structuring elements from the Golay alphabet. The parameter [GolayElement](#) determines the name of the structuring element, while [Rotation](#) determines its rotation. The structuring elements are intended for use in [hit_or_miss](#): In [StructElement1](#) the structuring element for the foreground is returned, while in [StructElement2](#) the structuring element for the background is returned. [Row](#) and [Column](#) determine the reference point of the structuring element.

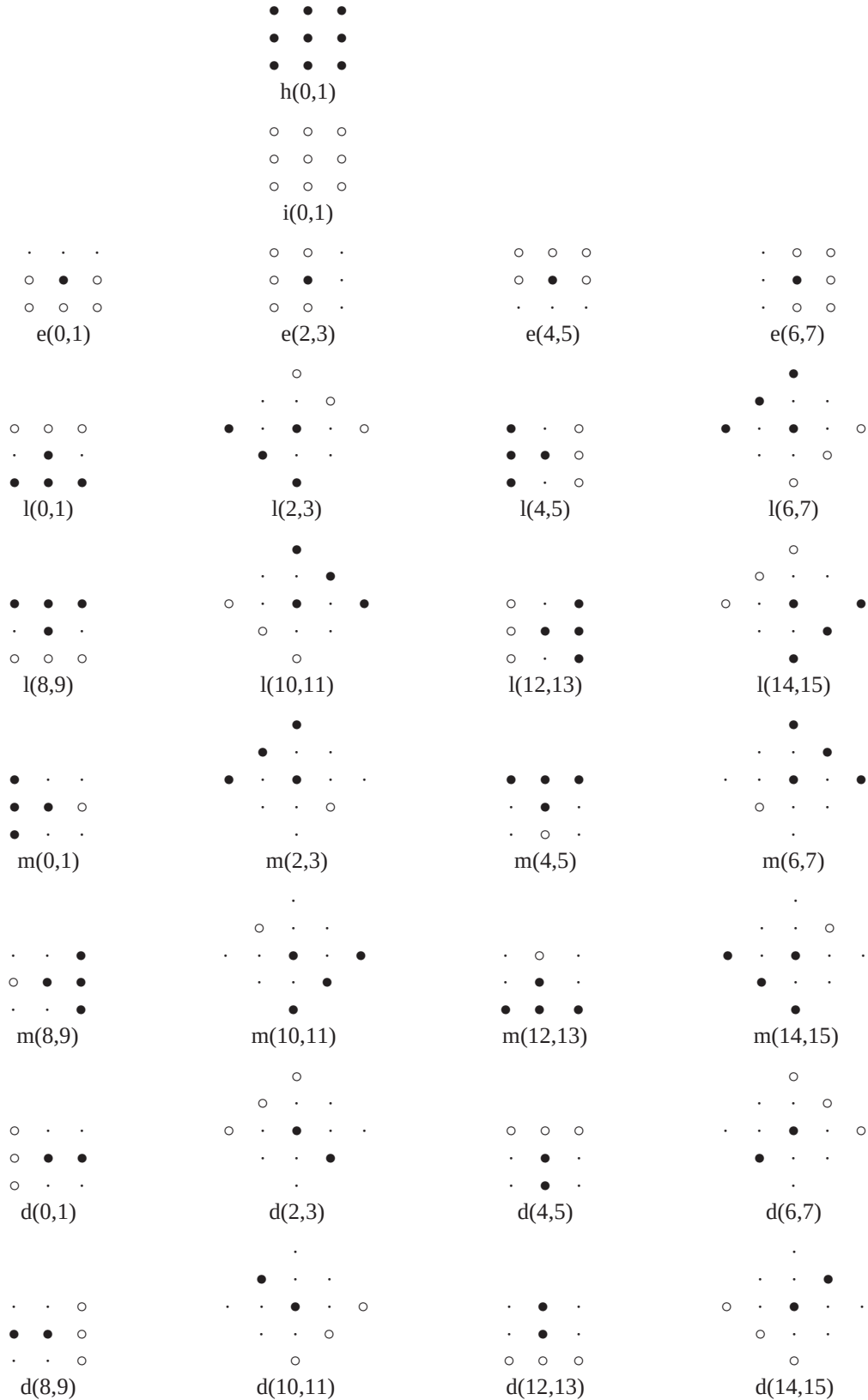
The rotations are numbered from 0 to 15. This does not mean, however, that there are 16 different rotations: Even values denote rotations of the foreground elements, while odd values denote rotations of the background elements.

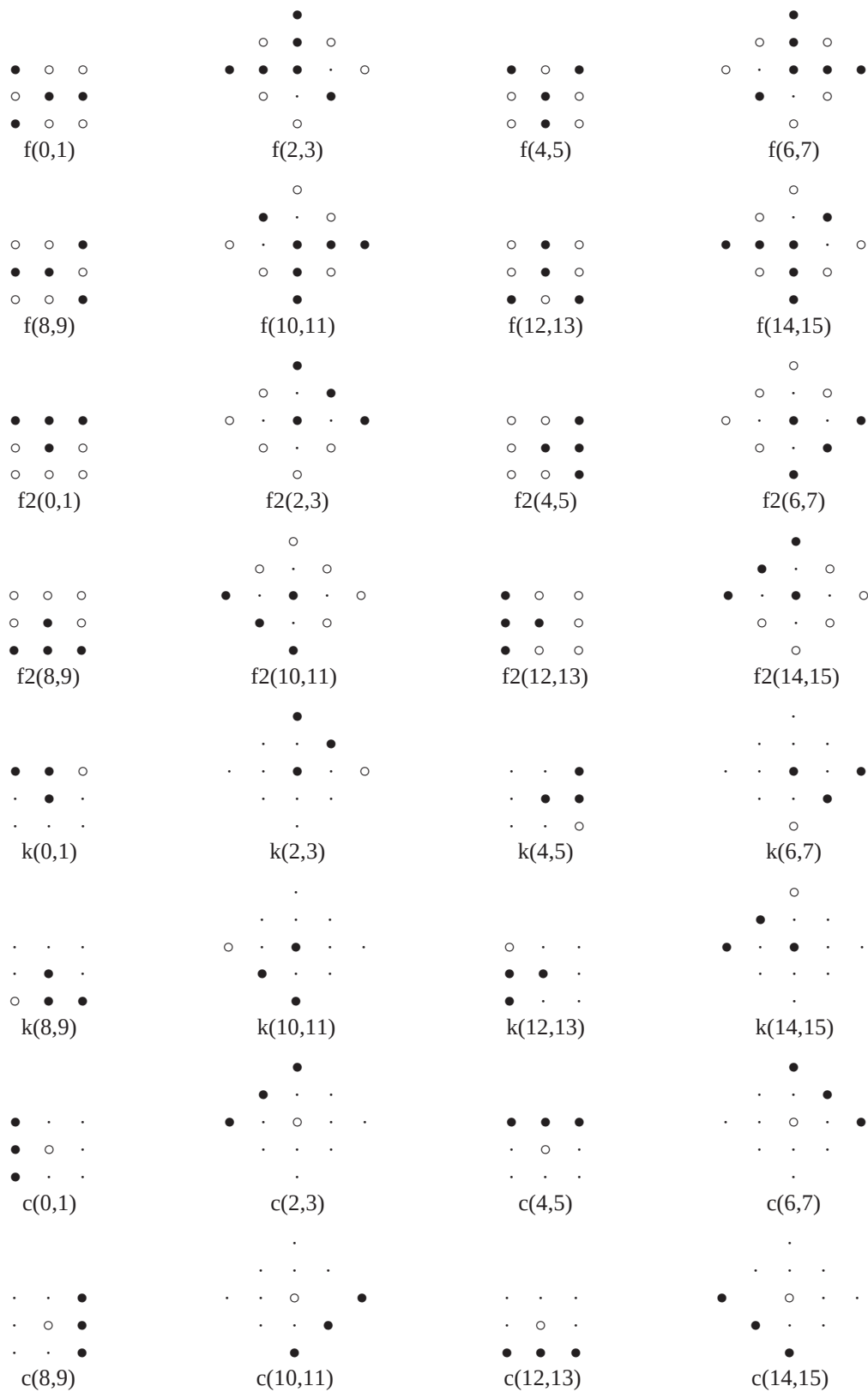
For [golay_elements](#) only even values are accepted, and determine the Golay element for [StructElement1](#). The next larger odd value is used for [StructElement2](#). There are no rotations for the Golay elements 'h' and 'i'. Therefore, only the values 0 and 1 are possible as "rotations" (and hence only 0 for [golay_elements](#)). The element 'e' has only four possible rotations, and hence the rotation must be between 0 and 7 (for [golay_elements](#) the values 0, 2, 4, or 6 must be used).

The tables below show the elements of the Golay alphabet with all possible rotations. The characters used have the following meaning:

- Foreground pixel
- Background pixel
- Don't care pixel

The names of the elements and their rotation numbers are displayed below the respective element. The elements with even number contain the foreground pixels, while the elements with odd numbers contain the background pixels.





Parameter

- ▷ **StructElement1** (output_object) region $\leadsto$ Hobject
Structuring element for the foreground.
- ▷ **StructElement2** (output_object) region $\leadsto$ Hobject
Structuring element for the background.

- ▷ **GolayElement** (input_control) string $\leadsto$ *string*
Name of the structuring element.
Default Value : 'l'
List of values : GolayElement $\in \{ 'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k' \}$
- ▷ **Rotation** (input_control) integer $\leadsto$ *integer*
Rotation of the Golay element. Depending on the element, not all rotations are valid.
Default Value : 0
List of values : Rotation $\in \{ 0, 2, 4, 6, 8, 10, 12, 14 \}$
- ▷ **Row** (input_control) point.y $\leadsto$ *integer*
Row coordinate of the reference point.
Default Value : 16
Suggested values : Row $\in \{ 0, 16, 32, 128, 256 \}$
Typical range of values : $0 \leq \text{Row} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Column** (input_control) point.x $\leadsto$ *integer*
Column coordinate of the reference point.
Default Value : 16
Suggested values : Column $\in \{ 0, 16, 32, 128, 256 \}$
Typical range of values : $0 \leq \text{Column} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1

Result

[golay_elements](#) returns 2 (H_MSG_TRUE) if all parameters are correct. Otherwise, an exception is raised.

Parallelization Information

[golay_elements](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[hit_or_miss](#)

Alternatives

[gen_region_points](#), [gen_struct_elements](#), [gen_region_polygon_filled](#)

See also

[dilation_golay](#), [erosion_golay](#), [opening_golay](#), [closing_golay](#), [hit_or_miss_golay](#), [thickening_golay](#)

References

J. Serra: "Image Analysis and Mathematical Morphology". Volume I. Academic Press, 1982

Module

Morphology

hit_or_miss (Region, StructElement1,
StructElement2 : RegionHitMiss : Row, Column :)

Hit-or-miss operation for regions.

[hit_or_miss](#) performs the hit-or-miss-transformation. First, an erosion with the structuring element [StructElement1](#) is done on the input region [Region](#). Then an erosion with the structuring element [StructElement2](#) is performed on the complement of the input region. The intersection of the two resulting regions is the result [RegionHitMiss](#) of [hit_or_miss](#).

The hit-or-miss-transformation selects precisely the points for which the conditions given by the structuring elements [StructElement1](#) and [StructElement2](#) are fulfilled. [StructElement1](#) determines the condition for the foreground pixels, while [StructElement2](#) determines the condition for the background pixels. In order to obtain sensible results, [StructElement1](#) and [StructElement2](#) must fit like key and lock. In any case, [StructElement1](#) and [StructElement2](#) must be disjunct. [Row](#) and [Column](#) determine the reference point of the structuring elements.

Structuring elements ([StructElement1](#), [StructElement2](#)) can be generated by calling operators like [gen_struct_elements](#), [gen_region_points](#), etc.

Parameter	
▷ Region (input_object)	region(-array) $\leadsto$ Hobject Regions to be processed.
▷ StructElement1 (input_object)	region $\leadsto$ Hobject Erosion mask for the input regions.
▷ StructElement2 (input_object)	region $\leadsto$ Hobject Erosion mask for the complements of the input regions.
▷ RegionHitMiss (output_object)	region(-array) $\leadsto$ Hobject Result of the hit-or-miss operation.
▷ Row (input_control)	point.y $\leadsto$ integer Row coordinate of the reference point. Default Value : 16 Suggested values : Row $\in \{0, 16, 32, 128, 256\}$ Typical range of values : $0 \leq \text{Row} \leq 511$ (lin) Minimum Increment : 1 Recommended Increment : 1
▷ Column (input_control)	point.x $\leadsto$ integer Column coordinate of the reference point. Default Value : 16 Suggested values : Column $\in \{0, 16, 32, 128, 256\}$ Typical range of values : $0 \leq \text{Column} \leq 511$ (lin) Minimum Increment : 1 Recommended Increment : 1

Complexity

Let F be the area of an input region, $F1$ the area of the structuring element 1, and $F2$ the area of the structuring element 2. Then the runtime complexity for one object is:

$$O\left(\sqrt{F} \cdot \left(\sqrt{F1} + \sqrt{F2}\right)\right) .$$

Result

hit_or_miss returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

hit_or_miss is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`golya_elements`, `gen_struct_elements`, `threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`

Possible Successors

`difference`, `reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`hit_or_miss_golya`, `hit_or_miss_seq`, `erosion2`, `dilation2`

See also

`thinning`, `thickening`, `gen_region_points`, `gen_region_polygon_filled`

Module

Morphology

hit_or_miss_golya (Region : RegionHitMiss : GolyaElement, Rotation :)

Hit-or-miss operation for regions using the Golay alphabet.

`hit_or_miss_golay` performs the hit-or-miss-transformation for the input regions `Region` (using structuring elements from the Golay alphabet). First, an erosion with the foreground of the structuring element `GolayElement` is done on the input region `Region`. Then an erosion with the background of the structuring element `GolayElement` is performed on the complement of the input region. The intersection of the two resulting regions is the result `RegionHitMiss` of `hit_or_miss_golay`. The following structuring elements are available:

'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'.

The rotation number `Rotation` determines which rotation of the element should be used. The hit-or-miss-transformation selects precisely the points for which the conditions given by the selected Golay element are fulfilled.

Attention

Not all values of `Rotation` are valid for any Golay element.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be processed.
- ▷ **RegionHitMiss** (output_object) region(-array) $\leadsto$ *Hobject*
Result of the hit-or-miss operation.
- ▷ **GolayElement** (input_control) string $\leadsto$ *string*
Structuring element from the Golay alphabet.
Default Value : 'h'
List of values : `GolayElement` $\in$ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}
- ▷ **Rotation** (input_control) integer $\leadsto$ *integer*
Rotation of the Golay element. Depending on the element, not all rotations are valid.
Default Value : 0
List of values : `Rotation` $\in$ {0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15}

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(6 \cdot \sqrt{F}) .$$

Result

`hit_or_miss_golay` returns 2 (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

`hit_or_miss_golay` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`hit_or_miss_seq`, `hit_or_miss`

See also

`erosion_golay`, `dilation_golay`, `opening_golay`, `closing_golay`, `thinning_golay`, `thickening_golay`, `golay_elements`

Module

Morphology

hit_or_miss_seq (Region : RegionHitMiss : GolayElement :)

Hit-or-miss operation for regions using the Golay alphabet (sequential).

hit_or_miss_golay performs the hit-or-miss-transformation for the input regions **Region** using all rotations of a structuring element from the Golay alphabet. The result of the operator is the union of all intermediate results of the respective rotations. The following structuring elements are available:

'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'.

The Golay elements, together with all possible rotations, are described with the operator **golay_elements**.

Parameter	
▷ Region (input_object)	region(-array) $\leadsto$ Hobject Regions to be processed.
▷ RegionHitMiss (output_object)	region(-array) $\leadsto$ Hobject Result of the hit-or-miss operation.
▷ GolayElement (input_control)	string $\leadsto$ string Structuring element from the Golay alphabet. Default Value : 'h' List of values : GolayElement $\in$ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}

Complexity

Let F be the area of an input region, and R be the number of rotations. Then the runtime complexity for one region is:

$$O(R \cdot 6 \cdot \sqrt{F}) .$$

Result

hit_or_miss_seq returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: **set_system('no_object_result', <RegionResult>)**
- empty region: **set_system('empty_region_result', <RegionResult>)**

Otherwise, an exception is raised.

Parallelization Information
hit_or_miss_seq is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).
Possible Predecessors
threshold , regiongrowing , connection , union1 , watersheds , class_ndim_norm
Possible Successors
reduce_domain , select_shape , area_center , connection
Alternatives
hit_or_miss_golay , hit_or_miss
See also
thinning_seq , thickening_seq
Module

Morphology

minkowski_add1 (Region, StructElement : RegionMinkAdd : Iterations :)

Perform a Minkowski addition on a region.

minkowski_add1 dilates the input regions with a structuring element. By applying **minkowski_add1** to a region, its boundary gets smoothed. In the process, the area of the region is enlarged. Furthermore, disconnected regions may be merged. Such regions, however, remain logically distinct region. The Minkowski addition is a set-theoretic region operation. It is based on translations and union operations.

Let M (**StructElement**) and R (**Region**) be two regions, where M is the structuring element and R is the region to be processed. Furthermore, let m be a point in M . Then the displacement vector $\vec{v}_m = (dx, dy)$ is defined as the difference of the center of gravity of M and the vector $\vec{m}$. Let $t_{\vec{v}_m}(R)$ denote the translation of a region R by a vector $\vec{v}$. Then

$$\text{minkowski\_add1}(R, M) := \bigcup_{m \in M} t_{\vec{v}_m}(R)$$

For each point m in M a translation of the region R is performed. The union of all these translations is the Minkowski addition of R with M . **minkowski_add1** is similar to the operator **dilation1**, the difference is that in **dilation1** the structuring element is mirrored at the origin. The position of **StructElement** is meaningless, since the displacement vectors are determined with respect to the center of gravity of M .

The parameter **Iterations** determines the number of iterations which are to be performed with the structuring element. The result of iteration $n - 1$ is used as input for iteration n . From the above definition it follows that an empty region is generated in case of an empty structuring element.

Structuring elements (**StructElement**) can be generated with operators such as **gen_circle**, **gen_rectangle1**, **gen_rectangle2**, **gen_ellipse**, **draw_region**, **gen_region_polygon**, **gen_region_points**, etc.

Attention

A Minkowski addition always results in enlarged regions. Closely spaced regions which may touch or overlap as a result of the dilation are still treated as two separate regions. If the desired behavior is to merge them into one region, the operator **union1** has to be called first.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be dilated.
- ▷ **StructElement** (input_object) region $\leadsto$ *Hobject*
Structuring element.
- ▷ **RegionMinkAdd** (output_object) region(-array) $\leadsto$ *Hobject*
Dilated regions.
- ▷ **Iterations** (input_control) integer $\leadsto$ *integer*
Number of iterations.
Default Value : 1
Suggested values : Iterations $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50\}$
Typical range of values : $1 \leq \text{Iterations}$ (lin)
Minimum Increment : 1
Recommended Increment : 1

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \text{Iterations}) .$$

Result

minkowski_add1 returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: **set_system('no_object_result', <RegionResult>)**
- empty region: **set_system('empty_region_result', <RegionResult>)**

Otherwise, an exception is raised.

Parallelization Information

minkowski_add1 is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

threshold, **regiongrowing**, **connection**, **union1**, **watersheds**, **class_ndim_norm**, **gen_circle**, **gen_ellipse**, **gen_rectangle1**, **gen_rectangle2**, **draw_region**, **gen_region_points**, **gen_struct_elements**, **gen_region_polygon_filled**

<i>Possible Successors</i>
reduce_domain , select_shape , area_center , connection
<i>Alternatives</i>
minkowski_add2 , dilation1
<i>See also</i>
transpose_region , minkowski_sub1
<i>Module</i>
Morphology

minkowski_add2 (<i>Region</i> , <i>StructElement</i> : <i>RegionMinkAdd</i> : <i>Row</i> , <i>Column</i> , <i>Iterations</i> :)

Dilate a region (using a reference point).

minkowski_add2 computes the Minkowski addition of the input regions with a structuring element (**StructElement**) having the reference point (**Row**,**Column**). **minkowski_add2** has a similar effect as **minkowski_add1**, the difference is that the reference point of the structuring element can be chosen arbitrarily. The parameter **Iterations** determines the number of iterations which are to be performed with the structuring element. The result of iteration $n - 1$ is used as input for iteration n .

An empty region is generated in case of an empty structuring element.

Structuring elements (**StructElement**) can be generated with operators such as [gen_circle](#), [gen_rectangle1](#), [gen_rectangle2](#), [gen_ellipse](#), [draw_region](#), [gen_region_polygon](#), [gen_region_points](#), etc.

<i>Attention</i>
A Minkowski addition always results in enlarged regions. Closely spaced regions which may touch or overlap as a result of the dilation are still treated as two separate regions. If the desired behavior is to merge them into one region, the operator union1 has to be called first.
<i>Parameter</i>

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be dilated.
- ▷ **StructElement** (input_object) region $\leadsto$ *Hobject*
Structuring element.
- ▷ **RegionMinkAdd** (output_object) region(-array) $\leadsto$ *Hobject*
Dilated regions.
- ▷ **Row** (input_control) point.y $\leadsto$ *integer*
Row coordinate of the reference point.
Typical range of values : $1 \leq \text{Row} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Column** (input_control) point.x $\leadsto$ *integer*
Column coordinate of the reference point.
Typical range of values : $1 \leq \text{Column} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Iterations** (input_control) integer $\leadsto$ *integer*
Number of iterations.
Default Value : 1
Suggested values : $\text{Iterations} \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50\}$
Typical range of values : $1 \leq \text{Iterations}$ (lin)
Minimum Increment : 1
Recommended Increment : 1

<i>Complexity</i>
Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \text{Iterations}) .$$

Result

`minkowski_add2` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

`minkowski_add2` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`,
`gen_circle`, `gen_ellipse`, `gen_rectangle1`, `gen_rectangle2`, `draw_region`,
`gen_region_points`, `gen_struct_elements`, `gen_region_polygon_filled`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`minkowski_add1`, `dilation1`

See also

`transpose_region`

Module

Morphology

minkowski_sub1 (Region,
 StructElement : RegionMinkSub : Iterations :)

Erode a region.

`minkowski_sub1` computes the Minkowski subtraction of the input regions with a structuring element. By applying `minkowski_sub1` to a region, its boundary gets smoothed. In the process, the area of the region is reduced. Furthermore, connected regions may be split. Such regions, however, remain logically one region. The Minkowski subtraction is a set-theoretic region operation. It uses the intersection operation.

Let M (**StructElement**) and R (**Region**) be two regions, where M is the structuring element and R is the region to be processed. Furthermore, let m be a point in M . Then the displacement vector $\vec{v}_m = (dx, dy)$ is defined as the difference of the center of gravity of M and the vector $\vec{m}$. Let $t_{\vec{v}_m}(R)$ denote the translation of a region R by a vector $\vec{v}$. Then

$$\text{minkowski\_sub1}(R, M) := \bigcap_{m \in M} t_{\vec{v}_m}(R)$$

For each point m in M a translation of the region R is performed. The intersection of all these translations is the Minkowski subtraction of R with M . `minkowski_sub1` is similar to the operator `erosion1`, the difference is that in `erosion1` the structuring element is mirrored at the origin. The position of **StructElement** is meaningless, since the displacement vectors are determined with respect to the center of gravity of M .

The parameter **Iterations** determines the number of iterations which are to be performed with the structuring element. The result of iteration $n - 1$ is used as input for iteration n . From the above definition it follows that the maximum region is generated in case of an empty structuring element.

Structuring elements (**StructElement**) can be generated with operators such as `gen_circle`, `gen_rectangle1`, `gen_rectangle2`, `gen_ellipse`, `draw_region`, `gen_region_polygon`, `gen_region_points`, etc.

Parameter	
▷ Region (input_object)	region(-array) $\leadsto$ Hobject Regions to be eroded.
▷ StructElement (input_object)	region $\leadsto$ Hobject Structuring element.
▷ RegionMinkSub (output_object)	region(-array) $\leadsto$ Hobject Eroded regions.
▷ Iterations (input_control)	integer $\leadsto$ integer Number of iterations. Default Value : 1 Suggested values : Iterations $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50\}$ Typical range of values : $1 \leq \text{Iterations}$ (lin) Minimum Increment : 1 Recommended Increment : 1

Complexity
Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \text{Iterations}) .$$

Result
`minkowski_sub1` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information
`minkowski_sub1` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors
`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`, `gen_circle`, `gen_ellipse`, `gen_rectangle1`, `gen_rectangle2`, `draw_region`, `gen_region_points`, `gen_struct_elements`, `gen_region_polygon_filled`

Possible Successors
`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives
`minkowski_sub2`, `erosion1`

See also
`transpose_region`

Module
Morphology

minkowski_sub2 (Region, StructElement : RegionMinkSub : Row, Column, Iterations :)

Erode a region (using a reference point).

`minkowski_sub2` computes the Minkowski subtraction of the input regions with a structuring element (`StructElement`) having the reference point (`Row, Column`). `minkowski_sub2` has a similar effect as `minkowski_sub1`, the difference is that the reference point of the structuring element can be chosen arbitrarily. The parameter `Iterations` determines the number of iterations which are to be performed with the structuring element. The result of iteration $n - 1$ is used as input for iteration n .

A maximum region is generated in case of an empty structuring element.

Structuring elements (`StructElement`) can be generated with operators such as `gen_circle`, `gen_rectangle1`, `gen_rectangle2`, `gen_ellipse`, `draw_region`, `gen_region_polygon`, `gen_region_points`, etc.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be eroded.
- ▷ **StructElement** (input_object) region $\leadsto$ *Hobject*
Structuring element.
- ▷ **RegionMinkSub** (output_object) region(-array) $\leadsto$ *Hobject*
Eroded regions.
- ▷ **Row** (input_control) point.y $\leadsto$ *integer*
Row coordinate of the reference point.
Default Value : 0
Suggested values : Row $\in \{0, 10, 16, 32, 64, 100, 128\}$
Typical range of values : $0 \leq \text{Row} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Column** (input_control) point.x $\leadsto$ *integer*
Column coordinate of the reference point.
Default Value : 0
Suggested values : Column $\in \{0, 10, 16, 32, 64, 100, 128\}$
Typical range of values : $0 \leq \text{Column} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Iterations** (input_control) integer $\leadsto$ *integer*
Number of iterations.
Default Value : 1
Suggested values : Iterations $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50\}$
Typical range of values : $1 \leq \text{Iterations}$ (lin)
Minimum Increment : 1
Recommended Increment : 1

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \text{Iterations}) .$$

Result

`minkowski_sub2` returns 2 (`H_MSG_TRUE`) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

`minkowski_sub2` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`threshold`, `regiongrowing`, `watersheds`, `class_ndim_norm`, `gen_circle`, `gen_ellipse`, `gen_rectangle1`, `gen_rectangle2`, `draw_region`, `gen_region_points`, `gen_struct_elements`, `gen_region_polygon_filled`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`minkowski_sub1`, `erosion1`, `erosion2`, `erosion_golay`, `erosion_seq`

See also

[gen_circle](#), [gen_rectangle2](#), [gen_region_polygon](#)

Module

Morphology

morph_hat (Region, StructElement : RegionMorphHat : :)

Compute the union of [bottom_hat](#) and [top_hat](#).

morph_hat computes the union of the regions that are removed by an [opening](#) operation with the regions that are added by a [closing](#) operation. Hence this is the union of the results of [top_hat](#) and [bottom_hat](#). The position of **StructElement** does not influence the result.

Structuring elements (**StructElement**) can be generated with operators such as [gen_circle](#), [gen_rectangle1](#), [gen_rectangle2](#), [gen_ellipse](#), [draw_region](#), [gen_region_polygon](#), [gen_region_points](#), etc.

Attention

The individual regions are processed separately.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be processed.
- ▷ **StructElement** (input_object) region $\leadsto$ *Hobject*
Structuring element (position-invariant).
- ▷ **RegionMorphHat** (output_object) region(-array) $\leadsto$ *Hobject*
Union of top hat and bottom hat.

Example

```
#include <iostream.h>
#include "HalconCpp.h"

main()
{
    cout << "Reproduction of 'dilation_circle ()'" << endl;
    cout << "First = original image " << endl;
    cout << "Red   = after segmentation " << endl;
    cout << "Blue  = after erosion " << endl;

    HByteImage img("monkey");
    HWindow      w;

    HRegion      circ   = HRegion::GenCircle (10, 10, 1.5);
    HRegionArray regs   = (img >= 128).Connection();
    HRegionArray tophat = regs.TopHat (circ);
    HRegionArray bothat = regs.BottomHat (circ);
    HRegionArray unionX = tophat.Union2 (bothat);

    img.Display (w);      w.Click ();
    w.SetColor ("red");   regs.Display (w);   w.Click ();
    w.SetColor ("blue");  tophat.Display (w);  w.Click ();
    w.SetColor ("green"); bothat.Display (w);  w.Click ();
    w.SetColor ("white"); unionX.Display (w);  w.Click ();

    return(0);
}
```

Result

morph_hat returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

<i>Parallelization Information</i>	
morph_hat	is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).
<i>Possible Predecessors</i>	
threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm, gen_circle, gen_ellipse, gen_rectangle1, gen_rectangle2, draw_region, gen_region_points, gen_struct_elements, gen_region_polygon_filled	
<i>Possible Successors</i>	
reduce_domain, select_shape, area_center, connection	
<i>Alternatives</i>	
top_hat, bottom_hat, union2	
<i>See also</i>	
opening, closing	
<i>Module</i>	
Morphology	

morph_skeleton (Region : RegionSkeleton : :)

Compute the morphological skeleton of a region.

morph_skeleton computes the skeleton of the input regions (**Region**) using morphological transformations. The computation yields a disconnected skeleton (gaps in the diagonals) having a width of one or two pixels. The calculation uses the Golay element 'h', i.e., an 8-neighborhood. This is equivalent to the maximum-norm.

<i>Parameter</i>	
▷ Region (input_object)	region(-array) $\leadsto$ Hobject Regions to be processed.
▷ RegionSkeleton (output_object)	region(-array) $\leadsto$ Hobject Resulting morphological skeleton.
<i>Result</i>	
morph_skeleton returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:	
• no region: <code>set_system('no_object_result', <RegionResult>)</code> • empty region: <code>set_system('empty_region_result', <RegionResult>)</code>	
Otherwise, an exception is raised.	
<i>Parallelization Information</i>	
morph_skeleton	is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).
<i>Possible Predecessors</i>	
threshold, regiongrowing, connection, union1, watersheds, class_ndim_norm	
<i>Possible Successors</i>	
skeleton, reduce_domain, select_shape, area_center, connection	
<i>Alternatives</i>	
skeleton, thinning	
<i>See also</i>	
thinning_seq, morph_skiz	
<i>Module</i>	
Morphology	

morph_skiz (Region : RegionSkiz : Iterations1, Iterations2 :)

Thinning of a region.

morph_skiz first performs a sequential thinning (**thinning_seq**) of the input region with the element 'l' of the Golay alphabet. The number of iterations is determined by the parameter **Iterations1**. Then a sequential thinning of the resulting region with the element 'e' of the Golay alphabet is carried out. The number of iterations for this step is determined by the parameter **Iterations2**. The skiz operation serves to compute a kind of skeleton of the input regions, and to prune the branches of the resulting skeleton. If the skiz operation is applied to the complement of the region, the region and the resulting skeleton are separated.

If very large values or 'maximal' are passed for **Iterations1** or **Iterations2**, the processing stops if no more changes occur.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be thinned.
- ▷ **RegionSkiz** (output_object) region(-array) $\leadsto$ *Hobject*
Result of the skiz operator.
- ▷ **Iterations1** (input_control) integer $\leadsto$ integer / string
Number of iterations for the sequential thinning with the element 'l' of the Golay alphabet.
Default Value : 100
Suggested values : $\text{Iterations1} \in \{\text{'maximal'}, 0, 1, 2, 3, 5, 7, 10, 15, 20, 30, 40, 50, 70, 100, 150, 200, 300, 400\}$
Typical range of values : $0 \leq \text{Iterations1} \text{ (lin)}$
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Iterations2** (input_control) integer $\leadsto$ integer / string
Number of iterations for the sequential thinning with the element 'e' of the Golay alphabet.
Default Value : 1
Suggested values : $\text{Iterations2} \in \{\text{'maximal'}, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50\}$
Typical range of values : $0 \leq \text{Iterations2} \text{ (lin)}$
Minimum Increment : 1
Recommended Increment : 1

Complexity

Let F be the area of the input region. Then the runtime complexity for one region is

$$O((\text{Iterations1} + \text{Iterations2}) \cdot 3 \cdot \sqrt{F}) .$$

Result

morph_skiz returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: **set_system('no_object_result', <RegionResult>)**
- empty region: **set_system('empty_region_result', <RegionResult>)**

Otherwise, an exception is raised.

Parallelization Information

morph_skiz is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

threshold, **regiongrowing**, **connection**, **union1**, **watersheds**, **class_ndim_norm**

Possible Successors

pruning, **reduce_domain**, **select_shape**, **area_center**, **connection**, **background_seg**, **complement**

Alternatives

skeleton, **thinning_seq**, **morph_skeleton**, **interjacent**

See also

thinning, **hit_or_miss_seq**, **difference**

Module

Morphology

opening (Region, StructElement : RegionOpening : :)

Open a region.

An **opening** operation is defined as an erosion followed by a Minkowski addition. By applying **opening** to a region, larger structures remain mostly intact, while small structures like lines or points are eliminated. In contrast, a **closing** operation results in small gaps being retained or filled up (see **closing**).

opening serves to eliminate small regions (smaller than **StructElement**) and to smooth the boundaries of a region. The position of **StructElement** is meaningless, since an opening operation is invariant with respect to the choice of the reference point.

Structuring elements (**StructElement**) can be generated with operators such as **gen_circle**, **gen_rectangle1**, **gen_rectangle2**, **gen_ellipse**, **draw_region**, **gen_region_polygon**, **gen_region_points**, etc.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ Hobject
Regions to be opened.
- ▷ **StructElement** (input_object) region $\leadsto$ Hobject
Structuring element (position-invariant).
- ▷ **RegionOpening** (output_object) region(-array) $\leadsto$ Hobject
Opened regions.

Example

```
/* Large regions in an aerial picture (beech trees or meadows): */
read_image(Image, 'wald1')
threshold(Image, Light, 80, 255)
gen_circle(StructElement1, 100, 100, 2)
gen_circle(StructElement2, 100, 100, 20)
/* close the small gap */
closing(Light, StructElement1, H)
/* selecting the large regions */
opening(H, StructElement2, Large).
```

```
/* Selecting of edges with certain orientation: */
read_image(Image, 'fabrik')
sobel_amp(Image, Sobel, 'sum_abs', 3)
threshold(Sobel, Edges, 10, 255)
gen_rectangle2(StructElement, 100, 100, 3.07819, 20, 1)
opening(Edges, StructElement, Direction).
```

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(2 \cdot \sqrt{F1} \cdot \sqrt{F2}) .$$

Result

opening returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: **set_system('no_object_result', <RegionResult>)**
- empty region: **set_system('empty_region_result', <RegionResult>)**

Otherwise, an exception is raised.

Parallelization Information

`opening` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`,
`gen_circle`, `gen_ellipse`, `gen_rectangle1`, `gen_rectangle2`, `draw_region`,
`gen_region_points`, `gen_struct_elements`, `gen_region_polygon_filled`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`minkowski_add1`, `erosion1`, `opening_circle`

See also

`gen_circle`, `gen_rectangle2`, `gen_region_polygon`

Module

Morphology

opening_circle (Region : RegionOpening : Radius :)

Open a region with a circular structuring element.

`opening_circle` is defined as an erosion followed by a Minkowski addition with a circular structuring element (see example). `opening` serves to eliminate small regions (smaller than the circular structuring element) and to smooth the boundaries of a region.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be opened.
- ▷ **RegionOpening** (output_object) region(-array) $\leadsto$ *Hobject*
Opened regions.
- ▷ **Radius** (input_control) real $\leadsto$ *real* / integer
Radius of the circular structuring element.
Default Value : 3.5
Suggested values : Radius $\in \{1.5, 2.5, 3.5, 4.5, 5.5, 7.5, 9.5, 12.5, 15.5, 19.5, 25.5, 33.5, 45.5, 60.5, 110.5\}$
Typical range of values : $0.5 \leq \text{Radius} \leq 511.5$ (lin)
Minimum Increment : 1.0
Recommended Increment : 1.0

Example

```
/* Large regions in an aerial picture (beech trees or meadows): */
read_image(Image, 'wald1')
threshold(Image, Light, 80, 255)
/* close the small gap */
closing_circle(LightH, 2)
/* selecting the large regions */
opening_circle(H, Large, 20).
```

Complexity

Let $F1$ be the area of the input region. Then the runtime complexity for one region is:

$$O(4 \cdot \sqrt{F1} \cdot \text{Radius}) .$$

Result

`opening_circle` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

	Parallelization Information	
<code>opening_circle</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>).		
	Possible Predecessors	
<code>threshold</code> , <code>regiongrowing</code> , <code>connection</code> , <code>union1</code> , <code>watersheds</code> , <code>class_ndim_norm</code>		
	Possible Successors	
<code>reduce_domain</code> , <code>select_shape</code> , <code>area_center</code> , <code>connection</code>		
	Alternatives	
<code>opening</code> , <code>dilation1</code> , <code>minkowski_add1</code> , <code>gen_circle</code>		
	See also	
<code>transpose_region</code>		
	Module	
Morphology		

opening_golay (Region : RegionOpening : GolayElement, Rotation :)

Open a region with an element from the Golay alphabet.

`opening_golay` is defined as a Minkowski subtraction followed by a Minkowski addition. First the Minkowski subtraction of the input region (`Region`) with the structuring element from the Golay alphabet defined by `GolayElement` and `Rotation` is computed. Then the Minkowski addition of the result and the structuring element rotated by 180° is performed.

The following structuring elements are available:

'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'.

The rotation number `Rotation` determines which rotation of the element should be used, and whether the foreground (even) or background version (odd) of the selected element should be used. The Golay elements, together with all possible rotations, are described with the operator `golay_elements`.

`opening_golay` serves to eliminate regions smaller than the structuring element, and to smooth regions' boundaries.

	Attention	
Not all values of <code>Rotation</code> are valid for any Golay element. For some of the values of <code>Rotation</code> , the resulting regions are identical to the input regions.		

	Parameter	
▷	Region (input_object) region(-array) $\leadsto$ <i>Hobject</i> Regions to be opened.	
▷	RegionOpening (output_object) region(-array) $\leadsto$ <i>Hobject</i> Opened regions.	
▷	GolayElement (input_control) string $\leadsto$ <i>string</i> Structuring element from the Golay alphabet. Default Value : 'h' List of values : GolayElement $\in$ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}	
▷	Rotation (input_control) integer $\leadsto$ <i>integer</i> Rotation of the Golay element. Depending on the element, not all rotations are valid. Default Value : 0 List of values : Rotation $\in$ {0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15}	

	Complexity	
Let F be the area of an input region. Then the runtime complexity for one region is:		

$$O(6 \cdot \sqrt{F}) .$$

Result

`opening_golay` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

`opening_golay` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`opening_seg`, `opening`

See also

`erosion_golay`, `dilation_golay`, `closing_golay`, `hit_or_miss_golay`, `thinning_golay`, `thickening_golay`, `golay_elements`

Module

Morphology

<code>opening_rectangle1</code> (Region : RegionOpening : Width, Height :)

Open a region with a rectangular structuring element.

`opening_rectangle1` performs an `erosion_rectangle1` followed by a `dilation_rectangle1` on the input region `Region`. The size of the rectangular structuring element is determined by the parameters `Width` and `Height`. As is the case for all `opening` variants, larger structures are preserved, while small regions like lines or points are eliminated.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be opened.
- ▷ **RegionOpening** (output_object) region(-array) $\leadsto$ *Hobject*
Opened regions.
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer / real*
Width of the structuring rectangle.
Default Value : 10
Suggested values : `Width` $\in \{1, 2, 3, 4, 5, 7, 9, 12, 15, 19, 25, 33, 45, 60, 110, 150, 200\}$
Typical range of values : $1 \leq \text{Width} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer / real*
Height of the structuring rectangle.
Default Value : 10
Suggested values : `Height` $\in \{1, 2, 3, 4, 5, 7, 9, 12, 15, 19, 25, 33, 45, 60, 110, 150, 200\}$
Typical range of values : $1 \leq \text{Height} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1

Complexity

Let $F1$ be the area of an input region and H be the height of the rectangle. Then the runtime complexity for one region is:

$$O(2 \cdot \sqrt{F1} \cdot \text{ld}(H)) \text{ .}$$

Result

`opening_rectangle1` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

`opening_rectangle1` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`threshold`, `regiongrowing`, `watersheds`, `class_ndim_norm`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`opening`, `gen_rectangle1`, `dilation_rectangle1`, `erosion_rectangle1`

See also

`opening_seg`, `opening_golay`

Module

Morphology

opening_seg (Region, StructElement : RegionOpening : :)

Separate overlapping regions.

The `opening_seg` operation is defined as a sequence of the following operators: `erosion1`, `connection` and `dilation1` (see example). Only one iteration is done in `erosion1` and `dilation1`.

`opening_seg` serves to separate overlapping regions whose area of overlap is smaller than `StructElement`. It should be noted that the resulting regions can overlap without actually merging (see `expand_region`). `opening_seg` uses the center of gravity as the reference point of the structuring element.

Structuring elements (`StructElement`) can be generated with operators such as `gen_circle`, `gen_rectangle1`, `gen_rectangle2`, `gen_ellipse`, `draw_region`, `gen_region_polygon`, `gen_region_points`, etc.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ Hobject
Regions to be opened.
- ▷ **StructElement** (input_object) region $\leadsto$ Hobject
Structuring element (position-invariant).
- ▷ **RegionOpening** (output_object) region-array $\leadsto$ Hobject
Opened regions.

Example

```
/* Simulation of opening_seg */
opening_seg(Region, StructElement, RegionOpening):
    erosion1(Region, StructElement, H1, 1) >
    connection(H1, H2)
    dilation1(H2, StructElement, RegionOpening, 1)
    clear_obj([H1, H2]).
```

Complexity

Let $F1$ be the area of the input region, and $F2$ be the area of the structuring element. Then the runtime complexity for one region is:

$$O(\sqrt{F1} \cdot \sqrt{F2} \cdot \sqrt{\sqrt{F1}}) .$$

Result

opening_seg returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: **set_system('no_object_result', <RegionResult>)**
- empty region: **set_system('empty_region_result', <RegionResult>)**

Otherwise, an exception is raised.

Parallelization Information

opening_seg is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

threshold, **regiongrowing**, **connection**, **union1**, **watersheds**, **class_ndim_norm**,
gen_circle, **gen_ellipse**, **gen_rectangle1**, **gen_rectangle2**, **draw_region**,
gen_region_points, **gen_struct_elements**, **gen_region_polygon_filled**

Possible Successors

expand_region, **reduce_domain**, **select_shape**, **area_center**, **connection**

Alternatives

erosion1, **connection**, **dilation1**

Module

Morphology

pruning (**Region** : **RegionPrune** : **Length** :)

Prune the branches of a region.

pruning removes branches from a skeleton (**Region**) having a length less than **Length**. All other branches are preserved.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be processed.
- ▷ **RegionPrune** (output_object) region(-array) $\leadsto$ *Hobject*
Result of the pruning operation.
- ▷ **Length** (input_control) integer $\leadsto$ *integer*
Length of the branches to be removed.
Default Value : 2
Suggested values : $\text{Length} \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50\}$
Typical range of values : $1 \leq \text{Length} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1

Complexity

Let F be the area of the input region. Then the runtime complexity for one region is

$$O(\text{Length} \cdot 3 \cdot \sqrt{F}) .$$

Result

pruning returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: **set_system('no_object_result', <RegionResult>)**
- empty region: **set_system('empty_region_result', <RegionResult>)**

Otherwise, an exception is raised.

Parallelization Information

pruning is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

morph_skiz, **skeleton**, **thinning_seq**

Possible Successors

reduce_domain, **select_shape**, **area_center**, **connection**

See also

morph_skeleton, **junctions_skeleton**

Module

Morphology

thickening (**Region**, **StructElement1**,
StructElement2 : **RegionThick** : **Row**, **Column**, **Iterations** :)

Add the result of a hit-or-miss operation to a region.

thickening performs a thickening of the input regions using morphological operations. The operator first applies a hit-or-miss-transformation to **Region** (cf. **hit_or_miss**), and then adds the detected points to the input region. The parameter **Iterations** determines the number of iterations performed.

For the choice of the structuring elements **StructElement1** and **StructElement2**, as well as for **Row** and **Column**, the same restrictions described under **hit_or_miss** apply.

The structuring elements (**StructElement1** and **StructElement2**) can be generated by calling **golay_elements**, for example.

Attention

If the reference point is contained in **StructElement1** the input region remains unchanged.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be processed.
- ▷ **StructElement1** (input_object) region $\leadsto$ *Hobject*
Structuring element for the foreground.
- ▷ **StructElement2** (input_object) region $\leadsto$ *Hobject*
Structuring element for the background.
- ▷ **RegionThick** (output_object) region(-array) $\leadsto$ *Hobject*
Result of the thickening operator.
- ▷ **Row** (input_control) point.y $\leadsto$ *integer*
Row coordinate of the reference point.
Default Value : 16
Suggested values : $\text{Row} \in \{0, 2, 4, 8, 16, 32, 128\}$
Typical range of values : $0 \leq \text{Row} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Column** (input_control) point.x $\leadsto$ *integer*
Column coordinate of the reference point.
Default Value : 16
Suggested values : $\text{Column} \in \{0, 2, 4, 8, 16, 32, 128\}$
Typical range of values : $0 \leq \text{Column} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1

- ▷ **Iterations** (input_control) integer $\leadsto$ integer
 Number of iterations.
Default Value : 1
Suggested values : Iterations $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50, 70, 100, 200, 400\}$
Typical range of values : $1 \leq \text{Iterations} \text{ (lin)}$
Minimum Increment : 1
Recommended Increment : 1

Complexity

Let F be the area of an input region, $F1$ the area of the structuring element 1, and $F2$ the area of the structuring element 2. Then the runtime complexity for one object is:

$$O\left(\text{Iterations} \cdot \sqrt{F} \cdot \left(\sqrt{F1} + \sqrt{F2}\right)\right) .$$

Result

thickening returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

thickening is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`golay_elements`, `threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`,
`class_ndim_norm`, `gen_circle`, `gen_ellipse`, `gen_rectangle1`, `gen_rectangle2`,
`draw_region`, `gen_region_points`, `gen_struct_elements`, `gen_region_polygon_filled`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`thickening_golay`, `thickening_seq`

See also

`hit_or_miss`

Module

Morphology

thickening_golay (Region : RegionThick : GolayElement, Rotation :)

Add the result of a hit-or-miss operation to a region (using a Golay structuring element).

thickening_golay performs a thickening of the input regions using morphological operations and structuring elements from the Golay alphabet. The operator first applies a hit-or-miss-transformation to **Region** (cf. `hit_or_miss_golay`), and then adds the detected points to the input region. The following structuring elements are available:

'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'.

The rotation number **Rotation** determines which rotation of the element should be used. The Golay elements, together with all possible rotations, are described with the operator `golay_elements`.

Attention

Not all values of **Rotation** are valid for any Golay element.

Parameter	
▷ Region (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Regions to be processed.
▷ RegionThick (output_object)	region(-array) $\leadsto$ <i>Hobject</i> Result of the thickening operator.
▷ GolayElement (input_control)	string $\leadsto$ <i>string</i> Structuring element from the Golay alphabet. Default Value : 'h' List of values : GolayElement $\in \{ 'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k' \}$
▷ Rotation (input_control)	integer $\leadsto$ <i>integer</i> Rotation of the Golay element. Depending on the element, not all rotations are valid. Default Value : 0 List of values : Rotation $\in \{ 0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15 \}$

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(6 \cdot \sqrt{F}) .$$

Result

thickening_golay returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

thickening_golay is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`thickening`, `thickening_seq`

See also

`erosion_golay`, `hit_or_miss_golay`

Module

Morphology

thickening_seq (Region : RegionThick : GolayElement, Iterations :)

Add the result of a hit-or-miss operation to a region (sequential).

thickening_seq calculates the sequential thickening of the input regions with a structuring element from the Golay alphabet (**GolayElement**). To do so, **thickening_seq** calls the operator **thickening_golay** with all possible rotations of the structuring element **Iterations** times. The following structuring elements are available:

'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'.

The Golay elements, together with all possible rotations, are described with the operator **golay_elements**. For all elements of the Golay alphabet, except for 'c', the foreground and background masks are exchanged in order to have an effect for them on the outer boundary of the region. The element 'c' can be used to generate the convex hull of the input region if enough iterations are performed.

Parameter	
▷ Region (input_object)	region(-array) $\leadsto$ Hobject Regions to be processed.
▷ RegionThick (output_object)	region(-array) $\leadsto$ Hobject Result of the thickening operator.
▷ GolayElement (input_control)	string $\leadsto$ string Structuring element from the Golay alphabet. Default Value : 'h' List of values : GolayElement $\in \{ 'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k' \}$
▷ Iterations (input_control)	integer $\leadsto$ integer Number of iterations. Default Value : 1 Suggested values : Iterations $\in \{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50, 70, 100, 200 \}$ Typical range of values : $1 \leq \text{Iterations} \text{ (lin)}$ Minimum Increment : 1 Recommended Increment : 1

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(\text{Iterations} \cdot 6 \cdot \sqrt{F}) .$$

Result

thickening_seq returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

thickening_seq is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`thickening_golay`, `thickening`

See also

`erosion_golay`, `thinning_seq`

Module

Morphology

thinning (Region, StructElement1, StructElement2 : RegionThin : Row, Column, Iterations :)

Remove the result of a hit-or-miss operation from a region.

thinning performs a thinning of the input regions using morphological operations. The operator first applies a hit-or-miss-transformation to **Region** (cf. **hit_or_miss**), and then removes the detected points from the input region. The parameter **Iterations** determines the number of iterations performed.

For the choice of the structuring elements **StructElement1** and **StructElement2**, as well as for **Row** and **Column**, the same restrictions described under **hit_or_miss** apply.

Structuring elements (**StructElement1**, **StructElement2**) can be generated with operators such as `gen_circle`, `gen_rectangle1`, `gen_rectangle2`, `gen_ellipse`, `draw_region`, `gen_region_polygon`, `gen_region_points`, etc.

Parameter	
▷ Region (input_object)	region(-array) $\leadsto$ Hobject Regions to be processed.
▷ StructElement1 (input_object)	region $\leadsto$ Hobject Structuring element for the foreground.
▷ StructElement2 (input_object)	region $\leadsto$ Hobject Structuring element for the background.
▷ RegionThin (output_object)	region(-array) $\leadsto$ Hobject Result of the thinning operator.
▷ Row (input_control)	point.y $\leadsto$ integer Row coordinate of the reference point. Default Value : 0 Typical range of values : $0 \leq \text{Row} \leq 511$ (lin) Minimum Increment : 1 Recommended Increment : 1
▷ Column (input_control)	point.x $\leadsto$ integer Column coordinate of the reference point. Default Value : 0 Typical range of values : $0 \leq \text{Column} \leq 511$ (lin) Minimum Increment : 1 Recommended Increment : 1
▷ Iterations (input_control)	integer $\leadsto$ integer Number of iterations. Default Value : 1 Suggested values : Iterations $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 17, 20, 30, 40, 50\}$ Typical range of values : $1 \leq \text{Iterations}$ (lin) Minimum Increment : 1 Recommended Increment : 1

Complexity

Let F be the area of an input region, $F1$ the area of the structuring element 1, and $F2$ the area of the structuring element 2. Then the runtime complexity for one object is:

$$O\left(\text{Iterations} \cdot \sqrt{F} \cdot \left(\sqrt{F1} + \sqrt{F2}\right)\right) .$$

Result

thinning returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

thinning is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `union1`, `watersheds`, `class_ndim_norm`, `gen_circle`, `gen_ellipse`, `gen_rectangle1`, `gen_rectangle2`, `draw_region`, `gen_region_points`, `gen_struct_elements`, `gen_region_polygon_filled`

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

Alternatives

`thinning_golay`, `thinning_seq`

See also

`hit_or_miss`

Module

Morphology

thinning_golay (Region : RegionThin : GolayElement, Rotation :)

Remove the result of a hit-or-miss operation from a region (using a Golay structuring element).

thinning_golay performs a thinning of the input regions using morphological operations and structuring elements from the Golay alphabet. The operator first applies a hit-or-miss-transformation to **Region** (cf. **hit_or_miss_golay**), and then removes the detected points from the input region. The following structuring elements are available:

'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'.

The rotation number **Rotation** determines which rotation of the element should be used. The Golay elements, together with all possible rotations, are described with the operator **golay_elements**.

Attention

Not all values of **Rotation** are valid for any Golay element.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be processed.
- ▷ **RegionThin** (output_object) region(-array) $\leadsto$ *Hobject*
Result of the thinning operator.
- ▷ **GolayElement** (input_control) string $\leadsto$ *string*
Structuring element from the Golay alphabet.
Default Value : 'h'
List of values : GolayElement $\in$ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}
- ▷ **Rotation** (input_control) integer $\leadsto$ *integer*
Rotation of the Golay element. Depending on the element, not all rotations are valid.
Default Value : 0
List of values : Rotation $\in$ {0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15}

Complexity

Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(6 \cdot \sqrt{F}) .$$

Result

thinning_golay returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: **set_system('no_object_result', <RegionResult>)**
- empty region: **set_system('empty_region_result', <RegionResult>)**

Otherwise, an exception is raised.

Parallelization Information

thinning_golay is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

reduce_domain, **select_shape**, **area_center**, **connection**

Alternatives

thinning_seq, **thinning**

See also

erosion_golay, **hit_or_miss_golay**

Module

Morphology

thinning_seq (Region : RegionThin : GolayElement, Iterations :)

Remove the result of a hit-or-miss operation from a region (sequential).

thinning_seq calculates the sequential thinning of the input regions with a structuring element from the Golay alphabet (**GolayElement**). To do so, **thinning_seq** calls the operator **thinning_golay** with all possible rotations of the structuring element **Iterations** times. If **Iterations** is chosen large enough, the operator calculates the skeleton of a region if the structuring elements 'l' or 'm' are used. For the element 'c' the background and foreground are exchanged in order to have an effect on the interior boundary of a region. If a very large value or 'maximal' is passed for **Iterations** the iteration stops if no more changes occur. The following structuring elements are available:

- 'l' Skeleton, similar to **skeleton**. This structuring element is also used in **morph_skiz**.
- 'm' A skeleton with many "hairs" and multiple (parallel) branches.
- 'd' A skeleton without multiple branches, but with many gaps, similar to **morph_skeleton**.
- 'c' Uniform erosion of the region.
- 'e' One pixel wide lines are shortened. This structuring element is also used in **morph_skiz**.
- 'i' Isolated points are removed. (Only **Iterations** = 1 is useful.)
- 'f' Y-junctions are eliminated. (Only **Iterations** = 1 is useful.)
- 'f2' One pixel long branches and corners are removed. (Only **Iterations** = 1 is useful.)
- 'h' A kind of inner boundary, which, however, is thicker than the result of **boundary**, is generated. (Only **Iterations** = 1 is useful.)
- 'k' Junction points are eliminated, but also new ones are generated.

The Golay elements, together with all possible rotations, are described with the operator **golay_elements**.

Parameter	
▷ Region (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Regions to be processed.
▷ RegionThin (output_object)	region(-array) $\leadsto$ <i>Hobject</i> Result of the thinning operator.
▷ GolayElement (input_control)	string $\leadsto$ <i>string</i> Structuring element from the Golay alphabet. Default Value : 'l' List of values : GolayElement $\in$ {'l', 'm', 'd', 'c', 'e', 'i', 'f', 'f2', 'h', 'k'}
▷ Iterations (input_control)	integer $\leadsto$ <i>integer</i> / <i>string</i> Number of iterations. For 'f', 'f2', 'h' and 'i' the only useful value is 1. Default Value : 20 Suggested values : Iterations $\in$ {'maximal', 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 15, 20, 30, 40, 50, 70, 100, 150, 200} Typical range of values : $1 \leq \text{Iterations} \text{ (lin)}$ Minimum Increment : 1 Recommended Increment : 1

Complexity
Let F be the area of an input region. Then the runtime complexity for one region is:

$$O(\text{Iterations} \cdot 6 \cdot \sqrt{F}) .$$

Result
thinning_seq returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: **set_system('no_object_result', <RegionResult>)**
- empty region: **set_system('empty_region_result', <RegionResult>)**

Otherwise, an exception is raised.

Parallelization Information
thinning_seq is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).

Possible Predecessors

[threshold](#), [regiongrowing](#), [connection](#), [union1](#), [watersheds](#), [class_ndim_norm](#),
[gen_circle](#), [gen_ellipse](#), [gen_rectangle1](#), [gen_rectangle2](#), [draw_region](#),
[gen_region_points](#), [gen_struct_elements](#), [gen_region_polygon_filled](#)

Possible Successors

[pruning](#), [reduce_domain](#), [select_shape](#), [area_center](#), [connection](#), [complement](#)

Alternatives

[skeleton](#), [morph_skiz](#), [expand_region](#)

See also

[hit_or_miss_seq](#), [erosion_golay](#), [difference](#), [thinning_golay](#), [thinning](#),
[thickening_seq](#)

Module

Morphology

top_hat (Region , StructElement : RegionTopHat : :)

Compute the top hat of regions.

[top_hat](#) computes the [opening](#) of [Region](#) with [StructElement](#). The difference between the original region and the result of the opening is called the top hat. In contrast to [opening](#), which splits regions under certain circumstances, [top_hat](#) computes the regions removed by such a splitting.

The position of [StructElement](#) is meaningless, since an opening operation is invariant with respect to the choice of the reference point.

Structuring elements ([StructElement](#)) can be generated with operators such as [gen_circle](#), [gen_rectangle1](#), [gen_rectangle2](#), [gen_ellipse](#), [draw_region](#), [gen_region_polygon](#), [gen_region_points](#), etc.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be processed.
- ▷ **StructElement** (input_object) region $\leadsto$ *Hobject*
Structuring element (position independent).
- ▷ **RegionTopHat** (output_object) region(-array) $\leadsto$ *Hobject*
Result of the top hat operator.

Result

[top_hat](#) returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: [set_system\('no_object_result', <RegionResult>\)](#)
- empty region: [set_system\('empty_region_result', <RegionResult>\)](#)

Otherwise, an exception is raised.

Parallelization Information

[top_hat](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[threshold](#), [regiongrowing](#), [connection](#), [union1](#), [watersheds](#), [class_ndim_norm](#),
[gen_circle](#), [gen_ellipse](#), [gen_rectangle1](#), [gen_rectangle2](#), [draw_region](#),
[gen_region_points](#), [gen_struct_elements](#), [gen_region_polygon_filled](#)

Possible Successors

[reduce_domain](#), [select_shape](#), [area_center](#), [connection](#)

Alternatives

[opening](#), [difference](#)

See also

[bottom_hat](#), [morph_hat](#), [gray_tophat](#), [opening](#)

Morphology	Module

Chapter 11

Object

11.1 Information

count_obj (Objects : : : Number)

Number of objects in a tuple.

The operator [count_obj](#) determines for the object parameter [Objects](#) the number of objects it contains. In this connection it should be noted that object is not the same as connection component (see [connection](#)). For example, the number of objects of a region not consisting of three connected parts is 1.

Attention

In Prolog and Lisp the length of the list is not necessarily identical with the number of objects. This is the case when object keys are contained which were created in the compact mode (keys from compact and normal mode can be used as a mixture). See in this connection [set_system\(::'compact_object',<true/false>:\)](#).

Parameter

- ▷ **Objects** (input_object) object-array $\leadsto$ *Hobject*
Objects to be examined.
- ▷ **Number** (output_control) integer $\leadsto$ *integer*
Number of objects in the tuple [Objects](#).

Complexity

Runtime complexity: $O(|\text{Objects}|)$.

Result

If the surrogates are correct, i.e. all objects are present in the HALCON operator data base, the operator [count_obj](#) returns the value 2 (H_MSG.TRUE). The behavior in case of empty input (no input objects available) is set via the operator [set_system\(::'no_object_result',<Result>:\)](#).

Parallelization Information

[count_obj](#) is *reentrant* and processed *without* parallelization.

See also

[copy_obj](#), [obj_to_integer](#), [connection](#), [set_system](#)

Module

Basic operators

get_channel_info (Object : : Request, Channel : Information)

Informations about the components of an image object.

The operator [get_channel_info](#) gives information about the components of an image object. The following requests ([Request](#)) are currently possible:

'creator' Output of the names of the procedures which initially created the image components (not the object).

'type' Output of the type of image component ('byte', 'int1', 'int2', 'uint2', 'int4', 'real', 'direction', 'cyclic', 'complex', 'dvt', 'lut'). The component 0 is of type 'region' or 'xld'.

In the tuple `Channel` the numbers of the components about which information is required are stated. After carrying out `get_channel_info`, `Information` contains a tuple of strings (one string per entry in `Channel`) with the required information.

Parameter	
▷ Object (input_object)	object $\leadsto$ <i>Hobject</i> Image object to be examined.
▷ Request (input_control)	string $\leadsto$ <i>string</i> Required information about object components. Default Value : 'creator' List of values : Request $\in$ {'creator', 'type'}
▷ Channel (input_control)	channel(-array) $\leadsto$ <i>integer</i> Components to be examined (0 for region/XLD). Default Value : 0 Suggested values : Channel $\in$ {0, 1, 2, 3, 4, 5, 6, 7, 8}
▷ Information (output_control)	string(-array) $\leadsto$ <i>string</i> Requested information.

Result
If the parameters are correct the operator `get_channel_info` returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Parallelization Information
`get_channel_info` is *reentrant* and processed *without* parallelization.

Possible Predecessors
`read_image`

See also
`count_relation`

Module
Basic operators

get_obj_class (Object : : : Class)

Name of the class of an image object.

`get_obj_class` returns the name of the corresponding class to each object. The following classes are possible:

'image' Object with region (definition domain) and at least one channel.

'region' Object with a region without gray values.

'xld_cont' XLD object as contour

'xld_poly' XLD object as polygon

'xld_parallel' XLD object with parallel polygons

Parameter	
▷ Object (input_object)	object(-array) $\leadsto$ <i>Hobject</i> Image objects to be examined.
▷ Class (output_control)	string(-array) $\leadsto$ <i>string</i> Name of class.

Result
If the parameter values are correct the operator `get_obj_class` returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Parallelization Information

`get_obj_class` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Successors

`disp_image`, `disp_region`, `disp_xld`

See also

`get_channel_info`, `count_relation`

Module

Basic operators

test_equal_obj (Objects1, Objects2 : : :)

Compare image objects regarding equality.

The operator `test_equal_obj` compares the regions and gray value components of all objects of the two input parameters. The n-th object in `Objects1` is compared to the n-th object in `Objects2` (for all n). If all corresponding regions are equal and the number of regions is also identical the operator `test_equal_obj` returns the value TRUE, otherwise FALSE.

Attention

Image matrices are not compared regarding their contents. Thus, two images are “equal” if they are at the same place in the store. If the input parameters are empty and the behavior was set via the operator `set_system (::'no_object_result', 'true':)`, the operator `test_equal_obj` returns TRUE, since all input (= empty set) is equal.

Parameter

- ▷ **Objects1** (input_object) object-array $\leadsto$ Hobject
Test objects.
- ▷ **Objects2** (input_object) object-array $\leadsto$ Hobject
Comparative objects.

Complexity

If F is the area of a region the runtime complexity is $O(1)$ or $O(\sqrt{F})$ if the result is TRUE and $O(\sqrt{F})$ if the result is FALSE.

Result

The operator `test_equal_obj` returns the value 2 (H.MSG_TRUE) if both object tuples are identical. If the tuples differ in at least one place `test_equal_obj` returns 3 (H.MSG_FALSE). The behavior in case of empty input (no input objects available) is set via the operator `set_system (::'no_object_result', <Result>:)`. If the number of objects differs an exception is raised.

Parallelization Information

`test_equal_obj` is *reentrant* and processed *without parallelization*.

See also

`test_equal_region`

Return Value

bool

Module

Basic operators

test_equal_region (Regions1, Regions2 : : :)

Test whether the regions of two objects are identical.

The operator `test_equal_region` compares the regions of the two input parameters. The n-th element in `Regions1` is compared to the n-th object in `Regions2` (for all n). If all regions are equal and the number of regions is identical the operator `test_equal_region` returns the value TRUE, otherwise FALSE.

Parameter
▷ Regions1 (input_object) region(-array) $\leadsto$ Hobject Test regions. ▷ Regions2 (input_object) region(-array) $\leadsto$ Hobject Comparative regions. Number of elements : Regions1 = Regions2
Complexity
If F is the area of a region the runtime complexity is $O(1)$ or $O(\sqrt{F})$ if the result is TRUE, $O(\sqrt{F})$ if the result is FALSE.
Result
The operator <code>test_equal_region</code> returns the value 2 (H_MSG_TRUE) if both object tuples are identical. If the tuples differ in at least one place <code>test_equal_region</code> returns 3 (H_MSG_FALSE). The behavior in case of empty input (no input objects available) is set via the operator <code>set_system (::'no_object_result', <Result>:)</code>. If the number of objects differs an exception is raised.
Parallelization Information
<code>test_equal_region</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
<code>intersection</code>, <code>complement</code>, <code>area.center</code>
See also
<code>test_equal_obj</code>
Return Value
bool
Module
Basic operators

test_obj_def (Object : : :)

Test whether an object is already deleted.

The operator `test_obj_def` checks whether the object still exists in the HALCON operator data base (i.e. whether the surrogate is still valid). This check especially makes sense before deleting an object if it is not sure that the object has already been deleted by a prior deleting operator (`clear_obj`).

Attention

The operator `test_obj_def` can return TRUE even if the object was already deleted because the surrogates of deleted objects are re-used for new objects. In this context see the example.

Parameter
▷ Object (input_object) object $\leadsto$ Hobject Object to be checked.
Example

```

Herror err;
circle(&Circle,100.0,100.0,100.0);
err = test_obj_def(Circle);
printf("Result for test_obj_def (H_MSG_TRUE): %d\n",err);
clear_obj(Circle);
err = test_obj_def(Circle);
printf("Result for test_obj_def (H_MSG_FALSE): %d\n",err);
gen_rectangle1(&Rectangle,200.0,200.0,300.0,300.0);
err = test_obj_def(Circle);
printf("Result for test_obj_def (H_MSG_TRUE!!!): %d\n",err);

```

Complexity

The runtime complexity is $O(1)$.

Result

The operator `test_obj_def` returns the value 2 (H_MSG_TRUE) if an object with this surrogate is present in the HALCON operator data base, otherwise the value 3 (H_MSG_FALSE) is returned. The behavior in case of empty input (no input objects available) is set via the operator `set_system (:: 'no_object_result', <Result>:)`.

Parallelization Information

`test_obj_def` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`clear_obj`, `gen_circle`, `gen_rectangle1`

See also

`set_check`, `clear_obj`, `reset_obj_db`

Return Value

bool

Module

Basic operators

11.2 Manipulation

clear_obj (Objects : : :)

Delete an iconic object from the HALCON database.

`clear_obj` deletes iconic objects, which are no longer needed, from the HALCON database. It should be noted that `clear_obj` is the only way to delete objects from the database, and hence to reclaim their memory, in all host languages except Smalltalk and C++.

Images and regions are normally used by several iconic objects at the same time (uses less memory!). This has the consequence that a region or an image is only deleted if all objects using it have been deleted.

The operator `reset_obj_db` can be used to reset the system and clear all remaining iconic objects.

Attention

Regarding the use of local variables: Because only local variables are deleted on exit of a subroutine (or a rule in Prolog), while the HALCON database is not updated, it is necessary to clear local objects before exiting the subroutine. Special care has to be taken if backtracking is possible in Prolog (before reaching the `clear_obj` statements).

Parameter

▷ **objects** (input_object) object(-array) $\leadsto$ *Hobject*
Objects to be deleted.

Result

`clear_obj` returns 2 (H_MSG_TRUE) if all objects are contained in the HALCON database. If not all objects are valid (e.g., already cleared), an exception is raised, which also clears all valid objects. The operator `set_check (:: '~clear':)` can be used to suppress the raising of this exception. If the input is empty the behavior can be set via `set_system (:: 'no_object_result', <Result>:)`. If necessary, an exception is raised.

Parallelization Information

`clear_obj` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`test_obj_def`

Alternatives

`reset_obj_db`

See also

`test_obj_def`, `set_check`

Module

Basic operators

concat_obj (Objects1, Objects2 : ObjectsConcat : :)

Concatenate two iconic object tuples.

concat_obj concatenates the two tuples of iconic objects **Objects1** and **Objects2** into a new object tuple **ObjectsConcat**. Hence, this tuple contains all the objects of the two input tuples:

ObjectsConcat = [**Objects1**, **Objects2**]

In **ObjectsConcat** the objects of **Objects1** are stored first, followed by the objects of **Objects2**. The order of the objects is preserved. As usual, only the objects are copied, and not the corresponding images and regions, i.e., no new memory is allocated. **concat_obj** is designed especially for HALCON/C. In languages like Smalltalk, Prolog, and C++ it is not needed.

concat_obj should not be confused with **union1** or **union2**, in which regions are merged, i.e., in which the number of objects is modified.

concat_obj can be used to concatenate objects of different image object types (e.g., images and XLD contours) into a single object. This is only recommended if it is necessary to accumulate in a single object variable, for example, the results of an image processing sequence. It should be noted that the only operators that can handle such object tuples of mixed type are **concat_obj**, **copy_obj**, **select_obj**, and **disp_obj**. For technical reasons, object tuples of mixed type must not be created in HDevelop.

Parameter

- ▷ **Objects1** (input_object) object(-array) $\leadsto$ *Hobject*
Object tuple 1.
- ▷ **Objects2** (input_object) object(-array) $\leadsto$ *Hobject*
Object tuple 2.
- ▷ **ObjectsConcat** (output_object) object-array $\leadsto$ *Hobject*
Concatenated objects.

Example

```
/* generate a tuple of a circle and a rectangle */
```

```
gen_circle(&Circle, 200.0, 400.0, 23.0);
gen_rectangle1(&Rectangle, 23.0, 44.0, 203.0, 201.0);
concat_obj(Circle, Rectangle, &CircleAndRectangle);
clear_obj(Circle); clear_obj(Rectangle);
disp_region(CircleAndRectangle, WindowHandle);
```

Complexity

Runtime complexity: $O(|\mathbf{Objects1}| + |\mathbf{Objects2}|)$;

Memory complexity of the result objects: $O(|\mathbf{Objects1}| + |\mathbf{Objects2}|)$

Result

concat_obj returns 2 (H_MSG_TRUE) if all objects are contained in the HALCON database. If the input is empty the behavior can be set via **set_system(:'no_object_result', <Result>:)**. If necessary, an exception is raised.

Parallelization Information

concat_obj is *reentrant* and processed *without* parallelization.

See also

count_obj, **copy_obj**, **select_obj**, **disp_obj**

Module

Basic operators

copy_obj (Objects : ObjectsSelected : Index, NumObj :)

Copy an iconic object in the HALCON database.

copy_obj copies **NumObj** iconic objects beginning with index **Index** (starting with 1) from the iconic input object tuple **Objects** to the output object **ObjectsSelected**. If -1 is passed for **NumObj** all objects beginning with **Index** are copied. No new storage is allocated for the regions and images. Instead, new objects containing references to the existing objects are created. The number of objects in an object tuple can be queried with the operator **count_obj**.

Parameter

- ▷ **Objects** (input_object) object(-array) $\leadsto$ *Hobject*
Objects to be copied.
- ▷ **ObjectsSelected** (output_object) object(-array) $\leadsto$ *Hobject*
Copied objects.
- ▷ **Index** (input_control) integer $\leadsto$ *integer*
Starting index of the objects to be copied.
Default Value : 1
Suggested values : $\text{Index} \in \{1, 2, 3, 4, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000\}$
Typical range of values : $1 \leq \text{Index}$
Restriction : $\text{Index} \leq \text{number}(\text{Objects})$
- ▷ **NumObj** (input_control) integer $\leadsto$ *integer*
Number of objects to be copied or -1.
Default Value : 1
Suggested values : $\text{NumObj} \in \{-1, 1, 2, 3, 4, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000\}$
Typical range of values : $-1 \leq \text{NumObj}$
Restriction : $((\text{NumObj} + \text{Index}) \leq \text{number}(\text{Objects})) \wedge (\text{NumObj} \neq 0)$

Example

```
/* Access all regions */
```

```
count_obj(Regions, Num)
for(1, Num, i)
    copy_obj(Regions, Single, i, 1)
    get_region_polygon(Single, 5.0, Line, Column)
    disp_polygon(WindowHandle, Line, Column)
    clear_obj(Single)
loop().
```

Complexity

Runtime complexity: $O(|\text{Objects}| + \text{NumObj})$;

Memory complexity of the result object: $O(\text{NumObj})$

Result

copy_obj returns 2 (H_MSG_TRUE) if all objects are contained in the HALCON database and all parameters are correct. If the input is empty the behavior can be set via **set_system** (**::'no_object_result', <Result>:**). If necessary, an exception is raised.

Parallelization Information

copy_obj is *reentrant* and processed *without* parallelization.

Possible Predecessors

count_obj

Alternatives

select_obj

See also

count_obj, **concat_obj**, **obj_to_integer**, **copy_image**

Module

Basic operators

gen_empty_obj (: EmptyObject : :)

Create an empty object tuple.

The operator `gen_empty_obj` creates an empty tuple. This means that the output parameter does not contain any objects. Thus, the operator `count_obj` returns 0. However, `clear_obj` can be called for the output. It should be noted that no objects must not be confused with an empty region. In case of an empty region, i.e. a region with 0 pixels `count_obj` returns the value 1.

Parameter
▷ EmptyObject (output_object) object $\leadsto$ Hobject No objects.
Parallelization Information
<code>gen_empty_obj</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
Module
Image / region / XLD management

integer_to_obj (: Objects : SurrogateTuple :)

Convert an “integer number” into an iconic object.

`integer_to_obj` is the inverse operator to `obj_to_integer`. All surrogates of objects passed in `SurrogateTuple` are stored as objects. In contrast to `obj_to_integer`, the objects are duplicated. `integer_to_obj` is intended especially for use in HALCON/C, because iconic objects and control parameters are treated differently in C.

Attention
The objects are duplicated in the database.
Parameter
▷ Objects (output_object) object(-array) $\leadsto$ Hobject Created objects.
▷ SurrogateTuple (input_control) integer(-array) $\leadsto$ integer Tuple of object surrogates.

Result
<code>integer_to_obj</code> returns 2 (H_MSG_TRUE) if all parameters are correct, i.e., if they are valid object keys. If the input is empty the behavior can be set via <code>set_system(: : 'no_object_result', <Result> :)</code> . If necessary, an exception is raised.

Parallelization Information
<code>integer_to_obj</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
See also
<code>obj_to_integer</code>
Module
Basic operators

obj_to_integer (Objects : : Index, Number : SurrogateTuple)

Convert an iconic object into an “integer number.”

`obj_to_integer` stores `Number`, starting at index `Index`, of the database keys of the input object `Objects` as integer numbers in the output parameter `SurrogateTuple`. If -1 is passed for `Number` all objects beginning with `Index` are copied. This facilitates a direct access to an arbitrary element of `Objects`. In conjunction with `count_obj` (returns the number of objects in `Objects`) the elements of `Objects` can be processed successively. The objects are not duplicated by `obj_to_integer` and thus must not be cleared by `clear_obj`.

Attention
The objects’ data is not duplicated.

Parameter	
▷ Objects (input_object)	object(-array) $\leadsto$ <i>Hobject</i> Objects for which the surrogates are to be returned.
▷ Index (input_control)	integer $\leadsto$ <i>integer</i> Starting index of the surrogates to be returned. Default Value : 1 Typical range of values : $1 \leq \text{Index}$
▷ Number (input_control)	integer $\leadsto$ <i>integer</i> Number of surrogates to be returned. Default Value : -1 Restriction : $(\text{Number} = -1) \vee ((\text{Number} + \text{Index}) \leq \text{number}(\text{Objects}))$
▷ SurrogateTuple (output_control)	integer(-array) $\leadsto$ <i>integer</i> Tuple containing the surrogates.

Example

```
/* Access the i-th element: */
obj_to_integer(Objects,i,1,Surrogat).
```

Complexity

Runtime complexity: $O(|\text{Objects}| + \text{Number})$

Result

obj_to_integer returns 2 (H.MSG_TRUE) if all parameters are correct. If the input is empty the behavior can be set via **set_system**(::'no_object_result',<Result>:). If necessary, an exception is raised.

Parallelization Information

obj_to_integer is *reentrant* and processed *without* parallelization.

Possible Predecessors

test_obj_def

Alternatives

copy_obj, **select_obj**, **copy_image**, **gen_image_proto**

See also

integer_to_obj, **count_obj**

Module

Basic operators

select_obj (Objects : ObjectSelected : Index :)

Select an object from an object tuple.

select_obj copies the iconic object with index **Index** (starting with 1) from the iconic input object tuple **Objects** to the output object **ObjectSelected**. No new storage is allocated for the regions and images. Instead, new objects containing references to the existing objects are created. The number of objects in an object tuple can be queried with the operator **count_obj**.

Parameter	
▷ Objects (input_object)	object(-array) $\leadsto$ <i>Hobject</i> Objects, of which one is to be selected.
▷ ObjectSelected (output_object)	object $\leadsto$ <i>Hobject</i> Selected object.
▷ Index (input_control)	integer $\leadsto$ <i>integer</i> Index of the object to be selected. Default Value : 1 Suggested values : $\text{Index} \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 50, 100, 200, 500, 1000, 2000, 5000\}$ Typical range of values : $1 \leq \text{Index}$

Example

```

/* Access to all Regions */

count_obj(Regions,&Num);
for (i=1; i<=Num; i++)
{
    select_obj(Regions,&Single,i);
    T_get_region_polygon(Single,5.0,&Row,&Column);
    T_disp_polygon(WindowHandleTuple,Row,Column);
    destroy_tuple(Row);
    destroy_tuple(Column);
    clear_obj(Single);
}

```

Complexity

Runtime complexity: $O(|\text{Objects}|)$

Result

[select_obj](#) returns 2 (H_MSG_TRUE) if all objects are contained in the HALCON database and all parameters are correct. If the input is empty the behavior can be set via [set_system \(::'no-object-result', <Result>:\)](#). If necessary, an exception is raised.

Parallelization Information

[select_obj](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[count_obj](#)

Alternatives

[copy_obj](#)

See also

[count_obj](#), [concat_obj](#), [obj_to_integer](#)

Module

Basic operators

Chapter 12

Regions

12.1 Access

get_region_chain (Region : : : Row, Column, Chain)

Contour of an object as chain code.

The operator `get_region_chain` returns the contour of a region. A contour is a series of pixels describing the outline of the region. The contour “lies on” the region. It starts at the smallest line number; in that line at the pixel with the largest column index. The rotation occurs clockwise. Holes of the region are ignored. The direction code (chain code) is defined as follows:

3	2	1
4	*	0
5	6	7

The operator `get_region_chain` returns the code in the form of a tuple. In case of an empty region the parameters `Row` and `Column` are zero and `Chain` is the empty tuple.

Attention

Holes of the region are ignored. Only one region may be passed, and it must have exactly one connection component.

Parameter

- ▷ **Region** (input_object) region $\rightsquigarrow$ *Hobject*
Region to be transformed.
 - ▷ **Row** (output_control) chain.begin.y $\rightsquigarrow$ *integer*
Line of starting point.
 - ▷ **Column** (output_control) chain.begin.x $\rightsquigarrow$ *integer*
Column of starting point.
 - ▷ **Chain** (output_control) chain.code-array $\rightsquigarrow$ *integer*
Direction code of the contour (from starting point).
- Typical range of values :** $0 \leq \text{Chain} \leq 7$

Result

The operator `get_region_chain` normally returns the value 2 (H_MSG.TRUE). If more than one connection component is passed an exception handling is caused. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result', <Result>)`. The behavior in case of empty region (the region is the empty set) is set via the operator `set_system('empty_region_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`get_region_chain` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`sobel_amp`, `threshold`, `skeleton`, `edges_image`, `gen_rectangle1`, `gen_circle`

<i>Possible Successors</i>
approx_chain , approx_chain_simple
<i>See also</i>
copy_obj , get_region_contour , get_region_polygon
<i>Module</i>
Region processing

get_region_contour (Region : : : Rows, Columns)

Access the contour of an object.

The operator [get_region_contour](#) returns the contour of a region. A contour is a result of line ([Rows](#)) and column coordinates ([Columns](#)), describing the boundary of the region. The contour lies on the region. It starts at the smallest line number. In that line at the pixel with the largest column index. The rotation direction is clockwise. The first pixel of the contour is identical with the last. Holes of the region are ignored. The operator [get_region_contour](#) returns the coordinates in the form of tuples. An empty region is passed as empty tuple.

Attention
Holes of the region are ignored. Only one region may be passed, and this region must have exactly one connection component.

<i>Parameter</i>
▷ Region (input_object) region $\leadsto$ <i>Hobject</i> Output region.
▷ Rows (output_control) contour.y-array $\leadsto$ <i>integer</i> Line numbers of the contour pixels.
▷ Columns (output_control) contour.x-array $\leadsto$ <i>integer</i> Column numbers of the contour pixels.
Number of elements : Columns = Rows

Result
The operator [get_region_contour](#) normally returns the value 2 (H_MSG_TRUE). If more than one connection component is passed an exception handling is caused. The behavior in case of empty input (no input regions available) is set via the operator [set_system\('no.object.result', <Result>\)](#).

Parallelization Information
[get_region_contour](#) is *reentrant* and processed *without parallelization*.

<i>Possible Predecessors</i>
sobel_amp , threshold , skeleton , edges_image , gen_rectangle1 , gen_circle
<i>See also</i>
copy_obj , get_region_chain , get_region_polygon
<i>Module</i>
Region processing

get_region_convex (Region : : : Rows, Columns)

Access convex hull as contour.

The operator [get_region_convex](#) returns the convex hull of a region as polygon. The polygon is the minimum result of line ([Rows](#)) and column coordinates ([Columns](#)) describing the hull of the region. The polygon pixels lie on the region. The polygon starts at the smallest line number; in this line at the pixel with the largest column index. The rotation direction is clockwise. The first pixel of the polygon is identical with the last. The operator [get_region_convex](#) returns the coordinates in the form of tuples. An empty region is passed as empty tuple.

Parameter
▷ Region (input_object) region $\leadsto$ <i>Hobject</i> Output region. ▷ Rows (output_control) contour.y-array $\leadsto$ <i>integer</i> Line numbers of contour pixels. ▷ Columns (output_control) contour.x-array $\leadsto$ <i>integer</i> Column numbers of the contour pixels. Number of elements : Columns = Rows
Result
The operator <code>get_region_convex</code> returns the value 2 (H_MSG_TRUE).
Parallelization Information
<code>get_region_convex</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
Possible Predecessors
<code>threshold</code> , <code>skeleton</code> , <code>dyn_threshold</code>
Possible Successors
<code>disp_polygon</code>
Alternatives
<code>shape_trans</code>
See also
<code>select_obj</code> , <code>get_region_contour</code>
Module
Region processing

get_region_points (Region : : : Rows, Columns)

Access the pixels of a region.

The operator `get_region_points` returns the region data in the form of coordinate lists. The coordinates are sorted in the following order:

$$(r_1, c_1) \leq (r_2, c_2) := (r_1 < r_2) \vee (r_1 = r_2) \wedge (c_1 \leq c_2)$$

`get_region_points` returns the coordinates in the form of tuples. An empty region is passed as empty tuple.

Attention	Only one region may be passed.

Parameter
▷ Region (input_object) region $\leadsto$ <i>Hobject</i> This region is accessed. ▷ Rows (output_control) coordinates.y-array $\leadsto$ <i>integer</i> Line numbers of the pixels in the region ▷ Columns (output_control) coordinates.x-array $\leadsto$ <i>integer</i> Column numbers of the pixels in the region. Number of elements : Columns = Rows

Result	The operator <code>get_region_points</code> normally returns the value 2 (H_MSG_TRUE). If more than one connection component is passed an exception handling is caused. The behavior in case of empty input (no input regions available) is set via the operator <code>set_system('no_object_result', <Result>)</code> .

Parallelization Information	<code>get_region_points</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
Possible Predecessors	<code>sobel_amp</code> , <code>threshold</code> , <code>connection</code>

get_region_runs	Alternatives
copy_obj , gen_region_points	See also
	Module
Region processing	

get_region_polygon (Region : : Tolerance : Rows, Columns)

Polygon approximation of a region.

The operator [get_region_polygon](#) calculates a polygon to approximate the edge of a region. A polygon is a sequence of line ([Rows](#)) and column coordinates ([Columns](#)). It describes the contour of the region. Only the base points of the polygon are returned. The parameter [Tolerance](#) indicates how large the maximum distance between the polygon and the edge of the region may be. Holes of the region are ignored. The operator [get_region_polygon](#) returns the coordinates in the form of tuples.

Attention
Holes of the region are ignored. Only one region may be passed, and this region must have exactly one connection component.

<i>Parameter</i>	
▷ Region (input_object) region $\leadsto$ <i>Hobject</i> Region to be approximated.	
▷ Tolerance (input_control) number $\leadsto$ <i>real</i> / integer Maximum distance between the polygon and the edge of the region. Default Value : 5.0 Suggested values : Tolerance $\in$ {0.0, 2.0, 5.0, 10.0} Typical range of values : $0.0 \leq \text{Tolerance} \leq 10.0$ (lin) Minimum Increment : 0.01 Recommended Increment : 1.0	
▷ Rows (output_control) polygon.y-array $\leadsto$ <i>integer</i> Line numbers of the base points of the contour.	
▷ Columns (output_control) polygon.x-array $\leadsto$ <i>integer</i> Column numbers of the base points of the contour. Number of elements : Columns = Rows	

Result
The operator [get_region_polygon](#) normally returns the value 2 (H_MSG_TRUE). If more than one connection component is passed an exception handling is caused. The behavior in case of empty input (no input regions available) is set via the operator [set_system\('no_object_result', <Result>\)](#).

Parallelization Information
[get_region_polygon](#) is *reentrant* and processed *without parallelization*.

Possible Predecessors
[sobel_amp](#), [threshold](#), [skeleton](#), [edges_image](#)

See also
[copy_obj](#), [gen_region_polygon](#), [disp_polygon](#), [get_region_chain](#), [get_region_contour](#), [set_line_approx](#)

	Module
Region processing	

get_region_runs (Region : : : Row, ColumnBegin, ColumnEnd)

Access the runlength coding of a region.

The operator `get_region_runs` returns the region data in the form of chord tuples. The chord representation is caused by examining a region line by line with ascending line number (= from “top” to “bottom”). Every line is passed from left to right (ascending column number); storing all starting and ending points of region segments (= chords). Thus a region can be described by a sequence of chords, a chord being defined by line number, starting and ending points (column number). The operator `get_region_runs` returns the three components of the chords in the form of tuples. In case of an empty region three empty tuples are returned.

Attention

Only one region may be passed.

Parameter

- ▷ **Region** (input_object) region $\leadsto$ *Hobject*
Output region.
- ▷ **Row** (output_control) chord.y-array $\leadsto$ *integer*
Line numbers of the chords.
- ▷ **ColumnBegin** (output_control) chord.x1-array $\leadsto$ *integer*
Column numbers of the starting points of the chords.
Number of elements : ColumnBegin = Row
- ▷ **ColumnEnd** (output_control) chord.x2-array $\leadsto$ *integer*
Column numbers of the ending points of the chords.
Number of elements : ColumnEnd = Row

Result

The operator `get_region_runs` normally returns the value 2 (H_MSG_TRUE). If more than one region is passed an exception handling is caused. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result', <Result>)`.

Parallelization Information

`get_region_runs` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`threshold`, `connection`

Alternatives

`get_region_points`

See also

`copy_obj`, `gen_region_runs`

Module

Region processing

12.2 Affine-Transformations

affine_trans_region (Region : RegionAffineTrans : HomMat2D,
Interpolate :)

Apply an arbitrary affine 2D transformation to regions.

`affine_trans_region` applies an arbitrary affine 2D transformation, i.e., scaling, rotation, translation, and slant (skewing), to the regions given in `Region` and returns the transformed regions in `RegionAffineTrans`. The affine transformation is described by the homogeneous transformation matrix given in `HomMat2D`, which can be created using the operators `hom_mat2d_identity`, `hom_mat2d_scale`, `hom_mat2d_rotate`, `hom_mat2d_translate`, etc., or be the result of operators like `vector_angle_to_rigid`.

The components of the homogeneous transformation matrix are interpreted as follows: The *row* coordinate of the image corresponds to the *x* coordinate of the matrix, while the *column* coordinate of the image corresponds to the *y* coordinate of the matrix. This is necessary to obtain a right-handed coordinate system for the image. In particular, this assures that rotations are performed in the correct direction. Note that the (*x,y*) order of the matrices quite naturally corresponds to the usual (row,column) order for coordinates in the image.

The parameter `Interpolate` determines whether the transformation is to be done by using interpolation internally. This can lead to smoother region boundaries, especially if regions are enlarged. However, the runtime increases drastically.

Attention

[affine_trans_region](#) in general is not reversible (clipping and discretization during rotation and scaling).

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Region(s) to be rotated and scaled.
- ▷ **RegionAffineTrans** (output_object) region(-array) $\leadsto$ *Hobject*
Transformed output region(s).
Number of elements : RegionAffineTrans = Region
- ▷ **HomMat2D** (input_control) affine2d-array $\leadsto$ *real*
Input transformation matrix.
Number of elements : 6
- ▷ **Interpolate** (input_control) string $\leadsto$ *string*
Should the transformation be done using interpolation?
Default Value : 'false'
List of values : Interpolate $\in$ {'true', 'false'}

Result

[affine_trans_region](#) returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via [set_system\('no_object_result', <Result>\)](#), the behavior in case of an empty input region via [set_system\('empty_region_result', <Result>\)](#), and the behavior in case of an empty result region via [set_system\('store_empty_region', <true/false>\)](#). If necessary, an exception handling is raised.

Parallelization Information

[affine_trans_region](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[hom_mat2d_identity](#), [hom_mat2d_scale](#), [hom_mat2d_translate](#), [hom_mat2d_invert](#),
[hom_mat2d_rotate](#)

Possible Successors

[select_shape](#)

Alternatives

[move_region](#), [mirror_region](#), [zoom_region](#)

See also

[affine_trans_image](#)

Module

Region processing

mirror_region (Region : RegionMirror : RowColumn, WidthHeight :)

Reflect a region about the x- or y-axis.

[mirror_region](#) reflects a region about the x- or y-axis (parameter [RowColumn](#)). The parameter [WidthHeight](#) specifies the corresponding image width or height respectively, i.e., it is two times the coordinate of the axis of symmetry.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Region(s) to be reflected.
- ▷ **RegionMirror** (output_object) region(-array) $\leadsto$ *Hobject*
Reflected region(s).
Number of elements : RegionMirror = Region
- ▷ **RowColumn** (input_control) string $\leadsto$ *string*
Coordinate of the axis of symmetry.
Default Value : 'row'
List of values : RowColumn $\in$ {'column', 'row'}

- ▷ **WidthHeight** (input_control) integer $\leadsto$ integer
 Height or width of the corresponding image.
Default Value : 512
Suggested values : WidthHeight $\in \{128, 256, 512, 525, 768, 1024\}$
Typical range of values : $1 \leq \text{WidthHeight} \leq 1024$ (lin)
Minimum Increment : 1
Recommended Increment : 1
Restriction : WidthHeight > 0

Example

```
read_image(&Image, "affe");
threshold(Image, &Seg, 128.0, 255.0);
mirror_region(Seg, &Mirror, "row", 512);
disp_region(Mirror, WindowHandle);
```

Parallelization Information

[mirror_region](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[threshold](#), [connection](#), [regiongrowing](#), [pouring](#)

Possible Successors

[select_shape](#), [disp_region](#)

Alternatives

[affine_trans_region](#)

See also

[zoom_region](#)

Module

Region processing

move_region (Region : RegionMoved : Row, Column :)

Translate a region.

[move_region](#) translates the input regions by the vector given by ([Row](#), [Column](#)). If necessary, the resulting regions are clipped with the current image format.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ Hobject
 Region(s) to be moved.
- ▷ **RegionMoved** (output_object) region(-array) $\leadsto$ Hobject
 Translated region(s).
Number of elements : RegionMoved = Region
- ▷ **Row** (input_control) point.y $\leadsto$ integer
 Row coordinate of the vector by which the region is to be moved.
Default Value : 30
Suggested values : Row $\in \{-128, -64, -32, -16, -10, -8, -4, -2, -1, 0, 1, 2, 4, 5, 8, 10, 16, 32, 64, 128\}$
Typical range of values : $-512 \leq \text{Row} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Column** (input_control) point.x $\leadsto$ integer
 Row coordinate of the vector by which the region is to be moved.
Default Value : 30
Suggested values : Column $\in \{-128, -64, -32, -16, -10, -8, -4, -2, -1, 0, 1, 2, 4, 5, 8, 10, 16, 32, 64, 128\}$
Typical range of values : $-512 \leq \text{Column} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10

<i>Complexity</i>
Let F be the area of the input region. Then the runtime complexity is $O(F)$.
<i>Result</i>
<code>move_region</code> always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via <code>set_system('no_object_result', <Result>)</code> , the behavior in case of an empty input region via <code>set_system('empty_region_result', <Result>)</code> , and the behavior in case of an empty result region via <code>set_system('store_empty_region', <true/false>)</code> . If necessary, an exception handling is raised.
<i>Parallelization Information</i>
<code>move_region</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).
<i>Possible Predecessors</i>
<code>threshold</code> , <code>connection</code> , <code>regiongrowing</code> , <code>pouring</code>
<i>Possible Successors</i>
<code>select_shape</code> , <code>disp_region</code>
<i>See also</i>
<code>affine_trans_image</code> , <code>mirror_region</code> , <code>zoom_region</code>
<i>Module</i>
Region processing

transpose_region (Region : Transposed : Row, Column :)

Reflect a region about a point.

`transpose_region` reflects a region about a point. The fixed point is given by `Column` and `Row`. The image P' of a point P is determined by the following requirement:

If $P = S$, then $P' = S$, i.e., the point S is the fixed point of the mapping. If $P \neq S$, S is the center point of a line segment connecting P and P' . Therefore, the following equations result:

$$\begin{aligned}\text{Column} &= \frac{x + x'}{2} \\ \text{Row} &= \frac{y + y'}{2}.\end{aligned}$$

If `Row` and `Column` are set to the origin, the in morphology often used transposition results. Hence `transpose_region` is often used to reflect (transpose) a structuring element.

<i>Parameter</i>
▷ Region (input_object) region(-array) $\leadsto$ <i>Hobject</i> Region to be reflected. ▷ Transposed (output_object) region(-array) $\leadsto$ <i>Hobject</i> Transposed region. ▷ Row (input_control) point.y $\leadsto$ <i>integer</i> Row coordinate of the reference point. Default Value : 0 Suggested values : Row $\in \{0, 64, 128, 256, 512\}$ Typical range of values : $0 \leq \text{Row} \leq 511$ (lin) Minimum Increment : 1 Recommended Increment : 1 ▷ Column (input_control) point.x $\leadsto$ <i>integer</i> Column coordinate of the reference point. Default Value : 0 Suggested values : Column $\in \{0, 64, 128, 256, 512\}$ Typical range of values : $0 \leq \text{Column} \leq 511$ (lin) Minimum Increment : 1 Recommended Increment : 1

Complexity

Let F be the area of the input region. Then the runtime complexity for one region is

$$O(\sqrt{F}) .$$

Result

`transpose_region` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty or no input region can be set via:

- no region: `set_system('no_object_result', <RegionResult>)`
- empty region: `set_system('empty_region_result', <RegionResult>)`

Otherwise, an exception is raised.

Parallelization Information

`transpose_region` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

`reduce_domain`, `select_shape`, `area_center`, `connection`

See also

`dilation1`, `opening`, `closing`

Module

Morphology

zoom_region (Region : RegionZoom : ScaleWidth, ScaleHeight :)

Zoom a region.

`zoom_region` enlarges or reduces the regions given in `Region` in the x- and y-direction by the given scale factors `ScaleWidth` and `ScaleHeight`.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Region(s) to be zoomed.
- ▷ **RegionZoom** (output_object) region(-array) $\leadsto$ *Hobject*
Zoomed region(s).
Number of elements : RegionZoom = Region
- ▷ **ScaleWidth** (input_control) extent.x $\leadsto$ *real*
Scale factor in x-direction.
Default Value : 2.0
Suggested values : ScaleWidth $\in \{0.25, 0.5, 1.0, 2.0, 3.0\}$
Typical range of values : $0.0 \leq \text{ScaleWidth} \leq 100.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.5
- ▷ **ScaleHeight** (input_control) extent.y $\leadsto$ *real*
Scale factor in y-direction.
Default Value : 2.0
Suggested values : ScaleHeight $\in \{0.25, 0.5, 1.0, 2.0, 3.0\}$
Typical range of values : $0.0 \leq \text{ScaleHeight} \leq 100.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.5

Parallelization Information

`zoom_region` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `connection`, `regiongrowing`, `pouring`

Possible Successors

`select_shape`, `disp_region`

	See also	
zoom_image_size , zoom_image_factor		
	Module	
Region processing		

12.3 Creation

gen_checker_region (: RegionChecker : WidthRegion, HeightRegion, WidthPattern, HeightPattern :)

Create a checkered region.

The operator [gen_checker_region](#) returns a checkered region. Every black field of the checkerboard belongs to the region. The horizontal and vertical expansion of the region is limited by [WidthRegion](#), [HeightRegion](#) respectively, the size of the fields of the checkerboard by [WidthPattern](#) × [HeightPattern](#).

	Attention	
If a very small pattern is chosen (WidthPattern < 4) the created region requires much storage.		
	Parameter	

- ▷ **RegionChecker** (output_object) region $\rightsquigarrow$ *Hobject*
Created checkerboard region.
- ▷ **WidthRegion** (input_control) extent.x $\rightsquigarrow$ *integer*
Largest occurring *x* value of the region.
Default Value : 511
Suggested values : WidthRegion ∈ {10, 20, 31, 63, 127, 255, 300, 400, 511}
Typical range of values : $1 \leq \text{WidthRegion} \leq 1024$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : WidthRegion ≥ 1
- ▷ **HeightRegion** (input_control) extent.y $\rightsquigarrow$ *integer*
Largest occurring *y* value of the region.
Default Value : 511
Suggested values : HeightRegion ∈ {10, 20, 31, 63, 127, 255, 300, 400, 511}
Typical range of values : $1 \leq \text{HeightRegion} \leq 1024$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : HeightRegion ≥ 1
- ▷ **WidthPattern** (input_control) extent.y $\rightsquigarrow$ *integer*
Width of a field of the checkerboard.
Default Value : 64
Suggested values : WidthPattern ∈ {1, 2, 4, 8, 16, 20, 32, 64, 100, 128, 200, 300, 500}
Typical range of values : $1 \leq \text{WidthPattern} \leq 1024$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : (WidthPattern > 0) ∧ (WidthPattern < WidthRegion)
- ▷ **HeightPattern** (input_control) extent.y $\rightsquigarrow$ *integer*
Height of a field of the checkerboard.
Default Value : 64
Suggested values : HeightPattern ∈ {1, 2, 4, 8, 16, 20, 32, 64, 100, 128, 200, 300, 500}
Typical range of values : $1 \leq \text{HeightPattern} \leq 1024$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : (HeightPattern > 0) ∧ (HeightPattern < HeightRegion)

	Example	

```
gen_checker_region(Checker, 512, 512, 32, 64:)
set_draw(WindowHandle, 'fill')
```

```
set_part(WindowHandle,0,0,511,511)
disp_region(Checker,WindowHandle)
```

Complexity

The required storage (in bytes) for the region is:

$O((\text{WidthRegion} * \text{HeightRegion}) / \text{WidthPattern})$

Result

The operator `gen_checker_region` returns the value 2 (H_MSG_TRUE) if the parameter values are correct. Otherwise an exception handling is raised. The clipping according to the current image format is set via the operator `set_system('clip_region', <'true'/'false'>)`.

Parallelization Information

`gen_checker_region` is *reentrant* and processed *without* parallelization.

Possible Successors

`paint_region`

Alternatives

`gen_grid_region`, `gen_region_polygon_filled`, `gen_region_points`, `gen_region_runs`, `gen_rectangle1`, `concat_obj`, `gen_random_region`, `gen_random_regions`

See also

`hamming_change_region`, `reduce_domain`

Module

Region processing

gen_circle (: Circle : Row, Column, Radius :)

Create a circle.

The operator `gen_circle` generates one or more circles described by the center and `Radius`. If several circles shall be generated the coordinates must be passed in the form of tuples.

The coordinate system runs from (0,0) (upper left corner) to (Width-1,Height-1). See `get_system` and `reset_obj_db` in this context.

If an integer value (1,2,3..) is given for the radius the result is an even-numbered diameter and thus an asymmetrical circle. In case of an odd-numbered diameter (radius = 1.5,2.5,3.5...) a symmetrical circle is obtained.

If the circle extends beyond the image edge it is clipped to the current image format according to the value of the system flag 'clip_region' (`set_system`).

Parameter

- ▷ **Circle** (output_object) region(-array) $\leadsto$ *Hobject*
Generated circle.
- ▷ **Row** (input_control) circle.center.y(-array) $\leadsto$ *real* / integer
Line index of center.
Default Value : 200.0
Suggested values : Row $\in$ {0.0, 10.0, 50.0, 100.0, 200.0, 300.0}
Typical range of values : $1.0 \leq \text{Row} \leq 1024.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
- ▷ **Column** (input_control) circle.center.x(-array) $\leadsto$ *real* / integer
Column index of center.
Default Value : 200.0
Suggested values : Co lumn $\in$ {0.0, 10.0, 50.0, 100.0, 200.0, 300.0}
Typical range of values : $1.0 \leq \text{Co lumn} \leq 1024.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0

- ▷ **Radius** (input_control) circle.radius(-array) $\leadsto$ *real* / integer
 Radius of circle.
Default Value : 100.5
Suggested values : Radius $\in \{1.0, 1.5, 2.0, 2.5, 3, 3.5, 4, 4.5, 5.5, 6.5, 7.5, 9.5, 11.5, 15.5, 20.5, 25.5, 31.5, 50.5\}$
Typical range of values : $1.0 \leq \text{Radius} \leq 1024.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
Restriction : Radius > 0.0

Example

```
open_window(0,0,-1,-1,'root','visible','',WindowHandle)
read_image(Image,'meer')
gen_circle(Circle,300.0,200.0,150.5)
reduce_domain(Image,Circle,Mask)
disp_color(Mask,WindowHandle).
```

Complexity

Runtime complexity: $O(\text{Radius} * 2)$

Storage complexity (byte): $O(\text{Radius} * 8)$

Result

If the parameter values are correct, the operator `gen_circle` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised. The clipping according to the current image format is set via the operator `set_system('clip_region', <'true'/'false'>)`. If an empty region is created by clipping (the circle is completely outside of the image format) the operator `set_system('store_empty_region', <true/false>)` determines whether the empty region is put out.

Parallelization Information

`gen_circle` is *reentrant* and processed *without parallelization*.

Possible Successors

`paint_region`, `reduce_domain`

Alternatives

`gen_ellipse`, `gen_region.polygon-filled`, `gen_region.points`, `gen_region.runs`, `draw_circle`

See also

`disp_circle`, `set_shape`, `smallest_circle`, `reduce_domain`

Module

Region processing

gen_ellipse (: Ellipse : Row, Column, Phi, Radius1, Radius2 :)

Create an ellipse.

The operator `gen_ellipse` generates one or more ellipses with the center (**Row**, **Column**), the orientation **Phi** and the half-radii **Radius1** and **Radius2**. The angle is indicated in arc measure according to the x axis in mathematically positive direction. More than one region can be created by passing tuples of parameter values.

The center must be located within the image coordinates. The coordinate system runs from (0,0) (upper left corner) to (Width-1,Height-1). See `get_system` and `reset_obj_db` in this context. If the ellipse reaches beyond the edge of the image it is clipped to the current image format according to the value of the system flag 'clip_region' (`set_system`).

Parameter

- ▷ **Ellipse** (output_object) region(-array) $\leadsto$ *Hobject*
 Created ellipse(s).

- ▷ **Row** (input_control) ellipse.center.y(-array) $\leadsto$ *real* / integer
Line index of center.
Default Value : 200.0
Suggested values : Row $\in$ {0.0, 10.0, 20.0, 50.0, 100.0, 256.0, 300.0, 400.0}
Typical range of values : $1.0 \leq \text{Row} \leq 1024.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
- ▷ **Column** (input_control) ellipse.center.x(-array) $\leadsto$ *real* / integer
Column index of center.
Default Value : 200.0
Suggested values : Column $\in$ {0.0, 10.0, 20.0, 50.0, 100.0, 256.0, 300.0, 400.0}
Typical range of values : $1.0 \leq \text{Column} \leq 1024.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
- ▷ **Phi** (input_control) ellipse.angle.rad(-array) $\leadsto$ *real* / integer
Orientation of the longer radius (Radius1).
Default Value : 0.0
Suggested values : Phi $\in$ {-1.178097, -0.785398, -0.392699, 0.0, 0.392699, 0.785398, 1.178097}
Typical range of values : $-1.178097 \leq \text{Phi} \leq 1.178097$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
- ▷ **Radius1** (input_control) ellipse.radius1(-array) $\leadsto$ *real* / integer
Longer radius.
Default Value : 100.0
Suggested values : Radius1 $\in$ {2.0, 5.0, 10.0, 20.0, 50.0, 100.0, 256.0, 300.0, 400.0}
Typical range of values : $1.0 \leq \text{Radius1} \leq 1024.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
Restriction : Radius1 > 0
- ▷ **Radius2** (input_control) ellipse.radius2(-array) $\leadsto$ *real* / integer
Shorter radius.
Default Value : 60.0
Suggested values : Radius2 $\in$ {1.0, 2.0, 4.0, 5.0, 10.0, 20.0, 50.0, 100.0, 256.0, 300.0, 400.0}
Typical range of values : $1.0 \leq \text{Radius2} \leq 1024.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
Restriction : (Radius2 > 0) $\wedge$ (Radius2 $\leq$ Radius1)

Example

```

open_window(0,0,-1,-1,'root','visible','',WindowHandle)
set_insert(WindowHandle,'xor')
repeat()
    get_mbutton(WindowHandle,Row,Column,Button)
    gen_ellipse(Ellipse,Row,Column,Column / 300.0,
        (Row mod 100)+1,(Column mod 50) + 1)
    disp_region(Ellipse,WindowHandle)
    clear_obj(Ellipse)
until(Button = 1).

```

Complexity

Runtime complexity: $O(\text{Radius1} * 2)$

Storage complexity (byte): $O(\text{Radius1} * 8)$

Result

If the parameter values are correct, the operator `gen_ellipse` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised. The clipping according to the current image format is set via the operator `set_system('clip_region', '<'true'/'false'>')`.

Parallelization Information

`gen_ellipse` is *reentrant* and processed *without* parallelization.

<i>Possible Successors</i>
paint_region , reduce_domain
<i>Alternatives</i>
gen_circle , gen_region_polygon_filled , draw_ellipse
<i>See also</i>
disp_ellipse , set_shape , smallest_circle , reduce_domain
<i>Module</i>
Region processing

gen_empty_region (: EmptyRegion : :)

Create an empty region.

The operator [gen_empty_region](#) creates an empty region. This means that the output parameter contains an object. Thus, [count_obj](#) returns 1. The area of the region is 0. Most of the shape features are undefined (0). It should be noted that an empty region must not be confused with the empty tuple.

<i>Parameter</i>
▷ EmptyRegion (output_object) region $\leadsto$ Hobject Empty region (no pixels).
<i>Parallelization Information</i>
gen_empty_region is <i>reentrant</i> and processed <i>without parallelization</i> .
<i>Module</i>
Region processing

gen_grid_region (: RegionGrid : RowSteps, ColumnSteps, Type, Width, Height :)

Create a region from lines or pixels.

The operator [gen_grid_region](#) creates a grid constructed of lines (**Type** = 'lines') or pixels (**Type** = 'points'). In case of 'lines' continuous lines are returned, in case of 'points' only the intersections of the lines. Starting from the pixel (0,0) to the pixel (**Height**-1,**Width**-1) the grid is built up at stepping width **RowSteps** in line direction and **ColumnSteps** in column direction. In the 'lines' mode **RowSteps**, **ColumnSteps** respectively, can be set to zero. In this case only columns, lines respectively, are created.

<i>Attention</i>
If a very small pattern is chosen (RowSteps < 4 oder ColumnSteps < 4) the created region requires much storage.
In the 'points' mode RowSteps and ColumnSteps must not be set to zero.

<i>Parameter</i>
▷ RegionGrid (output_object) region $\leadsto$ Hobject Created lines/pixel region.
▷ RowSteps (input_control) extent.y $\leadsto$ integer / real Step width in line direction or zero. Default Value : 10 Suggested values : RowSteps $\in$ {0, 2, 3, 4, 5, 7, 10, 15, 20, 30, 50, 100} Typical range of values : $0 \leq \text{RowSteps} \leq 512$ (lin) Minimum Increment : 1 Recommended Increment : 10 Restriction : (RowSteps > 1) $\vee$ (RowSteps = 0)

- ▷ **ColumnSteps** (input_control) extent.x $\leadsto$ integer / real
 Step width in column direction or zero.
Default Value : 10
Suggested values : ColumnSteps $\in \{0, 2, 3, 4, 5, 7, 10, 15, 20, 30, 50, 100\}$
Typical range of values : $0 \leq \text{ColumnSteps} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : $(\text{ColumnSteps} > 1) \vee (\text{ColumnSteps} = 0)$
- ▷ **Type** (input_control) string $\leadsto$ string
 Type of created pattern.
Default Value : 'lines'
List of values : Type $\in \{'lines', 'points'\}$
- ▷ **Width** (input_control) extent.x $\leadsto$ integer
 Maximum width of pattern.
Default Value : 512
Suggested values : Width $\in \{128, 256, 512, 1024\}$
Typical range of values : $1 \leq \text{Width} \leq 1024$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : Width ≥ 1
- ▷ **Height** (input_control) extent.y $\leadsto$ integer
 Maximum height of pattern.
Default Value : 512
Suggested values : Height $\in \{128, 256, 512, 1024\}$
Typical range of values : $1 \leq \text{Height} \leq 1024$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : Height ≥ 1

Example

```
read_image(Image, 'fabrik')
gen_grid_region(Raster, 10, 10, 'lines', 512, 512)
reduce_domain(Image, Raster, Mask)
sobel_amp(Mask, GridSobel, 'sum_abs', 3)
disp_image(GridSobel, WindowHandle).
```

Complexity

The necessary storage (in bytes) for the region is:

$O((\text{ImageWidth}/\text{ColumnSteps}) * (\text{ImageHeight}/\text{RowSteps}))$

Result

If the parameter values are correct the operator [gen_grid_region](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised. The clipping according to the current image format is set via the operator [set_system\('clip_region', <'true'/'false'>\)](#).

Parallelization Information

[gen_grid_region](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[reduce_domain](#), [paint_region](#)

Alternatives

[gen_region_line](#), [gen_region_polygon](#), [gen_region_points](#), [gen_region_runs](#)

See also

[gen_checker_region](#), [reduce_domain](#)

Module

Region processing

gen_random_region (: RegionRandom : Width, Height :)

Create a random region.

The operator `gen_random_region` returns a random region. During this process every pixel in the image area $[0 \dots \text{Width} - 1][0 \dots \text{Height} - 1]$ is adapted into the region with the probability 0.5. The created region can be imagined as the threshold formation in an image with noise.

This procedure is particularly important for the creation of uncorrelated binary patterns. The random pattern is created by the C function “`nrnd48()`”.

Attention

If `Width` and `Height` are chosen large (> 100) the created region may require much storage space due to the internally used runlength coding. The gray values of the output region are undefined.

Parameter

- ▷ **RegionRandom** (output_object) region $\leadsto$ Hobject
Created random region with expansion `Width` x `Height`.
- ▷ **Width** (input_control) extent.x $\leadsto$ integer
Maximum horizontal expansion of random region.
Default Value : 128
Suggested values : `Width` $\in \{16, 32, 50, 64, 100, 128, 256, 300, 400, 512\}$
Typical range of values : $1 \leq \text{Width} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : `Width` > 0
- ▷ **Height** (input_control) extent.y $\leadsto$ integer
Maximum vertical expansion of random region.
Default Value : 128
Suggested values : `Height` $\in \{16, 32, 50, 64, 100, 128, 256, 300, 400, 512\}$
Typical range of values : $1 \leq \text{Height} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : `Height` > 0

Complexity

The worst case for the storage complexity for the created region (in byte) is: $O(\text{Width} * \text{Height} * 2)$.

Result

If the parameter values are correct, the operator `gen_random_region` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised. The clipping according to the current image format is set via the operator `set_system('clip-region', <'true'/'false'>)`.

Parallelization Information

`gen_random_region` is *reentrant* and processed *without* parallelization.

Possible Successors

`paint_region`, `reduce_domain`

See also

`gen_checker_region`, `hamming_change_region`, `add_noise_distribution`,
`add_noise_white`, `reduce_domain`

Module

Region processing

gen_random_regions (: Regions : Type, WidthMin, WidthMax, HeightMin, HeightMax, PhiMin, PhiMax, NumRegions, Width, Height :)

Create random regions like circles, rectangles and ellipses.

The operator `gen_random_region` generates circles, rectangles and ellipses whose parameters are determined at random. In each case only one lower, upper limit respectively, is given. The position is always random and cannot be determined by parameters. The parameter `NumRegions` indicates how many regions shall be created.

Parameter

- ▷ **Regions** (output_object) region-array $\leadsto$ *Hobject*
Created regions.
- ▷ **Type** (input_control) string $\leadsto$ *string*
Type of regions to be created.
Default Value : 'circle'
List of values : Type $\in$ {'circle', 'ring', 'ellipse', 'rectangle1', 'rectangle2'}
- ▷ **WidthMin** (input_control) number $\leadsto$ *real* / integer
Minimum width of the region.
Default Value : 10.0
Suggested values : WidthMin $\in$ {1.0, 3.0, 5.0, 10.0, 20.0, 40.0, 80.0}
Typical range of values : $1.0 \leq \text{WidthMin} \leq 511.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
Restriction : WidthMin > 0
- ▷ **WidthMax** (input_control) number $\leadsto$ *real* / integer
Maximum width of the region.
Default Value : 20.0
Suggested values : WidthMax $\in$ {1.0, 3.0, 5.0, 10.0, 20.0, 40.0, 80.0}
Typical range of values : $1.0 \leq \text{WidthMax} \leq 511.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
Restriction : WidthMax > 0
- ▷ **HeightMin** (input_control) number $\leadsto$ *real* / integer
Minimum height of the region.
Default Value : 10.0
Suggested values : HeightMin $\in$ {1.0, 3.0, 5.0, 10.0, 20.0, 40.0, 80.0}
Typical range of values : $1.0 \leq \text{HeightMin} \leq 511.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
Restriction : HeightMin > 0
- ▷ **HeightMax** (input_control) number $\leadsto$ *real* / integer
Maximum height of the region.
Default Value : 30.0
Suggested values : HeightMax $\in$ {1.0, 3.0, 5.0, 10.0, 20.0, 40.0, 80.0}
Typical range of values : $1.0 \leq \text{HeightMax} \leq 511.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
Restriction : HeightMax > 0
- ▷ **PhiMin** (input_control) number $\leadsto$ *real* / integer
Minimum rotation angle of the region.
Default Value : -0.7854
Suggested values : PhiMin $\in$ {0.0, 0.1, 0.3, 0.6, 0.9, 1.2, 1.5}
Typical range of values : $0.0 \leq \text{PhiMin} \leq 6.28$ (lin)
Minimum Increment : 0.001
Recommended Increment : 0.10
Restriction : PhiMin > 0
- ▷ **PhiMax** (input_control) number $\leadsto$ *real* / integer
Maximum rotation angle of the region.
Default Value : 0.7854
Suggested values : PhiMax $\in$ {0.0, 0.1, 0.3, 0.6, 0.9, 1.2, 1.5}
Typical range of values : $0.0 \leq \text{PhiMax} \leq 6.28$ (lin)
Minimum Increment : 0.001
Recommended Increment : 0.10
Restriction : PhiMax > 0

- ▷ **NumRegions** (input_control) integer $\leadsto$ integer
 Number of regions.
Default Value : 100
Suggested values : NumRegions $\in \{1, 5, 20, 100, 200, 500, 1000, 2000\}$
Typical range of values : $1 \leq \text{NumRegions} \leq 2000$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : NumRegions > 0
- ▷ **Width** (input_control) integer $\leadsto$ integer
 Maximum horizontal expansion.
Default Value : 512
Suggested values : Width $\in \{128, 256, 512, 1024\}$
Typical range of values : $1 \leq \text{Width} \leq 1024$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : Width > 0
- ▷ **Height** (input_control) integer $\leadsto$ integer
 Maximum vertical expansion.
Default Value : 512
Suggested values : Height $\in \{128, 256, 512, 1024\}$
Typical range of values : $1 \leq \text{Height} \leq 1024$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : Height > 0

Result

If the parameter values are correct `gen_random_regions` returns the value 2 (H.MSG_TRUE). Otherwise an exception handling is raised. The clipping according to the current image format is determined by the operator `set_system('clip_region', <'true'/'false'>)`.

Parallelization Information

`gen_random_regions` is *reentrant* and processed *without* parallelization.

Possible Successors

`paint_region`

Module

Region processing

gen_rectangle1 (: Rectangle : Row1, Column1, Row2, Column2 :)

Create a rectangle parallel to the coordinate axes.

The operator `gen_rectangle1` generates one or more rectangles parallel to the coordinate axes which are described by the upper left corner (**Row1**, **Column1**) and the lower right corner (**Row2**, **Column2**). More than one region can be created by passing a tuple of corner points. The coordinate system runs from (0,0) (upper left corner) to (Width-1, Height-1). See `get_system` and `reset_obj_db` in this context.

Parameter

- ▷ **Rectangle** (output_object) region(-array) $\leadsto$ Hobject
 Created rectangle.
- ▷ **Row1** (input_control) rectangle.origin.y(-array) $\leadsto$ real / integer
 Line of upper left corner point.
Default Value : 30.0
Suggested values : Row1 $\in \{0.0, 10.0, 20.0, 50.0, 100.0, 200.0\}$
Typical range of values : $-\infty \leq \text{Row1} \leq \infty$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0

- ▷ **Column1** (input_control) rectangle.origin.x(-array) $\leadsto$ real / integer
 Column of upper left corner point.
Default Value : 20.0
Suggested values : Column1 $\in$ {0.0, 10.0, 20.0, 50.0, 100.0, 200.0}
Typical range of values : $-\infty \leq \text{Column1} \leq \infty$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
- ▷ **Row2** (input_control) rectangle.corner.y(-array) $\leadsto$ real / integer
 Line of lower right corner point.
Default Value : 100.0
Suggested values : Row2 $\in$ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0, 511.0}
Typical range of values : $-\infty \leq \text{Row2} \leq \infty$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
Restriction : Row2 $\geq$ Row1
- ▷ **Column2** (input_control) rectangle.corner.x(-array) $\leadsto$ real / integer
 Column of lower right corner point.
Default Value : 200.0
Suggested values : Column2 $\in$ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0, 511.0}
Typical range of values : $-\infty \leq \text{Column2} \leq \infty$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
Restriction : Column2 $\geq$ Column1

Example

```
/* Contrast improvement in a rectangular region of interest */
```

```
read_image(Image, 'fabrik')
open_window(0, 0, -1, -1, 'root', 'visible', '', WindowHandle)
disp_image(Image, WindowHandle)
draw_rectangle1(WindowHandle, Row1, Column1, Row2, Column2)
gen_rectangle1(Rectangle, Row1, Column1, Row2, Column2)
reduce_domain(Image, Rectangle, Mask)
emphasize(Mask, Emphasize, 9, 9, 1.0)
disp_image(Emphasize, WindowHandle).
```

Result

If the parameter values are correct, the operator [gen_rectangle1](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised. The clipping according to the current image format is set via the operator [set_system\('clip.region', <'true'/'false'>\)](#).

Parallelization Information

[gen_rectangle1](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[paint_region](#), [reduce_domain](#)

Alternatives

[gen_rectangle2](#), [gen_region_polygon](#), [fill_up](#), [gen_region_runs](#), [gen_region_points](#), [gen_region_line](#)

See also

[draw_rectangle1](#), [reduce_domain](#), [smallest_rectangle1](#)

Module

Region processing

gen_rectangle2 (: Rectangle : Row, Column, Phi, Length1, Length2 :)

Create a rectangle of any orientation.

The operator `gen_rectangle2` generates one or more rectangles with the center (`Row`, `Column`), the orientation `Phi` and the half edge lengths `Length1` and `Length2`. The orientation is given in arc measure and indicates the angle between the horizontal axis and `Length1` (mathematically positive). The coordinate system runs from (0,0) (upper left corner) to (Width-1,Height-1). See `get_system` and `reset_obj_db` in this context. More than one region can be created by passing one tuple of corner points.

Attention

The gray values of the output objects are undefined.

Parameter

- ▷ **Rectangle** (output_object) region(-array) $\leadsto$ Hobject
Created rectangle.
- ▷ **Row** (input_control) rectangle2.center.y(-array) $\leadsto$ real / integer
Line index of the center.
Default Value : 50.0
Suggested values : Row $\in$ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
Typical range of values : $-\infty \leq \text{Row} \leq \infty$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
- ▷ **Column** (input_control) rectangle2.center.x(-array) $\leadsto$ real / integer
Column index of the center.
Default Value : 100.0
Suggested values : Column $\in$ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
Typical range of values : $-\infty \leq \text{Column} \leq \infty$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
- ▷ **Phi** (input_control) rectangle2.angle.rad(-array) $\leadsto$ real / integer
Angle of longitudinal axis to the horizontal (in radians).
Default Value : 0.0
Suggested values : Phi $\in$ {-1.178097, -0.785398, -0.392699, 0.0, 0.392699, 0.785398, 1.178097}
Typical range of values : $-1.178097 \leq \text{Phi} \leq 1.178097$ (lin)
Minimum Increment : 0.001
Recommended Increment : 0.1
Restriction : $((-\pi/2) < \text{Phi}) \wedge (\text{Phi} \leq (\pi/2))$
- ▷ **Length1** (input_control) rectangle2.hwidth(-array) $\leadsto$ real / integer
Half width.
Default Value : 200.0
Suggested values : Length1 $\in$ {3.0, 5.0, 10.0, 15.0, 20.0, 50.0, 100.0, 200.0, 300.0, 500.0}
Typical range of values : $-\infty \leq \text{Length1} \leq \infty$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
- ▷ **Length2** (input_control) rectangle2.hheight(-array) $\leadsto$ real / integer
Half height.
Default Value : 100.0
Suggested values : Length2 $\in$ {1.0, 2.0, 3.0, 5.0, 10.0, 15.0, 20.0, 50.0, 100.0, 200.0}
Typical range of values : $-\infty \leq \text{Length2} \leq \infty$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
Restriction : Length2 $\leq$ Length1

Result

If the parameter values are correct the operator `gen_rectangle2` returns the value 2 (H_MSG.TRUE). Otherwise an exception handling is raised. The clipping according to the current image format is set via the operator `set_system('clip_region', <'true'/'false'>)`.

Parallelization Information

`gen_rectangle2` is *reentrant* and processed *without* parallelization.

Possible Successors

`paint_region`, `reduce_domain`

Alternatives

[gen_rectangle1](#), [gen_region_polygon_filled](#), [gen_region_polygon](#), [gen_region_points](#), [fill_up](#)

See also

[draw_rectangle2](#), [reduce_domain](#), [smallest_rectangle2](#), [gen_ellipse](#)

Module

Region processing

gen_region_contour_xld (Contour : Region : Mode :)

Create a region from an XLD contour.

[gen_region_contour_xld](#) creates a region [Region](#) from a subpixel XLD contour [Contour](#). The contour is sampled according to the Bresenham algorithm and influenced by the parameter *neighborhood* of the operator [set_system](#). Open contours are closed before converting them to regions. Finally, the parameter [Mode](#) defines whether the region is filled up (*filled*) or returned by its contour (*margin*).

Parameter

- ▷ **Contour** (input_object) xld_cont $\leadsto$ *Hobject*
Input contour.
- ▷ **Region** (output_object) region $\leadsto$ *Hobject*
Created region.
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Fill mode of the region.
Default Value : 'filled'
Suggested values : Mode $\in$ {'filled', 'margin'}

Parallelization Information

[gen_region_contour_xld](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_contour_polygon_xld](#), [gen_contour_polygon_rounded_xld](#)

Alternatives

[gen_region_polygon](#), [gen_region_polygon_xld](#)

See also

[set_system](#)

Module

Sub-pixel operators

gen_region_histo (: Region : Histogram, Row, Column, Scale :)

Convert a histogram into a region.

[gen_region_histo](#) converts a histogram created with [gray_histo](#) into a region. The effect of the three control parameters is the same as in [disp_image](#) and [set_paint](#).

Parameter

- ▷ **Region** (output_object) region $\leadsto$ *Hobject*
Region containing the histogram.
- ▷ **Histogram** (input_control) histogram-array $\leadsto$ *integer*
Input histogram.
- ▷ **Row** (input_control) point.y $\leadsto$ *integer*
Row coordinate of the center of the histogram.
Default Value : 255
Suggested values : Row $\in$ {100, 200, 255, 300, 400}
Typical range of values : $0 \leq \text{Row} \leq 511$

- ▷ **Column** (input_control) point.x $\leadsto$ integer
 Column coordinate of the center of the histogram.
Default Value : 255
Suggested values : Column $\in \{100, 200, 255, 300, 400\}$
Typical range of values : $0 \leq \text{Column} \leq 511$
- ▷ **Scale** (input_control) integer $\leadsto$ integer
 Scale factor for the histogram.
Default Value : 1
List of values : Scale $\in \{1, 2, 3, 4, 5, 6, 7\}$
Typical range of values : $1 \leq \text{Scale} \leq 10$ (lin)
Minimum Increment : 1
Recommended Increment : 1

Result

`gen_region_histo` returns 2 (H_MSG_TRUE) if all parameters are correct. If necessary, an exception handling is raised.

Parallelization Information

`gen_region_histo` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`gray_histo`

See also

`disp_channel`, `set_paint`

Module

Region processing

gen_region_hline (: Regions : Orientation, Distance :)

Store input lines described in Hesse normal shape as regions.

The operator `gen_region_hline` stores the lines described in Hesse normal shape as regions. A line is determined by the distance from the line to the origin (**Distance**, corresponds to the length of the normal vector) and the direction of the normal vector (**Orientation**, corresponds to the orientation of the line $\pm\pi/2$). The directions were defined in such a way that at **Orientation** = 0 the normal vector lies in the direction of the X axis, which corresponds to a vertical line. At **Orientation** = $\pi/2$ the normal vector points in the direction of the Y axis, i.e. a horizontal line is described.

Attention

The lines are clipped to the current maximum image format.

Parameter

- ▷ **Regions** (output_object) region(-array) $\leadsto$ Hobject
 Created regions (one for every line), clipped to maximum image format.
Number of elements : Regions = Distance
- ▷ **Orientation** (input_control) hesseline.angle.rad(-array) $\leadsto$ real / integer
 Orientation of the normal vector in radians.
Default Value : 0.0
Suggested values : Orientation $\in \{-0.78, 0.0, 0.78, 1.57\}$
Typical range of values : $-\infty \leq \text{Orientation} \leq \infty$ (lin)
Recommended Increment : 0.02
Number of elements : Orientation = Distance
- ▷ **Distance** (input_control) hesseline.distance(-array) $\leadsto$ real / integer
 Distance from the line to the coordinate origin (0.0).
Default Value : 200
Suggested values : Distance $\in \{10, 50, 100, 200, 300, 400\}$
Typical range of values : $-\infty \leq \text{Distance} \leq \infty$ (lin)
Recommended Increment : 1

Result

The operator `gen_region_hline` always returns the value 2 (H_MSG_TRUE).

Parallelization Information

[gen_region_hline](#) is *reentrant* and processed *without* parallelization.

Alternatives

[gen_region_line](#)

See also

[hough_lines](#)

Module

Region processing

gen_region_line (: RegionLines : BeginRow, BeginCol, EndRow,
EndCol :)

Store input lines as regions.

The operator [gen_region_line](#) stores the given lines (with starting point [[BeginRow](#),[BeginCol](#)] and ending point [[EndRow](#), [EndCol](#)]) as region.

Parameter

- ▷ **RegionLines** (output_object) region(-array) $\leadsto$ *Hobject*
Created regions.
- ▷ **BeginRow** (input_control) line.begin.y(-array) $\leadsto$ *integer*
Line coordinates of the starting points of the input lines.
Default Value : 100
Suggested values : $\text{BeginRow} \in \{10, 50, 100, 200, 300, 400\}$
Typical range of values : $-\infty \leq \text{BeginRow} \leq \infty$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **BeginCol** (input_control) line.begin.x(-array) $\leadsto$ *integer*
Column coordinates of the starting points of the input lines.
Default Value : 50
Suggested values : $\text{BeginCol} \in \{10, 50, 100, 200, 300, 400\}$
Typical range of values : $-\infty \leq \text{BeginCol} \leq \infty$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **EndRow** (input_control) line.end.y(-array) $\leadsto$ *integer*
Line coordinates of the ending points of the input lines.
Default Value : 150
Suggested values : $\text{EndRow} \in \{50, 100, 200, 300, 400, 500\}$
Typical range of values : $-\infty \leq \text{EndRow} \leq \infty$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **EndCol** (input_control) line.end.x(-array) $\leadsto$ *integer*
Column coordinates of the ending points of the input lines.
Default Value : 250
Suggested values : $\text{EndCol} \in \{50, 100, 200, 300, 400, 500\}$
Typical range of values : $-\infty \leq \text{EndCol} \leq \infty$ (lin)
Minimum Increment : 1
Recommended Increment : 1

Result

The operator [gen_region_line](#) always returns the value 2 (H_MSG_TRUE). The clipping according to the current image format is determined by the operator [set_system\('clip_region', <'true'/'false'>\)](#).

Parallelization Information

[gen_region_line](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[split_skeleton_lines](#)

	Alternatives
gen_region_hline	
	Module
Region processing	

gen_region_points (: Region : Rows, Columns :)

Store individual pixels as image region.

The operator [gen_region_points](#) creates a region described by a number of pixels. The pixels do not have to be stored in a fixed order, but the best runtime behavior is obtained when the pixels are stored in ascending order. The order is as follows:

$$(l_1, c_1) \leq (l_2, c_2) := (l_1 < l_2) \vee (l_1 = l_2) \wedge (c_1 \leq c_2)$$

The indicated coordinates stand for two consecutive pixels in the tuple.

	Parameter
▷ Region (output_object)	region $\leadsto$ Hobject Created region.
▷ ROWS (input_control)	coordinates.y(-array) $\leadsto$ integer Lines of the pixels in the region. Default Value : 100 Suggested values : Rows $\in \{0, 10, 30, 50, 100, 200, 300, 500\}$ Typical range of values : $-\infty \leq \text{ROWS} \leq \infty$ (lin) Minimum Increment : 1 Recommended Increment : 1
▷ Columns (input_control)	coordinates.x(-array) $\leadsto$ integer Columns of the pixels in the region. Default Value : 100 Suggested values : Columns $\in \{0, 10, 30, 50, 100, 200, 300, 500\}$ Typical range of values : $-\infty \leq \text{Columns} \leq \infty$ (lin) Minimum Increment : 1 Recommended Increment : 1 Number of elements : Columns = Rows

	Complexity
F shall be the number of pixels. If the pixels are sorted in ascending order the runtime complexity is: $O(F)$, otherwise $O(\log(F) * F)$.	

	Result
The operator gen_region_points returns the value 2 (H_MSG_TRUE) if the pixels are located within the image format. Otherwise an exception handling is raised. The clipping according to the current image format is set via the operator set_system('clip_region', <'true'/'false'>) . If an empty region is created (by the clipping or by an empty input) the operator set_system('store_empty_region', <true/false>) determines whether the region is returned or an empty object tuple.	

	Parallelization Information
gen_region_points is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.	

	Possible Predecessors
get_region_points	

	Possible Successors
paint_region , reduce_domain	

	Alternatives
gen_region_polygon , gen_region_runs , gen_region_line	

	See also
reduce_domain	

	Module
Region processing	

gen_region_polygon (: Region : Rows, Columns :)

Store a polygon as an image object.

The operator `gen_region_polygon` creates a region from a polygon row described by a series of line and column coordinates. The created region consists of the pixels of the routes defined thereby, wherein it is linearly interpolated between the base points.

Attention

The region is not automatically closed and not filled. The gray values of the output regions are undefined.

Parameter

- ▷ **Region** (output_object) region $\leadsto$ Hobject
Created region.
- ▷ **Rows** (input_control) polygon.y-array $\leadsto$ integer
Line indices of the base points of the region contour.
Default Value : 100
Suggested values : Rows $\in \{0, 10, 30, 50, 100, 200, 300, 500\}$
Typical range of values : $-\infty \leq \text{Rows} \leq \infty$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Columns** (input_control) polygon.x-array $\leadsto$ integer
Column indices of the base points of the region contour.
Default Value : 100
Suggested values : Columns $\in \{0, 10, 30, 50, 100, 200, 300, 500\}$
Typical range of values : $-\infty \leq \text{Columns} \leq \infty$ (lin)
Minimum Increment : 1
Recommended Increment : 1
Number of elements : Columns = Rows

Example

```
/* Polygon-approximation*/
get_region_polygon(Region, 7, Row, Column)
/* store it as a region */
gen_region_polygon(Pol, Row, Column)
/* fill up the hole */
fill_up(Pol, Filled).
```

Result

If the base points are correct the operator `gen_region_polygon` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised. The clipping according to the current image format is set via the operator `set_system('clip_region', <'true'/'false'>)`. If an empty region is created (by the clipping or by an empty input) the operator `set_system('store_empty_region', <true/false>)` determines whether the region is returned or an empty object tuple.

Parallelization Information

`gen_region_polygon` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`get_region_polygon`, `draw_polygon`

Alternatives

`gen_region_polygon-filled`, `gen_region_points`, `gen_region_runs`

See also

`fill_up`, `reduce_domain`, `get_region_polygon`, `draw_polygon`

Module

Region processing

gen_region_polygon_filled (: Region : Rows, Columns :)

Store a polygon as a “filled” region.

The operator [gen_region_polygon_filled](#) creates a region from a polygon containing the corner points of the region (line and column coordinates) either clockwise or anti-clockwise. Contrary to [gen_region_polygon](#) a “filled” region is returned here.

Parameter

- ▷ **Region** (output_object) region $\leadsto$ *Hobject*
Created region.
- ▷ **Rows** (input_control) polygon.y-array $\leadsto$ *integer*
Line indices of the base points of the region contour.
Default Value : 100
Suggested values : Rows $\in \{0, 10, 30, 50, 100, 200, 300, 500\}$
Typical range of values : $-\infty \leq \text{Rows} \leq \infty$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Columns** (input_control) polygon.x-array $\leadsto$ *integer*
Column indices of the base points of the region contour.
Default Value : 100
Suggested values : Columns $\in \{0, 10, 30, 50, 100, 200, 300, 500\}$
Typical range of values : $-\infty \leq \text{Columns} \leq \infty$ (lin)
Minimum Increment : 1
Recommended Increment : 1
Number of elements : Columns = Rows

Example

```
/* Polygon approximation */
T_get_region_polygon(Region, 7, &Row, &Column);
T_gen_region_polygon_filled(&Pol, Row, Column);
/* fill up with original gray value */
reduce_domain(Image, Pol, &New);
```

Result

If the base points are correct the operator [gen_region_polygon_filled](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised. The clipping according to the current image format is set via the operator [set_system\('clip_region', <'true'/'false'>\)](#). If an empty region is created (by the clipping or by an empty input) the operator [set_system\('store_empty_region', <true/false>\)](#) determines whether the region is returned or an empty object tuple.

Parallelization Information

[gen_region_polygon_filled](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[get_region_polygon](#), [draw_polygon](#)

Alternatives

[gen_region_polygon](#), [gen_region_points](#), [draw_polygon](#)

See also

[gen_region_polygon](#), [reduce_domain](#), [get_region_polygon](#), [gen_region_runs](#)

Module

Region processing

gen_region_polygon_xld (Polygon : Region : Mode :)

Create a region from an XLD polygon.

gen_region_polygon_xld creates a region **Region** from a subpixel XLD polygon **Polygon**. The polygon is sampled according to the Bresenham algorithm and influenced by the parameter *neighborhood* of the operator **set_system**. Open polygons are closed before converting them to regions. Finally, the parameter **Mode** defines whether the region is filled up (*filled*) or returned by its contour (*margin*).

Parameter

- ▷ **Polygon** (input_object) xld_poly $\leadsto$ *Hobject*
Input polygon.
- ▷ **Region** (output_object) region $\leadsto$ *Hobject*
Created region.
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Fill mode of the region.
Default Value : 'filled'
Suggested values : Mode $\in$ {'filled', 'margin'}

Parallelization Information

gen_region_polygon_xld is *reentrant* and processed *without* parallelization.

Possible Predecessors

gen_contour_polygon_xld, **gen_contour_polygon_rounded_xld**

Alternatives

gen_region_polygon, **gen_region_contour_xld**

See also

set_system

Module

Sub-pixel operators

gen_region_runs (: Region : Row, ColumnBegin, ColumnEnd :)

Create an image region from a runlength coding.

The operator **gen_region_runs** creates a region described by the input runlength structure. The runlength representation is created by examining a region line by line with ascending line number (= from “top” to “bottom”). Every line runs through from left to right (ascending column number). All starting and ending points being stored by region segments (=runs). Thus a region can be described by a sequence of runs, a run being defined by line number as well as starting and ending points (column number).

The storing is fastest when the runs are sorted. The order is as follows:

$$(l_1, b_1, e_1) \leq (l_2, b_2, e_2) := (l_1 < l_2) \vee (l_1 = l_2) \wedge (b_1 \leq b_2)$$

.

Parameter

- ▷ **Region** (output_object) region $\leadsto$ *Hobject*
Created region.
- ▷ **Row** (input_control) chord.y(-array) $\leadsto$ *integer*
Lines of the runs.
Default Value : 100
Suggested values : Row $\in$ {0, 50, 100, 200, 300, 500}
Typical range of values : $-\infty \leq \text{Row} \leq \infty$ (lin)
Minimum Increment : 1
Recommended Increment : 10

- ▷ **ColumnBegin** (input_control) chord.x1(-array) $\leadsto$ integer
Columns of the starting points of the runs.
Default Value : 50
Suggested values : ColumnBegin $\in \{0, 50, 100, 200, 300, 500\}$
Typical range of values : $-\infty \leq \text{ColumnBegin} \leq \infty$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Number of elements : ColumnBegin = Row
- ▷ **ColumnEnd** (input_control) chord.x2(-array) $\leadsto$ integer
Columns of the ending points of the runs.
Default Value : 200
Suggested values : ColumnEnd $\in \{50, 100, 200, 300, 500\}$
Typical range of values : $-\infty \leq \text{ColumnEnd} \leq \infty$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Number of elements : ColumnEnd = Row
Restriction : ColumnEnd $\geq$ ColumnBegin

Complexity

F shall be the number of pixels. If the pixels are sorted in ascending order the runtime complexity is: $O(F)$, otherwise it is $O(\log(F) * F)$.

Result

If the data is correct the operator `gen_region_runs` returns the value 2 (H_MSG_TRUE), otherwise an exception handling is raised. The clipping according to the current image format is set via the operator `set_system('clip_region', <'true'/'false'>)`. If an empty region is created (by the clipping or by an empty input) the operator `set_system('store_empty_region', <true/false>)` determines whether the region is returned or an empty object tuple.

Parallelization Information

`gen_region_runs` is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

`get_region_runs`

Alternatives

`gen_region_points`, `gen_region_polygon`, `gen_region_line`, `gen_region_polygon_filled`

See also

`reduce_domain`

Module

Region processing

label_to_region (LabelImage : Regions : :)

Extract regions with equal gray values from an image.

`label_to_region` segments an image into regions of equal gray value. One output region is generated for each gray value occurring in the image. This is similar to calling `threshold` multiple times, and accumulating the results with `concat_obj`. Another related operator is `regiongrowing`. However, `label_to_region` does not perform a `connection` operation on the resulting regions, i.e., they may be disconnected. A typical application of `label_to_region` is the segmentation of label images, hence its name.

The number of output regions is limited by the system parameter 'max_outp_obj_par', which can be read via

`get_system(, 'max_outp_obj_par' : <Anzahl>)`.

Attention

`label_to_region` is not implemented for images of type 'real'. The input images must not contain negative gray values.

Parameter
▷ LabelImage (input_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / int2 / int4 Label image. ▷ Regions (output_object) region-array $\leadsto$ <i>Hobject</i> Regions having a constant gray value.
Complexity
Let $x1$ be the minimum x-coordinate, $x2$ the maximum x-coordinate, $y1$ be the minimum y-coordinate, and $y2$ the maximum y-coordinate of a particular gray value. Furthermore, let N be the number of different gray values in the image. Then the runtime complexity is $O(N * (x2 - x1 + 1) * (y2 - y1 + 1))$
Result
label_to_region returns 2 (H_MSG_TRUE) if the gray values lie within a correct range. The behavior with respect to the input images and output regions can be determined by setting the values of the flags 'no_object_result', 'empty_region_result', and 'store_empty_region' with set_system. If necessary, an exception is raised.
Parallelization Information
label_to_region is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).
Possible Predecessors
min_max_gray, sobel_amp, gauss_image, reduce_domain, diff_of_gauss
Possible Successors
connection, dilation1, erosion1, opening, closing, rank_region, shape_trans, skeleton
See also
threshold, concat_obj, regiongrowing, region_to_label
Module
Region processing

12.4 Features

area_center (Regions : : : Area, Row, Column)

Area and center of regions.

The operator [area_center](#) calculates the area and the center of the input regions. The area is defined as the number of pixels of a region. The center is calculated as the mean value of the line or column coordinates, respectively, of all pixels.

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of the input region. In case of empty region all parameters have the value 0.0 if no other behavior was set (see [set_system](#)).

Parameter
▷ Regions (input_object) region(-array) $\leadsto$ <i>Hobject</i> Region(s) to be examined. ▷ Area (output_control) integer(-array) $\leadsto$ <i>integer</i> Area of the region. ▷ Row (output_control) point.y(-array) $\leadsto$ <i>real</i> Line index of the center. ▷ Column (output_control) point.x(-array) $\leadsto$ <i>real</i> Column index of the center.
Example

```
#include <iostream.h>
#include "HalconCpp.h"

main()
{
```

```

Tuple    area, row, column;

HImage   img ("affe");
HWindow  w;

img.Display (w);
w.Click ();

HRegionArray  reg = (img >= 164).Connection ();

reg.Display (w);
w.Click ();

area = reg.AreaCenter (&row, &column);

for (int i = 0; i < reg.Num (); i++)
{
    cout << "Row    [" << i << "]" << "= " << row[i].D ();
    cout << "\t\tColumn [" << i << "]" << "= " << column[i].D () << endl;
}

cout << "Total number of regions: " << reg.Num () << endl;
return(0);
}

```

Complexity

If F is the area of a region the mean runtime complexity is $O(\sqrt{F})$.

Result

The operator `area_center` returns the value 2 (H_MSG.TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result', <Result>)`. The behavior in case of empty region (the region is the empty set) is set via `set_system('empty_region_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`area_center` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`

See also

`select_shape`

Module

Region processing

circularity (Regions : : : Circularity)

Shape factor for the circularity (similarity to a circle) of a region.

The operator `circularity` calculates the similarity of the input region with a circle.

Calculation: If F is the area of the region and max is the maximum distance from the center to all contour pixels, the shape factor C is defined as:

$$C = \frac{F}{(max^2 * \pi)}$$

The shape factor C of a circle is 1. If the region is long or has holes C is smaller than 1. The operator `circularity` especially responds to large bulges, holes and unconnected regions.

In case of an empty region the operator `circularity` returns the value 0 (if no other behavior was set (see `set_system`)). If more than one region is passed the numerical values of the shape factor are stored in a tuple, the position of a value in the tuple corresponding to the position of the region in the input tuple.

Parameter	
▷ Regions (input_object)	region(-array) $\leadsto$ Hobject Region(s) to be examined.
▷ Circularity (output_control)	real(-array) $\leadsto$ real Roundness of the input region(s). Assertion: $(0 \leq \text{Circularity}) \wedge (\text{Circularity} \leq 1.0)$
Example	
<pre>/* Comparison between shape factors of rectangle, circle and ellipse: */ gen_rectangle1(R1,10,10,20,20) gen_rectangle2(R2,100,100,0.0,100,20) gen_ellipse(E100,100,0.0,100,20) gen_circle(C,100,100,20) circularity([R1,R2,E,C],M) fwrite_string(FileId,['quadrante: ',M[1]]) fnew_line(FileId) fwrite_string(FileId,['rectangle: ',M[2]]) fnew_line(FileId) fwrite_string(FileId,['ellipse: ',M[3]]) fnew_line(FileId) fwrite_string(FileId,['circle: ',M[4]]) fnew_line(FileId)</pre>	
Result	
The operator circularity returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system('no_object_result', <Result>). The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result', <Result>). If necessary an exception handling is raised.	
Parallelization Information	
circularity is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).	
Possible Predecessors	
threshold , regiongrowing , connection	
Alternatives	
roundness , compactness , convexity , eccentricity	
See also	
area_center , select_shape	
Module	
Region processing	

compactness (Regions : : : Compactness)

Shape factor for the compactness of a region.

The operator [compactness](#) calculates the compactness of the input regions.

Calculation: If L is the length of the contour (see [contlength](#)) and F the area of the region the shape factor C is defined as:

$$C = \frac{L^2}{4F\pi}$$

The shape factor C of a circle is 1. If the region is long or has holes C is larger than 1. The operator [compactness](#) responds to the course of the contour (roughness) and to holes. In case of an empty region the operator [compactness](#) returns the value 0 if no other behavior was set (see [set_system](#)). If more than one region is passed the numerical values of the shape factor are stored in a tuple, the position of a value in the tuple corresponding to the position of the region in the input tuple.

Parameter	
▷ Regions (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Region(s) to be examined.
▷ Compactness (output_control)	real(-array) $\leadsto$ <i>real</i> Compactness of the input region(s).
Assertion : (Compactness ≥ 1.0) $\vee$ (Compactness = 0)	
Result	
The operator <code>compactness</code> returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator <code>set_system('no_object_result', <Result>)</code> . The behavior in case of empty region (the region is the empty set) is set via <code>set_system('empty_region_result', <Result>)</code> . If necessary an exception handling is raised.	
Parallelization Information	
<code>compactness</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).	
Possible Predecessors	
<code>threshold</code> , <code>regiongrowing</code> , <code>connection</code>	
Alternatives	
<code>compactness</code> , <code>convexity</code> , <code>eccentricity</code>	
See also	
<code>contlength</code> , <code>area_center</code> , <code>select_shape</code>	
Module	
Region processing	

connect_and_holes (Regions : : : NumConnected, NumHoles)

Number of connection components and holes

The operator `connect_and_holes` calculates the number of connection components and the number of holes of each region of **Regions**.

If more than one region is passed the numerical values of the output control parameters **NumConnected** and **NumHoles** are each stored in a tuple, the position of a value in the tuple corresponding to the position of the region in the input tuple.

Parameter	
▷ Regions (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Region(s) to be examined.
▷ NumConnected (output_control)	integer(-array) $\leadsto$ <i>integer</i> Number of connection components of a region.
▷ NumHoles (output_control)	integer(-array) $\leadsto$ <i>integer</i> Number of holes of a region.
Result	
The operator <code>connect_and_holes</code> returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator <code>set_system('no_object_result', <Result>)</code> . The behavior in case of empty region (the region is the empty set) is set via <code>set_system('empty_region_result', <Result>)</code> .	
Parallelization Information	
<code>connect_and_holes</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).	
Possible Predecessors	
<code>threshold</code> , <code>regiongrowing</code> , <code>connection</code>	
Alternatives	
<code>euler_number</code>	
See also	
<code>connection</code> , <code>fill_up</code> , <code>fill_up_shape</code> , <code>union1</code>	
Module	
Region processing	

contlength (Regions : : : ContLength)

Contour length of a region.

The operator **contlength** calculates the total length of the contour (sum of all connection components of the region) for each region of **Regions**. The distance between two neighboring contour points parallel to the coordinate axes is rated 1, the distance in the diagonal is rated $\sqrt{2}$. If more than one region is passed the numerical values of the contour length are stored in a tuple, the position of a value in the tuple corresponding to the position of the region in the input tuple. In case of an empty region the operator **contlength** returns the value 0.

Attention

The contour of holes is not calculated.

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ *Hobject*
Region(s) to be examined.
 - ▷ **ContLength** (output_control) real(-array) $\leadsto$ *real*
Contour length of the input region(s).
- Assertion :** ContLength ≥ 0

Example

```
#include <iostream.h>
#include "HalconCpp.h"

int main (int argc, char *argv[])
{
    if (argc < 2)
    {
        cout << "Usage: " << argv[0] << " <# of regions> " << endl;
        return (-1);
    }

    HWindow          w;
    HRegionArray      reg;

    int NumOfElements = atoi (argv[1]);

    cout << "Draw " << NumOfElements << " regions " << endl;

    for (int i=0; i < NumOfElements; i++)
    {
        reg[i] = w.DrawRegion ();
    }

    Tuple circ = reg.Circularity ();
    Tuple cont = reg.Contlength ();

    for (i = 0; i < NumOfElements; i++)
    {
        cout << "Circularity of " << i+1 << ". region = " << circ[i].D();
        cout << "\t\t Contour Length of" << i+1 <<
            "\t\t ". region = " << cont[i].D() << endl;
    }

    w.Click ();
    return(0);
}
```

Result

The operator **contlength** returns the value 2 (H_MSG_TRUE) if the input is not empty. The

behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information	
<code>contlength</code>	is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>).
Possible Predecessors	
<code>threshold</code> , <code>regiongrowing</code> , <code>connection</code>	
Possible Successors	
<code>get_region_contour</code>	
Alternatives	
<code>compactness</code>	
See also	
<code>area_center</code> , <code>get_region_contour</code>	
Module	
Region processing	

convexity (Regions : : : Convexity)

Shape factor for the convexity of a region.

The operator `convexity` calculates the convexity of each input region of `Regions`.

Calculation: If F_c is the area of the convex hull and F_o the original area of the region the shape factor C is defined as:

$$C = \frac{F_o}{F_c}$$

The shape factor C is 1 if the region is convex (e.g., rectangle, circle etc.). If there are indentations or holes C is smaller than 1.

In case of an empty region the operator `convexity` returns the value 0 (if no other behavior was set (see `set_system`)). If more than one region is passed the numerical values of the contour length are stored in a tuple, the position of a value in the tuple corresponding to the position of the region in the input tuple.

Parameter	
▷ Regions (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Region(s) to be examined.
▷ Convexity (output_control)	real(-array) $\leadsto$ <i>real</i> Convexity of the input region(s).
Assertion : Convexity ≤ 1	

Result	
The operator <code>convexity</code> returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator <code>set_system('no_object_result', <Result>)</code> . The behavior in case of empty region (the region is the empty set) is set via <code>set_system('empty_region_result', <Result>)</code> . If necessary an exception handling is raised.	

Parallelization Information	
<code>convexity</code>	is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>).
Possible Predecessors	
<code>threshold</code> , <code>regiongrowing</code> , <code>connection</code>	
See also	
<code>select_shape</code> , <code>area_center</code> , <code>shape_trans</code>	
Module	
Region processing	

diameter_region (Regions : : : Row1, Column1, Row2, Column2, Diameter)

Maximal distance between two boundary points of a region.

The operator **diameter_region** calculates the maximal distance between two boundary points of a region. The coordinates of these two extremes and the distance between them will be returned.

Attention

If the region is empty, the results of **Row1**, **Column1**, **Row2** and **Column2** (all of them = 0) may lead to confusion.

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be examined.
- ▷ **Row1** (output_control) line.begin.y(-array) $\leadsto$ *integer*
Row index of the first extreme point.
- ▷ **Column1** (output_control) line.begin.x(-array) $\leadsto$ *integer*
Column index of the first extreme point.
- ▷ **Row2** (output_control) line.end.y(-array) $\leadsto$ *integer*
Row index of the second extreme point.
- ▷ **Column2** (output_control) line.end.x(-array) $\leadsto$ *integer*
Column index of the second extreme point.
- ▷ **Diameter** (output_control) number(-array) $\leadsto$ *real*
Distance of the two extreme points.

Complexity

If F is the area of a region, the runtime complexity amounts to $O(\sqrt{F})$ on average.

Result

The operator **diameter_region** returns the value 2 (H_MSG_TRUE), if the input is not empty. The reaction to empty input (no input regions are available) may be determined with the help of the operator **set_system** ('no_object_result', <Result>). The reaction concerning an empty region (region is the empty set) will be determined by the operator **set_system**('empty_region_result', <Result>). If necessary an exception handling is raised.

Parallelization Information

diameter_region is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

threshold, **regiongrowing**, **connection**, **runlength_features**

Possible Successors

disp_line

Alternatives

smallest_rectangle2

Module

Region processing

eccentricity (Regions : : : Anisometry, Bulkiness, StructureFactor)

Shape features derived from the ellipse parameters.

The operator **eccentricity** calculates three shape features derived from the geometric moments.

Definition: If the parameters Ra , Rb and the area A of the region are given (see **elliptic_axis**), the following applies:

$$\text{Anisometry} = \frac{Ra}{Rb}$$

$$\text{Bulkiness} = \frac{\pi \cdot Ra \cdot Rb}{A}$$

$$\text{StructureFactor} = \text{Anisometry} \cdot \text{Bulkiness} - 1$$

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see [set_system](#)).

Parameter	
▷ Regions (input_object)	region(-array) $\leadsto$ Hobject Region(s) to be examined.
▷ Anisometry (output_control)	real(-array) $\leadsto$ real Shape feature (in case of a circle = 1.0). Assertion : $\text{Anisometry} \geq 1.0$
▷ Bulkiness (output_control)	real(-array) $\leadsto$ real Calculated shape feature.
▷ StructureFactor (output_control)	real(-array) $\leadsto$ real Calculated shape feature.

Complexity

If F is the area of the region the mean runtime complexity is $O(\sqrt{F})$.

Result

The operator [eccentricity](#) returns the value 2 (H_MSG.TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator [set_system](#) ('no_object_result', <Result>). The behavior in case of empty region (the region is the empty set) is set via [set_system](#) ('empty_region_result', <Result>). If necessary an exception handling is raised.

Parallelization Information

[eccentricity](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[threshold](#), [regiongrowing](#), [connection](#)

See also

[elliptic_axis](#), [moments_region_2nd](#), [select_shape](#), [area_center](#)

Module

Region processing

elliptic_axis (Regions : : : Ra, Rb, Phi)

Parameters of the equivalent ellipse.

The operator [elliptic_axis](#) calculates the radii and the orientation of the ellipse having the “same orientation” and the “same side relation” as the input region. Several input regions can be passed in [Regions](#) as tuples. The length of the main radius [Ra](#) and the secondary radius [Rb](#) as well as the orientation of the main axis with regard to the horizontal ([Phi](#)) are determined. The angle is indicated in arc measure.

Calculation:

If the moments M_{20} , M_{02} and M_{11} are normalized and passed to the area (see [moments_region_2nd](#)), the radii [Ra](#) and [Rb](#) are calculated as:

$$\text{Ra} = \frac{\sqrt{8(M_{20} + M_{02} + \sqrt{(M_{20} - M_{02})^2 + 4M_{11}^2})}}{2}$$

$$\text{Rb} = \frac{\sqrt{8(M_{20} + M_{02} - \sqrt{(M_{20} - M_{02})^2 + 4M_{11}^2})}}{2}$$

The orientation [Phi](#) is defined by:

$$\text{Phi} = -0.5 \text{atan2}(2M_{11}, M_{02} - M_{20})$$

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see [set_system\('no_object_result', <Result>\)](#)).

Parameter	
▷ Regions (input_object)	region(-array) $\leadsto$ Hobject Region(s) to be examined.
▷ Ra (output_control)	real(-array) $\leadsto$ real Main radius (normalized to the area). Assertion : $Ra \geq 0.0$
▷ Rb (output_control)	real(-array) $\leadsto$ real Secondary radius (normalized to the area). Assertion : $(Rb \geq 0.0) \wedge (Rb \leq Ra)$
▷ Phi (output_control)	real(-array) $\leadsto$ real Angle between main radius and x axis (arc measure). Assertion : $((-\pi/2) < \Phi) \wedge (\Phi \leq (\pi/2))$
Example	

```
read_image(Image, 'fabrik')
open_window(0, 0, -1, -1, 'root', 'visible', '', WindowHandle)
regiongrowing(Image, Seg, 5, 5, 6, 100)
elliptic_axis(Seg, Ra, Rb, Phi)
area_center(Seg, _, Row, Column)
gen_ellipse(Ellipses, Row, Column, Phi, Ra, Rb)
set_draw(WindowHandle, 'margin')
disp_region(Ellipses, WindowHandle)
```

Complexity
If F is the area of a region the mean runtime complexity is $O(\sqrt{F})$.

Result
The operator [elliptic_axis](#) returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator [set_system\('no_object_result', <Result>\)](#). The behavior in case of empty region (the region is the empty set) is set via [set_system\('empty_region_result', <Result>\)](#). If necessary an exception handling is raised.

Parallelization Information
[elliptic_axis](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors
[threshold](#), [regiongrowing](#), [connection](#)

Possible Successors
[gen_ellipse](#)

Alternatives
[smallest_rectangle2](#), [orientation_region](#)

See also
[moments_region_2nd](#), [select_shape](#), [set_shape](#)

References
R. Haralick, L. Shapiro “Computer and Robot Vision” Addison-Wesley, 1992, pp. 73-75

Module
Region processing

euler_number (Regions : : : EulerNumber)

Calculate the Euler number.

The procedure [euler_number](#) calculates the Euler number, i.e., the difference between the number of connection components and the number of holes.

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

Parameter	
▷ Regions (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Region(s) to be examined.
▷ EulerNumber (output_control)	integer(-array) $\leadsto$ <i>integer</i> Calculated Euler number.
Result	
The operator <code>euler_number</code> returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator <code>set_system('no_object_result', <Result>)</code> . The behavior in case of empty region (the region is the empty set) is set via <code>set_system('empty_region_result', <Result>)</code> . If necessary an exception handling is raised.	
Parallelization Information	
<code>euler_number</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).	
Possible Predecessors	
<code>threshold</code> , <code>regiongrowing</code> , <code>connection</code>	
Alternatives	
<code>connect_and_holes</code>	
Module	
Region processing	

find_neighbors (Regions1, Regions2 : : MaxDistance : RegionIndex1, RegionIndex2)

Search direct neighbors.

The operator `find_neighbors` determines neighboring regions with `Regions1` and `Regions2` containing the regions to be examined. `Regions1` can have three different states:

- `Regions1` is empty:
In this case all regions in `Regions2` are permutatively checked for neighborhood.
- `Regions1` consists of one region:
The regions of `Regions1` are compared to all regions in `Regions2`.
- `Regions1` consists of the same number of regions as `Regions2`:
Here all regions at the n-th position in `Regions1` and `Regions2` are checked for the neighboring relation.

The operator `find_neighbors` uses the city block distance between neighboring regions. It can be specified by the parameter `MaxDistance`. neighboring regions are located at the n-th position in `RegionIndex1` and `RegionIndex2`, i.e., the region with index `RegionIndex1[n]` from `Regions1` is the neighbor of the region with index `RegionIndex2[n]` from `Regions2`.

Attention	
Covered regions are not found!	
Parameter	
▷ Regions1 (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Starting regions.
▷ Regions2 (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Comparative regions.
▷ MaxDistance (input_control)	integer $\leadsto$ <i>integer</i> Maximal distance of regions. Default Value : 1 Suggested values : <code>MaxDistance</code> $\in$ {1, 2, 3, 4, 5, 6, 7, 8, 10, 15, 20, 50} Typical range of values : $1 \leq \text{MaxDistance} \leq 255$ Minimum Increment : 1 Recommended Increment : 1

- ▷ **RegionIndex1** (output_control) integer-array $\leadsto$ integer
Indices of the found regions from [Regions1](#).
- ▷ **RegionIndex2** (output_control) integer-array $\leadsto$ integer
Indices of the found regions from [Regions2](#).

Result

The operator [find_neighbors](#) returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator [set_system](#) ('no_object_result', <Result>). The behavior in case of empty region (the region is the empty set) is set via [set_system](#) ('empty_region_result', <Result>). If necessary an exception handling is raised.

Parallelization Information

[find_neighbors](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[threshold](#), [regiongrowing](#), [connection](#)

See also

[spatial_relation](#), [select_region_spatial](#), [expand_region](#), [interjacent](#), [boundary](#)

Module

Region processing

get_region_index (Regions : : Row, Column : Index)

Index of all regions containing a given pixel.

The operator [get_region_index](#) returns the index of all regions in [Regions](#) (range of values: 1 to n) containing the test pixel ([Row](#),[Column](#)), i.e.:

$$|\text{Regions}[n] \cap \{(\text{Row}, \text{Column})\}| \neq \emptyset$$

The returned indices can be used, e.g., in [select_obj](#) to select the regions containing the test pixel.

Attention

If the regions overlap more than one region might contain the pixel. In this case all these regions are returned. If no region contains the indicated pixel the empty tuple (= no region) is returned.

Parameter

- ▷ **Regions** (input_object) region-array $\leadsto$ *Hobject*
Regions to be examined.
- ▷ **Row** (input_control) point.y $\leadsto$ integer
Line index of the test pixel.
Default Value : 100
Typical range of values : $-\infty \leq \text{Row} \leq \infty$ (lin)
- ▷ **Column** (input_control) point.x $\leadsto$ integer
Column index of the test pixel.
Default Value : 100
Typical range of values : $-\infty \leq \text{Column} \leq \infty$ (lin)
- ▷ **Index** (output_control) integer(-array) $\leadsto$ integer
Index of the regions containing the test pixel.

Complexity

If F is the area of the region and N is the number of regions the mean runtime complexity is $O(\ln(\sqrt{F}) * N)$.

Result

The operator [get_region_index](#) returns the value 2 (H_MSG_TRUE) if the parameters are correct. The behavior in case of empty input (no input regions available) is set via the operator [set_system](#) ('no_object_result', <Result>). If necessary an exception handling is raised.

Parallelization Information

[get_region_index](#) is *reentrant* and automatically *parallelized* (on tuple level).

<i>Possible Predecessors</i>
threshold , regiongrowing , connection
<i>Alternatives</i>
select_region_point
<i>See also</i>
get_mbutton , get_mposition , test_region_point
<i>Module</i>
Region processing

get_region_thickness (Region : : : Thickness, Histogramm)

Access the thickness of a region along the main axis.

The operator [get_region_thickness](#) calculates the thickness of the regions along the main axis (see [elliptic_axis](#)) for each pixel of the section. The thickness at one point on the main axis is defined as the distance between the intersections of the contour with the plumb on the main axis in the respective point which are the furthest apart. Additionally the operator [get_region_thickness](#) returns the [Histogramm](#) of the thicknesses of the region. The length of the histogram corresponds to the largest occurring thickness in the observed region.

Attention
Only one region may be passed. If the region has several connection components, only the first one is investigated. All other components are ignored.

<i>Parameter</i>
▷ Region (input_object) region $\leadsto$ <i>Hobject</i> Region to be analysed. ▷ Thickness (output_control) integer-array $\leadsto$ <i>integer</i> Thickness of the region along its main axis. ▷ Histogramm (output_control) integer-array $\leadsto$ <i>integer</i> Histogram of the thickness of the region along its main axis.

Result
The operator [get_region_thickness](#) returns the value 2 (H_MSG_TRUE) if exactly one region is passed. The behavior in case of empty input (no input regions available) is set via the operator [set_system](#) ('no_object_result', <Result>).

Parallelization Information
[get_region_thickness](#) is *reentrant* and processed *without* parallelization.

<i>Possible Predecessors</i>
sobel_amp , threshold , connection , select_shape , select_obj
<i>See also</i>
copy_obj , elliptic_axis
<i>Module</i>
Region processing

hamming_distance (Regions1, Regions2 : : : Distance, Similarity)

Hamming distance between two regions.

The operator [hamming_distance](#) returns the hamming distance between two regions, i.e., the number of pixels of the regions which are different (**Distance**), i.e., the number of pixels contained in one region but not in the other:

$$\text{Distance} = |\text{Regions1} \cap \overline{\text{Regions2}}| + |\text{Regions2} \cap \overline{\text{Regions1}}|$$

The parameter **Similarity** describes the similarity between the two regions based on the hamming distance **Distance**:

$$\text{Similarity} = 1 - \frac{\text{Distance}}{|\text{Regions1}| + |\text{Regions2}|}$$

If both regions are empty **Similarity** is set to 0. The regions with the same index from both input parameters are always compared.

Attention

In both input parameters the same number of regions must be passed.

Parameter

- ▷ **Regions1** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be examined.
- ▷ **Regions2** (input_object) region(-array) $\leadsto$ *Hobject*
Comparative regions.
- ▷ **Distance** (output_control) integer(-array) $\leadsto$ *integer*
Hamming distance of two regions.
Assertion : $\text{Distance} \geq 0$
- ▷ **Similarity** (output_control) real(-array) $\leadsto$ *real*
Similarity of two regions.
Assertion : $(0 \leq \text{Similarity}) \wedge (\text{Similarity} \leq 1)$

Complexity

If F is the area of a region the mean runtime complexity is $O(\sqrt{F})$.

Result

hamming_distance returns the value 2 (H_MSG_TRUE) if the number of objects in both parameters is the same and is not 0. The behavior in case of empty input (no input objects available) is set via the operator **set_system('no_object_result', <Result>)**. The behavior in case of empty region (the region is the empty set) is set via **set_system('empty_region_result', <Result>)**. If necessary an exception handling handling is raised.

Parallelization Information

hamming_distance is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

threshold, **regiongrowing**, **connection**

Alternatives

intersection, **complement**, **area_center**

See also

hamming_change_region

Module

Region processing

hamming_distance_norm (Regions1, Regions2 : : Norm : Distance, Similarity)

Hamming distance between two regions using normalization.

The operator **hamming_distance_norm** returns the hamming distance between two regions, i.e., the number of pixels of the regions which are different (**Distance**). Before calculating the difference the region in **Regions1** is normalized onto the regions in **Regions2**. The result is the number of pixels contained in one region but not in the other:

$$\text{Distance} = |\text{Norm}(\text{Regions1}) \cap \overline{\text{Regions2}}| + |\text{Regions2} \cap \overline{\text{Norm}(\text{Regions1})}|$$

The parameter **Similarity** describes the similarity between the two regions based on the hamming distance **Distance**:

$$\text{Similarity} = 1 - \frac{\text{Distance}}{|Norm(\text{Regions1})| + |\text{Regions2}|}$$

The following types of normalization are available:

'center': The region is moved so that both regions have the same center of gravity.

If both regions are empty **Similarity** is set to 0. The regions with the same index from both input parameters are always compared.

Attention

In both input parameters the same number of regions must be passed.

Parameter

- ▷ **Regions1** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be examined.
- ▷ **Regions2** (input_object) region(-array) $\leadsto$ *Hobject*
Comparative regions.
- ▷ **Norm** (input_control) string(-array) $\leadsto$ *string*
Type of normalization.
Default Value : 'center'
List of values : Norm $\in$ {'center'}
- ▷ **Distance** (output_control) integer(-array) $\leadsto$ *integer*
Hamming distance of two regions.
Assertion : Distance ≥ 0
- ▷ **Similarity** (output_control) real(-array) $\leadsto$ *real*
Similarity of two regions.
Assertion : $(0 \leq \text{Similarity}) \wedge (\text{Similarity} \leq 1)$

Complexity

If F is the area of a region the mean runtime complexity is $O(\sqrt{F})$.

Result

hamming_distance_norm returns the value 2 (H_MSG_TRUE) if the number of objects in both parameters is the same and is not 0. The behavior in case of empty input (no input objects available) is set via the operator `set_system('no_object_result', <Result>)`. The behavior in case of empty region (the region is the empty set) is set via `set_system('empty_region_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

hamming_distance_norm is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

threshold, **regiongrowing**, **connection**

Alternatives

intersection, **complement**, **area.center**

See also

hamming_change_region

Module

Region processing

inner_circle (Regions : : : Row, Column, Radius)

Largest inner circle of a region.

The operator **inner_circle** determines the largest inner circle of a region. This is the biggest discrete circle region that completely fits into the region. For this circle the center (**Row**, **Column**) and the radius (**Radius**) are calculated. If the position of the circle is ambiguous, the "first possible" position (as far upper left as possible) is returned.

The output of the procedure is chosen in such a way that it can be used as an input for the HALCON procedures [disp_circle](#), [gen_circle](#), and [gen_ellipse_contour_xld](#).

If several regions are passed in [Regions](#) corresponding tuples are returned as output parameters. In case of an empty input region all parameters have the value 0.0 if no other behavior was set with [set_system](#).

Attention

If several inner circles are present at a region only the most upper left solution is returned.

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be examined.
 - ▷ **Row** (output_control) circle.center.y(-array) $\leadsto$ *real*
Line index of the center.
 - ▷ **Column** (output_control) circle.center.x(-array) $\leadsto$ *real*
Column index of the center.
 - ▷ **Radius** (output_control) circle.radius(-array) $\leadsto$ *real*
Radius of the inner circle.
- Assertion :** Radius ≥ 0

Example

```
read_image(Image, 'fabrik')
open_window(0, 0, -1, -1, 'root', 'visible', '', WindowHandle)
regiongrowing(Image, Seg, 5, 5, 6, 100)
select_shape(Seg, H, 'area', 'and', 100, 2000)
inner_circle(H, Row, Column, Radius)
gen_circle(Circles, Row, Column, Radius:)
set_draw(WindowHandle, 'margin')
disp_region(Circles, WindowHandle)
```

Complexity

If F is the area of the region and R is the radius of the inner circle the runtime complexity is $O(\sqrt{F} * R)$.

Result

The operator [inner_circle](#) returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator [set_system](#) ('no_object_result', <Result>), the behavior in case of empty region is set via [set_system](#) ('empty_region_result', <Result>). If necessary an exception handling is raised.

Parallelization Information

[inner_circle](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[threshold](#), [regiongrowing](#), [connection](#), [runlength_features](#)

Possible Successors

[gen_circle](#), [disp_circle](#)

Alternatives

[erosion_circle](#)

See also

[set_shape](#), [select_shape](#), [smallest_circle](#)

Module

Region processing

moments_region_2nd (Regions : : : M11, M20, M02, Ia, Ib)

Geometric moments of regions.

The operator [moments_region_2nd](#) calculates the moments ([M20](#), [M02](#)) and the product of inertia of the axes through the center parallel to the coordinate axes ([M11](#)). Furthermore the main axes of inertia ([Ia](#), [Ib](#)) are calculated.

Calculation: Z_0 and S_0 are the coordinates of the center of a region R with the area F . Then the moments M_{ij} are defined by:

$$M_{ij} = \sum_{(Z,S) \in R} (Z_0 - Z)^i (S_0 - S)^j$$

wherein Z and S run through all pixels of the region R .

Furthermore,

$$h = \frac{M20 + M02}{2}$$

then **Ia** and **Ib** are defined by:

$$\mathbf{Ia} = h + \sqrt{h^2 - M20 * M02 + M11^2}$$

$$\mathbf{Ib} = h - \sqrt{h^2 - M20 * M02 + M11^2}$$

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see [set_system](#)).

Parameter	
▷ Regions (input_object)	region(-array) $\leadsto$ Hobject Regions to be examined.
▷ M11 (output_control)	real(-array) $\leadsto$ real Product of inertia of the axes through the center parallel to the coordinate axes.
▷ M20 (output_control)	real(-array) $\leadsto$ real Moment of 2nd order (line-dependent).
▷ M02 (output_control)	real(-array) $\leadsto$ real Moment of 2nd order (column-dependent).
▷ Ia (output_control)	real(-array) $\leadsto$ real The one main axis of inertia.
▷ Ib (output_control)	real(-array) $\leadsto$ real The other main axis of inertia.

Complexity

If F is the area of the region the mean runtime complexity is $O(\sqrt{F})$.

Result

The operator [moments_region_2nd](#) returns the value 2 (H_MSG.TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator [set_system](#) ('no_object_result', <Result>). The behavior in case of empty region (region is the empty set) is set via [set_system](#) ('empty_region_result', <Result>). If necessary an exception handling is raised.

Parallelization Information

[moments_region_2nd](#) is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

[threshold](#), [regiongrowing](#), [connection](#)

Alternatives

[moments_region_2nd_invar](#)

See also

[elliptic_axis](#)

Module

Region processing

moments_region_2nd_invar (Regions : : : M11, M20, M02)

Geometric moments of regions.

The operator [moments_region_2nd_invar](#) calculates the scaled moments (**M20**, **M02**) and the procut of inertia of the axes through the center parallel to the coordinate axes (**M11**).

Calculation: Z_0 and S_0 are the coordinates of the center of a region R with the area F . Then the moments M_{ij} are defined by:

$$M_{ij} = \frac{1}{F^2} \sum_{(Z,S) \in R} (Z_0 - Z)^i (S_0 - S)^j$$

wherein Z and S run through all pixels of the region R .

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see [set_system](#)).

Parameter	
▷ Regions (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Regions to be examined.
▷ M11 (output_control)	real(-array) $\leadsto$ <i>real</i> Product of inertia of the axes through the center parallel to the coordinate axes.
▷ M20 (output_control)	real(-array) $\leadsto$ <i>real</i> Moment of 2nd order (line-dependent).
▷ M02 (output_control)	real(-array) $\leadsto$ <i>real</i> Moment of 2nd order (column-dependent).

Complexity
If F is the area of the region the mean runtime complexity is $O(\sqrt{F})$.

Result
The operator moments_region_2nd_invar returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system('no_object_result', <Result>) . The behavior in case of empty region (the region is the empty set) is set via set_system('empty_region_result', <Result>) . If necessary an exception handling is raised.

Parallelization Information
moments_region_2nd_invar is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).

Possible Predecessors
threshold , regiongrowing , connection

Alternatives
moments_region_2nd

See also
elliptic_axis

Module
Region processing

moments_region_2nd_rel_invar (Regions : : : PHI1, PHI2)

Geometric moments of regions.

The operator [moments_region_2nd_rel_invar](#) calculates the scaled relative moments ([PHI1](#), [PHI2](#)).

Calculation: The moments PHI_1 and PHI_2 are defined by:

$$PHI_1 = V_{20} + V_{02}$$

$$PHI_2 = (V_{20} + V_{02})^2 + V_{11}^2$$

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see [set_system](#)).

Parameter
▷ Regions (input_object) region(-array) $\leadsto$ <i>Hobject</i> Regions to be examined. ▷ PHI1 (output_control) real(-array) $\leadsto$ <i>real</i> Moment of 2nd order. ▷ PHI2 (output_control) real(-array) $\leadsto$ <i>real</i> Moment of 2nd order.
Result
The operator <code>moments_region_2nd_rel_invar</code> returns the value 2 (H_MSG.TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator <code>set_system('no_object_result', <Result>)</code> . The behavior in case of empty region (the region is the empty set) is set via <code>set_system('empty_region_result', <Result>)</code> . If necessary an exception handling is raised.
Parallelization Information
<code>moments_region_2nd_rel_invar</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).
Possible Predecessors
<code>threshold</code> , <code>regiongrowing</code> , <code>connection</code>
Alternatives
<code>moments_region_2nd</code>
See also
<code>elliptic_axis</code>
Module
Region processing

moments_region_3rd (Regions : : : M21, M12, M03, M30)

Geometric moments of regions.

The operator `moments_region_3rd` calculates the translation-invariant central moments (**M21**, **M12**, **M03**, **M30**) of order $(p + q)$.

Calculation: x and y are the coordinates of the center of a region R with the area Z . Then the moments M_{pq} are defined by:

$$M_{pq} = \sum_{i=1} MZ(x_i, y_i)(x_i - x)^p(y_i - y)^q$$

wherein are $x = \frac{m_{10}}{m_{00}}$ and $y = \frac{m_{01}}{m_{00}}$.

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see `set_system`).

Parameter
▷ Regions (input_object) region(-array) $\leadsto$ <i>Hobject</i> Regions to be examined. ▷ M21 (output_control) real(-array) $\leadsto$ <i>real</i> oment of 3nd order (line-dependent). ▷ M12 (output_control) real(-array) $\leadsto$ <i>real</i> Moment of 3nd order (column-dependent). ▷ M03 (output_control) real(-array) $\leadsto$ <i>real</i> Moment of 3nd order (column-dependent). ▷ M30 (output_control) real(-array) $\leadsto$ <i>real</i> Moment of 3nd order (line-dependent).

<i>Complexity</i>
If Z is the area of the region the mean runtime complexity is $O(\sqrt{Z})$.
<i>Result</i>
The operator <code>moments_region_3rd</code> returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator <code>set_system('no_object_result', <Result>)</code> . The behavior in case of empty region (the region is the empty set) is set via <code>set_system('empty_region_result', <Result>)</code> . If necessary an exception handling is raised.
<i>Parallelization Information</i>
<code>moments_region_3rd</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>).
<i>Possible Predecessors</i>
<code>threshold</code> , <code>regiongrowing</code> , <code>connection</code>
<i>Alternatives</i>
<code>moments_region_2nd</code>
<i>See also</i>
<code>elliptic_axis</code>
<i>Module</i>
Region processing

moments_region_3rd_invar (Regions : : : M21, M12, M03, M30)

Geometric moments of regions.

The operator `moments_region_3rd_invar` calculates the scale-invariant moments (**M21**, **M12**, **M03**, **M30**).

Calculation: Then the moments M_{ij} are defined by:

$$M_{pq} = \frac{\mu_{pq}}{\mu^3}$$

wobei $p + q \geq 2$ und $\mu = \mu_{00} = m_{00}$ sind.
 wherein are $p + q \geq 2$ and $\mu = \mu_{00} = m_{00}$.

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see `set_system`).

<i>Parameter</i>
▷ Regions (input_object) region(-array) $\leadsto$ <i>Hobject</i> Regions to be examined. ▷ M21 (output_control) real(-array) $\leadsto$ <i>real</i> Moment of 3nd order (line-dependent). ▷ M12 (output_control) real(-array) $\leadsto$ <i>real</i> Moment of 3nd order (column-dependent). ▷ M03 (output_control) real(-array) $\leadsto$ <i>real</i> Moment of 3nd order (column-dependent). ▷ M30 (output_control) real(-array) $\leadsto$ <i>real</i> Moment of 3nd order (line-dependent).

<i>Complexity</i>
If Z is the area of the region the mean runtime complexity is $O(\sqrt{Z})$.
<i>Result</i>
The operator <code>moments_region_3rd_invar</code> returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator <code>set_system('no_object_result', <Result>)</code> . The behavior in case of empty region (the region is the empty set) is set via <code>set_system('empty_region_result', <Result>)</code> . If necessary an exception handling is raised.

	Parallelization Information	
<code>moments_region_3rd_invar</code>	is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).	
	Possible Predecessors	
<code>threshold</code> , <code>regiongrowing</code> , <code>connection</code>		
	Alternatives	
<code>moments_region_2nd</code>		
	See also	
<code>elliptic_axis</code>		
	Module	
Region processing		

moments_region_central (Regions : : : I1, I2, I3, I4)

Geometric moments of regions.

The operator `moments_region_central` calculates the central moments (**I1**, **I2**, **I3**, **I4**).

Calculation: Then the moments I_i are defined by: $I_1 = \mu_{20}\mu_{02} - \mu_{11}^2$

$$I_2 = (\mu_{30}\mu_{03} - \mu_{21}\mu_{12})^2 - 4(\mu_{30}\mu_{12} - \mu_{21}^2)(\mu_{21}\mu_{03} - \mu_{12}^2)$$

$$I_3 = \mu_{20}(\mu_{21}\mu_{03} - \mu_{12}^2) - \mu_{11}(\mu_{30}\mu_{03} - \mu_{21}\mu_{12}) + \mu_{02}(\mu_{30}\mu_{12} - \mu_{21}^2)$$

$$I_4 = \mu_{30}^2\mu_{02}^3 - 6\mu_{30}\mu_{21}\mu_{11}\mu_{02}^2 + 6\mu_{30}\mu_{12}\mu_{02}(2\mu_{11}^2 - \mu_{20}\mu_{02}) \\ + \mu_{30}\mu_{03}(6\mu_{20}\mu_{11}\mu_{02} - 8\mu_{11}^3) + 9\mu_{21}^2\mu_{20}\mu_{02}^2 - 18\mu_{21}\mu_{12}\mu_{20}\mu_{11}\mu_{02} \\ + 6\mu_{21}\mu_{03}\mu_{20}(2\mu_{11}^2 - \mu_{20}\mu_{02}) + 9\mu_{12}^2\mu_{20}^2\mu_{02} - 6\mu_{12}\mu_{03}\mu_{11}\mu_{20}^2 + \mu_{03}^2\mu_{20}^3$$

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see `set_system`).

	Parameter	
▷ Regions (input_object)	region(-array) $\leadsto$ <i>Hobject</i>	
Regions to be examined.		
▷ I1 (output_control)	real(-array) $\leadsto$ <i>real</i>	
Moment of 2nd order.		
▷ I2 (output_control)	real(-array) $\leadsto$ <i>real</i>	
Moment of 2nd order.		
▷ I3 (output_control)	real(-array) $\leadsto$ <i>real</i>	
Moment of 2nd order.		
▷ I4 (output_control)	real(-array) $\leadsto$ <i>real</i>	
Moment of 3rd order.		

	Complexity	
If Z is the area of the region the mean runtime complexity is $O(\sqrt{Z})$.		

	Result	
The operator <code>moments_region_central</code> returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator <code>set_system('no_object_result', <Result>)</code> . The behavior in case of empty region (the region is the empty set) is set via <code>set_system('empty_region_result', <Result>)</code> . If necessary an exception handling is raised.		

	Parallelization Information	
<code>moments_region_central</code>	is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).	
	Possible Predecessors	
<code>threshold</code> , <code>regiongrowing</code> , <code>connection</code>		

	Alternatives
moments_region_2nd	
	See also
elliptic_axis	
	Module
Region processing	

moments_region_central_invar (Regions : : : PSI1, PSI2, PSI3, PSI4)

Geometric moments of regions.

The operator [moments_region_central_invar](#) calculates the moments ([PSI1](#), [PSI2](#), [PSI3](#), [PSI4](#)) that are invariant under translation and general linear transformations.

Calculation: Then the moments ψ_i are defined by: $\psi_1 = \frac{I_1}{\mu^4}$

$$\psi_2 = \frac{I_2}{\mu^5}$$

$$\psi_3 = \frac{I_3}{\mu^7}$$

$$\psi_4 = \frac{I_4}{\mu^9}$$

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see [set_system](#)).

	Parameter
▷ Regions (input_object)	region(-array) $\leadsto$ Hobject Regions to be examined.
▷ PSI1 (output_control)	real(-array) $\leadsto$ real Moment of 2nd order.
▷ PSI2 (output_control)	real(-array) $\leadsto$ real Moment of 2nd order.
▷ PSI3 (output_control)	real(-array) $\leadsto$ real Moment of 2nd order.
▷ PSI4 (output_control)	real(-array) $\leadsto$ real Moment of 2nd order.

	Complexity
If Z is the area of the region the mean runtime complexity is $O(\sqrt{Z})$.	

	Result
The operator moments_region_central_invar returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object_result', <Result>). The behavior in case of empty region (the region is the empty set) is set via set_system ('empty_region_result', <Result>). If necessary an exception handling is raised.	

	Parallelization Information
moments_region_central_invar is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).	

	Possible Predecessors
threshold , regiongrowing , connection	

	Alternatives
moments_region_2nd	
	See also
elliptic_axis	

Module

Region processing

orientation_region (Regions : : : Phi)

Orientation of a region.

The operator `orientation_region` calculates the orientation of the region. The operator is based on `elliptic_axis`. In addition the point on the contour with maximal distance to the center of gravity is calculated. If the column coordinate of this point is less than the column coordinate of the center of gravity the value of π is added to the angle.

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see `set_system('no_object_result', <Result>)`).

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ *Hobject*
Region(s) to be examined.
 - ▷ **Phi** (output_control) real(-array) $\leadsto$ *real*
Orientation of region (arc measure).
- Assertion :** $((-\pi/2) < \text{Phi}) \wedge (\text{Phi} \leq ((3 \cdot \pi)/2))$

Complexity

If F is the area of a region the mean runtime complexity is $O(\sqrt{F})$.

Result

The operator `orientation_region` returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result', <Result>)`. The behavior in case of empty region (the region is the empty set) is set via `set_system('empty_region_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`orientation_region` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`

Possible Successors

`disp_arrow`

Alternatives

`elliptic_axis`, `smallest_rectangle2`

See also

`moments_region_2nd`, `line_orientation`

Module

Region processing

roundness (Regions : : : Distance, Sigma, Roundness, Sides)

Shape factors from contour.

The operator `roundness` examines the distance between the contour and the center of the area. In particular the mean distance (**Distance**), the deviation from the mean distance (**Sigma**) and two shape features derived therefrom are determined. **Roundness** is the relation between mean value and standard deviation, and **Sides** indicates the number of polygon pieces if a regular polygon is concerned.

The contour for calculating the features is determined depending on the global neighborhood (see `set_system`).

Calculation:

If p is the center of the area, p_i the pixels and F the area of the contour.

$$\begin{aligned}\text{Distance} &= \frac{1}{F} \sum ||p - p_i|| \\ \text{Sigma}^2 &= \frac{1}{F} \sum (||p - p_i|| - \text{Distance})^2 \\ \text{Roundness} &= 1 - \frac{\text{Sigma}}{\text{Distance}} \\ \text{Sides} &= 1.4111 \left(\frac{\text{Distance}}{\text{Sigma}} \right)^{0.4724}\end{aligned}$$

If more than one region is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input.

In case of empty region all parameters have the value 0.0 if no other behavior was set (see [set_system](#)).

Parameter	
▷ Regions (input_object)	region(-array) $\leadsto$ Hobject Region(s) to be examined.
▷ Distance (output_control)	real(-array) $\leadsto$ real Mean distance from the center. Assertion : Distance ≥ 0.0
▷ Sigma (output_control)	real(-array) $\leadsto$ real Standard deviation of Distance . Assertion : Sigma ≥ 0.0
▷ Roundness (output_control)	real(-array) $\leadsto$ real Shape factor for roundness. Assertion : Roundness ≤ 1.0
▷ Sides (output_control)	real(-array) $\leadsto$ real Number of polygon sides. Assertion : Sides ≥ 0

Complexity
If F is the area of a region the mean runtime complexity is $O(\sqrt{F})$.

Result
The operator roundness returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator set_system ('no_object.result', <Result>), the behavior in case of empty region is set via set_system ('empty_region.result', <Result>). If necessary an exception handling is raised.

Parallelization Information
roundness is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).

Possible Predecessors
threshold , regiongrowing , connection

Alternatives
compactness

See also
contlength

References
R. Haralick, L. Shapiro "Computer and Robot Vision" Addison-Wesley, 1992, pp. 61

Module
Region processing

runlengthdistribution (Region : : : Foreground, Background)

Distribution of runs needed for runlength encoding of a region.

The operator `runlength_distribution` calculates the distribution of the runs of a region of the fore- and background. The frequency of the occurrence of a certain length is calculated. Runs of infinite length are not counted. Therefore the background are the holes of the region. As many values are passed as set by the maximum length of fore- or background, respectively. The length of both tuples usually differs. The first entry of the tuples is always 0 (no runs of the length 0). If there are no blanks the empty tuple is passed at `Background`. Analogously the empty tuple is passed in case of an empty region at `Foreground`.

<i>Parameter</i>	
▷ Region (input_object)	region $\leadsto$ <i>Hobject</i> Region to be examined.
▷ Foreground (output_control)	integer-array $\leadsto$ <i>integer</i> Length distribution of the region (foreground).
▷ Background (output_control)	integer-array $\leadsto$ <i>integer</i> Length distribution of the background.

Complexity

If n is the number of runs of the region the runtime complexity is $O(n)$.

Result

The operator `runlength_distribution` returns the value 2 (`H_MSG_TRUE`) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result', <Result>)`. If more than one region is passed an exception handling is raised.

Parallelization Information

`runlength_distribution` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`threshold`, `select_obj`

Alternatives

`runlength_features`

See also

`runlength_features`

Module

Region processing

runlength_features (Regions : : : NumRuns, KFactor, LFactor, MeanLength, Bytes)

Characteristic values for runlength coding of regions.

The operator `runlength_features` calculates for every input region from `Regions` the number of runs necessary for storing this region with the aid of runlength coding. Furthermore the so-called "K-factor" is determined, which indicates by how much the number of runs differs from the ideal of the square in which this value is 1.0.

The K-factor (`KFactor`) is calculated according to the formula:

$$\text{KFactor} = \frac{\text{NumRuns}}{\sqrt{\text{Area}}}$$

wherein *Area* indicates the area of the region. It should be noted that the K-factor can be smaller than 1.0 (in case of long horizontal regions).

The L-factor (`LFactor`) indicates the mean number of runs for each line index occurring in the region.

`MeanLength` indicates the mean length of the runs. The parameter `Bytes` indicates how many bytes are necessary for coding the region with runlengths.

Attention

All features calculated by the operator `runlength_features` are not rotation invariant because the runlength coding depends on the direction. The operator `runlength_features` does not serve for calculating shape features but for controlling and analysing the efficiency of the runlength coding.

Parameter
▷ Regions (input_object) region(-array) $\leadsto$ <i>Hobject</i> Regions to be examined. ▷ NumRuns (output_control) integer(-array) $\leadsto$ <i>integer</i> Number of runs. Assertion : $0 \leq \text{NumRuns}$ ▷ KFactor (output_control) real(-array) $\leadsto$ <i>real</i> Storing factor in relation to a square. Assertion : $0 \leq \text{KFactor}$ ▷ LFactor (output_control) real(-array) $\leadsto$ <i>real</i> Mean number of runs per line. Assertion : $0 \leq \text{LFactor}$ ▷ MeanLength (output_control) real(-array) $\leadsto$ <i>real</i> Mean length of runs. Assertion : $0 \leq \text{MeanLength}$ ▷ Bytes (output_control) integer(-array) $\leadsto$ <i>integer</i> Number of bytes necessary for coding the region. Assertion : $0 \leq \text{Bytes}$
Complexity
The mean runtime complexity is $O(1)$.
Result
The operator <code>runlength_features</code> returns the value 2 (H_MSG_TRUE) if the input is not empty. If necessary an exception handling is raised.
Parallelization Information
<code>runlength_features</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).
Possible Predecessors
<code>threshold</code> , <code>regiongrowing</code> , <code>connection</code> , <code>runlength_features</code>
See also
<code>runlength_features</code> , <code>runlength_distribution</code>
Module
Region processing

select_region_point (Regions : DestRegions : Row, Column :)

Choose all regions containing a given pixel.

The operator `select_region_point` selects all regions from `Regions` containing the test pixel (`Row,Column`), i.e.:

$$|\text{Regions}[n] \cap \{(\text{Row}, \text{Column})\}| = 1$$

Attention

If the regions overlap more than one region might contain the pixel. In this case all these regions are returned. If no region contains the indicated pixel the empty tuple (= no region) is returned.

Parameter
▷ Regions (input_object) region-array $\leadsto$ <i>Hobject</i> Regions to be examined. ▷ DestRegions (output_object) region-array $\leadsto$ <i>Hobject</i> All regions containing the test pixel. ▷ Row (input_control) point.y $\leadsto$ <i>integer</i> Line index of the test pixel. Default Value : 100 Typical range of values : $-\infty \leq \text{Row} \leq \infty$ (lin)

▷ **Column** (input_control) point.x $\leadsto$ integer
 Column index of the test pixel.
Default Value : 100
Typical range of values : $-\infty \leq \text{Column} \leq \infty$ (lin)

Example

```
read_image(Image, 'fabrik')
open_window(0, 0, -1, -1, 'root', 'visible', '', WindowHandle)
disp_image(Image)
regiongrowing(Image, Seg, 3, 3, 5, 0)
set_color(WindowHandle, 'red')
set_draw(WindowHandle, 'margin')
Button := 1
while (Button = 1)
    fwrite_string(FileId, 'Select the region with the mouse (End right button)')
    fnew_line(FileId)
    get_mbutton(WindowHandle, Row, Column, Button)
    select_region_point(Seg, Single, Row, Column)
    disp_region(Single, WindowHandle)
endwhile
```

Complexity

If F is the area of the region and N is the number of regions, the mean runtime complexity is $O(\ln(\sqrt{F}) * N)$.

Result

The operator `select_region_point` returns the value 2 (H_MSG_TRUE) if the parameters are correct. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`select_region_point` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`

Alternatives

`test_region_point`

See also

`get_mbutton`, `get_mposition`

Module

Region processing

select_region_spatial (Regions1,
 Regions2 : : Direction : RegionIndex1, RegionIndex2)

Pose relation of regions.

The operator `select_region_spatial` chooses the regions from `Regions2` which are sufficient for the neighboring relation `Direction`. The regions to be examined have to be passed in `Regions1` or `Regions2`, respectively. `Regions1` can have three different states:

- `Regions1` is empty:
 In this case all regions in `Regions2` are permutatively checked for neighborhood.
- `Regions1` consists of one region:
 The regions of `Regions1` are compared to all regions in `Regions2`.
- `Regions1` consists of the same number of regions as `Regions2`:
 The regions at the n -th position in `Regions1` and `Regions2` are each checked for a neighboring relation.

Possible values for `Direction` are:

'left': [Regions2](#) is left of [Regions1](#)

'right': [Regions2](#) is right of [Regions1](#)

'above': [Regions2](#) is above [Regions1](#)

'below': [Regions2](#) is below [Regions1](#)

The operator [select_region_spatial](#) calculates the centers of the regions to be compared and decides according to the angle between the center straight lines and the x axis whether the direction relation is fulfilled. The relation is fulfilled within the area of -45 degree to +45 degree around the coordinate axes. Thus, the direction relation can be understood in such a way that the center of the second region must be located left (or right, above, below) of the center of the first region. The indices of the regions fulfilling the direction relation are located at the n-th position in [RegionIndex1](#) and [RegionIndex2](#), i.e., the region with the index [RegionIndex2\[n\]](#) has the indicated relation with the region with the index [RegionIndex1\[n\]](#). Access to regions via the index can be obtained via the operator [copy_obj](#).

Parameter	
▷ Regions1 (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Starting regions
▷ Regions2 (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Comparative regions
▷ Direction (input_control)	string $\leadsto$ <i>string</i> Desired neighboring relation. Default Value : 'left' List of values : $\text{Direction} \in \{\text{'left'}, \text{'right'}, \text{'above'}, \text{'below'}\}$
▷ RegionIndex1 (output_control)	integer-array $\leadsto$ <i>integer</i> Indices in the input tuples (Regions1 or Regions2), respectively.
▷ RegionIndex2 (output_control)	integer-array $\leadsto$ <i>integer</i> Indices in the input tuples (Regions1 or Regions2), respectively.

The operator [select_region_spatial](#) returns the value 2 (H_MSG_TRUE) if [Regions2](#) is not empty. The behavior in case of empty parameter [Regions2](#) (no input regions available) is set via the operator [set_system](#) ('no_object_result', <Result>). The behavior in case of empty region (the region is the empty set) is set via [set_system](#) ('empty_region_result', <Result>). If necessary an exception handling is raised.

[select_region_spatial](#) is *reentrant* and processed *without parallelization*.

Possible Predecessors
[threshold](#), [regiongrowing](#), [connection](#)

Alternatives
[area_center](#), [intersection](#)

See also
[spatial_relation](#), [find_neighbors](#), [copy_obj](#), [obj_to_integer](#)

Module
Region processing

select_shape (Regions : SelectedRegions : Features, Operation, Min, Max :)

Choose regions with the aid of shape features.

The operator [select_shape](#) chooses regions according to shape. For each input region from [Regions](#) the indicated features ([Features](#)) are calculated. If each ([Operation](#) = 'and') or at least one ([Operation](#) = 'or') of the calculated features is within the default limits ([Min,Max](#)) the region is adapted into the output (duplicated). Condition: $\text{Min}_i \leq \text{Feature}_i(\text{Object}) \leq \text{Max}_i$

Possible values for [Features](#):

'area': Area of the object

'row': Row index of the center
'column': Column index of the center
'width': Width of the region
'height': Height of the region
'row1': Row index of upper left corner
'column1': Column index of upper left corner
'row2': Row index of lower right corner
'column2': Column index of lower right corner
'circularity': Circularity (see [circularity](#))
'compactness': Compactness (see [compactness](#))
'contlength': Total length of contour (see operator [contlength](#))
'convexity': Convexity (see [convexity](#))
'ra': Main radius of the equivalent ellipse (see [elliptic.axis](#))
'rb': Secondary radius of the equivalent ellipse (see [elliptic.axis](#))
'phi': Orientation of the equivalent ellipse (see [elliptic.axis](#))
'anisometry': Anisometry (see [eccentricity](#))
'bulkiness': Bulkiness (see operator [eccentricity](#))
'struct.factor': Structur Factor (see operator [eccentricity](#))
'outer.radius': Radius of smallest surrounding circle (see [smallest.circle](#))
'inner.radius': Radius of largest inner circle (see [inner.circle](#))
'dist.mean': Mean distance from the region border to the center (see operator [roundness](#))
'dist.deviation': Deviation of the distance from the region border from the center (see operator [roundness](#))
'roundness': Roundness (see operator [roundness](#))
'num.sides': Number of polygon sides (see operator [roundness](#))
'connect.num': Number of connection components (see operator [connect.and.holes](#))
'holes.num': Number of holes (see operator [connect.and.holes](#))
'max.diameter': Maximum diameter of the region (see operator [diameter.region](#))
'orientation': Orientation of the region (see operator [orientation.region](#))
'euler.number': Euler number (see operator [euler.number](#))
'rect2.phi': Orientation of the smallest surrounding rectangle (see operator [smallest.rectangle2](#))
'rect2.len1': Half the length of the smallest surrounding rectangle (see operator [smallest.rectangle2](#))
'rect2.len2': Half the width of the smallest surrounding rectangle (see operator [smallest.rectangle2](#))
'moments_m11': Geometric moments of the region (see operator [moments.region.2nd](#))
'moments_m20': Geometric moments of the region (see operator [moments.region.2nd](#))
'moments_m02': Geometric moments of the region (see operator [moments.region.2nd](#))
'moments_ia': Geometric moments of the region (see operator [moments.region.2nd](#))
'moments_ib': Geometric moments of the region (see operator [moments.region.2nd](#))
'moments_m11.invar': Geometric moments of the region (see operator [moments.region.2nd.invar](#))
'moments_m20.invar': Geometric moments of the region (see operator [moments.region.2nd.invar](#))
'moments_m02.invar': Geometric moments of the region (see operator [moments.region.2nd.invar](#))
'moments_phi1': Geometric moments of the region (see operator [moments.region.2nd.rel.invar](#))
'moments_phi2': Geometric moments of the region (see operator [moments.region.2nd.rel.invar](#))
'moments_m21': Geometric moments of the region (see operator [moments.region.3rd](#))
'moments_m12': Geometric moments of the region (see operator [moments.region.3rd](#))
'moments_m03': Geometric moments of the region (see operator [moments.region.3rd](#))

'**moments_m30**': Geometric moments of the region (see operator [moments_region_3rd](#))
 '**moments_m21_invar**': Geometric moments of the region (see operator [moments_region_3rd_invar](#))
 '**moments_m12_invar**': Geometric moments of the region (see operator [moments_region_3rd_invar](#))
 '**moments_m03_invar**': Geometric moments of the region (see operator [moments_region_3rd_invar](#))
 '**moments_m30_invar**': Geometric moments of the region (see operator [moments_region_3rd_invar](#))
 '**moments_i1**': Geometric moments of the region (see operator [moments_region_central](#))
 '**moments_i2**': Geometric moments of the region (see operator [moments_region_central](#))
 '**moments_i3**': Geometric moments of the region (see operator [moments_region_central](#))
 '**moments_i4**': Geometric moments of the region (see operator [moments_region_central](#))
 '**moments_psi1**': Geometric moments of the region (see operator [moments_region_central_invar](#))
 '**moments_psi2**': Geometric moments of the region (see operator [moments_region_central_invar](#))
 '**moments_psi3**': Geometric moments of the region (see operator [moments_region_central_invar](#))
 '**moments_psi4**': Geometric moments of the region (see operator [moments_region_central_invar](#))

If only one feature ([Features](#)) is used the value of [Operation](#) is meaningless. Several features are processed in the sequence in which they are entered.

<i>Parameter</i>	
▷ Regions (input_object)	region-array $\leadsto$ <i>Hobject</i> Regions to be examined.
▷ SelectedRegions (output_object)	region-array $\leadsto$ <i>Hobject</i> Regions fulfilling the condition.
▷ Features (input_control)	string(-array) $\leadsto$ <i>string</i> Shape features to be checked. Default Value : 'area' List of values : $\text{Features} \in \{\text{'area', 'row', 'column', 'width', 'height', 'row1', 'column1', 'row2', 'column2', 'circularity', 'compactness', 'contlength', 'convexity', 'ra', 'rb', 'phi', 'anisometry', 'bulkiness', 'struct_factor', 'outer_radius', 'inner_radius', 'max_diameter', 'dist_mean', 'dist_deviation', 'roundness', 'num_sides', 'orientation', 'connect_num', 'holes_num', 'euler_number', 'rect2_phi', 'rect2_len1', 'rect2_len2', 'moments_m11', 'moments_m20', 'moments_m02', 'moments_ia', 'moments_ib', 'moments_m11_invar', 'moments_m20_invar', 'moments_m02_invar', 'moments_phi1', 'moments_phi2', 'moments_m21', 'moments_m12', 'moments_m03', 'moments_m30', 'moments_m21_invar', 'moments_m12_invar', 'moments_m03_invar', 'moments_m30_invar', 'moments_i1', 'moments_i2', 'moments_i3', 'moments_i4', 'moments_psi1', 'moments_psi2', 'moments_psi3', 'moments_psi4'}\}$
▷ Operation (input_control)	string $\leadsto$ <i>string</i> Linkage type of the individual features. Default Value : 'and' List of values : $\text{Operation} \in \{\text{'and', 'or'}\}$
▷ Min (input_control)	real(-array) $\leadsto$ <i>real / integer / string</i> Lower limits of the features or 'min'. Default Value : 150.0 Typical range of values : $0.0 \leq \text{Min} \leq 99999.0$ Minimum Increment : 0.001 Recommended Increment : 1.0
▷ Max (input_control)	real(-array) $\leadsto$ <i>real / integer / string</i> Upper limits of the features or 'max'. Default Value : 99999.0 Typical range of values : $0.0 \leq \text{Max} \leq 99999.0$ Minimum Increment : 0.001 Recommended Increment : 1.0 Restriction : $\text{Max} \geq \text{Min}$

Example

```

/* where are the eyes of the ape ? */
read_image(Image'affe')
threshold(Image,S1,160,255)
connection(S1,S2)
select_shape(S2,Eyes,['area','anisometry'],'and',[500,1.0],[50000,1.7])
disp_region(Eyes,WindowHandle)

```

Result

The operator `select_shape` returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input objects available) is set via the operator `set_system('no_object_result',<Result>)`. The behavior in case of empty region (the region is the empty set) is set via `set_system('empty_region_result',<Result>)`. If necessary an exception handling is raised.

Parallelization Information

`select_shape` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `runlength_features`

Possible Successors

`select_shape`, `select_gray`, `shape_trans`, `reduce_domain`, `count_obj`

See also

`area_center`, `circularity`, `compactness`, `contlength`, `convexity`, `elliptic_axis`, `eccentricity`, `inner_circle`, `smallest_circle`, `smallest_rectangle1`, `smallest_rectangle2`, `roundness`, `connect_and_holes`, `diameter_region`, `orientation_region`, `moments_region_2nd`, `moments_region_2nd_invar`, `moments_region_2nd_rel_invar`, `moments_region_3rd`, `moments_region_3rd_invar`, `moments_region_central`, `moments_region_central_invar`, `select_obj`

Module

Region processing

select_shape_proto (Regions, Pattern : SelectedRegions : Feature, Min, Max :)

Choose regions having a certain relation to each other.

The operator `select_shape_proto` selects regions based on certain relations between the regions. Every region from `Regions` is compared to the union of regions from `Pattern`. The limits (`Min` and `Max`) are specified absolutely or in percent (0..100), depending on the feature. Possible values for `Feature` are:

- '**distance_dilate**' The minimum distance in the maximum norm from the edge of `Pattern` to the edge of every region from `Regions` is determined (see `distance_rr_min_dil`).
- '**distance_contour**' The minimum Euclidean distance from the edge of `Pattern` to the edge of every region from `Regions` is determined. (see `distance_rr_min`).
- '**distance_center**' The Euclidean distance from the center of `Pattern` to the center of every region from `Regions` is determined.
- '**covers**' It is examined how well the region `Pattern` fits into the regions from `Regions`. If there is no shift so that `Pattern` is a subset of `Regions` the overlap is 0. If `Pattern` corresponds to the region after a corresponding shift the overlap is 100. Otherwise the area of the opening of `Regions` with `Pattern` is put into relation with the area of `Regions` (in percent).
- '**fits**' It is examined whether `Pattern` can be shifted in such a way that it fits in `Regions`. If this is possible the corresponding region is copied from `Regions`. The parameters `Min` and `Max` are ignored.
- '**overlaps_abs**' The area of the intersection of `Pattern` and every region in `Regions` is computed.
- '**overlaps_rel**' The area of the intersection of `Pattern` and every region in `Regions` is computed. The relative overlap is the ratio of the area of the intersection and the area of the respective region in `Regions` (in percent).

	Parameter
▷ Regions (input_object)	region(-array) $\leadsto$ Hobject Regions to be examined.
▷ Pattern (input_object)	region(-array) $\leadsto$ Hobject Region compared to Regions .
▷ SelectedRegions (output_object)	region(-array) $\leadsto$ Hobject Regions fulfilling the condition.
▷ Feature (input_control)	string(-array) $\leadsto$ string Shape features to be checked. Default Value : 'covers' List of values : Feature $\in$ {'distance_center', 'distance_dilate', 'distance_contour', 'covers', 'fits', 'overlaps_abs', 'overlaps_rel'}
▷ Min (input_control)	number $\leadsto$ real / integer Lower border of feature. Default Value : 50.0 Suggested values : Min $\in$ {0.0, 1.0, 5.0, 10.0, 20.0, 30.0, 50.0, 60.0, 70.0, 80.0, 90.0, 95.0, 99.0, 100.0, 200.0, 400.0} Typical range of values : $0.0 \leq \text{Min}$ Minimum Increment : 0.001 Recommended Increment : 5.0
▷ Max (input_control)	number $\leadsto$ real / integer Upper border of the feature. Default Value : 100.0 Suggested values : Max $\in$ {0.0, 10.0, 20.0, 30.0, 50.0, 60.0, 70.0, 80.0, 90.0, 95.0, 99.0, 100.0, 200.0, 300.0, 400.0} Typical range of values : $0.0 \leq \text{Max}$ Minimum Increment : 0.001 Recommended Increment : 5.0
	Example

```
#include <iostream.h>
#include "HalconCpp.h"

int main (int argc, char *argv[])
{
    if (argc < 2)
    {
        cout << "Usage: " << argv[0] << " <radius of circle>" << endl;
        exit (1);
    }

    double    rad = atof (argv[1]);
    HImage    img ("affe");
    HWindow   w;

    img.Display (w);

    HRegion    circ = HRegion::GenCircle (100, 100, rad);
    HRegionArray reg = img.Regiongrowing (3, 3, 5, 0);
    HRegionArray seg = reg.SelectShapeProto (circ, "fits", 0, 0);

    w.SetColor ("red");
    seg.Display (w);
    w.Click ();
    return(0);
}
```

	Result

The operator [select_shape_proto](#) returns the value 2 (H_MSG_TRUE) if the input is not empty.

The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result', <Result>)`. The behavior in case of empty region (the region is the empty set) is set via `set_system('empty_region_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`select_shape_proto` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`connection`, `draw_region`, `gen_circle`, `gen_rectangle1`, `gen_rectangle2`, `gen_ellipse`

Possible Successors

`select_gray`, `shape_trans`, `reduce_domain`, `count_obj`

Alternatives

`select_shape`

See also

`opening`, `erosion1`, `distance_rr_min_dil`, `distance_rr_min`

Module

Region processing

select_shape_std (Regions : SelectedRegions : Shape, Percent :)

Select regions of a given shape.

The operator `select_shape_std` compares the shape of the given regions with default shapes. If the region has a similar shape it is adopted into the output. Possible values for `Shape` are:

'max_area' The largest region is selected.

'rectangle1' The surrounding rectangle parallel to the coordinate axes is determined via the operator `smallest_rectangle1`. If the area difference in percent is larger than `Percent` the region is adopted.

'rectangle1' The smallest surrounding rectangle with any orientation is determined via the operator `smallest_rectangle2`. If the area difference in percent is larger than `Percent` the region is adopted.

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ *Hobject*
Input regions to be selected.
- ▷ **SelectedRegions** (output_object) region(-array) $\leadsto$ *Hobject*
Regions with desired shape.
- ▷ **Shape** (input_control) string $\leadsto$ *string*
Shape features to be checked.
Default Value : 'max_area'
List of values : Shape $\in$ { 'max_area', 'rectangle1', 'rectangle2' }
- ▷ **Percent** (input_control) real $\leadsto$ *real*
Similarity measure.
Default Value : 70.0
Suggested values : Percent $\in$ { 10.0, 30.0, 50.0, 60.0, 70.0, 80.0, 90.0, 95.0, 100.0 }
Typical range of values : $0.0 \leq \text{Percent} \leq 100.0$ (lin)
Minimum Increment : 0.1
Recommended Increment : 10.0

Parallelization Information

`select_shape_std` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `smallest_rectangle1`, `smallest_rectangle2`

Alternatives

`intersection`, `complement`, `area_center`

See also

`smallest_rectangle1`, `smallest_rectangle2`

Module

Region processing

smallest_circle (Regions : : : Row, Column, Radius)

Smallest surrounding circle of a region.

The operator `smallest_circle` determines the smallest surrounding circle of a region, i.e., the circle with the smallest area of all circles containing the region. For this circle the center (`Row,Column`) and the radius (`Radius`) are calculated. The procedure is applied when, for example, the location and size of circular objects (e.g., coins) which, however, are not homogeneous inside or have broken edges due to bad segmentation, has to be determined. The output of the procedure is selected in such a way that it can be used as input for the HALCONprocedures `disp_circle` and `gen_circle`.

If several regions are passed in `Regions` corresponding tuples are returned as output parameter. In case of empty region all parameters have the value 0.0 if no other behavior was set (see `set_system`).

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be examined.
 - ▷ **Row** (output_control) circle.center.y(-array) $\leadsto$ *real*
Line index of the center.
 - ▷ **Column** (output_control) circle.center.x(-array) $\leadsto$ *real*
Column index of the center.
 - ▷ **Radius** (output_control) circle.radius(-array) $\leadsto$ *real*
Radius of the surrounding circle.
- Assertion :** $\text{Radius} \geq 0$

Example

```
read_image(Image, 'fabrik')
open_window(0,0,-1,-1,'root','visible','',WindowHandle)
regiongrowing(Image,Seg,5,5,6,100:)
select_shape(Seg,H,'area','and',100,2000)
smallest_circle(H,Row,Column,Radius)
gen_circle(Circles,Row,Column,Radius)
set_draw(WindowHandle,'margin')
disp_region(Circles,WindowHandle)
```

Complexity

If F is the area of the region, then the mean runtime complexity is $O(\sqrt{F})$.

Result

The operator `smallest_circle` returns the value 2 (H.MSG.TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result', <Result>)`. The behavior in case of empty region (the region is the empty set) is set via `set_system('empty_region_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`smallest_circle` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `runlength_features`

Possible Successors

`gen_circle`, `disp_circle`

Alternatives

`elliptic_axis`, `smallest_rectangle1`, `smallest_rectangle2`

See also

`set_shape`, `select_shape`, `inner_circle`

Module

Region processing

smallest_rectangle1 (Regions : : : Row1, Column1, Row2, Column2)

Surrounding rectangle parallel to the coordinate axes.

The operator `smallest_rectangle1` calculates the surrounding rectangle of all input regions (parallel to the coordinate axes). The surrounding rectangle is described by the coordinates of the corner pixels (`Row1, Column1, Row2, Column2`)

If more than one region is passed in `Regions`, the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input. In case of empty region all parameters have the value 0 if no other behavior was set (see `set_system`).

Attention

In case of empty region the result of `Row1, Column1, Row2` and `Column2` (all are 0) can lead to confusion.

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be examined.
- ▷ **Row1** (output_control) rectangle.origin.y(-array) $\leadsto$ *integer*
Line index of upper left corner point.
- ▷ **Column1** (output_control) rectangle.origin.x(-array) $\leadsto$ *integer*
Column index of upper left corner point.
- ▷ **Row2** (output_control) rectangle.corner.y(-array) $\leadsto$ *integer*
Line index of lower right corner point.
- ▷ **Column2** (output_control) rectangle.corner.x(-array) $\leadsto$ *integer*
Column index of lower right corner point.

Complexity

If F is the area of the region the mean runtime complexity is $O(\sqrt{F})$.

Result

The operator `smallest_rectangle1` returns the value 2 (`H_MSG_TRUE`) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result', <Result>)`. The behavior in case of empty region (the region is the empty set) is set via `set_system('empty_region_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`smallest_rectangle1` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`, `runlength_features`

Possible Successors

`disp_rectangle1`, `gen_rectangle1`

Alternatives

`smallest_rectangle2`, `area_center`

See also

`select_shape`

Module

Region processing

smallest_rectangle2 (Regions : : : Row, Column, Phi, Length1, Length2)

Smallest surrounding rectangle with any orientation.

The operator `smallest_rectangle2` determines the smallest surrounding rectangle of a region, i.e., the rectangle with the smallest area of all rectangles containing the region. For this rectangle the center, the inclination and the two radii are calculated.

The procedure is applied when, for example, the location of a scenery of several regions (e.g., printed text on a rectangular paper or in rectangular print (justified lines)) must be found. The parameters of

`smallest_rectangle2` are chosen in such a way that they can be used directly as input for the HALCON-procedures `disp_rectangle2` and `gen_rectangle2`.

If more than one region is passed in `Regions` the results are stored in tuples, the index of a value in the tuple corresponding to the index of a region in the input. In case of empty region all parameters have the value 0.0 if no other behavior was set (see `set_system`).

Parameter	
▷ Regions (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Regions to be examined.
▷ Row (output_control)	rectangle2.center.y(-array) $\leadsto$ <i>real</i> Line index of the center.
▷ Column (output_control)	rectangle2.center.x(-array) $\leadsto$ <i>real</i> Column index of the center.
▷ Phi (output_control)	rectangle2.angle.rad(-array) $\leadsto$ <i>real</i> Orientation of the surrounding rectangle (arc measure) Assertion : $((-\pi/2) < \text{Phi}) \wedge (\text{Phi} \leq (\pi/2))$
▷ Length1 (output_control)	rectangle2.hwidth(-array) $\leadsto$ <i>real</i> First radius (half length) of the surrounding rectangle. Assertion : $\text{Length1} \geq 0.0$
▷ Length2 (output_control)	rectangle2.hheight(-array) $\leadsto$ <i>real</i> Second radius (half width) of the surrounding rectangle. Assertion : $(\text{Length2} \geq 0.0) \wedge (\text{Length2} \leq \text{Length1})$
Example	

```
read_image(Image, 'fabrik')
open_window(0, 0, -1, -1, 'root', 'visible', '', WindowHandle)
regiongrowing(Image, Seg, 5, 5, 6, 100)
smallest_rectangle2(Seg, Row, Column, Phi, Length1, Length2)
gen_rectangle2(Rectangle, Row, Column, Phi, Length1, Length2)
set_draw(WindowHandle, 'margin')
disp_region(Rectangle, WindowHandle)
```

Complexity
If F is the area of the region and N is the number of supporting points of the convex hull, the runtime complexity is $O(\sqrt{F} + N^2)$.

Result
The operator <code>smallest_rectangle2</code> returns the value 2 (<code>H_MSG_TRUE</code>) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator <code>set_system('no_object_result', <Result>)</code> . The behavior in case of empty region (the region is the empty set) is set via <code>set_system('empty_region_result', <Result>)</code> . If necessary an exception handling is raised.

Parallelization Information
<code>smallest_rectangle2</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).

Possible Predecessors
<code>threshold</code> , <code>regiongrowing</code> , <code>connection</code> , <code>runlength_features</code>

Possible Successors
<code>disp_rectangle2</code> , <code>gen_rectangle2</code>

Alternatives
<code>elliptic_axis</code> , <code>smallest_rectangle1</code>

See also
<code>smallest_circle</code> , <code>set_shape</code>

Module
Region processing

spatial_relation (Regions1, Regions2 : : Percent : RegionIndex1, RegionIndex2, Relation1, Relation2)

Pose relation of regions with regard to the coordinate axes.

The operator **spatial_relation** selects regions located by **Percent** percent “left”, “right”, “above” or “below” other regions. **Regions1** and **Regions2** contain the regions to be compared. **Regions1** can have three states:

- **Regions1** is empty:
In this case all regions in **Regions2** are permutatively checked for neighborhood.
- **Regions1** consists of one region:
The regions of **Regions1** are compared to all regions in **Regions2**.
- **Regions1** consists of the same number of regions as **Regions2**:
Regions1 and **Regions2** are checked for a neighboring relation.

The percentage **Percent** is interpreted in such a way that the area of the second region has to be located really left/right or above/below the region margins of the first region by at least **Percent** percent. The indices of the regions that fulfill at least one of these conditions are then located at the n-th position in the output parameters **RegionIndex1** and **RegionIndex2**. Additionally the output parameters **Relation1** and **Relation2** contain at the n-th position the type of relation of the region pair (**RegionIndex1**[n], **RegionIndex2**[n]), i.e., region with index **RegionIndex2**[n] has the relation **Relation1**[n] and **Relation2**[n] with region with index **RegionIndex1**[n].

Possible values for **Relation1** and **Relation2** are:

Relation1: 'left', 'right' or "

Relation2: 'above', 'below' or "

In **RegionIndex1** and **RegionIndex2** the indices of the regions in the tuples of the input regions (**Regions1** or **Regions2**), respectively, are entered as image identifiers. Access to chosen regions via the index can be obtained by the operator **copy_obj**.

Parameter

- ▷ **Regions1** (input_object) region(-array) $\leadsto$ *Hobject*
Starting regions.
- ▷ **Regions2** (input_object) region(-array) $\leadsto$ *Hobject*
Comparative regions.
- ▷ **Percent** (input_control) integer $\leadsto$ *integer*
Percentage of the area of the comparative region which must be located left/right or above/below the region margins of the starting region.
Default Value : 50
Suggested values : Percent $\in \{0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$
Typical range of values : $0 \leq \text{Percent} \leq 100$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : $(0 \leq \text{Percent}) \wedge (\text{Percent} \leq 100)$
- ▷ **RegionIndex1** (output_control) integer-array $\leadsto$ *integer*
Indices of the regions in the tuple of the input regions which fulfill the pose relation.
- ▷ **RegionIndex2** (output_control) integer-array $\leadsto$ *integer*
Indices of the regions in the tuple of the input regions which fulfill the pose relation.
- ▷ **Relation1** (output_control) string-array $\leadsto$ *string*
Horizontal pose relation in which **RegionIndex2**[n] stands with **RegionIndex1**[n].
- ▷ **Relation2** (output_control) string-array $\leadsto$ *string*
Vertical pose relation in which **RegionIndex2**[n] stands with **RegionIndex1**[n].

Result

The operator **spatial_relation** returns the value 2 (H_MSG_TRUE) if **Regions2** is not empty and **Percent** is correctly choosen. The behavior in case of empty parameter **Regions2** (no input regions available) is set via the operator **set_system('no_object_result', <Result>)**. The behavior in case of

empty region (the region is the empty set) is set via `set_system('empty_region_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`spatial_relation` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`

Alternatives

`area_center`, `intersection`

See also

`select_region_spatial`, `find_neighbors`, `copy_obj`, `obj_to_integer`

Module

Region processing

test_region_point (Regions : : Row, Column :)

Test if the region consists of the given point.

`test_region_point` tests if at least one input region of `Regions` consists of the test point (`Row, Column`).

Attention

In case of empty input (= no region) and `set_system('no_object_result', 'true')` FALSE is returned (no region contains the pixel).

The test pixel is not contained in an empty region (no pixel of the region corresponds to the pixel). If all regions are empty FALSE is also returned, i.e., an empty region behaves as if it did not exist.

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ *Hobject*
Region(s) to be examined.
- ▷ **Row** (input_control) point.y $\leadsto$ *integer*
Line index of the test pixel.
Default Value : 100
Typical range of values : $0 \leq \text{Row} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Column** (input_control) point.x $\leadsto$ *integer*
Column index of the test pixel.
Default Value : 100
Typical range of values : $0 \leq \text{Column} \leq 512$ (lin)
Minimum Increment : 1
Recommended Increment : 1

Complexity

If F is the area of one region and N is the number of regions, the runtime complexity is $O(\ln(\sqrt{F}) * N)$.

Result

The operator `test_region_point` returns the value 2 (H_MSG_TRUE) if a region contains the test pixel. If this is not the case `test_region_point` returns 3 (H_MSG_FALSE). The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`test_region_point` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`threshold`, `regiongrowing`, `connection`

Alternatives

`union1`, `intersection`, `area_center`

See also

`select_region_point`

	Return Value
bool	
	Module
Region processing	

12.5 Sets

complement (Region : RegionComplement : :)

Return the complement of a region.

complement determines the complement of the input region(s).

If the system flag 'clip_region' is 'false' (see [set_system](#)) the complement is done virtually by setting the complement flag of [Region](#) to TRUE. For succeeding operations the de Morgan laws are applied while calculating results.

If the system flag 'clip_region' is 'true' the difference of the largest image processed so far (see [reset_obj_db](#)) and the input region is returned.

	Parameter
▷ Region (input_object)	region(-array) $\leadsto$ Hobject Input region(s).
▷ RegionComplement (output_object)	region(-array) $\leadsto$ Hobject Complemented regions.
Number of elements : RegionComplement = Region	
	Result

complement always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via [set_system\('no_object_result', <Result>\)](#) and the behavior in case of an empty input region via [set_system\('empty_region_result', <Result>\)](#). If necessary, an exception handling is raised.

	Parallelization Information
complement is reentrant and automatically parallelized (on tuple level).	
	Possible Predecessors
threshold , connection , regiongrowing , pouring , class_ndim_norm	
	Possible Successors
select_shape	
	See also
difference , union1 , union2 , intersection , reset_obj_db , set_system	
	Module
Region processing	

difference (Region, Sub : RegionDifference : :)

Calculate the difference of two regions.

difference calculates the set-theoretic difference of two regions:

$$(\text{Regions in } \text{Region}) - (\text{Regions in } \text{Sub})$$

The resulting region is defined as the input region ([Region](#)) with all points from [Sub](#) removed.

Attention
Empty regions are valid for both parameters. On output, empty regions may result. The value of the system flag 'store_empty_region' determines the behavior in this case.

Parameter	
▷ Region (input_object)	region(-array) $\leadsto$ Hobject Regions to be processed.
▷ Sub (input_object)	region(-array) $\leadsto$ Hobject The union of these regions is subtracted from Region.
▷ RegionDifference (output_object)	region(-array) $\leadsto$ Hobject Resulting region.
Example	
<pre>/* provides the region X without the points in Y */ difference(X,Y,RegionDifference)</pre>	
Complexity	
Let N be the number of regions, F_1 be their average area, and F_2 be the total area of all regions in Sub . Then the runtime complexity is $O(F_1 * \log(F_1) + N * (\sqrt{F_1} + \sqrt{F_2}))$.	
Result	
difference always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via set_system('no_object_result', <Result>) and the behavior in case of an empty input region via set_system('empty_region_result', <Result>) . If necessary, an exception handling is raised.	
Parallelization Information	
difference is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Predecessors	
threshold , connection , regiongrowing , pouring , class_ndim_norm	
Possible Successors	
select_shape , disp_region	
See also	
intersection , union1 , union2 , complement	
Module	
Region processing	

intersection (Region1, Region2 : RegionIntersection : :)

Calculate the intersection of two regions.

intersection calculates the intersection of the regions in **Region1** with the regions in **Region2**. Each region in **Region1** is intersected with all regions in **Region2**. The order of regions in **RegionIntersection** is identical to the order of regions in **Region1**.

Attention
Empty input regions are permitted. Because empty result regions are possible, the system flag 'store_empty_region' should be set appropriately.

Parameter	
▷ Region1 (input_object)	region(-array) $\leadsto$ Hobject Regions to be intersected with all regions in Region2.
▷ Region2 (input_object)	region(-array) $\leadsto$ Hobject Regions with which Region1 is intersected.
▷ RegionIntersection (output_object)	region(-array) $\leadsto$ Hobject Result of the intersection.
Number of elements : RegionIntersection $\leq$ Region1	
Complexity	

Let N be the number of regions in **Region1**, F_1 be their average area, and F_2 be the total area of all regions in **Region2**. Then the runtime complexity is $O(F_1 \log(F_1) + N * (\sqrt{F_1} + \sqrt{F_2}))$.

Result

`intersection` always returns 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via `set_system('no_object_result', <Result>)` and the behavior in case of an empty input region via `set_system('empty_region_result', <Result>)`. If necessary, an exception handling is raised.

Parallelization Information

`intersection` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`threshold`, `connection`, `regiongrowing`, `pouring`

Possible Successors

`select_shape`, `disp_region`

See also

`union1`, `union2`, `complement`

Module

Region processing

union1 (Region : RegionUnion : :)

Return the union of all input regions.

`union1` computes the union of all input regions and returns the result in `RegionUnion`.

Parameter

- ▷ **Region** (input_object) region-array $\leadsto$ Hobject
Regions of which the union is to be computed.
 - ▷ **RegionUnion** (output_object) region $\leadsto$ Hobject
Union of all input regions.
- Number of elements** : RegionUnion $\leq$ Region

Example

```
/* Union of segmentation results: */
threshold(Image, Region1, 128, 255)
dyn_threshold(Image, Mean, Region2, 5, 'light')
concat_obj(Region1, Region2, Regions)
union1(Regions, RegionUnion).
```

Complexity

Let F be the sum of all areas of the input regions. Then the runtime complexity is $O(\log(\sqrt{F}) * \sqrt{F})$.

Result

`union1` always returns 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via `set_system('no_object_result', <Result>)` and the behavior in case of an empty input region via `set_system('empty_region_result', <Result>)`. If necessary, an exception handling is raised.

Parallelization Information

`union1` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`threshold`, `connection`, `regiongrowing`, `pouring`

Possible Successors

`select_shape`, `disp_region`

Alternatives

`union2`

See also

`intersection`, `complement`

Module

Region processing

union2 (Region1, Region2 : RegionUnion : :)

Return the union of two regions.

union2 computes the union of the region in **Region1** with all regions in **Region2**. This means that **union2** is not commutative!

Parameter

- ▷ **Region1** (input_object) region(-array) $\leadsto$ *Hobject*
Region for which the union with all regions in Region2 is to be computed.
- ▷ **Region2** (input_object) region(-array) $\leadsto$ *Hobject*
Regions which should be added to Region1.
- ▷ **RegionUnion** (output_object) region(-array) $\leadsto$ *Hobject*
Resulting regions.

Number of elements : RegionUnion = Region1

Complexity

Let F be the sum of all areas of the input regions. Then the runtime complexity is $O(\log(\sqrt{F}) * \sqrt{F})$.

Result

union2 always returns 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via **set_system('no_object_result', <Result>)** and the behavior in case of an empty input region via **set_system('empty_region_result', <Result>)**. If necessary, an exception handling is raised.

Parallelization Information

union2 is *reentrant* and processed *without* parallelization.

Possible Predecessors

threshold, **connection**, **regiongrowing**, **pouring**

Possible Successors

select_shape, **disp_region**

Alternatives

union1

See also

intersection, **complement**

Module

Region processing

12.6 Transformation

background_seg (Foreground : BackgroundRegions : :)

Determine the connected components of the background of given regions.

background_seg determines connected components of the background of the foreground regions given in **Foreground**. This operator is normally used after an edge operator in order to determine the regions enclosed by the extracted edges. The connected components are determined using 4-neighborhood.

Parameter

- ▷ **Foreground** (input_object) region(-array) $\leadsto$ *Hobject*
Input regions.
- ▷ **BackgroundRegions** (output_object) region-array $\leadsto$ *Hobject*
Connected components of the background.

Example

```
/* Simulation of background_seg: */
background_seg(Foreground, BackgroundRegions):
    complement(Foreground, Background)
```

```

get_system('neighborhood', Save)
set_system('neighborhood', 4)
connection(Background, BackgroundRegions)
    clear_obj(Background)
set_system('neighborhood', Save).

/* Segmentation with edge filter: */
read_image(Image, 'fabrik')
sobel_dir(Image, Sobel, Dir, 'sum_sqrt', 3)
threshold(Sobel, Edges, 20, 255)
skeleton(Edges, Margins)
background_seg(Margins, Regions).

```

Complexity

Let F be the area of the background, H and W be the height and width of the image, and N be the number of resulting regions. Then the runtime complexity is $O(H + \sqrt{F} * \sqrt{N})$.

Result

[background_seg](#) always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via [set_system\('no-object-result', <Result>\)](#) and the behavior in case of an empty input region via [set_system\('empty-region-result', <Result>\)](#). If necessary, an exception handling is raised.

Parallelization Information

[background_seg](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[threshold](#), [connection](#), [regiongrowing](#), [pouring](#), [class_ndim_norm](#)

Possible Successors

[select_shape](#)

Alternatives

[complement](#), [connection](#)

See also

[threshold](#), [hysteresis_threshold](#), [skeleton](#), [expand_region](#), [set_system](#), [sobel_amp](#), [edges_image](#), [roberts](#), [bandpass_image](#)

Module

Region processing

clip_region (Region : RegionClipped : Row1, Column1, Row2,
Column2 :)

Clip a region to a rectangle.

[clip_region](#) clips the input regions to the rectangle given by the four control parameters. [clip_region](#) is more efficient than calling [intersection](#) with a rectangle generated by [gen_rectangle1](#).

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Region to be clipped.
- ▷ **RegionClipped** (output_object) region(-array) $\leadsto$ *Hobject*
Clipped regions.
- ▷ **Row1** (input_control) rectangle.origin.y $\leadsto$ *integer*
Row coordinate of the upper left corner of the rectangle.
Default Value : 0
Suggested values : Row1 $\in$ {0, 128, 200, 256}
Typical range of values : $-\infty \leq \text{Row1} \leq \infty$ (lin)

- ▷ **Column1** (input_control) rectangle.origin.x $\leadsto$ integer
Column coordinate of the upper left corner of the rectangle.
Default Value : 0
Suggested values : Column1 $\in$ {0, 128, 200, 256}
Typical range of values : $-\infty \leq \text{Column1} \leq \infty$ (lin)
- ▷ **Row2** (input_control) rectangle.corner.y $\leadsto$ integer
Row coordinate of the lower right corner of the rectangle.
Default Value : 256
Suggested values : Row2 $\in$ {128, 200, 256, 512}
Typical range of values : $0 \leq \text{Row2} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Column2** (input_control) rectangle.corner.x $\leadsto$ integer
Column coordinate of the lower right corner of the rectangle.
Default Value : 256
Suggested values : Column2 $\in$ {128, 200, 256, 512}
Typical range of values : $0 \leq \text{Column2} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 10

Result

clip_region returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via **set_system('no_object_result', <Result>)** and the behavior in case of an empty input region via **set_system('empty_region_result', <Result>)**. If necessary, an exception handling is raised.

Parallelization Information

clip_region is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

threshold, **connection**, **regiongrowing**, **pouring**

Possible Successors

select_shape, **disp_region**

Alternatives

intersection, **gen_rectangle1**, **clip_region_rel**

Module

Region processing

clip_region_rel (Region : RegionClipped : Top, Bottom, Left, Right :)

Clip a region relative to its size.

clip_region_rel clips a region to a rectangle lying within the region. The size of the rectangle is determined by the enclosing rectangle of the region, which is reduced by the values given in the four control parameters. All four parameters must contain numbers larger or equal to zero, and determine by which amount the rectangle is reduced at the top (**Top**), at the bottom (**Bottom**), at the left (**Left**), and at the right (**Right**). If all parameters are set to zero, the region remains unchanged.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ Hobject
Regions to be clipped.
- ▷ **RegionClipped** (output_object) region(-array) $\leadsto$ Hobject
Clipped regions.

- ▷ **Top** (input_control) integer $\rightsquigarrow$ integer
 Number of rows clipped at the top.
Default Value : 1
Suggested values : Top $\in \{0, 1, 2, 3, 4, 5, 7, 10, 20, 30, 50\}$
Typical range of values : $0 \leq \text{Top} \text{ (lin)}$
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Bottom** (input_control) integer $\rightsquigarrow$ integer
 Number of rows clipped at the bottom.
Default Value : 1
Suggested values : Bottom $\in \{0, 1, 2, 3, 4, 5, 7, 10, 20, 30, 50\}$
Typical range of values : $0 \leq \text{Bottom} \text{ (lin)}$
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Left** (input_control) integer $\rightsquigarrow$ integer
 Number of columns clipped at the left.
Default Value : 1
Suggested values : Left $\in \{0, 1, 2, 3, 4, 5, 7, 10, 20, 30, 50\}$
Typical range of values : $0 \leq \text{Left} \text{ (lin)}$
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Right** (input_control) integer $\rightsquigarrow$ integer
 Number of columns clipped at the right.
Default Value : 1
Suggested values : Right $\in \{0, 1, 2, 3, 4, 5, 7, 10, 20, 30, 50\}$
Typical range of values : $0 \leq \text{Right} \text{ (lin)}$
Minimum Increment : 1
Recommended Increment : 1

Result

`clip_region_rel` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via `set_system('no_object_result', <Result>)` and the behavior in case of an empty input region via `set_system('empty_region_result', <Result>)`. If necessary, an exception handling is raised.

Parallelization Information

`clip_region_rel` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`threshold`, `connection`, `regiongrowing`, `pouring`

Possible Successors

`select_shape`, `disp_region`

Alternatives

`smallest_rectangle1`, `intersection`, `gen_rectangle1`, `clip_region`

Module

Region processing

connection (Region : ConnectedRegions : :)

Compute connected components of a region.

`connection` determines the connected components of the input regions given in `Region`. The neighborhood used for this can be set via `set_system('neighborhood', <4/8>)`. The default is 8-neighborhood, which is useful for determining the connected components of the foreground. The maximum number of connected components that is returned by `connection` can be set via `set_system('max_connection', <Num>)`. The default value of 0 causes all connected components to be returned. The inverse operator of `connection` is `union1`.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ Hobject
Input region.
- ▷ **ConnectedRegions** (output_object) region-array $\leadsto$ Hobject
Connected components.

Example

```

read_image(Image, 'affe')
set_colored(WindowHandle, 12)
threshold(Image, Light, 150.0, 255.0)
count_obj(Light, Number1)
fwrite_string('Nummber of regions after threshold = '+Number1)
fnew_line()
disp_region(Light, WindowHandle)
connection(Light, Many)
count_obj(Many, Number2)
fwrite_string('Nummber of regions after threshold = '+Number2)
fnew_line()
disp_region(Many, WindowHandle).

```

Complexity

Let F be the area of the input region and N be the number of generated connected components. Then the runtime complexity is $O(\sqrt{F} * \sqrt{N})$.

Result

connection always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via **set_system('no_object_result', <Result>)** and the behavior in case of an empty input region via **set_system('empty_region_result', <Result>)**. If necessary, an exception handling is raised.

Parallelization Information

connection is *reentrant* and processed *without* parallelization.

Possible Predecessors

auto_threshold, **threshold**, **dyn_threshold**, **erosion1**

Possible Successors

select_shape, **select_gray**, **shape_trans**, **set_colored**, **dilation1**, **count_obj**,
reduce_domain, **add_channels**

Alternatives

background_seg

See also

set_system, **union1**

Module

Region processing

distance_transform (Region : DistanceImage : Metric, Foreground,
Width, Height :)

Compute the distance transformation of a region.

distance_transform computes for every point of the input region **Region** (or its complement, respectively) the distance of the point to the border of the region. The parameter **Foreground** determines whether the distances are calculated for all points within the region (**Foreground** = 'true') or for all points outside the region (**Foreground** = 'false'). The distance is computed for every point of the output image **DistanceImage**, which has the specified dimensions **Width** and **Height**. The input region is always clipped to the extent of the output image. If it is important that the distances within the entire region should be computed, the region should be moved (see **move_region**) so that it has only positive coordinates and the width and height of the

output image should be large enough to contain the region. The extent of the input region can be obtained with `smallest_rectangle1`.

The parameter `Metric` determines which metric is used for the calculation of the distances. If `Metric = 'city-block'`, the distance is calculated from the shortest path from the point to the border of the region, where only horizontal and vertical “movements” are allowed. They are weighted with a distance of 1. If `Metric = 'chessboard'`, the distance is calculated from the shortest path to the border, where horizontal, vertical, and diagonal “movements” are allowed. They are weighted with a distance of 1. If `Metric = 'octagonal'`, a combination of these approaches is used, which leads to diagonal paths getting a higher weight. If `Metric = 'chamfer-3-4'`, horizontal and vertical movements are weighted with a weight of 3, while diagonal movements are weighted with a weight of 4. To normalize the distances, the resulting distance image is divided by 3. Since this normalization step takes some time, and one usually is interested in the relative distances of the points, the normalization can be suppressed with `Metric = 'chamfer-3-4-unnormalized'`. Finally, if `Metric = 'euclidean'`, the computed distance is approximately Euclidean.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Region for which the distance to the border is computed.
- ▷ **DistanceImage** (output_object) image $\leadsto$ *Hobject*: int4
Image containing the distance information.
- ▷ **Metric** (input_control) string $\leadsto$ *string*
Type of metric to be used for the distance transformation.
Default Value : "city-block"
List of values : `Metric` $\in$ {"city-block", "chessboard", "octagonal", "chamfer-3-4", "chamfer-3-4-unnormalized", "euclidean"}
- ▷ **Foreground** (input_control) string $\leadsto$ *string*
Compute the distance for pixels inside (*true*) or outside (*false*) the input region.
Default Value : 'true'
List of values : `Foreground` $\in$ {'true', 'false'}
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer*
Width of the output image.
Default Value : 640
Suggested values : `Width` $\in$ {160, 192, 320, 384, 640, 768}
Typical range of values : $1 \leq \text{Width}$
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer*
Height of the output image.
Default Value : 480
Suggested values : `Height` $\in$ {120, 144, 240, 288, 480, 576}
Typical range of values : $1 \leq \text{Height}$

Example

```
/* Step towards extracting the medial axis of a shape: */
gen_rectangle1 (Rectangle1, 0, 0, 200, 400)
gen_rectangle1 (Rectangle2, 200, 0, 400, 200)
union2 (Rectangle1, Rectangle2, Shape)
distance_transform (Shape, DistanceImage, 'chessboard', 'true', 640, 480)
```

Complexity

The runtime complexity is $O(\text{Width} * \text{Height})$.

Result

`distance_transform` returns H_MSG_2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

`distance_transform` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `dyn_threshold`, `regiongrowing`

Possible Successors

`threshold`

See also

[skeleton](#)

References

P. Soille: “Morphological Image Analysis, Principles and Applications”; Springer Verlag Berlin Heidelberg New York, 1999.

G. Borgefors: “Distance Transformations in Arbitrary Dimensions”; Computer Vision, Graphics, and Image Processing, Vol. 27, pages 321–345, 1984.

P.E. Danielsson: “Euclidean Distance Mapping”; Computer Graphics and Image Processing, Vol. 14, pages 227–248, 1980.

Module

Region processing

eliminate_runs (Region : RegionClipped : ElimShorter, ElimLonger :)

Eliminate runs of a given length.

[eliminate_runs](#) eliminates all runs of the run length encoding of the input regions which are shorter than [ElimShorter](#) or longer as [ElimLonger](#).

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Region to be clipped.
- ▷ **RegionClipped** (output_object) region(-array) $\leadsto$ *Hobject*
Clipped regions.
- ▷ **ElimShorter** (input_control) integer $\leadsto$ integer
All runs which are shorter are eliminated.
Default Value : 3
Suggested values : $\text{ElimShorter} \in \{2, 3, 4, 5, 6, 8, 10, 12, 15\}$
Typical range of values : $1 \leq \text{ElimShorter} \leq 500$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **ElimLonger** (input_control) integer $\leadsto$ integer
All runs which are longer are eliminated.
Default Value : 1000
Suggested values : $\text{ElimLonger} \in \{50, 100, 200, 500, 1000, 2000\}$
Typical range of values : $1 \leq \text{ElimLonger} \leq 10000$ (lin)
Minimum Increment : 1
Recommended Increment : 10

Result

[eliminate_runs](#) returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via [set_system\('no_object_result', <Result>\)](#) and the behavior in case of an empty input region via [set_system\('empty_region_result', <Result>\)](#). If necessary, an exception handling is raised.

Parallelization Information

[eliminate_runs](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[threshold](#), [connection](#), [regiongrowing](#), [pouring](#)

Possible Successors

[erosion1](#), [dilation1](#), [disp_region](#)

Alternatives

[shape_trans](#)

Module

Region processing

expand_region (Regions, ForbiddenArea : RegionExpanded : Iterations, Mode :)

Fill gaps between regions or split overlapping regions.

expand_region closes gaps between the input regions, which resulted from the suppression of small regions in a segmentation operator, for example, (mode *'image'*), or to separate overlapping regions (mode *'region'*). Both uses result from the expansion of regions. The operator works by adding or removing a one pixel wide “strip” to a region.

The expansion takes place only in regions, which are designated as not “forbidden” (parameter **ForbiddenArea**). The number of iterations is determined by the parameter **Iterations**. By passing *'maximal'*, **expand_region** iterates until convergence, i.e., until no more changes occur. By passing 0 for this parameter, all non-overlapping regions are returned. The two modes of operation (*'image'* and *'region'*) are different in the following ways:

'image' The input regions are expanded iteratively until they touch another region or the image border. Because **expand_region** processes all regions simultaneously, gaps between regions are distributed evenly to all regions. Overlapping regions are split by distributing the area of overlap evenly to both regions.

'region' No expansion of the input regions is performed. Instead, only overlapping regions are split by distributing the area of overlap evenly to the respective regions. Because the intersection with the original region is computed after the shrinking operation gaps in the output regions may result, i.e., the segmentation is not complete. This can be prevented by calling **expand_region** a second time with the complement of the original regions as “forbidden area.”

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ *Hobject*
Regions for which the gaps are to be closed, or which are to be separated.
- ▷ **ForbiddenArea** (input_object) region $\leadsto$ *Hobject*
Regions in which no expansion takes place.
- ▷ **RegionExpanded** (output_object) region(-array) $\leadsto$ *Hobject*
Expanded or separated regions.
- ▷ **Iterations** (input_control) integer $\leadsto$ *integer / string*
Number of iterations.
Default Value : *'maximal'*
Suggested values : $\text{Iterations} \in \{\text{'maximal'}, 0, 1, 2, 3, 5, 7, 10, 15, 20, 30, 50, 70, 100, 200\}$
Typical range of values : $0 \leq \text{Iterations} \leq 1000$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Expansion mode.
Default Value : *'image'*
List of values : $\text{Mode} \in \{\text{'image'}, \text{'region'}\}$

Example

```
read_image(Image, 'fabrik')
threshold(Image, Light, 100, 255)
disp_region(Light, WindowHandle)
connection(Light, Seg)
expand_region(Seg, [], Exp1, 'maximal', 'image')
set_colored(WindowHandle, 12)
set_draw(WindowHandle, 'margin')
disp_region(Exp1, WindowHandle)
```

Result

expand_region always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via **set_system('no_object_result', <Result>)**, the behavior in case of an empty input region via **set_system('empty_region_result', <Result>)**, and the behavior in case of an empty

result region via `set_system('store_empty_region', <true/false>)`. If necessary, an exception handling is raised.

Parallelization Information

`expand_region` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`pouring`, `threshold`, `dyn_threshold`, `regiongrowing`

Alternatives

`dilation1`

See also

`expand_gray`, `interjacent`, `skeleton`

Module

Region processing

fill_up (Region : RegionFillUp : :)

Fill up holes in regions.

`fill_up` fills up holes in regions. The number of regions remains unchanged. The neighborhood type is set via `set_system('neighborhood', <4/8>)` (default: 8-neighborhood).

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Input regions containing holes.
- ▷ **RegionFillUp** (output_object) region(-array) $\leadsto$ *Hobject*
Regions without holes.

Result

`fill_up` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via `set_system('no_object_result', <Result>)` and the behavior in case of an empty input region via `set_system('empty_region_result', <Result>)`. If necessary, an exception handling is raised.

Parallelization Information

`fill_up` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `connection`, `regiongrowing`, `pouring`

Possible Successors

`select_shape`, `disp_region`

Alternatives

`fill_up_shape`

See also

`boundary`

Module

Region processing

fill_up_shape (Region : RegionFillUp : Feature, Min, Max :)

Fill up holes in regions having given shape features.

`fill_up_shape` fills up those holes in the input region `Region` having given shape features. The parameter `Feature` determines the shape feature to be used, while `Min` and `Max` determine the range the shape feature has to lie in in order for the hole to be filled up.

Parameter	
▷ Region (input_object)	region(-array) $\leadsto$ Hobject Input region(s).
▷ RegionFillUp (output_object)	region(-array) $\leadsto$ Hobject Output region(s) with filled holes.
▷ Feature (input_control)	string $\leadsto$ string Shape feature used. Default Value : 'area' List of values : Feature $\in$ {'area', 'compactness', 'convexity', 'anisometry', 'phi', 'ra', 'rb', 'inner_circle', 'outer_circle'}
▷ Min (input_control)	number $\leadsto$ real / integer Minimum value for Feature. Default Value : 1.0 Suggested values : Min $\in$ {0.0, 1.0, 10.0, 50.0, 100.0, 500.0, 1000.0, 10000.0} Typical range of values : $0.0 \leq \text{Min}$
▷ Max (input_control)	number $\leadsto$ real / integer Maximum value for Feature. Default Value : 100.0 Suggested values : Max $\in$ {10.0, 50.0, 100.0, 500.0, 1000.0, 10000.0, 100000.0} Typical range of values : $0.0 \leq \text{Max}$
Example	

```
read_image(&Image, "affe");
threshold(Image, &Seg, 120.0, 255.0);
fill_up_shape(Seg, &Filled, "area", 0.0, 200.0);
```

fill_up_shape returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via `set_system('no_object_result', <Result>)` and the behavior in case of an empty input region via `set_system('empty_region_result', <Result>)`. If necessary, an exception handling is raised.

Parallelization Information	
fill_up_shape is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).	
Possible Predecessors	
threshold , connection , regiongrowing , pouring	
Possible Successors	
select_shape , disp_region	
Alternatives	
fill_up	
See also	
select_shape , connection , area_center	
Module	
Region processing	

hamming_change_region (InputRegion : OutputRegion : Width, Height, Distance :)

Generate a region having a given Hamming distance.

hamming_change_region changes the region in the left upper part of the image given by **Width** and **Height** such that the resulting regions have a Hamming distance of **Distance** to the input regions. This is done by adding or removing **Distance** points from the input region.

Attention
If **Width** and **Height** are chosen too large the resulting region requires a lot of memory.

Parameter	
▷ InputRegion (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Region to be modified.
▷ OutputRegion (output_object)	region(-array) $\leadsto$ <i>Hobject</i> Regions having the required Hamming distance.
▷ Width (input_control)	extent.x $\leadsto$ <i>integer</i> Width of the region to be changed. Default Value : 100 Suggested values : Width $\in \{64, 128, 256, 512\}$ Typical range of values : $1 \leq \text{Width} \leq 512$ (lin) Minimum Increment : 1 Recommended Increment : 10 Restriction : Width > 0
▷ Height (input_control)	extent.y $\leadsto$ <i>integer</i> Height of the region to be changed. Default Value : 100 Suggested values : Height $\in \{64, 128, 256, 512\}$ Typical range of values : $1 \leq \text{Height} \leq 512$ (lin) Minimum Increment : 1 Recommended Increment : 10 Restriction : Height > 0
▷ Distance (input_control)	integer $\leadsto$ <i>integer</i> Hamming distance between the old and new regions. Default Value : 1000 Suggested values : Distance $\in \{100, 500, 1000, 5000, 10000\}$ Typical range of values : $0 \leq \text{Distance} \leq 10000$ (lin) Minimum Increment : 1 Recommended Increment : 10 Restriction : $(\text{Distance} \geq 0) \wedge (\text{Distance} < (\text{Width} \cdot \text{Height}))$
Complexity	
Memory requirement of the generated region (worst case): $O(2 * \text{Width} * \text{Height})$.	
Result	
hamming_change_region returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via set_system('no_object_result', <Result>) . If necessary, an exception handling is raised.	
Parallelization Information	
hamming_change_region is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).	
Possible Predecessors	
connection , regiongrowing , pouring , class_ndim_norm	
Possible Successors	
select_shape	
See also	
hamming_distance	
Module	
Region processing	

interjacent (Region : RegionInterjacent : Mode :)

Partition the image plane using given regions.

[interjacent](#) partitions the image plane using the regions given in [Region](#). The result is a region containing the extracted separating lines. The following modes of operation can be used:

'medial_axis' This mode is used for regions that do not touch or overlap. The operator will find separating lines between the regions which partition the background evenly between the input regions. This corresponds to the following calls:

```
complement('full', Region, Tmp) skeleton(Tmp, Result)
```

'border' If the input regions do not touch or overlap this mode is equivalent to `boundary(Region, Result)`, i.e., it replaces each region by its boundary. If regions are touching they are aggregated into one region. The corresponding output region then contains the boundary of the aggregated region, as well as the one pixel wide separating line between the original regions. This corresponds to the following calls:

```
boundary(Region, Tmp1, 'inner') union1(Tmp1, Tmp2)
skeleton(Tmp2, Result)
```

'mixed' In this mode the operator behaves like the mode `'medial_axis'` for non-overlapping regions. If regions touch or overlap, again separating lines between the input regions are generated on output, but this time including the “touching line” between regions, i.e., touching regions are separated by a line in the output region. This corresponds to the following calls:

```
erosion1(Region, Mask, Tmp1, 1) union1(Tmp1, Tmp2)
complement('full', Tmp2, Tmp3) skeleton(Tmp3, Result)
```

where Mask denotes the following “cross mask”:

```

      ×
    × × ×
      ×

```

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions for which the separating lines are to be determined.
- ▷ **RegionInterjacent** (output_object) region $\leadsto$ *Hobject*
Output region containing the separating lines.
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Mode of operation.
Default Value : 'mixed'
List of values : Mode $\in$ {'medial_axis', 'border', 'mixed'}

Example

```
read_image(Image, 'wald1_rot')
mean(Image, Mean, 31, 31)
dyn_threshold(Mean, Seg, 20)
interjacent(Seg, Graph, 'medial_axis')
disp_region(Graph, WindowHandle)
```

Result

`interjacent` always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via `set_system('no_object_result', <Result>)`, the behavior in case of an empty input region via `set_system('empty_region_result', <Result>)`, and the behavior in case of an empty result region via `set_system('store_empty_region', <true/false>)`. If necessary, an exception handling is raised.

Parallelization Information

`interjacent` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`threshold`, `connection`, `regiongrowing`, `pouring`

Possible Successors

`select_shape`, `disp_region`

See also

`expand_region`, `junctions_skeleton`, `boundary`

Module

Region processing

junctions_skeleton (Region : EndPoints, JuncPoints : :)

Find junctions and end points in a skeleton.

[junctions_skeleton](#) detects junctions and end points in a skeleton (see [skeleton](#)). The junctions in the input region [Region](#) are output as a region in [JuncPoints](#), while the end points are output as a region in [EndPoints](#).

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Input skeletons.
- ▷ **EndPoints** (output_object) region(-array) $\leadsto$ *Hobject*
Extracted end points.
Number of elements : EndPoints = Region
- ▷ **JuncPoints** (output_object) region(-array) $\leadsto$ *Hobject*
Extracted junctions.
Number of elements : JuncPoints = Region

Example

```
/* non-connected branches of a skeleton */
skeleton(Region,Skeleton)
junctions_skeleton(Skeleton,EPoints,JPoints)
difference(S,JPoints,Rows)
set_system('neighbourhood',4)
connection(Rows,Parts).
```

Complexity

Let F be the area of the input region. Then the runtime complexity is $O(F)$.

Result

[junctions_skeleton](#) always returns the value 2 (H_MSG.TRUE). The behavior in case of empty input (no regions given) can be set via [set_system\('no-object-result',<Result>\)](#), the behavior in case of an empty input region via [set_system\('empty-region-result',<Result>\)](#), and the behavior in case of an empty result region via [set_system\('store-empty-region',<true/false>\)](#). If necessary, an exception handling is raised.

Parallelization Information

[junctions_skeleton](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[skeleton](#)

Possible Successors

[area_center](#), [connection](#), [get_region_points](#), [difference](#)

See also

[pruning](#), [split_skeleton_region](#)

Module

Region processing

merge_regions_line_scan (CurrRegions,
PrevRegions : CurrMergedRegions, PrevMergedRegions : ImageHeight,
MergeBorder, MaxImagesRegion :)

Merge regions from line scan images.

The operator [merge_regions_line_scan](#) connects adjacent regions, which were segmented from adjacent images with the height [ImageHeight](#). This operator was especially designed to process regions that were extracted from images grabbed by a line scan camera. [CurrRegions](#) contains the regions from the current image and [PrevRegions](#) the regions from the previous one.

With the help of the parameter `MergeBorder` two cases can be distinguished: If the top (first) line of the current image touches the bottom (last) line of the previous image, `MergeBorder` must be set to 'top', otherwise set `MergeBorder` to 'bottom'.

If the operator `merge_regions_line_scan` is used recursively, the parameter `MaxImagesRegion` determines the maximum number of images which are covered by a merged region. All older region parts are removed.

The operator `merge_regions_line_scan` returns two region arrays. `PrevMergedRegions` contains all those regions from the previous input regions `PrevRegions`, which could not be merged with a current region. `CurrMergedRegions` collects all current regions together with the merged parts from the previous images. Merged regions will exceed the original image, because the previous regions are moved upward (`MergeBorder='top'`) or downward (`MergeBorder='bottom'`) according to the image height. For this the system parameter 'clip_region' (see also `set_system`) will internally be set to 'false'.

Parameter	
▷ CurrRegions (input_object)	region(-array) $\leadsto$ Hobject Current input regions.
▷ PrevRegions (input_object)	region(-array) $\leadsto$ Hobject Merged regions from the previous iteration.
▷ CurrMergedRegions (output_object)	region(-array) $\leadsto$ Hobject Current regions, merged with old ones where applicable.
▷ PrevMergedRegions (output_object)	region(-array) $\leadsto$ Hobject Regions from the previous iteration which could not be merged with the current ones.
▷ ImageHeight (input_control)	integer $\leadsto$ integer Height of the line scan images. Default Value : 512 List of values : ImageHeight $\in$ {240, 480, 512}
▷ MergeBorder (input_control)	string $\leadsto$ string Image line of the current image, which touches the previous image. Default Value : "top" List of values : MergeBorder $\in$ {"top", "bottom"}
▷ MaxImagesRegion (input_control)	integer $\leadsto$ integer Maximum number of images for a single region. Default Value : 3 Suggested values : MaxImagesRegion $\in$ {1, 2, 3, 4, 5}

Result
The operator `merge_regions_line_scan` returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Parallelization Information
`merge_regions_line_scan` is *reentrant* and processed *without* parallelization.

Module
Region processing

partition_dynamic (Region : Partitioned : Distance, Percent :)

Partition a region horizontally into rectangles.

`partition_dynamic` partitions the input region into rectangles having an average width of `Distance`. The region is not split exactly into n parts, but rather starting from the calculated splitting point positions having a minimum number of set pixels in the vertical directions are extracted. The search area is at most plus/minus the width of a rectangle. If `Percent` is passed as 100, this search area is used fully. If it is passed as 0, no search is done.

If the region is smaller than the given size its output remains unchanged. A partition is only done if the size of the region is at least 1.5 times the size of the rectangle given by the parameters.

Parameter
▷ Region (input_object) region(-array) $\leadsto$ <i>Hobject</i> Region to be partitioned. ▷ Partitioned (output_object) region-array $\leadsto$ <i>Hobject</i> Partitioned region. ▷ Distance (input_control) real $\leadsto$ <i>real</i> Width of the individual rectangles. ▷ Percent (input_control) real $\leadsto$ <i>real</i> Maximum shift of the partition point. Default Value : 20 Suggested values : Percent $\in \{0, 10, 20, 30, 40, 50, 70, 90, 100\}$ Typical range of values : $0 \leq \text{Percent} \leq 100$
Result
<code>partition_dynamic</code> returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via <code>set_system('no_object_result', <Result>)</code>, the behavior in case of an empty input region via <code>set_system('empty_region_result', <Result>)</code>, and the behavior in case of an empty result region via <code>set_system('store_empty_region', <true/false>)</code>. If necessary, an exception handling is raised.
Parallelization Information
<code>partition_dynamic</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).
Possible Predecessors
<code>threshold</code>, <code>connection</code>
Alternatives
<code>partition_rectangle</code>
See also
<code>intersection</code>, <code>smallest_rectangle1</code>, <code>shape_trans</code>, <code>clip_region</code>
Module
Region processing

partition_rectangle (Region : Partitioned : Width, Height :)

Partition a region into rectangles of equal size.

`partition_rectangle` partitions the input region into rectangles having an extent of **Width** times **Height**. The region is always split into rectangles of equal size. Therefore, **Width** and **Height** are adapted to the actual size of the region. If the region is smaller than the given size its output remains unchanged. A partition is only done if the size of the region is at least 1.5 times the size of the rectangle given by the paramters.

Parameter
▷ Region (input_object) region(-array) $\leadsto$ <i>Hobject</i> Region to be partitioned. ▷ Partitioned (output_object) region(-array) $\leadsto$ <i>Hobject</i> Partitioned region. ▷ Width (input_control) real $\leadsto$ <i>real</i> Width of the individual rectangles. ▷ Height (input_control) real $\leadsto$ <i>real</i> Height of the individual rectangles.
Result
<code>partition_rectangle</code> returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via <code>set_system('no_object_result', <Result>)</code>, the behavior in case of an empty input region via <code>set_system('empty_region_result', <Result>)</code>, and the behavior in case of an empty result region via <code>set_system('store_empty_region', <true/false>)</code>. If necessary, an exception handling is raised.

<i>Parallelization Information</i>
partition_rectangle is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).
<i>Possible Predecessors</i>
threshold , connection
<i>Alternatives</i>
partition_dynamic
<i>See also</i>
intersection , smallest_rectangle1 , shape_trans , clip_region
<i>Module</i>
Region processing

rank_region (<i>Region</i> : <i>RegionCount</i> : <i>Width</i> , <i>Height</i> , <i>Number</i> :)

Rank operator for regions.

[rank_region](#) calculates the binary rank operator. A filter mask of size [Height](#) x [Width](#) is used. In the process, for each point in the region the number of points of [Region](#) lying within the filter mask are counted. If this number is greater or equal to [Number](#), the current point is added to the output region. If

$$\text{Number} = \frac{\text{Height} * \text{Width}}{2},$$

is chosen, the median operator is obtained.

Attention
For [Height](#) and [Width](#) only odd values > 3 are valid. If invalid parameters are chosen they are converted automatically (without raising an exception handling) to the next larger even values.

<i>Parameter</i>
▷ Region (input_object) region(-array) $\leadsto$ <i>Hobject</i> Region(s) to be transformed. ▷ RegionCount (output_object) region(-array) $\leadsto$ <i>Hobject</i> Resulting region(s). ▷ Width (input_control) extent.x $\leadsto$ <i>integer</i> Width of the filter mask. Default Value : 15 Suggested values : $\text{Width} \in \{3, 5, 7, 9, 11, 13, 15, 17, 19, 21\}$ Typical range of values : $3 \leq \text{Width} \leq 511$ (lin) Minimum Increment : 2 Recommended Increment : 2 Restriction : $(\text{Width} \geq 3) \wedge \text{odd}(\text{Width})$ ▷ Height (input_control) extent.y $\leadsto$ <i>integer</i> Height of the filter mask. Default Value : 15 Suggested values : $\text{Height} \in \{3, 5, 7, 9, 11, 13, 15, 17, 19, 21\}$ Typical range of values : $3 \leq \text{Height} \leq 511$ (lin) Minimum Increment : 2 Recommended Increment : 2 Restriction : $(\text{Height} \geq 3) \wedge \text{odd}(\text{Height})$ ▷ Number (input_control) integer $\leadsto$ <i>integer</i> Minimum number of points lying within the filter mask. Default Value : 70 Suggested values : $\text{Number} \in \{5, 10, 20, 40, 60, 80, 90, 120, 150, 200\}$ Typical range of values : $1 \leq \text{Number} \leq 1000$ (lin) Minimum Increment : 1 Recommended Increment : 10 Restriction : $\text{Number} > 0$

Example

```

read_image(Image, 'affe')
mean_image(Image, Mean, 5, 5)
dyn_threshold(Mean, Points, 25)
rank_region(Points, Textur, 15, 15, 30)
gen_circle(Mask, 10, 10, 3)
opening1(Textur, Mask, Seg).

```

Complexity

Let F be the area of the input region. Then the runtime complexity is $O(F * 8)$.

Result

rank_region returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via **set_system('no_object_result', <Result>)** and the behavior in case of an empty input region via **set_system('empty_region_result', <Result>)**. If necessary, an exception handling is raised.

Parallelization Information

rank_region is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

threshold, **connection**, **regiongrowing**, **pouring**, **class_ndim_norm**

Possible Successors

select_shape, **disp_region**

Alternatives

closing_rectangle1, **expand_region**

See also

rank_image, **mean_image**

Module

Region processing

remove_noise_region (InputRegion : OutputRegion : Type :)

Remove noise from a region.

remove_noise_region removes noise from a region. In mode 'n_4', a structuring element consisting of the four neighbors of a point is generated. A dilation with this structuring element is performed, and the intersection of the result and the input region is calculated. Thus all pixels having no 4-connected neighbor are removed.

Parameter

- ▷ **InputRegion** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be modified.
- ▷ **OutputRegion** (output_object) region(-array) $\leadsto$ *Hobject*
Less noisy regions.
- ▷ **Type** (input_control) string $\leadsto$ *string*
Mode of noise removal.
Default Value : 'n_4'
List of values : Type $\in$ {'n_4', 'n_8', 'n_48'}

Complexity

Let F be the area of the input region. Then the runtime complexity is $O(\sqrt{F} * 4)$.

Result

remove_noise_region returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via **set_system('no_object_result', <Result>)**. If necessary, an exception handling is raised.

Parallelization Information

remove_noise_region is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors
connection , regiongrowing , pouring , class_ndim_norm
Possible Successors
select_shape
See also
dilation1 , intersection , gen_region_points
Module
Region processing

shape_trans (Region : RegionTrans : Type :)

Transform the shape of a region.

[shape_trans](#) transforms the shape of the input regions depending on the parameter [Type](#):

'convex' Convex hull.

'ellipse' Ellipse with the same moments and area as the input region.

'outer_circle' Smallest enclosing circle.

'inner_circle' Largest circle fitting into the region.

'rectangle1' Smallest enclosing rectangle parallel to the coordinate axes.

'rectangle2' Smallest enclosing rectangle.

'inner_center' The point on the skeleton of the input region having the smallest distance to the center of gravity of the input region.

Parameter
▷ Region (input_object) region(-array) $\leadsto$ <i>Hobject</i> Regions to be transformed. ▷ RegionTrans (output_object) region(-array) $\leadsto$ <i>Hobject</i> Transformed regions. ▷ Type (input_control) string $\leadsto$ <i>string</i> Type of transformation. Default Value : 'convex' List of values : Type $\in$ {'convex', 'ellipse', 'outer_circle', 'inner_circle', 'rectangle1', 'rectangle2', 'inner_center'}

Complexity
Let F be the area of the input region. Then the runtime complexity is $O(F)$.

Result
shape_trans returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via set_system('no_object_result', <Result>) . If necessary, an exception handling is raised.

Parallelization Information
shape_trans is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).

Possible Predecessors
connection , regiongrowing
Possible Successors
disp_region , regiongrowing_mean , area_center
See also
convexity , elliptic_axis , area_center , smallest_rectangle1 , smallest_rectangle2 , set_shape , select_shape , inner_circle
Module
Region processing

skeleton (Region : Skeleton : :)

Compute the skeleton of a region.

skeleton computes the skeleton of the input regions.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Region to be thinned.
 - ▷ **Skeleton** (output_object) region(-array) $\leadsto$ *Hobject*
Resulting skeleton.
- Number of elements** : Skeleton = Region

Complexity

Let F be the area of the enclosing rectangle of the input region. Then the runtime complexity is $O(F)$ (per region).

Result

skeleton returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior in case of empty input (no regions given) can be set via **set_system('no_object_result', <Result>)** and the behavior in case of an empty input region via **set_system('empty_region_result', <Result>)**. If necessary, an exception handling is raised.

Parallelization Information

skeleton is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

sobel_amp, **edges_image**, **bandpass_image**, **threshold**, **hysteresis_threshold**

Possible Successors

junctions_skeleton, **pruning**

Alternatives

morph_skeleton, **thinning**

See also

gray_skeleton, **sobel_amp**, **edges_image**, **roberts**, **bandpass_image**, **threshold**

References

Eckardt, U. "Verdünnung mit Perfekten Punkten", Proceedings 10. DAGM-Symposium, IFB 180, Zurich, 1988

Module

Region processing

sort_region (Regions : SortedRegions : SortMode, Order, RowOrCol :)

Sorting of regions with respect to their relative position.

The operator **sort_region** sorts the regions with respect to their relative position. All sorting methods with the exception of 'character' use one point of the region. With the help of the parameter **RowOrCol** = 'row' these points will be sorted according to their row and then according to their column. By using 'column', the column value will be used first. The following values are available for the parameter **SortMode**:

- 'character' The regions will be treated like characters in a row and will be sorted according to their order in the line: If two regions overlap horizontally, they will be sorted with respect to their column values, otherwise they will be sorted with regard to their row values.
- 'first_point' The point with the lowest column value in the first row of the region.
- 'last_point' The point with the highest column value in the last row of the region.
- 'upper_left' Upper left corner of the surrounding rectangle.
- 'upper_right' Upper right corner of the surrounding rectangle.
- 'lower_left' Lower left corner of the surrounding rectangle.
- 'lower_right' Lower right corner of the surrounding rectangle.

The parameter **Order** determines whether the sorting order is increasing or decreasing: using *'true'* the order will be increasing, using *'false'* the order will be decreasing.

Parameter	
▷ Regions (input_object)	region-array $\leadsto$ Hobject Regions to be sorted.
▷ SortedRegions (output_object)	region-array $\leadsto$ Hobject Sorted regions.
▷ SortMode (input_control)	string $\leadsto$ string Kind of sorting. Default Value : 'first_point' List of values : SortMode $\in$ {'character', 'first_point', 'last_point', 'upper_left', 'lower_left', 'upper_right', 'lower_right'}
▷ Order (input_control)	string $\leadsto$ string Increasing or decreasing sorting order. Default Value : 'true' List of values : Order $\in$ {'true', 'false'}
▷ RowOrCol (input_control)	string $\leadsto$ string Sorting first with respect to row, then to column. Default Value : 'row' List of values : RowOrCol $\in$ {'row', 'column'}

Result
If the parameters are correct, the operator **sort_region** returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Parallelization Information
sort_region is *reentrant* and processed *without* parallelization.

Possible Successors
do_ocr_multi, **do_ocr_single**

Module
Region processing

split_skeleton_lines (SkeletonRegion : : MaxDistance : BeginRow, BeginCol, EndRow, EndCol)

Split lines represented by one pixel wide, non-branching lines.

split_skeleton_lines splits lines represented by one pixel wide, non-branching regions into shorter lines based on their curvature. A line is split if the maximum distance of a point on the line to the line segment connecting its end points is larger than **MaxDistance** (split & merge algorithm). The start and end points of the approximating line segments are returned in **BeginRow**, **BeginCol**, **EndRow**, and **EndCol**.

Attention
The input regions must represent non-branching lines, that is single branches of the skeleton.

Parameter	
▷ SkeletonRegion (input_object)	region-array $\leadsto$ Hobject Input lines (represented by 1 pixel wide, non-branching regions).
▷ MaxDistance (input_control)	integer $\leadsto$ integer Maximum distance of the line points to the line segment connecting both end points. Default Value : 3 Suggested values : MaxDistance $\in$ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} Typical range of values : $1 \leq \text{MaxDistance} \leq 500$ (lin) Minimum Increment : 1 Recommended Increment : 1
▷ BeginRow (output_control)	line.begin.y-array $\leadsto$ integer Row coordinates of the start points of the output lines.

- ▷ **BeginCol** (output_control) line.begin.x-array $\leadsto$ *integer*
Column coordinates of the start points of the output lines.
- ▷ **EndRow** (output_control) line.end.y-array $\leadsto$ *integer*
Row coordinates of the end points of the output lines.
- ▷ **EndCol** (output_control) line.end.x-array $\leadsto$ *integer*
Column coordinates of the end points of the output lines.

Example

```
read_image(Image, 'fabrik')
edges_image (Image, ImaAmp, ImaDir, 'lanser2', 0.5, 'nms', 8, 16)
threshold (ImaAmp, RawEdges, 8, 255)
skeleton (RawEdges, Skeleton)
junctions_skeleton (Skeleton, EndPoints, JuncPoints)
difference (Skeleton, JuncPoints, SkelWithoutJunc)
connection (SkelWithoutJunc, SingleBranches)
select_shape (SingleBranches, SelectedBranches, 'area', 'and', 16, 99999)
split_skeleton_lines (SelectedBranches, 3, BeginRow, BeginCol, EndRow,
                     EndCol).
```

Result

[split_skeleton_lines](#) always returns the value 2 (H.MSG.TRUE). The behavior in case of empty input (no regions given) can be set via [set_system\('no_object_result', <Result>\)](#), the behavior in case of an empty input region via [set_system\('empty_region_result', <Result>\)](#), and the behavior in case of an empty result region via [set_system\('store_empty_region', <true/false>\)](#). If necessary, an exception handling is raised.

Parallelization Information

[split_skeleton_lines](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[connection](#), [select_shape](#), [skeleton](#), [junctions_skeleton](#), [difference](#)

Possible Successors

[select_lines](#), [partition_lines](#), [disp_line](#)

See also

[split_skeleton_region](#), [detect_edge_segments](#)

Module

Region processing

split_skeleton_region (
 SkeletonRegion : RegionLines : MaxDistance :)

Split lines represented by one pixel wide, non-branching regions.

[split_skeleton_region](#) splits lines represented by one pixel wide, non-branching regions into shorter lines based on their curvature. A line is split if the maximum distance of a point on the line to the line segment connecting its end points is larger than [MaxDistance](#) (split & merge algorithm). However, not the approximating lines are returned, but rather the original lines split into several output regions.

Attention

The input regions must represent non-branching lines, that is single branches of the skeleton.

Parameter

- ▷ **SkeletonRegion** (input_object) region(-array) $\leadsto$ *Hobject*
Input lines (represented by 1 pixel wide, non-branching regions).
- ▷ **RegionLines** (output_object) region-array $\leadsto$ *Hobject*
Split lines.

- ▷ **MaxDistance** (input_control) integer $\leadsto$ integer
 Maximum distance of the line points to the line segment connecting both end points.
Default Value : 3
Suggested values : MaxDistance $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$
Typical range of values : $1 \leq \text{MaxDistance} \leq 500$ (lin)
Minimum Increment : 1
Recommended Increment : 1

Example

```
read_image(Image, 'fabrik')
edges_image (Image, ImaAmp, ImaDir, 'lanser2', 0.5, 'nms', 8, 16)
threshold (ImaAmp, RawEdges, 8, 255)
skeleton (RawEdges, Skeleton)
junctions_skeleton (Skeleton, EndPoints, JuncPoints)
difference (Skeleton, JuncPoints, SkelWithoutJunc)
connection (SkelWithoutJunc, SingleBranches)
select_shape (SingleBranches, SelectedBranches, 'area', 'and', 16, 99999)
split_skeleton_region (SelectedBranches, Lines, 3)
```

Result

[split_skeleton_region](#) always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via [set_system\('no-object-result', <Result>\)](#), the behavior in case of an empty input region via [set_system\('empty-region-result', <Result>\)](#), and the behavior in case of an empty result region via [set_system\('store-empty-region', <true/false>\)](#). If necessary, an exception handling is raised.

Parallelization Information

[split_skeleton_region](#) is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

[connection](#), [select_shape](#), [skeleton](#), [junctions_skeleton](#), [difference](#)

Possible Successors

[count_obj](#), [select_shape](#), [select_obj](#), [area_center](#), [elliptic_axis](#),
[smallest_rectangle2](#), [get_region_polygon](#), [get_region_contour](#)

See also

[split_skeleton_lines](#), [get_region_polygon](#), [gen_polygons_xld](#)

Module

Region processing

Chapter 13

Segmentation

auto_threshold (Image : Regions : Sigma :)

Segment an image using thresholds determined from its histogram.

auto_threshold segments a single-channel image using multiple thresholding. First the relative histogram of the gray values is determined. Then relevant minima are extracted from the histogram, which are used successively as parameters for a thresholding operation. The thresholds used are 0, 255, and all minima extracted from the histogram (after the histogram has been smoothed). For each gray value interval *one* region is generated. Thus, the number of regions is the number of minima + 1. The larger the value of **Sigma** is chosen, the less regions will be extracted. This operator is particularly suited if the regions to be extracted exhibit similar gray values (homogeneous regions).

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ Hobject : byte
Image to be segmented.
- ▷ **Regions** (output_object) region-array $\leadsto$ Hobject
Regions with gray values within the automatically determined intervals.
- ▷ **Sigma** (input_control) number $\leadsto$ real / integer
Sigma for the Gaussian smoothing of the histogram.
Default Value : 2.0
Suggested values : $\text{Sigma} \in \{0.0, 0.5, 1.0, 2.0, 3.0, 4.0, 5.0\}$
Typical range of values : $0.0 \leq \text{Sigma} \leq 100.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.3
Restriction : $\text{Sigma} \geq 0.0$

Example

```
#include "HalconCpp.h"
#include <iostream.h>

int main (int argc, char *argv[])
{
    if (argc != 2)
    {
        cout << "Usage : " << argv[0] << " 'image' " << endl;
        return (-1);
    }

    HImage      image (argv[1]),
               med;
    HWindow     w;
```

```

w.SetDraw ("margin");
w.SetColored (12);

image.Display (w);

med = image.MedianImage ("circle", 3, -3);
med.Display (w);

HRegionArray reg = med.AutoThreshold2 (2.0);
HRegionArray con = reg.Connection ();

cout << "Display image after AutoThreshold2 segmentation " << endl;
con.Display (w);
w.Click ();

return (0);
}

```

Parallelization Information

[auto_threshold](#) is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

[anisotrope_diff](#), [median_image](#), [illuminate](#)

Possible Successors

[connection](#), [select_shape](#), [select_gray](#)

Alternatives

[gray_histo](#), [smooth_func_1d_gauss](#), [threshold](#)

Module

Region processing

bin_threshold (Image : Region : :)

Segment a black-and-white image using an automatically determined threshold.

[bin_threshold](#) segments a single-channel gray value image using an automatically determined threshold. First the relative histogram of the gray values is determined. Then relevant minima are extracted from the histogram, which are used as parameters for a thresholding operation. In order to reduce the number of minima, the histogram is smoothed with a Gaussian, as in [auto_threshold](#). The mask size is enlarged until there is only one minimum in the smoothed histogram. The selected region contains the pixels with gray values from 0 to the minimum. This operator is particularly suited for the segmentation of dark characters on a light paper.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject*: byte / uint2
Image to be segmented.
- ▷ **Region** (output_object) region(-array) $\leadsto$ *Hobject*
Dark regions of the image.

Example

```

#include "HalconCpp.h"
#include <iostream.h>

int main (int argc, char *argv[])
{
    if (argc != 2)
    {
        cout << "Usage : " << argv[0] << " 'image' " << endl;
        return (-1);
    }
}

```

```

}

HImage      image (argv[1]),
            med;
HWindow     w;

w.SetDraw ("margin");
w.SetShape ("rectangle1");
w.SetColored (12);

image.Display (w);

HRegionArray reg = image.BinThreshold();
HRegionArray con = reg.Connection ();

cout << "Display image after BinThreshold segmentation " << endl;
con.Display (w);
w.Click ();

return (0);
}

```

Parallelization Information

[bin_threshold](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[anisotrope_diff](#), [median_image](#), [illuminate](#)

Possible Successors

[connection](#), [select_shape](#), [select_gray](#)

Alternatives

[auto_threshold](#), [char_threshold](#), [gray_histo](#), [smooth_func1d_gauss](#), [threshold](#)

Module

Region processing

char_threshold (Image, HistoRegion : Characters : Sigma,
Percent : Threshold)

Perform a threshold segmentation for extracting characters.

The main application of [char_threshold](#) is to segment single-channel images of dark characters on bright paper. The operator works as follows: First, a histogram of the gray values in the image [Image](#) is computed for the points in the region [HistoRegion](#). To eliminate noise, the histogram is smoothed with the given [Sigma](#) (Gaussian smoothing). In the histogram, the background (white paper) corresponds to a large peak at high gray values, while the characters form a small peak at low gray values. In contrast to the operator [bin_threshold](#), which locates the minimum between the two peaks, here the threshold for the segmentation is determined in relation to the maximum of the histogram, i.e., the background, with the following condition:

$$histogram[threshold] * 100.0 < histogram[maximum] * (100.0 - Percent)$$

For example, if you choose [Percent](#) = 95 the operator locates the gray value whose frequency is at most 5 percent of the maximum frequency. Because [char_threshold](#) assumes that the characters are darker than the background, the threshold is searched for “to the left” of the maximum.

In comparison to [bin_threshold](#), this operator should be used if there is no clear minimum between the histogram peaks corresponding to the characters and the background, respectively, or if there is no peak corresponding to the characters at all. This may happen, e.g., if the image contains only few characters or in the case of a non-uniform illumination.

<i>Parameter</i>	
▷ Image (input_object)	image(-array) $\leadsto$ <i>Hobject</i> : byte Image to be segmented.
▷ HistoRegion (input_object)	region $\leadsto$ <i>Hobject</i> Region in which the histogram is computed.
▷ Characters (output_object)	region(-array) $\leadsto$ <i>Hobject</i> Dark regions (characters).
▷ Sigma (input_control)	number $\leadsto$ <i>real</i> Sigma for the Gaussian smoothing of the histogram. Default Value : 2.0 Suggested values : $\text{Sigma} \in \{0.0, 0.5, 1.0, 2.0, 3.0, 4.0, 5.0\}$ Typical range of values : $0.0 \leq \text{Sigma} \leq 50.0$ (lin) Minimum Increment : 0.01 Recommended Increment : 0.2
▷ Percent (input_control)	number $\leadsto$ <i>real</i> / integer Percentage for the gray value difference. Default Value : 95 Suggested values : $\text{Percent} \in \{90, 92, 95, 96, 97, 98, 99, 99.5, 100\}$ Typical range of values : $0.0 \leq \text{Percent} \leq 100.0$ (lin) Minimum Increment : 0.1 Recommended Increment : 0.5
▷ Threshold (output_control)	integer(-array) $\leadsto$ <i>integer</i> Calculated threshold.

Example

```
#include "HalconCpp.h"
#include <iostream.h>

int main (int argc, char *argv[])
{
    if (argc != 2)
    {
        cout << "Usage : " << argv[0] << " 'image' " << endl;
        return (-1);
    }

    HImage          image (argv[1]),
                   med;
    HWindow          w;

    w.SetDraw ("margin");
    w.SetShape ("rectangle1");
    w.SetColored (12);

    image.Display (w);

    HRegionArray reg = image.CharThreshold(0,5,&Threshold);
    HRegionArray con = reg.Connection ();

    cout << "Display image after CharThreshold segmentation " << endl;
    con.Display (w);
    w.Click ();

    return (0);
}
```

Parallelization Information

[char_threshold](#) is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors
anisotrope_diff , median_image , illuminate
Possible Successors
connection , select_shape , select_gray
Alternatives
bin_threshold , auto_threshold , gray_histo , smooth_func1d_gauss , threshold
Module
Region processing

check_difference (*Image*, *Pattern* : *Selected* : *Mode*, *DiffLowerBound*, *DiffUpperBound*, *GrayOffset*, *AddRow*, *AddCol* :)

Compare two images pixel by pixel.

check_difference selects from the input image *Image* those pixels ($g_o = g_{\text{Image}}$), whose gray value difference to the corresponding pixels in *Pattern* is inside (outside) of the interval [*DiffLowerBound*, *DiffUpperBound*]. The pixels of *Pattern* are translated by (*AddRow*, *AddCol*) with respect to *Image*. Let g_p be the gray value from *Pattern* translated by (*AddRow*, *AddCol*) w.r.t. g_o .

If the selected mode *Mode* is 'diff_inside', a pixel g_o is selected if

$$\begin{aligned} g_o - g_p - \text{GrayOffset} &> \text{DiffLowerBound} & \text{and} \\ g_o - g_p - \text{GrayOffset} &< \text{DiffUpperBound}. \end{aligned}$$

If the mode is set to 'diff_outside', a pixel g_o is selected if

$$\begin{aligned} g_o - g_p - \text{GrayOffset} &\leq \text{DiffLowerBound} & \text{or} \\ g_o - g_p - \text{GrayOffset} &\geq \text{DiffUpperBound}. \end{aligned}$$

This test is performed for all points of the domain (region) of *Image*, intersected with the domain of the translated *Pattern*. All points fulfilling the above condition are aggregated in the output region. The two images may be of different size. Typically, *Pattern* is smaller than *Image*.

Parameter
▷ Image (input_object) image(-array) $\rightsquigarrow$ <i>Hobject</i> : byte Image to be examined. ▷ Pattern (input_object) image(-array) $\rightsquigarrow$ <i>Hobject</i> : byte Comparison image. ▷ Selected (output_object) region(-array) $\rightsquigarrow$ <i>Hobject</i> Points in which the two images are similar/different. ▷ Mode (input_control) string $\rightsquigarrow$ <i>string</i> Mode: return similar or different pixels. Default Value : 'diff_outside' Suggested values : $\text{Mode} \in \{\text{'diff\_inside'}, \text{'diff\_outside'}\}$ ▷ DiffLowerBound (input_control) number $\rightsquigarrow$ <i>integer</i> Lower bound of the tolerated gray value difference. Default Value : -5 Suggested values : $\text{DiffLowerBound} \in \{0, -1, -2, -3, -5, -7, -10, -12, -15, -17, -20, -25, -30\}$ Typical range of values : $-255 \leq \text{DiffLowerBound} \leq 255$ (lin) Minimum Increment : 1 Recommended Increment : 2 Restriction : $(-255 \leq \text{DiffLowerBound}) \wedge (\text{DiffLowerBound} \leq 255)$

- ▷ **DiffUpperBound** (input_control) number $\leadsto$ *integer*
Upper bound of the tolerated gray value difference.
Default Value : 5
Suggested values : DiffUpperBound $\in \{0, 1, 2, 3, 5, 7, 10, 12, 15, 17, 20, 25, 30\}$
Typical range of values : $-255 \leq \text{DiffUpperBound} \leq 255$ (lin)
Minimum Increment : 1
Recommended Increment : 2
Restriction : $(-255 \leq \text{DiffUpperBound}) \wedge (\text{DiffUpperBound} \leq 255)$
- ▷ **GrayOffset** (input_control) number $\leadsto$ *integer*
Offset gray value subtracted from Image.
Default Value : 0
Suggested values : GrayOffset $\in \{-30, -25, -20, -17, -15, -12, -10, -7, -5, -3, -2, -1, 0, 1, 2, 3, 5, 7, 10, 12, 15, 17, 20, 25, 30\}$
Typical range of values : $-255 \leq \text{GrayOffset} \leq 255$ (lin)
Minimum Increment : 1
Recommended Increment : 2
Restriction : $(-255 \leq \text{GrayOffset}) \wedge (\text{GrayOffset} \leq 255)$
- ▷ **AddRow** (input_control) point.y $\leadsto$ *integer*
Row coordinate, by which Pattern is translated.
Default Value : 0
Suggested values : AddRow $\in \{-200, -100, -20, -10, 0, 10, 20, 100, 200\}$
Typical range of values : $-32000 \leq \text{AddRow} \leq 32000$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **AddCol** (input_control) point.x $\leadsto$ *integer*
Column coordinate, by which Pattern is translated.
Default Value : 0
Suggested values : AddCol $\in \{-200, -100, -20, -10, 0, 10, 20, 100, 200\}$
Typical range of values : $-32000 \leq \text{AddCol} \leq 32000$ (lin)
Minimum Increment : 1
Recommended Increment : 1

Complexity

Let F be the number of valid pixels. Then the runtime complexity is $O(F)$.

Result

`check_difference` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags '`no_object_result`', '`empty_region_result`', and '`store_empty_region`' with `set_system`. If necessary, an exception is raised.

Parallelization Information

`check_difference` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

`connection`, `select_shape`, `reduce_domain`, `select_gray`, `rank_region`, `dilation1`, `opening`

Alternatives

`sub_image`, `dyn_threshold`

Module

Region processing

```
class 2dim_sup ( ImageCol, ImageRow,
FeatureSpace : RegionClass2Dim : : )
```

Segment an image using two-dimensional pixel classification.

`class_2dim_sup` classifies the points in two-channel images using a two-dimensional feature space. For each point, two gray values (one from each image) are used as features. The feature space is represented by the input region. The classification is done as follows:

A point from the input region of an image is accepted if the point (g_c, g_l) , which is determined by the respective gray values, is contained in the region **FeatureSpace**. g_l is here a gray value from the image **ImageRow**, while g_c is the corresponding gray value from **ImageCol**.

Let P be a point with the coordinates $P = (L, C)$, g_l be the gray value at position (L, C) in the image **ImageRow**, and g_c be the gray value at position (L, C) in the image **ImageCol**. Then the point P is aggregated into the output region if

$$(g_c, g_l) \in \text{FeatureSpace}$$

g_l is interpreted as row coordinate and g_c as column coordinate.

For the generation of **FeatureSpace**, see **histo_2dim**. The feature space can be modified by applying region transformation operators, such as **rank_region**, **dilation1**, **shape_trans**, **elliptic_axis**, etc., before calling **class_2dim_sup**.

The parameters **ImageCol** and **ImageRow** must contain an equal number of images with the same respective size. The image points are taken from the intersection of the domains of both images (see **reduce_domain**).

Parameter

- ▷ **ImageCol** (input_object) image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1
Input image (first channel).
- ▷ **ImageRow** (input_object) image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1
Input image (second channel).
- ▷ **FeatureSpace** (input_object) region(-array) $\leadsto$ *Hobject*
Region defining the feature space.
- ▷ **RegionClass2Dim** (output_object) region(-array) $\leadsto$ *Hobject*
Classified regions.

Example

```
#include "HalconCpp.h"
#include <iostream.h>

int main (int argc, char *argv[])
{
    if (argc != 2)
    {
        cout << "Usage : " << argv[0] << " 'image' " << endl;
        return (-1);
    }

    HRegion    feats, cd2reg;
    HImage     image (argv[1]),
              text1, text2,
              mean1, mean2,
              histo;

    HWindow    win;
    long       nc;

    if ((nc = image.CountChannels ()) != 3)
    {
        cout << argv[1] << " is not a rgb-image " << endl;
        return (-2);
    }

    image.Display (win);

    win.SetColor ("green");
    cout << "Draw the region of interest " << endl;
```

```

HRegion  region = win.DrawRegion ();

text1 = image.TextureLaws ("el", 2, 5);
mean1 = text1.MeanImage (21, 21);
text2 = mean1.TextureLaws ("es", 2, 5);
mean2 = text2.MeanImage (21, 21);

histo = region.Histo2dim (mean1, mean2);
feats = histo.Threshold (1.0, 1000000.0);

win.SetDraw ("fill");
win.SetColor ("red");

feats.Display (win);

cout << "Charakteristics area in red" << endl;

cd2reg = mean1.Class2dimSup (mean2, feats);

win.SetColor ("blue");
cd2reg.Display (win);

cout << "Result of classification in blue " << endl;
win.Click ();
return (0);
}

```

Complexity

Let F be the area of the input region. Then the runtime complexity is $O(256^2 + F)$.

Result

[class_2dim_sup](#) always returns 2 (H_MSG_TRUE). The behavior with respect to the input images and output regions can be determined by setting the values of the flags 'no_object_result', 'empty_region_result', and 'store_empty_region' with [set_system](#). If necessary, an exception is raised.

Parallelization Information

[class_2dim_sup](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[histo_2dim](#), [threshold](#), [draw_region](#), [dilation1](#), [opening](#), [shape.trans](#)

Possible Successors

[connection](#), [select_shape](#), [select_gray](#)

Alternatives

[class_ndim_norm](#), [class_ndim_box](#), [threshold](#), [histo_2dim](#)

Module

Region processing

class_2dim_unsup (Image1, Image2 : Classes : Threshold,
NumClasses :)

Segment two images by clustering.

[class_2dim_unsup](#) performs a classification with two single-channel images. First a two-dimensional histogram of the two images is computed ([histo_2dim](#)). In this histogram the first maximum is extracted; it serves as the first cluster center. The histogram is computed with the intersection of the domains of both images (see [reduce_domain](#)). After this, all pixels in the images, which are at most [Threshold](#) pixels from the cluster center in the maximum norm, are determined. These pixels form one output region. Next, the pixels thus classified are deleted from the histogram so that they are not taken into account for the next class. In this modified histogram

again the maximum is extracted; it again serves as a cluster center. The above steps are repeated **NumClasses** times; thus, **NumClasses** output regions result. Only pixels defined in both images are returned.

Attention

Both input images must have the same size.

Parameter

- ▷ **Image1** (input_object) image $\rightsquigarrow$ *Hobject* : byte
First input image.
- ▷ **Image2** (input_object) image $\rightsquigarrow$ *Hobject* : byte
Second input image.
- ▷ **Classes** (output_object) region-array $\rightsquigarrow$ *Hobject*
Segmentation result.
- ▷ **Threshold** (input_control) integer $\rightsquigarrow$ *integer*
Threshold (maximum distance to the cluster's center).
Default Value : 15
Suggested values : Threshold $\in \{0, 2, 5, 8, 12, 17, 20, 30, 50, 70\}$
- ▷ **NumClasses** (input_control) integer $\rightsquigarrow$ *integer*
Number of classes (cluster centers).
Default Value : 5
Suggested values : NumClasses $\in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 20, 30, 40, 50\}$

Example

```
#include "HalconCpp.h"
#include <iostream.h>

int main (int argc, char *argv[])
{
    if (argc != 2)
    {
        cout << "Usage : " << argv[0] << " 'image' " << endl;
        return (-1);
    }

    HImage    colimg (argv[1]),
              green, blue;

    HWindow  w;
    long     nc;

    if ((nc = colimg.CountChannels ()) != 3)
    {
        cout << argv[1] << " is not a rgb-image " << endl;
        return (-2);
    }

    colimg.Display (w);

    HImage      red = colimg.Decompose3 (&green, &blue);
    HRegionArray seg = red.Class2dimUnsup (green, 15, 5);

    w.SetDraw ("margin");
    w.SetColored (12);
    seg.Display (w);
    w.Click ();

    return (0);
}
```

Result

`class_2dim_unsup` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags '*no_object_result*', '*empty_region_result*', and '*store_empty_region*' with `set_system`. If necessary, an exception is raised.

Parallelization Information

`class_2dim_unsup` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`decompose2`, `decompose3`, `median_image`, `anisotrope_diff`, `reduce_domain`

Possible Successors

`select_shape`, `select_gray`, `connection`

Alternatives

`threshold`, `histo_2dim`, `class_2dim_sup`, `class_ndim_norm`, `class_ndim_box`

Module

Region processing

class_ndim_box (MultiChannelImage : Regions : ClassifHandle :)

Classify pixels using hyper-cuboids.

`class_ndim_box` classifies the pixels of the multi-channel image given in `MultiChannelImage`. To do so, the classifier `ClassifHandle` created with `create_class_box` is used. The classifier can be trained using `learn_ndim_box` or as described with `create_class_box`. More information on the structure of the classifier can be found also under that operator.

`MultiChannelImage` is a multi channel image. Its pixel values are used for the classification.

Parameter

- ▷ **MultiChannelImage** (input_object) multichannel-image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / int4 / real
Multi-channel input image.
- ▷ **Regions** (output_object) region-array $\leadsto$ *Hobject*
Segmentation result.
- ▷ **ClassifHandle** (input_control) class_box $\leadsto$ *integer*
Classifier's handle number.

Example

```
read_image(Bild, 'meer')
disp_image(Image, WindowHandle)
set_color(WindowHandle, 'green')
fwrite_string('Draw the learning region')
fnew_line()
draw_region(Reg1, WindowHandle)
reduce_domain(Image, Reg1, Foreground)
set_color(WindowHandle, 'red')
fwrite_string('Draw Background')
fnew_line()
draw_region(Reg2, WindowHandle)
reduce_domain(Image, Reg2, Background)
fwrite_string('Start to learn')
fnew_line()
create_classif(ClassifHandle)
class_ndim_box(Foreground, Background, Image, ClassifHandle)
fwrite_string('start to classify')
fnew_line()
class_ndim_box(Image, Res, ClassifHandle)
set_draw(WindowHandle, 'fill')
```

```
disp_region(Res,WindowHandle)
free_classif(ClassifHandle).
```

Complexity

Let N be the number of hyper-cuboids and F be the area of the input region. Then the runtime complexity is $O(N * F)$.

Result

`class_ndim_box` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags 'no_object_result', 'empty_region_result', and 'store_empty_region' with `set_system`. If necessary, an exception is raised.

Parallelization Information

`class_ndim_box` is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

`create_class_box`, `learn_class_box`, `median_image`, `compose2`, `compose3`, `compose4`

Alternatives

`class_ndim_norm`, `class_2dim_sup`

See also

`class_ndim_box`, `descript_class_box`, `create_class_box`

Module

Region processing

```
class_ndim_norm ( MultiChannelImage : Regions : Metric,
SingleMultiple, Radius, Center : )
```

Classify pixels using hyper-spheres or hyper-cubes.

`class_ndim_norm` classifies the pixels of the multi-channel image given in `MultiChannelImage`. The result is returned in `Regions` as one region per classification object. The metric used ('euclid' or 'maximum') is determined by `Metric`. This parameter must be set to the same value used in `learn_ndim_norm`. The parameter is used to determine whether one region ('single') or multiple regions ('multiple') have to be generated for each cluster. `Radius` determines the radii or half edge length of the clusters, respectively. `Center` determines their centers.

Parameter

- ▷ **MultiChannelImage** (input_object) multichannel-image(-array) $\leadsto$ *Hobject* : byte
Multi-channel input image.
- ▷ **Regions** (output_object) region-array $\leadsto$ *Hobject*
Segmentation result.
- ▷ **Metric** (input_control) string $\leadsto$ *string*
Metric to be used.
Default Value : 'euclid'
List of values : `Metric` $\in$ {'euclid', 'maximum'}
- ▷ **SingleMultiple** (input_control) string $\leadsto$ *string*
Return one region or one region for each cluster.
Default Value : 'single'
List of values : `SingleMultiple` $\in$ {'single', 'multiple'}
- ▷ **Radius** (input_control) number(-array) $\leadsto$ *real* / integer
Cluster radii or half edge lengths (returned by `learn_ndim_norm`).
- ▷ **Center** (input_control) number(-array) $\leadsto$ *real* / integer
Coordinates of the cluster centers (returned by `learn_ndim_norm`).

Example

```
#include "HalconCpp.h"
#include <iostream.h>
```

```

int main ()
{
    HImage    image ("meer"),
              t1, t2, t3,
              m1, m2, m3, m;

    HWindow   w;

    w.SetColor ("green");
    image.Display (w);

    cout << "Draw your region of interest " << endl;

    HRegion testreg = w.DrawRegion ();

    t1 = image.TextureLaws ("el", 2, 5);    m1 = t1.MeanImage (21, 21);
    t2 = image.TextureLaws ("es", 2, 5);    m2 = t2.MeanImage (21, 21);
    t3 = image.TextureLaws ("le", 2, 5);    m3 = t3.MeanImage (21, 21);

    m = m1.Compose3 (m2, m3);

    Tuple Metric = "euclid";
    Tuple Radius = 20.0;
    Tuple MinNum = 5;
    Tuple NbrCha = 3;

    HRegion empty;
    Tuple cen, t;

    Radius = testreg.LearnNdimNorm (empty, m, Metric, Radius,
                                   MinNum, NbrCha, &cen, &t);

    Tuple RegMod = "multiple";

    HRegionArray reg = m.ClassNdimNorm (Metric, RegMod, Radius, cen, NbrCha);

    w.SetColored (12);
    reg.Display (w);
    cout << "Result of classification" << endl;
    return (0);
}

```

Complexity

Let N be the number of clusters and F be the area of the input region. Then the runtime complexity is $O(N * F)$.

Result

[class_ndim_norm](#) returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags '*no_object_result*', '*empty_region_result*', and '*store_empty_region*' with [set_system](#). If necessary, an exception is raised.

Parallelization Information

[class_ndim_norm](#) is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

[learn_ndim_norm](#), [compose2](#), [compose3](#), [compose4](#)

Possible Successors

[connection](#), [select_shape](#), [reduce_domain](#), [select_gray](#)

Alternatives

[class_ndim_box](#), [class_2dim_sup](#)

See also

[disp_circle](#), [disp_rectangle1](#)

detect_edge_segments (Image : : SobelSize, MinAmplitude, MaxDistance, MinLength : BeginRow, BeginCol, EndRow, EndCol)

Detect straight edge segments.

detect_edge_segments detects straight edge segments in the gray image **Image**. The extracted edge segments are returned as line segments with start point (**BeginRow**,**BeginCol**) and end point (**EndRow**,**EndCol**). Edge detection is based on the Sobel filter, using 'sum_abs' as parameter and **SobelSize** as the filter mask size (see **sobel_amp**). Only pixels with a filter response larger than **MinAmplitude** are used as candidates for edge points. These thresholded edge points are thinned and split into straight segments. Due to technical reasons, edge points in which several edges meet are lost. Therefore, **detect_edge_segments** usually does not return closed object contours. The parameter **MaxDistance** controls the maximum allowed distance of an edge point to its approximating line. For efficiency reasons, the sum of the absolute values of the coordinate differences is used instead of the Euclidean distance. **MinLength** controls the minimum length of the line segments. Lines shorter than **MinLength** are not returned.

Parameter

- ▷ **Image** (input_object)(multichannel-)image(-array) $\leadsto$ *Hobject* : byte
Input image.
- ▷ **SobelSize** (input_control) integer $\leadsto$ *integer*
Mask size of the Sobel operator.
Default Value : 5
List of values : SobelSize $\in \{3, 5, 7, 9, 11, 13\}$
- ▷ **MinAmplitude** (input_control) integer $\leadsto$ *integer*
Minimum edge strength.
Default Value : 32
Suggested values : MinAmplitude $\in \{10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 80, 90, 100, 110\}$
Typical range of values : $1 \leq \text{MinAmplitude} \leq 255$
Minimum Increment : 1
Recommended Increment : 1
Restriction : MinAmplitude ≥ 0
- ▷ **MaxDistance** (input_control) integer $\leadsto$ *integer*
Maximum distance of the approximating line to its original edge.
Default Value : 3
Suggested values : MaxDistance $\in \{2, 3, 4, 5, 6, 7, 8\}$
Typical range of values : $1 \leq \text{MaxDistance} \leq 30$
Minimum Increment : 1
Recommended Increment : 1
Restriction : MaxDistance ≥ 0
- ▷ **MinLength** (input_control) integer $\leadsto$ *integer*
Minimum length of the resulting line segments.
Default Value : 10
Suggested values : MinLength $\in \{3, 5, 7, 9, 11, 13, 16, 20\}$
Typical range of values : $1 \leq \text{MinLength} \leq 500$
Minimum Increment : 1
Recommended Increment : 1
Restriction : MinLength ≥ 0
- ▷ **BeginRow** (output_control) line.begin.y-array $\leadsto$ *integer*
Row coordinate of the line segments' start points.
- ▷ **BeginCol** (output_control) line.begin.x-array $\leadsto$ *integer*
Column coordinate of the line segments' start points.
- ▷ **EndRow** (output_control) line.end.y-array $\leadsto$ *integer*
Row coordinate of the line segments' end points.

▷ **EndCol** (output_control) line.end.x-array $\leadsto$ *integer*
 Column coordinate of the line segments' end points.

Example

```
Htuple  SobelSize, MinAmplitude, MaxDistance, MinLength;
Htuple  RowBegin, ColBegin, RowEnd, ColEnd;

create_tuple(&SobelSize, 1);
set_i(SobelSize, 5, 0);
create_tuple(&MinAmplitude, 1);
set_i(MinAmplitude, 32, 0);
create_tuple(&MaxDistance, 1);
set_i(MaxDistance, 3, 0);
create_tuple(&MinLength, 1);
set_i(MinLength, 10, 0);
T_detect_edge_segments(Image, SobelSize, MinAmplitude, MaxDistance, MinLength,
                      &RowBegin, &ColBegin, &RowEnd, &ColEnd);
```

Result

detect_edge_segments returns 2 (H_MSG_TRUE) if all parameters are correct. If the input is empty the behaviour can be set via **set_system('no-object-result', <Result>)**. If necessary, an exception handling is raised.

Parallelization Information

detect_edge_segments is *reentrant* and automatically *parallelized* (on tuple level, channel level).

Possible Predecessors

sigma_image, **median_image**

Possible Successors

select_lines, **partition_lines**, **select_lines_longest**, **line_position**,
line_orientation

Alternatives

sobel_amp, **threshold**, **skeleton**

Module

Image filters

dual_threshold (Image : RegionCrossings : MinSize, MinGray,
 Threshold :)

Threshold operator for signed images.

dual_threshold segments the input image into a region with gray values $\geq$ **Threshold** (“positive” regions) and a region with gray values $\leq$ **-Threshold** (“negative” regions). “Positive” or “negative” regions having a size of less than **MinSize** are suppressed, as well as regions whose maximum gray value is less than **MinGray** in absolute value.

The segmentation performed is not complete, i.e., the “positive” and “negative” regions together do not necessarily cover the entire image: Areas with a gray value between **-Threshold** and **Threshold**, **-MinGray** and **MinGray** respectively, are not taken into account.

dual_threshold is usually called after applying a Laplace operator (**laplace**, **derivate_gauss** or **diff_of_gauss**) or the difference of two images (**sub_image**) to an image.

The zero crossings of a Laplace image correspond to edges in an image, and are the separating regions of the “positive” and “negative” regions in the Laplace image. They can be determined by calling **dual_threshold** with **Threshold** = 1. The parameter **MinGray** controls the noise invariance, while **MinSize** controls the resolution of the edge detection.

Using byte images only the positive part of the operator is applied. Therefore **dual_threshold** behaves like a standard threshold operator (**threshold**) with successive **connection** and **select_gray**.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte / int2 / int4 / real
Input Laplace image.
- ▷ **RegionCrossings** (output_object) region-array $\leadsto$ *Hobject*
"Positive" and "negative" regions.
- ▷ **MinSize** (input_control) integer $\leadsto$ *integer*
Regions smaller than MinSize are suppressed.
Default Value : 20
Suggested values : MinSize $\in \{0, 10, 20, 50, 100, 200, 500, 1000\}$
Typical range of values : $0 \leq \text{MinSize} \leq 10000$ (lin)
Minimum Increment : 1
Recommended Increment : 10
- ▷ **MinGray** (input_control) real $\leadsto$ *real*
Regions whose maximum absolute gray value is smaller than MinGray are suppressed.
Default Value : 5.0
Suggested values : MinGray $\in \{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 9.0, 11.0, 15.0, 20.0\}$
Typical range of values : $0.001 \leq \text{MinGray} \leq 10000.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
Restriction : MinGray > 0
- ▷ **Threshold** (input_control) real $\leadsto$ *real*
Regions which have a gray value larger than Threshold (or smaller than -Threshold) are suppressed.
Default Value : 2.0
Suggested values : Threshold $\in \{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 9.0, 11.0, 15.0, 20.0\}$
Typical range of values : $0.001 \leq \text{Threshold} \leq 10000.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
Restriction : $(\text{Threshold} \geq 1) \wedge (\text{Threshold} \leq \text{MinGray})$

Example

```

/* Edge detection with the Laplace operator (and edge thinning) */
diff_of_gauss(Image,Laplace,2.0,1.6)
/* find "'positive'" and "'negative'" regions: */
dual_threshold(Laplace,Region,20,2,1)
/*The zero runnings are the complement to these image section: */
complement('full',Region,Nulldurchgaenge).

/* Simulation of \OpRef{dual_threshold} */
dual_threshold(Laplace,Result,MinS,MinG,Threshold):
  threshold(Laplace,Tmp1,Threshold,999999)
  connection(Tmp1,Tmp2)
  select_shape(Tmp2,Tmp3,'area','and',MinS,999999)
  select_gray(Laplace,Tmp3,Tmp4,'max','and',MinG,999999)
  threshold(Laplace,Tmp5,-999999,-Threshold)
  connection(Tmp5,Tmp6)
  select_shape(Tmp6,Tmp7,'area','and',MinS,999999)
  select_gray(Laplace,Tmp7,Tmp8,'min','and',-999999,-MinG)
  concat_obj(Tmp4,Tmp8,Result).

```

Result

dual_threshold returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags 'no_object_result', 'empty_region_result', and 'store_empty_region' with [set_system](#). If necessary, an exception is raised.

Parallelization Information

dual_threshold is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[min_max_gray](#), [sobel_amp](#), [gauss_image](#), [reduce_domain](#), [diff_of_gauss](#), [sub_image](#), [derivate_gauss](#)

Possible Successors

[connection](#), [dilation1](#), [erosion1](#), [opening](#), [closing](#), [rank_region](#), [shape_trans](#), [skeleton](#)

Alternatives

[threshold](#), [connection](#), [select_shape](#), [select_gray](#), [dyn_threshold](#), [check_difference](#)

See also

[laplace](#), [diff_of_gauss](#), [expand_region](#)

Module

Region processing

dyn_threshold ([OrigImage](#), [ThresholdImage](#) : [RegionDynThresh](#) : [Offset](#), [LightDark](#) :)

Segment an image using a local threshold.

[dyn_threshold](#) selects from the input image those regions in which the pixels fulfill a threshold condition. Let $g_o = g_{\text{OrigImage}}$ and $g_m = g_{\text{ThresholdImage}}$. Then the condition for [LightDark](#) = 'light' is:

$$g_o \geq g_m + \text{Offset}$$

For [LightDark](#) = 'dark' the condition is:

$$g_o \leq g_m - \text{Offset}$$

For [LightDark](#) = 'equal' it is:

$$g_m - \text{Offset} \leq g_o \leq g_m + \text{Offset}$$

Finally, for [LightDark](#) = 'not_equal' it is:

$$g_m - \text{Offset} > g_o \vee g_o > g_m + \text{Offset}$$

This means that all points in [OrigImage](#) whose gray value is larger then or equal to the gray value in [ThresholdImage](#) plus an offset are aggregated into the resulting region.

Typically, the threshold images are smoothed versions of the original image (e.g., by applying [mean_image](#), [gauss_image](#), etc.). Then the effect of [dyn_threshold](#) is similar to applying [threshold](#) to a highpass-filtered version of the original image (see [highpass_image](#)).

With [dyn_threshold](#) contours of an object can be extracted, where the objects' size (diameter) is determined by the mask size of the lowpass filter and the amplitude of the objects' edges:

The larger the mask size is chosen, the larger the found regions get. As a rule of thumb, the mask size should be about twice the diameter of the objects to be extracted. It is important not to set the parameter [Offset](#) to zero because in this case too many small regions will be found (noise). Values between 5 and 40 are a sensible choice. The larger [Offset](#) is chosen, the smaller the extracted regions get.

All points of the input image fulfilling the above condition are stored jointly in one region. If necessary, the connected components can be obtained by calling [connection](#).

Attention

If [Offset](#) is chosen from -1 to 1 usually a very noisy region is generated, requiring large storage. If [Offset](#) is chosen too large (> 60, say) it may happen that no points fulfill the threshold condition (i.e., an empty region is returned). If [Offset](#) is chosen too small (< -60, say) it may happen that all points fulfill the threshold condition (i.e., a full region is returned).

Parameter

- ▷ **OrigImage** (input_object) image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4 / real
Image to be segmented.
- ▷ **ThresholdImage** (input_object) image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2 / int4 / real
Image containing the local thresholds.
- ▷ **RegionDynThresh** (output_object) region(-array) $\leadsto$ *Hobject*
Segmented regions.
- ▷ **Offset** (input_control) number $\leadsto$ *real* / integer
Offset added to ThresholdImage.
Default Value : 5.0
Suggested values : Offset $\in \{1.0, 3.0, 5.0, 7.0, 10.0, 20.0, 30.0\}$
Typical range of values : $-255.0 \leq \text{Offset} \leq 255.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 5
Restriction : $(-255 < \text{Offset}) \wedge (\text{Offset} < 255)$
- ▷ **LightDark** (input_control) string $\leadsto$ *string*
Extract light, dark or similar areas?
Default Value : 'light'
List of values : LightDark $\in \{\text{'dark'}, \text{'light'}, \text{'equal'}, \text{'not\_equal'}\}$

Example

```
/* Looking for regions with the diameter D */
mean_image(Image, Mean, D*2+1, D*2+1)
dyn_threshold(Image, Mean, Seg:5, 'light')
connection(Seg, Regions).
```

Complexity

Let F be the area of the input region. Then the runtime complexity is $O(F)$.

Result

dyn_threshold returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags 'no_object_result', 'empty_region_result', and 'store_empty_region' with **set_system**. If necessary, an exception is raised.

Parallelization Information

dyn_threshold is *reentrant* and automatically *parallelized* (on *tuple level*, *domain level*).

Possible Predecessors

mean_image, **smooth_image**, **gauss_image**

Possible Successors

connection, **select_shape**, **reduce_domain**, **select_gray**, **rank_region**, **dilation1**, **opening**

Alternatives

check_difference, **highpass_image**, **sub_image**, **threshold**

See also

mean_image, **smooth_image**, **gauss_image**, **connection**, **rank_region**, **dilation1**

Module

Region processing

expand_gray (Regions, Image,
ForbiddenArea : RegionExpand : Iterations, Mode, Threshold :)

Fill gaps between regions (depending on gray value or color) or split overlapping regions.

expand_gray closes gaps between the input regions, which resulted from the suppression of small regions in a segmentation operator, (mode 'image'), for example, or separates overlapping regions 'region'). Both uses result from the expansion of regions. The operator works by adding a one pixel wide "strip" to a region, in which the

gray values or color are different from the gray values or color of neighboring pixels on the region's border by at most **Threshold** (in each channel). For images of type 'cyclic' (e.g., direction images), also points with a gray value difference of at least $255 - \text{Threshold}$ are added to the output region.

The expansion takes place only in regions, which are designated as not "forbidden" (parameter **ForbiddenArea**). The number of iterations is determined by the parameter **Iterations**. By passing 'maximal', **expand_gray** iterates until convergence, i.e., until no more changes occur. By passing 0 for this parameter, all non-overlapping regions are returned. The two modes of operation ('image' and 'region') are different in the following ways:

'image' The input regions are expanded iteratively until they touch another region or the image border, or the expansion stops because of too high gray value differences. Because **expand_gray** processes all regions simultaneously, gaps between regions are distributed evenly to all regions with a similar gray value. Overlapping regions are split by distributing the area of overlap evenly to both regions.

'region' No expansion of the input regions is performed. Instead, only overlapping regions are split by distributing the area of overlap evenly to regions having a matching gray value or color.

Attention

Because regions are only expanded into areas having a matching gray value or color, usually gaps will remain between the output regions, i.e., the segmentation is not complete.

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ *Hobject*
Regions for which the gaps are to be closed, or which are to be separated.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject*: byte / cyclic
Image (possibly multi-channel) for gray value or color comparison.
- ▷ **ForbiddenArea** (input_object) region $\leadsto$ *Hobject*
Regions in which no expansion takes place.
- ▷ **RegionExpand** (output_object) region(-array) $\leadsto$ *Hobject*
Expanded or separated regions.
- ▷ **Iterations** (input_control) string $\leadsto$ *string* / integer
Number of iterations.
Default Value : 'maximal'
Suggested values : $\text{Iterations} \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, \text{'maximal'}\}$
Typical range of values : $1 \leq \text{Iterations} \leq 500$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Expansion mode.
Default Value : 'image'
List of values : $\text{Mode} \in \{\text{'image'}, \text{'region'}\}$
- ▷ **Threshold** (input_control) integer(-array) $\leadsto$ *integer*
Maximum difference between the gray value or color at the region's border and a candidate for expansion.
Default Value : 32
Suggested values : $\text{Threshold} \in \{5, 10, 15, 20, 25, 30, 40, 50\}$
Typical range of values : $1 \leq \text{Threshold} \leq 255$ (lin)
Minimum Increment : 1
Recommended Increment : 5

Example

```
#include <iostream.h>
#include "HalconCpp.h"

int main (int argc, char *argv[])
{
    HImage    image (argv[1]);
    HRegion   empty_region;
    HWindow   win;
```

```

image.Display (win);

HRegionArray seg = (image >= 100).Connection ();

seg.Display (win);
HRegionArray exp = seg.ExpandGray1 (image, empty_region,
                                     "maximal", "image", 32);

win.SetDraw ("margin");
win.SetColored (12);
exp.Display (win);
win.Click ();

return (0);
}

```

Result

expand_gray always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via **set_system('no_object_result', <Result>)**, the behavior in case of an empty input region via **set_system('empty_region_result', <Result>)**, and the behavior in case of an empty result region via **set_system('store_empty_region', <true/false>)**. If necessary, an exception handling is raised.

Parallelization Information

expand_gray is *reentrant* and processed *without* parallelization.

Possible Predecessors

connection, **regiongrowing**, **pouring**, **class_ndim_norm**

Possible Successors

select_shape

See also

expand_gray_ref, **expand_region**

Module

Region processing

expand_gray_ref (Regions, Image,
ForbiddenArea : RegionExpand : Iterations, Mode, RefGray, Threshold :)

Fill gaps between regions (depending on gray value or color) or split overlapping regions.

expand_gray_ref closes gaps between the input regions, which resulted from the suppression of small regions in a segmentation operator, (mode *'image'*), for example, or separates overlapping regions *'region'*). Both uses result from the expansion of regions. The operator works by adding a one pixel wide “strip” to a region, in which the gray values or color are different from a reference gray value or color by at most **Threshold** (in each channel). For images of type *'cyclic'* (e.g., direction images), also points with a gray value difference of at least 255 – **Threshold** are added to the output region.

The expansion takes place only in regions, which are designated as not “forbidden” (parameter **ForbiddenArea**). The number of iterations is determined by the parameter **Iterations**. By passing *'maximal'*, **expand_gray_ref** iterates until convergence, i.e., until no more changes occur. By passing 0 for this parameter, all non-overlapping regions are returned. The two modes of operation (*'image'* and *'region'*) are different in the following ways:

'image' The input regions are expanded iteratively until they touch another region or the image border, or the expansion stops because of too high gray value differences. Because **expand_gray_ref** processes all regions simultaneously, gaps between regions are distributed evenly to all regions with a similar gray value. Overlapping regions are split by distributing the area of overlap evenly to both regions.

'region' No expansion of the input regions is performed. Instead, only overlapping regions are split by distributing the area of overlap evenly to regions having a matching gray value or color.

Attention

Because regions are only expanded into areas having a matching gray value or color, usually gaps will remain between the output regions, i.e., the segmentation is not complete.

Parameter

- ▷ **Regions** (input_object) region(-array) $\leadsto$ *Hobject*
Regions for which the gaps are to be closed, or which are to be separated.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject*: byte / cyclic
Image (possibly multi-channel) for gray value or color comparison.
- ▷ **ForbiddenArea** (input_object) region $\leadsto$ *Hobject*
Regions in which no expansion takes place.
- ▷ **RegionExpand** (output_object) region(-array) $\leadsto$ *Hobject*
Expanded or separated regions.
- ▷ **Iterations** (input_control) string $\leadsto$ *string* / integer
Number of iterations.
Default Value : 'maximal'
Suggested values : Iterations $\in$ {'maximal', 1, 2, 3, 4, 5, 7, 10, 15, 20, 30, 50, 70, 100, 150, 200, 300, 500}
Typical range of values : $1 \leq \text{Iterations} \leq 500$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Expansion mode.
Default Value : 'image'
List of values : Mode $\in$ {'image', 'region'}
- ▷ **RefGray** (input_control) integer(-array) $\leadsto$ *integer*
Reference gray value or color for comparison.
Default Value : 128
Suggested values : RefGray $\in$ {1, 10, 20, 50, 100, 128, 200, 255}
Typical range of values : $1 \leq \text{RefGray} \leq 255$ (lin)
Minimum Increment : 1
Recommended Increment : 10
- ▷ **Threshold** (input_control) integer(-array) $\leadsto$ *integer*
Maximum difference between the reference gray value or color and a candidate for expansion.
Default Value : 32
Suggested values : Threshold $\in$ {4, 10, 15, 20, 25, 30, 40}
Typical range of values : $1 \leq \text{Threshold} \leq 255$ (lin)
Minimum Increment : 1
Recommended Increment : 5

Example

```
#include <iostream.h>
#include "HalconCpp.h"

int main (int argc, char *argv[])
{
    HImage    image (argv[1]);
    HRegion   empty_region;
    HWindow   win;

    win.SetDraw ("margin");
    win.SetColored (12);

    image.Display (win);

    HRegionArray seg = (image >= 100).Connection ();
    seg.Display (win);
}
```



```

Tuple iter = "maximal";
Tuple mode = "image";
Tuple refg = 128;
Tuple thrs = 32;

HRegionArray exp = seg.ExpandGrayRef (image, empty_region,
                                     iter, mode, refg, thrs);

exp.Display (win);
win.Click ();

return (0);
}

```

Result

[expand_gray_ref](#) always returns the value 2 (H_MSG_TRUE). The behavior in case of empty input (no regions given) can be set via [set_system\('no_object_result', <Result>\)](#), the behavior in case of an empty input region via [set_system\('empty_region_result', <Result>\)](#), and the behavior in case of an empty result region via [set_system\('store_empty_region', <true/false>\)](#). If necessary, an exception handling is raised.

Parallelization Information

[expand_gray_ref](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[connection](#), [regiongrowing](#), [pouring](#), [class_ndim_norm](#)

Possible Successors

[select_shape](#)

See also

[expand_gray](#), [expand_region](#)

Module

Region processing

expand_line (Image : RegionExpand : Index, ExpandType, RowColumn, Threshold :)

Expand a region starting at a given line.

[expand_line](#) generates a region by expansion, starting at a given line (row or column). The expansion is terminated when the current gray value differs by more than [Threshold](#) from the mean gray value along the line ([ExpandType](#) = 'mean') or from the previously added gray value ([ExpandType](#) = 'gradient').

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte
Image to be segmented.
- ▷ **RegionExpand** (output_object) region(-array) $\leadsto$ *Hobject*
Extracted segments.
- ▷ **Index** (input_control) integer $\leadsto$ *integer*
Row or column index.
Default Value : 256
Suggested values : Index $\in \{16, 64, 128, 200, 256, 300, 400, 511\}$
Restriction : Index ≥ 0
- ▷ **ExpandType** (input_control) string $\leadsto$ *string*
Stopping criterion.
Default Value : 'gradient'
List of values : ExpandType $\in \{'gradient', 'mean'\}$
- ▷ **RowColumn** (input_control) string $\leadsto$ *string*
Segmentation mode (row or column).
Default Value : 'row'
List of values : RowColumn $\in \{'row', 'column'\}$

- ▷ **Threshold** (input_control) number $\leadsto$ real / integer
 Threshold for the expansion.
Default Value : 3.0
Suggested values : Threshold $\in \{0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 13.0, 17.0, 20.0, 30.0\}$
Typical range of values : $1.0 \leq \text{Threshold} \leq 255.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 1.0
Restriction : $(\text{Threshold} \geq 0.0) \wedge (\text{Threshold} \leq 255.0)$

Example

```
#include <iostream.h>
#include "HalconCpp.h"

int main (int argc, char *argv[])
{
    HImage    image (argv[1]),
              gauss;
    HWindow  win;

    win.SetDraw ("margin");
    win.SetColored (12);

    image.Display (win);

    gauss = image.GaussImage (5);

    HRegionArray reg = gauss.ExpandLine (100, "mean", "row", 5.0);

    reg.Display (win);
    win.Click ();

    return (0);
}
```

Parallelization Information

[expand_line](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[gauss_image](#), [smooth_image](#), [anisotrope_diff](#), [median_image](#), [affine_trans_image](#),
[rotate_image](#)

Possible Successors

[intersection](#), [opening](#), [closing](#)

Alternatives

[regiongrowing_mean](#), [expand_gray](#), [expand_gray_ref](#)

Module

Region processing

fast_threshold (Image : Region : MinGray, MaxGray, MinHeight :)

Fast selection of grayvalues within a given gray intervall.

[fast_threshold](#) selects the pixels from the input image whose gray values g fulfill the following condition:

$$\text{MinGray} \leq g \leq \text{MaxGray} .$$

To reduce procesing time, the selection is done in two steps: At first all pixels along rows with distances [MinHeight](#) are processed. In the next step the neighborhood (size [MinHeight](#) $\times$ [MinHeight](#)) of all previously selected points are processed.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte / uint2 / direction / cyclic
Image to be thresholded.
- ▷ **Region** (output_object) region(-array) $\leadsto$ *Hobject*
Regions with gray values lying in the specified interval.
- ▷ **MinGray** (input_control) number $\leadsto$ *real* / integer
Lower threshold for the gray values.
Default Value : 128
Suggested values : $\text{MinGray} \in \{0.0, 10.0, 30.0, 64.0, 128.0, 200.0, 220.0, 255.0\}$
Typical range of values : $0.0 \leq \text{MinGray} \leq 255.0$ (lin)
Minimum Increment : 1
Recommended Increment : 5.0
- ▷ **MaxGray** (input_control) number $\leadsto$ *real* / integer
Upper threshold for the gray values.
Default Value : 255.0
Suggested values : $\text{MaxGray} \in \{0.0, 10.0, 30.0, 64.0, 128.0, 200.0, 220.0, 255.0\}$
Typical range of values : $0.0 \leq \text{MaxGray} \leq 255.0$ (lin)
Minimum Increment : 1
Recommended Increment : 5.0
- ▷ **MinHeight** (input_control) number $\leadsto$ *integer*
Minimum height of objects to be extracted.
Default Value : 20
Suggested values : $\text{MinHeight} \in \{5, 10, 15, 20, 25, 30, 40, 50, 60, 70, 100\}$
Typical range of values : $2 \leq \text{MinHeight} \leq 200$ (lin)
Minimum Increment : 1
Recommended Increment : 2

Complexity

Let A be the area of the output region and *height* the height of **Image**. Then the runtime complexity is $O(A + \text{height}/\text{MinHeight})$.

Result

fast_threshold returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags '*no_object_result*', '*empty_region_result*', and '*store_empty_region*' with **set_system**. If necessary, an exception is raised.

Parallelization Information

fast_threshold is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

histo_to_thresh, **min_max_gray**, **sobel_amp**, **gauss_image**, **reduce_domain**, **fill_interlace**

Possible Successors

connection, **dilation1**, **erosion1**, **opening**, **closing**, **rank_region**, **shape_trans**, **skeleton**

Alternatives

threshold

See also

class_2dim_sup, **hysteresis_threshold**, **dyn_threshold**

Module

Region processing

histo_to_thresh (: : Histogramm, Sigma : MinThresh, MaxThresh)

Determine gray value thresholds from a histogram.

histo_to_thresh determines gray value thresholds from a histogram for a segmentation of an image using **threshold**. The thresholds returned are 0, 255, and all minima extracted from the histogram. Before the thresholds are determined the histogram is smoothed with a Gaussian.

Parameter

- ▷ **Histogramm** (input_control) histogram-array $\leadsto$ integer / real
Gray value histogram.
- ▷ **Sigma** (input_control) number $\leadsto$ real
Sigma for the Gaussian smoothing of the histogram.
Default Value : 2.0
Suggested values : $\text{Sigma} \in \{0.5, 1.0, 2.0, 3.0, 4.0, 5.0\}$
Typical range of values : $0.1 \leq \text{Sigma} \leq 30.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.2
- ▷ **MinThresh** (output_control) integer-array $\leadsto$ integer
Minimum thresholds.
- ▷ **MaxThresh** (output_control) integer-array $\leadsto$ integer
Maximum thresholds.

Example

```
#include <iostream.h>
#include "HalconCpp.h"

int main (int argc, char *argv[])
{
    if (argc != 2)
    {
        cout << "Usage : " << argv[0] << " <name of image>" << endl;
        return (-1);
    }

    HImage    image (argv[1]),
              Smoothed;
    HWindow   win;

    Tuple MinThres, MaxThres,
           HistoAbs, HistoRel,
           size    = 10,
           iter    = 3,
           thresh  = 0.0;

    HRegionArray reg = image.GetDomain ();

    HistoAbs = reg.GrayHisto (image, &HistoRel);
    Smoothed = HistoAbs.FunctionSmoothMean (size, iter);
    MinThres = Smoothed.HistoToThresh (thresh, &MaxThres);

    HRegionArray seg = image.Threshold (MinThres, MaxThres);
    HRegionArray con = seg.Connection ();

    /* Alternativkonstrukt fuer Threshold() in
       Aufrufkombination mit Connection()
       -----
       HRegionArray con = ((image >= MinThres) &
                          (image <= MaxThres)).Connection ();
       ----- */

    con.Display (win);
    win.Click ();

    return (0);
}
```

<i>Parallelization Information</i>
<code>histo_to_thresh</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
<i>Possible Predecessors</i>
<code>gray_histo</code>
<i>Possible Successors</i>
<code>threshold</code>
<i>Alternatives</i>
<code>auto_threshold</code>
<i>Module</i>
Region processing

hysteresis_threshold (Image : RegionHysteresis : Low, High, MaxLength :)

Perform a hysteresis threshold operation on an image.

`hysteresis_threshold` performs a hysteresis threshold operation (due to Canny) on an image. All points in the input image `Image` having a gray value larger than or equal to `High` are immediately accepted (“secure” points). Conversely, all points with gray values less than `Low` are immediately rejected. “Potential” points with gray values between both thresholds are accepted if they are connected to “secure” points by a path of “potential” points having a length of at most `MaxLength` points. This means that “secure” points influence their surroundings (hysteresis). The gray values of the input images remain unchanged. Only the regions of the image may get smaller.

<i>Parameter</i>
▷ Image (input_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / uint2 Image to be segmented. ▷ RegionHysteresis (output_object) region(-array) $\leadsto$ <i>Hobject</i> Segmentation result. ▷ Low (input_control) integer $\leadsto$ integer Lower threshold for the gray values. Default Value : 30 Suggested values : $\text{Low} \in \{5, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100\}$ Typical range of values : $0 \leq \text{Low} \leq 255$ (lin) Minimum Increment : 1 Recommended Increment : 5 Restriction : $(0 < \text{Low}) \wedge (\text{Low} < 255)$ ▷ High (input_control) integer $\leadsto$ integer Upper threshold for the gray values. Default Value : 60 Suggested values : $\text{High} \in \{5, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120, 130\}$ Typical range of values : $0 \leq \text{High} \leq 255$ (lin) Minimum Increment : 1 Recommended Increment : 5 Restriction : $((0 < \text{High}) \wedge (\text{High} < 255)) \wedge (\text{High} > \text{Low})$ ▷ MaxLength (input_control) integer $\leadsto$ integer Maximum length of a path of “potential” points to reach a “secure” point. Default Value : 10 Suggested values : $\text{MaxLength} \in \{1, 2, 3, 5, 7, 10, 12, 14, 17, 20, 25, 30, 35, 40, 50\}$ Typical range of values : $1 \leq \text{MaxLength} \leq 1000$ (lin) Minimum Increment : 1 Recommended Increment : 5 Restriction : $\text{MaxLength} > 1$

<i>Result</i>

`hysteresis_threshold` returns 2 (H_MSG.TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags ‘no_object_result’, ‘empty_region_result’, and ‘store_empty_region’ with `set_system`. If necessary, an exception is raised.

Parallelization Information

`hysteresis_threshold` is *reentrant* and automatically *parallelized* (on tuple level).

Alternatives

`dyn_threshold`, `threshold`, `class_2dim_sup`

See also

`edges_image`, `sobel_amp`, `background_seg`

References

J. Canny, "Finding Edges and Lines in Images"; Report, AI-TR-720, M.I.T. Artificial Intelligence Lab., Cambridge, MA, 1983.

Module

Region processing

learn_ndim_box (`Foreground`, `Background`,
`MultiChannelImage` : : `ClassifHandle` :)

Train the current classifier using a multi-channel image.

`learn_ndim_box` trains the classifier `ClassifHandle` with the gray values of `MultiChannelImage` using the points in `Foreground` as training sample. The points in `Background` are to be rejected by the classifier. The classifier trained thus can be used in `class_ndim_box` to segment multi-channel images. `Foreground` are the points that have to be found, `Background` contains the points which shall not be found.

Each pixel is trained once during the training process. For points in `Foreground` the class "0" is used, while for `Background` "1" is used. Pixels are trained by alternating points from `Foreground` with points from `Background`. If one region is smaller than the other, pixels are taken cyclically from the smaller region until the larger region is exhausted. `learn_ndim_box` later accepts only points which can be classified into class "0".

From a user's point of view the key difference between `learn_ndim_norm` and `learn_ndim_box` is that in the latter case the rejection class affects the classification process itself. Here, a hyper plane is generated, that separates `Foreground` and `Background` classes, so that no points in feature space are classified incorrectly. As for `learn_ndim_norm`, however, an overlap between `Foreground` and `Background` class is allowed. This has its effect on the return value Quality. The larger the overlap, the smaller this value.

Attention

All channels must be of the same type and have the same size.

Parameter

- ▷ **Foreground** (input_object) region(-array) $\leadsto$ *Hobject*
Foreground pixels to be trained.
- ▷ **Background** (input_object) region(-array) $\leadsto$ *Hobject*
Background pixels to be trained (rejection class).
- ▷ **MultiChannelImage** (input_object) multichannel-image(-array) $\leadsto$ *Hobject* : byte / direction /
cyclic / int1 / int2 / int4 / real
Multi-channel training image.
- ▷ **ClassifHandle** (input_control) class_box $\leadsto$ *integer*
Classifier's handle number.

Complexity

Let N be the number of generated hyper-cuboids and F be the area of the input region. Then the runtime complexity is $O(N * F)$.

Result

`class_ndim_box` returns 2 (H_MSG_TRUE) if all parameters are correct and there is an active classifier. The behavior with respect to the input images can be determined by setting the values of the flags '`no_object_result`' and '`empty_region_result`' with `set_system`. If necessary, an exception is raised.

Parallelization Information

`learn_ndim_box` is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

`create_class_box`, `draw_region`

Possible Successors

[class_ndim_box](#), [descript_class_box](#)

Alternatives

[learn_class_box](#)

Module

Region processing

learn_ndim_norm (Foreground, Background, Image : : Metric, Distance, MinNumberPercent : Radius, Center, Quality)

Construct clusters for [class_ndim_norm](#).

[learn_ndim_norm](#) generates classification clusters from the region [Foreground](#) and the corresponding gray values in the multi-channel image [Image](#), which can be used in [class_ndim_norm](#). [Background](#) determines a class of pixels not to be found in [class_ndim_norm](#). This parameter may be empty (empty tuple).

The parameter [Distance](#) determines the maximum distance [Radius](#) of the clusters. It describes the minimum distance between two cluster centers. If the parameter [Distance](#) is small the (small) hyper-cubes or hyper-spheres can approximate the feature space well. Simultaneously the runtime during classification increases.

The ratio of the number of pixels in a cluster to the total number of pixels (in percent) must be larger than the value of [MinNumberPercent](#), otherwise the cluster is not returned. [MinNumberPercent](#) serves to eliminate outliers in the training set. If it is chosen too large many clusters are suppressed.

Two different clustering procedures can be selected: The minimum distance algorithm (n-dimensional hyper-spheres) and the maximum algorithm (n-dimensional hyper-cubes) for describing the pixels of the image to classify in the n-dimensional histogram (parameter [Metric](#)). The Euclidian metric usually yields the better results, but takes longer to compute. The parameter [Quality](#) returns the quality of the clustering. It is a measure of overlap between the rejection class and the classifier classes. Values larger than 0 denote the corresponding ratio of overlap. If no rejection region is given, its value is set to 1. The regions in [Background](#) do not influence on the clustering. They are merely used to check the results that can be expected.

From a user's point of view the key difference between [learn_ndim_norm](#) and [learn_ndim_box](#) is that in the latter case the rejection class affects the classification process itself. Here, a hyper plane is generated, that separates [Foreground](#) and [Background](#) classes, so that no points in feature space are classified incorrectly. As for [learn_ndim_norm](#), however, an overlap between [Foreground](#) and [Background](#) class is allowed. This has its effect on the return value [Quality](#). The larger the overlap, the smaller this value.

Parameter

-
- ▷ **Foreground** (input_object) region(-array) $\leadsto$ Hobject
Foreground pixels to be trained.
 - ▷ **Background** (input_object) region(-array) $\leadsto$ Hobject
Background pixels to be trained (rejection class).
 - ▷ **Image** (input_object) image(-array) $\leadsto$ Hobject : byte
Multi-channel training image.
 - ▷ **Metric** (input_control) string $\leadsto$ string
Metric to be used.
Default Value : 'euclid'
List of values : `Metric` $\in$ { 'euclid', 'maximum' }
 - ▷ **Distance** (input_control) number $\leadsto$ real / integer
Maximum cluster radius.
Default Value : 10.0
Suggested values : `Distance` $\in$ { 1.0, 2.0, 3.0, 4.0, 6.0, 8.0, 10.0, 13.0, 17.0, 24.0, 30.0, 40.0 }
Typical range of values : $0.0 \leq \text{Distance} \leq 511.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 1.0
Restriction : `Radius` > 0.0
 - ▷ **MinNumberPercent** (input_control) number $\leadsto$ real / integer
The ratio of the number of pixels in a cluster to the total number of pixels (in percent) must be larger than

MinNumberPercent (otherwise the cluster is not output).

Default Value : 0.01

Suggested values : MinNumberPercent $\in \{0.001, 0.05, 0.1, 0.2, 0.5, 1.0, 2.0, 5.0, 10.0\}$

Typical range of values : $0.0 \leq \text{MinNumberPercent} \leq 100.0$ (lin)

Minimum Increment : 0.01

Recommended Increment : 0.1

Restriction : $(0 \leq \text{MinNumberPercent}) \wedge (\text{MinNumberPercent} \leq 100)$

- ▷ **Radius** (output_control) real-array $\leadsto$ real
Cluster radii of half edge lengths.
 - ▷ **Center** (output_control) real-array $\leadsto$ real
Coordinates of all cluster centers.
 - ▷ **Quality** (output_control) real $\leadsto$ real
Overlap of the rejection class with the classified objects (1: no overlapping).
- Assertion :** $(0 \leq \text{Quality}) \wedge (\text{Quality} \leq 1)$

Result

`learn_ndim_norm` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images can be determined by setting the values of the flags '`no_object_result`' and '`empty_region_result`' with `set_system`. If necessary, an exception is raised.

Parallelization Information

`learn_ndim_norm` is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

`min_max_gray`, `sobel_amp`, `gauss_image`, `reduce_domain`, `diff_of_gauss`

Possible Successors

`class_ndim_norm`, `connection`, `dilation1`, `erosion1`, `opening`, `closing`, `rank_region`, `shape_trans`, `skeleton`

See also

`class_ndim_norm`, `class_ndim_box`, `histo_2dim`

References

P. Haberäcker, "Digitale Bildverarbeitung"; Hanser-Studienbücher, München, Wien, 1987

Module

Region processing

local_max (Image : LocalMaxima : :)

Detect all local maxima in an image.

`local_max` extracts all points in an image having a gray value larger than the gray value of all its neighbors. The neighborhood used can be set by `set_system(:, 'neighborhood', <4/8>)`.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ Hobject : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Image to be processed.
 - ▷ **LocalMaxima** (output_object) region(-array) $\leadsto$ Hobject
Extracted local maxima as regions.
- Number of elements :** LocalMaxima = Image

Example

```
#include <iostream.h>
#include "HalconCpp.h"

int main (int argc, char *argv[])
{
    if (argc != 2)
    {
```



```

    cout << "Usage : " << argv[0] << " <name of image>" << endl;
    return (-1);
}

HImage  image (argv[1]);
HWindow win;

image.Display (win);

HImage      cres = image.CornerResponse (5, 0.04);
HRegionArray maxi = cres.LocalMax ();

win.SetColored (12);
maxi.Display (win);
win.Click ();

return (0);
}

```

Parallelization Information

[local_max](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[gauss_image](#), [smooth_image](#)

Possible Successors

[get_region_points](#), [connection](#)

Alternatives

[gray_skeleton](#), [nonmax_suppression_amp](#), [plateaus](#), [plateaus_center](#)

See also

[monotony](#), [topographic_sketch](#), [corner_response](#), [texture_laws](#)

Module

Region processing

nonmax_suppression_amp ([ImgAmp](#) : [ImageResult](#) : [Mode](#) :)

Suppress non-maximum points on an edge.

[nonmax_suppression_amp](#) suppresses in the regions of the image [ImgAmp](#) all points whose gray values are not local (directed) maxima. In contrast to [nonmax_suppression_dir](#), a direction image is not needed. Two modes of operation can be selected:

'hvnms' A point is labeled as a local maximum if its gray value is larger than or equal to the gray values within a search space of ± 5 pixels, either horizontally or vertically. Non-maximum points are removed from the region, gray values remain unchanged.

'loc_max' A point is labeled as a local maximum if its gray value is larger than or equal to the gray values of its eight neighbors.

Parameter

- ▷ **ImgAmp** (input_object) image(-array) $\leadsto$ *Hobject* : byte / uint2
Amplitude (gradient magnitude) image.
- ▷ **ImageResult** (output_object) image(-array) $\leadsto$ *Hobject* : byte / uint2
Image with thinned edge regions.
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Select horizontal/vertical or undirected NMS.
Default Value : 'hvnms'
List of values : Mode $\in$ { 'hvnms', 'loc_max' }

Result

`nonmax_suppression_amp` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags `'no_object_result'`, `'empty_region_result'`, and `'store_empty_region'` with `set_system`. If necessary, an exception is raised.

Parallelization Information

`nonmax_suppression_amp` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*).

Possible Predecessors

`sobel_amp`

Possible Successors

`threshold`, `hysteresis_threshold`

Alternatives

`gray_skeleton`, `local_max`, `gray_dilation_rect`

See also

`skeleton`

References

S.Lanser: "Detektion von Stufenkanten mittels rekursiver Filter nach Deriche"; Diplomarbeit; Technische Universität München, Institut für Informatik, Lehrstuhl Prof. Radig; 1991.

J.Canny: "Finding Edges and Rows in Images"; Report, AI-TR-720; M.I.T. Artificial Intelligence Lab., Cambridge, MA; 1983.

Module

Region processing

<code>nonmax_suppression_dir</code> (<code>ImgAmp</code> , <code>ImgDir</code> : <code>ImageResult</code> : <code>Mode</code> :)

Suppress non-maximum points on an edge.

`nonmax_suppression_dir` suppresses in the regions of the image `ImgAmp` all points whose gray values are not local (directed) maxima. `ImgDir` is a direction image giving the direction perpendicular for the local maximum (Unit: 2 degrees, i.e., 50 degrees are coded as 25 in the image). Such images are returned, for example, by `edges_image`. Two modes of operation can be selected:

'nms' Each point in the image is tested whether its gray value is a local maximum perpendicular to its direction. In this mode only the two neighbors closest to the given direction are examined. If one of the two gray values is greater than the gray of the point to be tested, it is suppressed (i.e., removed from the input region. The corresponding gray value remains unchanged).

'inms' Like 'nms'. However, the two gray values for the test are obtained by interpolation from four adjacent points.

Parameter

- ▷ **ImgAmp** (input_object) image(-array) $\leadsto$ *Hobject* : byte / uint2
Amplitude (gradient magnitude) image.
- ▷ **ImgDir** (input_object) image(-array) $\leadsto$ *Hobject* : direction
Direction image.
- ▷ **ImageResult** (output_object) image(-array) $\leadsto$ *Hobject* : byte / uint2
Image with thinned edge regions.
- ▷ **Mode** (input_control) string $\leadsto$ string
Select non-maximum-suppression or interpolating NMS.
Default Value : 'nms'
List of values : Mode $\in$ { 'nms', 'inms' }

Result

`nonmax_suppression_dir` returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags `'no_object_result'`, `'empty_region_result'`, and `'store_empty_region'` with `set_system`. If necessary, an exception is raised.

Parallelization Information

`nonmax_suppression_dir` is *reentrant* and automatically *parallelized* (on *tuple level*, *channel level*).

Possible Predecessors

`edges_image`, `sobel_dir`

Possible Successors

`threshold`, `hysteresis_threshold`

Alternatives

`nonmax_suppression_amp`, `gray_skeleton`, `gray_dilation_rect`

See also

`skeleton`

References

S.Lanser: "Detektion von Stufenkanten mittels rekursiver Filter nach Deriche"; Diplomarbeit; Technische Universität München, Institut für Informatik, Lehrstuhl Prof. Radig; 1991.

J.Canny: "Finding Edges and Rows in Images"; Report, AI-TR-720; M.I.T. Artificial Intelligence Lab., Cambridge; 1983.

Module

Region processing

plateaus (<code>Image</code> : <code>Plateaus</code> : :)

Detect all gray value plateaus.

`plateaus` extracts all points from `Image` with a gray value greater or equal to the gray value of its neighbors (8-neighborhood) and returns them in `Plateaus`. Each maximum is returned as a separate region.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Image to be processed.
- ▷ **Plateaus** (output_object) region-array $\leadsto$ *Hobject*
Extracted plateaus as regions (one region for each plateau).

Example

```
#include <iostream.h>
#include "HalconCpp.h"

int main (int argc, char *argv[])
{
    if (argc != 2)
    {
        cout << "Usage : " << argv[0] << " <name of image>" << endl;
        return (-1);
    }

    HImage    image (argv[1]);
    HWindow   win;

    image.Display (win);

    HImage      cres = image.CornerResponse (5, 0.04);
    HRegionArray maxi = cres.Plateaus ();

    win.SetColored (12);
    maxi.Display (win);
    win.Click ();
```

```
    return (0);
}
```

Parallelization Information

`plateaus` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`gauss_image`, `smooth_image`

Possible Successors

`area_center`, `get_region_points`, `select_shape`

Alternatives

`plateaus_center`, `gray_skeleton`, `nonmax_suppression_amp`, `local_max`

See also

`monotony`, `topographic_sketch`, `corner_response`, `texture_laws`

Module

Region processing

plateaus_center (Image : Plateaus : :)

Detect the centers of all gray value plateaus.

`plateaus_center` extracts all points from `Image` with a gray value greater or equal to the gray value of its neighbors (8-neighborhood) and returns them in `Plateaus`. If more than one of these points are connected (plateau), their center of gravity is returned. Each maximum is returned as a separate region.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real
Image to be processed.
- ▷ **Plateaus** (output_object) region-array $\leadsto$ *Hobject*
Centers of gravity of the extracted plateaus as regions (one region for each plateau).

Example

```
#include <iostream.h>
#include "HalconCpp.h"

int main (int argc, char *argv[])
{
    if (argc != 2)
    {
        cout << "Usage : " << argv[0] << " <name of image>" << endl;
        return (-1);
    }

    HImage  image (argv[1]);
    HWindow win;

    image.Display (win);

    HImage      cres = image.CornerResponse (5, 0.04);
    HRegionArray maxi = cres.PlateausCenter ();

    win.SetColored (12);
    maxi.Display (win);
    win.Click ();

    return (0);
}
```

Parallelization Information

`plateaus_center` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`gauss_image`, `smooth_image`

Possible Successors

`area_center`, `get_region_points`, `select_shape`

Alternatives

`plateaus`, `gray_skeleton`, `nonmax_suppression_amp`, `local_max`

See also

`monotony`, `topographic_sketch`, `corner_response`, `texture_laws`

Module

Region processing

<code>pouring</code> (<code>Image</code> : <code>Regions</code> : <code>Mode</code> , <code>MinGray</code> , <code>MaxGray</code> :)

Segment an image by “pouring water” over it.

`pouring` regards the input image as a “mountain range.” Larger gray values correspond to mountain peaks, while smaller gray values correspond to valley bottoms. `pouring` segments the input image in several steps. First the local maxima are extracted, i.e., pixels which either alone or in the form of an extended plateau have larger gray values than their immediate neighbors (in 4-neighborhood). In the next step, the maxima thus found are the starting points for an expansion until “valley bottoms” are reached. The expansion is done as long as there are chains of pixels in which the gray value gets smaller (like water running downhill from the maxima in all directions). Again, 4-neighborhood is used, but with a weaker condition (smaller or equal). This means that points at valley bottoms may belong to more than one maximum. These areas are at first not assigned to a region, but rather are split among all competing segments in the last step. The split is done by a uniform expansion of all involved segments, until all ambiguous pixels were assigned. The parameter `Mode` determines which steps are executed. The following values are possible:

‘all’ This is the normal mode of operation. All steps of the segmentation are performed. The regions are assigned to maxima, and overlapping regions are split.

‘maxima’ The segmentation only extracts the local maxima of the input image. No corresponding regions are extracted.

‘regions’ The segmentation extracts the local maxima of the input image and the corresponding regions, which are uniquely determined. Areas which could be assigned to more than one maximum are not split.

In order to prevent the algorithm from splitting a uniform background, which is different from the rest of the image, the parameters `MinGray` and `MaxGray` determine gray value thresholds for regions in the image which should be regarded as background. All parts of the image having a gray value smaller than `MinGray` or larger than `MaxGray` are disregarded for the extraction of the maxima as well as for the assignment of regions. For a complete segmentation of the image, `MinGray` = 0 und `MaxGray` = 255 should be selected. In any case, `MinGray` < `MaxGray` must be observed.

Parameter

- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte
Image to be segmented.
- ▷ **Regions** (output_object) region-array $\leadsto$ *Hobject*
Segmentation result.
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Mode of operation.
Default Value : ‘all’
List of values : `Mode` $\in$ { ‘all’, ‘maxima’, ‘regions’ }

- ▷ **MinGray** (input_control) integer $\leadsto$ integer
 All gray values smaller than this threshold are disregarded.
Default Value : 0
Suggested values : MinGray $\in \{0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110\}$
Typical range of values : $0 \leq \text{MinGray} \leq 256$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : MinGray ≥ 0
- ▷ **MaxGray** (input_control) integer $\leadsto$ integer
 All gray values larger than this threshold are disregarded.
Default Value : 255
Suggested values : MaxGray $\in \{100, 110, 120, 130, 140, 150, 160, 170, 180, 190, 200, 210, 220, 230, 240, 250, 255\}$
Typical range of values : $0 \leq \text{MaxGray} \leq 256$ (lin)
Minimum Increment : 1
Recommended Increment : 10
Restriction : $(\text{MaxGray} \leq 255) \wedge (\text{MaxGray} > \text{MinGray})$

Example

```
/* Segmentation of a filtering Image */
read_image(Image, 'br2')
mean_image(Image, Mean, 11, 11)
pouring(Mean, Seg, 'all', 0, 255)
disp_image(Mean, WindowHandle)
set_colored(WindowHandle, 12)
disp_region(Seg, WindowHandle).

/* Segmentation of a Image with masking of a dark background */
read_image(Image, 'hand')
mean_image(ImageMean, 15, 15)
pouring(Mean, Seg, 'all', 40, 255)
disp_image(Mean, WindowHandle)
set_colored(WindowHandle, 12)
disp_region(Seg, WindowHandle).

/* Segmentation of a histogram */
read_image(Image, 'monkey')
texture_laws(Image, Texture, 'el', 2, 5)
draw_region(Region, draw_region)
reduce_domain(Texture, Region, Testreg)
histo_2dim(Testreg, Texture, Region, Histo)
pouring(Histo, Seg, 'all', 0, 255).
```

Complexity

Let N be the number of pixels in the input image and M be the number of found segments, where the enclosing rectangle of the segment i contains m_i pixels. Furthermore, let K_i be the number of chords in segment i . Then the runtime complexity is

$$O(3 * N + \sum_M (3 * m_i) + \sum_M (K_i)) .$$

Result

pouring usually returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Parallelization Information

pouring is processed under *mutual exclusion* against itself and without parallelization.

Possible Predecessors

gauss_image, **smooth_image**

Alternatives

watersheds, **local_max**

See also

[histo_2dim](#), [expand_region](#), [expand_gray](#), [expand_gray_ref](#)

Module

Region processing

regiongrowing (Image : Regions : Row, Column, Tolerance, MinSize :)

Segment an image using *regiongrowing*.

regiongrowing segments images into regions of the same intensity — rastered into rectangles of size **Row** × **Column**. In order to decide whether two adjacent rectangles belong to the same region only the gray value of their center points is used. If the gray value difference is less then or equal to **Tolerance** the rectangles are merged into one region.

If g_1 und g_2 are two gray values to be examined, they are merged into the same region if:

$$|g_1 - g_2| < \text{**Tolerance**}$$

For images of type 'cyclic', the following formulas are used:

$$(|g_1 - g_2| < \text{**Tolerance**}) \wedge (|g_1 - g_2| \leq 127)$$

$$(256 - |g_1 - g_2| < \text{**Tolerance**}) \wedge (|g_1 - g_2| > 127)$$

For rectangles larger than one pixel, ususally the images should be smoothed with a lowpass filter with a size of at least **Row** × **Column** before calling **regiongrowing** (so that the gray values at the centers of the rectangles are “representative” for the whole rectangle). If the image contains little noise and the rectangles are small, the smoothing can be omitted in many cases. This, of course, makes the whole procedure faster.

The resulting regions are collections of rectangles of the chosen size **Row** × **Column**. Only regions containing at least **MinSize** points are returned.

Regiongrowing is a very fast operation, and thus suited for time-critical applications.

Attention

Column and **Row** are automatically converted to odd values if necessary.

Parameter

- ▷ **Image** (input_object) image(-array) $\rightsquigarrow$ *Hobject* : byte / direction / cyclic / int1 / int2 / int4 / real
Image to be segmented.
- ▷ **Regions** (output_object) region-array $\rightsquigarrow$ *Hobject*
Extracted segments.
- ▷ **Row** (input_control) extent.y $\rightsquigarrow$ *integer*
Vertical distance between tested pixels (height of the raster).
Default Value : 3
Suggested values : Row $\in \{1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21\}$
Typical range of values : $1 \leq \text{Row} \leq 99$ (lin)
Minimum Increment : 2
Recommended Increment : 2
Restriction : $(\text{Row} \geq 1) \wedge \text{odd}(\text{Row})$
- ▷ **Column** (input_control) extent.x $\rightsquigarrow$ *integer*
Horizontal distance between tested pixels (height of the raster).
Default Value : 3
Suggested values : Column $\in \{1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21\}$
Typical range of values : $1 \leq \text{Column} \leq 99$ (lin)
Minimum Increment : 2
Recommended Increment : 2
Restriction : $(\text{Column} \geq 1) \wedge \text{odd}(\text{Column})$

- ▷ **Tolerance** (input_control) number $\leadsto$ real / integer
 Points with a gray value difference less then or equal to tolerance are accumulated into the same object.
Default Value : 6.0
Suggested values : Tolerance $\in \{1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 12.0, 14.0, 18.0, 25.0\}$
Typical range of values : $1.0 \leq \text{Tolerance} \leq 127.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 1.0
Restriction : $(0 \leq \text{Tolerance}) \wedge (\text{Tolerance} < 127)$
- ▷ **MinSize** (input_control) integer $\leadsto$ integer
 Minimum size of the output regions.
Default Value : 100
Suggested values : MinSize $\in \{1, 5, 10, 20, 50, 100, 200, 500, 1000\}$
Typical range of values : $1 \leq \text{MinSize} \leq 1000$ (lin)
Minimum Increment : 1
Recommended Increment : 5
Restriction : MinSize ≥ 1

Example

```
read_image(Image, 'fabrik')
mean_image(Image, Mean, Row, Column)
regiongrowing(Mean, Result, Row, Column, 6.0, 100).
```

Complexity

Let N be the number of found regions and M the number of points in one of these regions. Then the runtime complexity is $O(N * \log(M) * M)$.

Result

regiongrowing returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags 'no_object_result', 'empty_region_result', and 'store_empty_region' with **set_system**. If necessary, an exception is raised.

Parallelization Information

regiongrowing is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

mean_image, **gauss_image**, **smooth_image**, **median_image**, **anisotrope_diff**

Possible Successors

select_shape, **reduce_domain**, **select_gray**

Alternatives

regiongrowing_n, **regiongrowing_mean**, **label_to_region**

Module

Region processing

regiongrowing_mean (Image : Regions : StartRows, StartColumns,
 Tolerance, MinSize :)

Regiongrowing using mean gray values.

regiongrowing_mean performs a regiongrowing using mean gray values of a region, starting from points given by **StartRows** and **StartColumns**. At any point in the process the mean gray value of the current region is calculated. Gray values at the boundary of the region are added to the region if they differ from the current mean by less than **Tolerance**. Regions smaller than **MinSize** are suppressed.

If no starting points are given (empty tuples), the expansion process starts at the upper leftmost point, and is continued with the first unprocessed point after a region has been created.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte / int4
Image to be segmented.
- ▷ **Regions** (output_object) region-array $\leadsto$ *Hobject*
Extracted segments.
- ▷ **StartRows** (input_control) point.y(-array) $\leadsto$ *integer*
Row coordinates of the starting points.
Default Value : '[]'
Typical range of values : $0 \leq \text{StartRows} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **StartColumns** (input_control) point.x(-array) $\leadsto$ *integer*
Column coordinates of the starting points.
Default Value : '[]'
Typical range of values : $0 \leq \text{StartColumns} \leq 511$ (lin)
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Tolerance** (input_control) number $\leadsto$ *real*
Maximum deviation from the mean.
Default Value : 5.0
Suggested values : Tolerance $\in \{0.5, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 12.0, 15.0, 17.0, 20.0, 25.0, 30.0, 40.0\}$
Typical range of values : $0.1 \leq \text{Tolerance} \leq 100.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 1.0
Restriction : Tolerance > 0.0
- ▷ **MinSize** (input_control) integer $\leadsto$ *integer*
Minimum size of a region.
Default Value : 100
Suggested values : MinSize $\in \{0, 10, 30, 50, 100, 500, 1000, 2000\}$
Typical range of values : $0 \leq \text{MinSize} \leq 50000$ (lin)
Minimum Increment : 1
Recommended Increment : 100
Restriction : MinSize ≥ 0

Result

[regiongrowing_mean](#) returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags 'no_object_result', 'empty_region_result', and 'store_empty_region' with [set_system](#). If necessary, an exception is raised.

Parallelization Information

[regiongrowing_mean](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[gauss_image](#), [sigma_image](#), [anisotrope_diff](#), [median_image](#)

Possible Successors

[select_shape](#), [reduce_domain](#), [opening](#), [expand_region](#)

Alternatives

[regiongrowing](#), [regiongrowing_n](#)

Module

Region processing

regiongrowing_n (MultiChannelImage : Regions : Metric, MinTolerance, MaxTolerance, MinSize :)

Regiongrowing for multi-channel images.

[regiongrowing_n](#) performs a multi-channel regiongrowing. The n channels give rise to an n -dimensional feature vector. Neighboring points are aggregated into the same region if the difference of their feature vectors

with respect to the given metric lies in the interval $[\text{MinTolerance}, \text{MaxTolerance}]$. Only neighbors of the 4-neighborhood are examined. The following metrics can be used:

Let g_A denote the gray value in the feature vector A at point a of the image, and likewise be g_B the gray value in the feature vector B at point a neighboring point b . Let $g(d)$ be the gray value with index d . Furthermore, let MinT denote MinTolerance and MaxT denote MaxTolerance .

'1-norm': Sum of absolute values

$$\text{MinT} \leq \frac{1}{n} \sum |g_A - g_B| \leq \text{MaxT}$$

'2-norm': Euclidian distance

$$\text{MinT} \leq \sqrt{\frac{\sum (g_A - g_B)^2}{n}} \leq \text{MaxT}$$

'3-norm': p - Norm with p = 3

$$\text{MinT} \leq \sqrt[3]{\frac{\sum (g_A - g_B)^3}{n}} \leq \text{MaxT}$$

'4-norm': p - Norm with p = 4

$$\text{MinT} \leq \sqrt[4]{\frac{\sum (g_A - g_B)^4}{n}} \leq \text{MaxT}$$

'n-norm': Minkowski distance

$$\text{MinT} \leq \sqrt[n]{\frac{\sum (g_A - g_B)^n}{n}} \leq \text{MaxT}$$

'max-diff': Supremum distance

$$\text{MinT} \leq \max \{|g_A - g_B|\} \leq \text{MaxT}$$

'min-diff': Infimum distance

$$\text{MinT} \leq \min \{|g_A - g_B|\} \leq \text{MaxT}$$

'variance': Variance of gray value differences

$$\text{MinT} \leq \text{Var}(g_A - g_B) \leq \text{MaxT}$$

'dot-product': Dot product

$$\text{MinT} \leq \frac{1}{n} \sqrt{\sum (g_A g_B)} \leq \text{MaxT}$$

'correlation': Correlation

$$\begin{aligned} m_A &= \frac{1}{n} \sum g_A \\ \text{Var}_A &= \frac{1}{n} \sqrt{\sum (g_A - m_A)^2} \\ m_B &= \frac{1}{n} \sum g_B \\ \text{Var}_B &= \frac{1}{n} \sqrt{\sum (g_B - m_B)^2} \\ \text{MinT} &\leq \frac{1}{n^2} \sum \frac{(g_A - m_A)(g_B - m_B)}{(\text{Var}_A \text{Var}_B)} \leq \text{MaxT} \end{aligned}$$

'mean-diff': Difference of arithmetic means

$$\begin{aligned} a &= \frac{1}{n} \sum g_A \\ b &= \frac{1}{n} \sum g_B \\ \text{MinT} &\leq |a - b| \leq \text{MaxT} \end{aligned}$$

'mean-ratio': Ratio of arithmetic means

$$a = \frac{1}{n} \sum g_A$$

$$b = \frac{1}{n} \sum g_B$$

$$\text{Min}T \leq \min \left\{ \frac{a}{b}, \frac{b}{a} \right\} \leq \text{Max}T$$

'length-diff': Difference of the vector lengths

$$a = \sqrt{\frac{\sum g_A^2}{n}}$$

$$b = \sqrt{\frac{\sum g_B^2}{n}}$$

$$\text{Min}T \leq |a - b| \leq \text{Max}T$$

'length-ratio': Ratio of the vector lengths

$$a = \sqrt{\frac{\sum g_A^2}{n}}$$

$$b = \sqrt{\frac{\sum g_B^2}{n}}$$

$$\text{Min}T \leq \min \left\{ \frac{a}{b}, \frac{b}{a} \right\} \leq \text{Max}T$$

'n-norm-ratio': Ratio of the vector lengths w.r.t the p-norm with $p = n$

$$a = \sqrt[n]{\frac{\sum g_A^n}{n}}$$

$$b = \sqrt[n]{\frac{\sum g_B^n}{n}}$$

$$\text{Min}T \leq \min \left\{ \frac{a}{b}, \frac{b}{a} \right\} \leq \text{Max}T$$

'gray-max-diff': Difference of the maximum gray values

$$a = \max \{|g_A|\}$$

$$b = \max \{|g_B|\}$$

$$\text{Min}T \leq |a - b| \leq \text{Max}T$$

'gray-max-ratio': Ratio of the maximum gray values

$$a = \max \{|g_A|\}$$

$$b = \max \{|g_B|\}$$

$$\text{Min}T \leq \min \left\{ \frac{a}{b}, \frac{b}{a} \right\} \leq \text{Max}T$$

'gray-min-diff': Difference of the minimum gray values

$$a = \min \{|g_A|\}$$

$$b = \min \{|g_B|\}$$

$$\text{Min}T \leq |a - b| \leq \text{Max}T$$

'gray-min-ratio': Ratio of the minimum gray values

$$a = \min \{|g_A|\}$$

$$b = \min \{|g_B|\}$$

$$MinT \leq \min \left\{ \frac{a}{b}, \frac{b}{a} \right\} \leq MaxT$$

'variance-diff': Difference of the variances over all gray values (channels)

$$MinT \leq |Var(g_A) - Var(g_B)| \leq MaxT$$

'variance-ratio': Ratio of the variances over all gray values (channels)

$$MinT \leq \frac{Var(g_B)}{Var(g_A)} \leq MaxT$$

'mean-abs-diff': Difference of the sum of absolute values over all gray values (channels)

$$a = \sum_{d,k,k < d} |g_A(d) - g_A(k)|$$

$$b = \sum_{d,k,k < d} |g_B(d) - g_B(k)|$$

$$MinT \leq \frac{|a - b|}{\text{Anzahl der Summen}} \leq MaxT$$

'mean-abs-ratio': Ratio of the sum of absolute values over all gray values (channels)

$$a = \sum_{d,k,k < d} |g_A(d) - g_A(k)|$$

$$b = \sum_{d,k,k < d} |g_B(d) - g_B(k)|$$

$$MinT \leq \min \left\{ \frac{a}{b}, \frac{b}{a} \right\} \leq MaxT$$

'max-abs-diff': Difference of the maximum distance of the components

$$a = \max \{g_A(d), g_A(k)\}$$

$$b = \max \{g_B(d), g_B(k)\}$$

$$MinT \leq |a - b| \leq MaxT$$

'max-abs-ratio': Ratio of the maximum distance of the components

$$a = \max \{g_A(d), g_A(k)\}$$

$$b = \max \{g_B(d), g_B(k)\}$$

$$MinT \leq \min \left\{ \frac{a}{b}, \frac{b}{a} \right\} \leq MaxT$$

'min-abs-diff': Difference of the minimum distance of the components

$$a = \min \{g_A(d), g_A(k)\}, k < d$$

$$b = \min \{g_B(d), g_B(k)\}, k < d$$

$$MinT \leq |a - b| \leq MaxT$$

'min-abs-ratio': Ratio of the minimum distance of the components

$$a = \min \{g_A(d), g_A(k)\}, k < d$$

$$b = \min \{g_B(d), g_B(k)\}, k < d$$

$$\text{Min}T \leq \min \left\{ \frac{a}{b}, \frac{b}{a} \right\} \leq \text{Max}T$$

'plane': The following has to hold for all $d_1, d_2 \in [1, n]$:

$$g_A(d_1) > g_A(d_2) \Rightarrow g_B(d_1) > g_B(d_2)$$

$$g_A(d_1) < g_A(d_2) \Rightarrow g_B(d_1) < g_B(d_2)$$

Regions with less than **MinSize** are suppressed.

Parameter	
▷ MultiChannelImage (input_object)	image(-array) $\leadsto$ <i>Hobject</i> : byte Multi-channel image to be segmented.
▷ Regions (output_object)	region-array $\leadsto$ <i>Hobject</i> Segmented regions.
▷ Metric (input_control)	string $\leadsto$ <i>string</i> Metric for the distance of the feature vectors. Default Value : '2-norm' List of values : Metric $\in$ {'1-norm', '2-norm', '3-norm', '4-norm', 'n-norm', 'max-diff', 'min-diff', 'variance', 'dot-product', 'correlation', 'mean-diff', 'mean-ratio', 'length-diff', 'length-ratio', 'n-norm-ratio', 'gray-max-diff', 'gray-max-ratio', 'gray-min-diff', 'gray-min-ratio', 'variance-diff', 'variance-ratio', 'mean-abs-diff', 'mean-abs-ratio', 'max-abs-diff', 'max-abs-ratio', 'min-abs-diff', 'min-abs-ratio', 'plane'}
▷ MinTolerance (input_control)	number $\leadsto$ <i>real</i> / integer Lower threshold for the features' distance. Default Value : 0.0 Suggested values : MinTolerance $\in$ {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 12.0, 14.0, 16.0, 18.0, 20.0, 25.0, 30.0} Typical range of values : $0.0 \leq \text{MinTolerance} \leq 255.0$ (lin) Minimum Increment : 0.01 Recommended Increment : 1.0
▷ MaxTolerance (input_control)	number $\leadsto$ <i>real</i> / integer Upper threshold for the features' distance. Default Value : 20.0 Suggested values : MaxTolerance $\in$ {0.0, 1.0, 2.0, 3.0, 4.0, 5.0, 6.0, 7.0, 8.0, 9.0, 10.0, 12.0, 14.0, 16.0, 18.0, 20.0, 25.0, 30.0} Typical range of values : $0.0 \leq \text{MaxTolerance} \leq 255.0$ (lin) Minimum Increment : 0.01 Recommended Increment : 1.0
▷ MinSize (input_control)	integer $\leadsto$ <i>integer</i> Minimum size of the output regions. Default Value : 30 Suggested values : MinSize $\in$ {1, 10, 25, 50, 100, 200, 500, 1000} Typical range of values : $1 \leq \text{MinSize} \leq 10000$ (lin) Minimum Increment : 1 Recommended Increment : 5

regiongrowing_n returns 2 (H_MSG_TRUE) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags 'no_object_result', 'empty_region_result', and 'store_empty_region' with **set_system**. If necessary, an exception is raised.

regiongrowing_n is *reentrant* and automatically *parallelized* (on *tuple level*).

compose2, **compose3**

	Alternatives
<code>class_2dim_sup</code> , <code>class_ndim_norm</code> , <code>class_ndim_box</code>	
	See also
<code>regiongrowing</code>	
	Module
Region processing	

threshold (Image : Region : MinGray, MaxGray :)

Select gray values lying within an interval.

`threshold` selects the pixels from the input image whose gray values g fulfill the following condition:

$$\text{MinGray} \leq g \leq \text{MaxGray} .$$

All points of an image fulfilling the condition are returned as one region. If more than one gray value interval is passed (tuples for `MinGray` and `MaxGray`), one separate region is returned for each interval.

	Parameter
▷ Image (input_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / dvf	
Image to be thresholded.	
▷ Region (output_object) region(-array) $\leadsto$ <i>Hobject</i>	
Regions with gray values lying in the specified interval.	
▷ MinGray (input_control) number(-array) $\leadsto$ <i>real</i> / integer	
Lower threshold for the gray values.	
Default Value : 128.0	
Suggested values : $\text{MinGray} \in \{0.0, 10.0, 30.0, 64.0, 128.0, 200.0, 220.0, 255.0\}$	
Typical range of values : $0.0 \leq \text{MinGray} \leq 255.0$ (lin)	
Minimum Increment : 0.01	
Recommended Increment : 5.0	
▷ MaxGray (input_control) number(-array) $\leadsto$ <i>real</i> / integer	
Upper threshold for the gray values.	
Default Value : 255.0	
Suggested values : $\text{MaxGray} \in \{0.0, 10.0, 30.0, 64.0, 128.0, 200.0, 220.0, 255.0\}$	
Typical range of values : $0.0 \leq \text{MaxGray} \leq 255.0$ (lin)	
Minimum Increment : 0.01	
Recommended Increment : 5.0	
Restriction : $\text{MaxGray} \geq \text{MinGray}$	

	Example

```

read_image(Image, 'fabrik')
sobel_dir(Image, EdgeAmp, EdgeDir, 'sum_abs', 3)
threshold(EdgeAmp, Seg, 50, 255, 2)
skeleton(Seg, Rand)
connection(Rand, Lines)
select_shape(Lines, Edges, 'area', 'and', 10, 1000000) .

```

	Complexity

Let F be the area of the input region. Then the runtime complexity is $O(F)$.

	Result

`threshold` returns 2 (`H_MSG_TRUE`) if all parameters are correct. The behavior with respect to the input images and output regions can be determined by setting the values of the flags `'no_object_result'`, `'empty_region_result'`, and `'store_empty_region'` with `set_system`. If necessary, an exception is raised.

	Parallelization Information

`threshold` is *reentrant* and automatically *parallelized* (on tuple level, domain level).

Possible Predecessors

[histo_to_thresh](#), [min_max_gray](#), [sobel_amp](#), [gauss_image](#), [reduce_domain](#),
[fill_interlace](#)

Possible Successors

[connection](#), [dilation1](#), [erosion1](#), [opening](#), [closing](#), [rank_region](#), [shape_trans](#),
[skeleton](#)

Alternatives

[class_2dim_sup](#), [hysteresis_threshold](#), [dyn_threshold](#)

See also

[dual_threshold](#), [zero_crossing](#), [background_seg](#), [regiongrowing](#)

Module

Region processing

threshold_sub_pix (Image : Border : Threshold :)

Extract level crossings from an image with subpixel accuracy.

[threshold_sub_pix](#) extracts the level crossings at the level [Threshold](#) of the input image [Image](#) with subpixel accuracy. The extracted level crossings are returned as XLD-contours in [Border](#). In contrast to the operator [threshold](#), [threshold_sub_pix](#) does not return regions, but the lines which separate regions with a gray value less than [Threshold](#) from regions with a gray value greater than [Threshold](#).

For the extraction, the input image is regarded as a surface, in which the gray values are interpolated bilinearly between the centers of the individual pixels. Consistent with the surface thus defined, level crossing lines are extracted for each pixel and linked into topologically sound contours. This means that the level crossing contours are correctly split at junction points. If the image contains extended areas of constant gray value [Threshold](#), only the border of such areas is returned as level crossings.

Parameter

- ▷ **Image** (input_object) singlechannel-image $\leadsto$ *Hobject* : byte / int1 / int2 / uint2 / int4 / real
Input image.
- ▷ **Border** (output_object) xld_cont-array $\leadsto$ *Hobject*
Extracted level crossings.
- ▷ **Threshold** (input_control) number $\leadsto$ *real* / integer
Threshold for the level crossings.
Default Value : 128
Suggested values : [Threshold](#) $\in$ {0.0, 10.0, 30.0, 64.0, 128.0, 200.0, 220.0, 255.0}

Example

```
read_image(Image, 'fabrik')
threshold_sub_pix(Image, Border, 35)
disp_xld(Border, windowHandle)
```

Result

[threshold_sub_pix](#) usually returns the value 2 (H_MSG.TRUE). If necessary, an exception is raised.

Parallelization Information

[threshold_sub_pix](#) is *reentrant* and processed *without* parallelization.

Alternatives

[threshold](#)

See also

[zero_crossing_sub_pix](#)

Module

Sub-pixel operators

watersheds (GrayImage : Basins, Watersheds : :)

Extract watersheds and basins from an image.

watersheds segments an image based on the topology of the gray values. The image is interpreted as a “mountain range.” Higher gray values correspond to “mountains,” while lower gray values correspond to “valleys.” In the resulting mountain range watersheds are extracted. These correspond to the bright ridges between dark basins. On output, the parameter **Basins** contains these basins, while **Watersheds** contains the watersheds, which are at least one pixel wide (points on the ridge which form a plateau). **Watersheds** always is a single region per input image, while **Basins** contains a separate region for each basin. It is advisable to apply a smoothing operator (e.g., **gauss_image**) to the input image before calling **watersheds** in order to reduce the number of output regions.

Attention

If the image contains many fine structures or is noisy, many output regions result, and thus the runtime increases considerably.

Parameter

- ▷ **GrayImage** (input_object) image(-array) $\leadsto$ *Hobject* : byte
Images to be segmented.
- ▷ **Basins** (output_object) region-array $\leadsto$ *Hobject*
Segments found (dark basins).
- ▷ **Watersheds** (output_object) region(-array) $\leadsto$ *Hobject*
Watersheds between the basins.

Example

```
#include <iostream.h>
#include "HalconCpp.h"

int main (int argc, char *argv[])
{
    HImage    image (argv[1]),
              invert, gauss;
    HWindow  win;

    cout << "Gauss of original " << endl;
    gauss = image.GaussImage (9);
    image.Display (win);

    cout << "Invert of Gauss " << endl;
    invert = gauss.InvertImage ();
    invert.Display (win);

    HRegion    watersheds;
    HRegionArray bassins = invert.Watersheds (&watersheds);

    win.SetColored (12);
    bassins.Display (win);
    win.Click ();

    return (0);
}
```

Result

watersheds always returns 2 (H_MSG_TRUE). The behavior with respect to the input images and output regions can be determined by setting the values of the flags *'no_object_result'*, *'empty_region_result'*, and *'store_empty_region'* with **set_system**. If necessary, an exception is raised.

Parallelization Information

watersheds is *reentrant* and automatically *parallelized* (on tuple level).

<i>Possible Predecessors</i>
gauss_image , smooth_image , invert_image
<i>Possible Successors</i>
expand_region , select_shape , reduce_domain , opening
<i>Alternatives</i>
pouring
<i>Module</i>
Region processing

zero_crossing (Image : RegionCrossing : :)

Detect zero crossings in an image.

[zero_crossing](#) returns the zero crossings of the input image as a region. A pixel is accepted as a zero crossing if its gray value (in [Image](#)) is zero, or if at least one of its neighbors of the 4-neighborhood has a different sign.

This operator is intended to be used after edge operators returning the second derivative of the image (e.g., [laplace_of_gauss](#)), which were possibly followed by a smoothing operator. In this case, the zero crossings are (candidates for) edges.

<i>Parameter</i>
▷ Image (input_object) image(-array) $\leadsto$ Hobject : int2 / int4 / real Input image. ▷ RegionCrossing (output_object) region(-array) $\leadsto$ Hobject Zero crossings (as region).

<i>Result</i>
zero_crossing usually returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

<i>Parallelization Information</i>
zero_crossing is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).

<i>Possible Predecessors</i>
laplace , laplace_of_gauss , derivate_gauss

<i>Possible Successors</i>
connection , skeleton , boundary , select_shape , fill_up

<i>Alternatives</i>
threshold , dual_threshold

<i>Module</i>
Region processing

zero_crossing_sub_pix (Image : ZeroCrossings : :)

Extract zero crossings from an image with subpixel accuracy.

[zero_crossing_sub_pix](#) extracts the zero crossings of the input image [Image](#) with subpixel accuracy. The extracted zero crossings are returned as XLD-contours in [ZeroCrossings](#). Thus, [zero_crossing_sub_pix](#) can be used as a sub-pixel precise edge extractor if the input image is a Laplace-filtered image (see [laplace](#), [laplace_of_gauss](#), [derivate_gauss](#)).

For the extraction, the input image is regarded as a surface, in which the gray values are interpolated bilinearly between the centers of the individual pixels. Consistent with the surface thus defined, zero crossing lines are extracted for each pixel and linked into topologically sound contours. This means that the zero crossing contours are correctly split at junction points. If the image contains extended areas of constant gray value 0, only the border of such areas is returned as zero crossings.

<i>Parameter</i>	
▷ Image (input_object)	singlechannel-image $\leadsto$ <i>Hobject</i> : int1 / int2 / int4 / real Input image.
▷ ZeroCrossings (output_object)	xld_cont-array $\leadsto$ <i>Hobject</i> Extracted zero crossings.

Example

```

/* Detection zero crossings of the Laplacian-of-Gaussian
   of an aerial image */
read_image(Image,'mreut')
derivate_gauss(Image,Laplace,3,'laplace')
zero_crossing_sub_pix(Laplace,ZeroCrossings)
disp_xld(ZeroCrossings,WindowHandle)

/* Detection of edges, i.e, zero crossings of the Laplacian-of-Gaussian
   that have a large gradient magnitude, in an aerial image */
read_image(Image,'mreut')
Sigma := 1.5
/* Compensate the threshold for the fact that derivate_gauss(...,'gradient')
   calculates a Gaussian-smoothed gradient, in which the edge amplitudes
   are too small because of the Gaussian smoothing, to correspond to a true
   edge amplitude of 20. */
Threshold := 20/(Sigma*sqrt(2*3.1415926))
derivate_gauss(Image,Gradient,Sigma,'gradient')
threshold(Gradient,Region,Threshold,255)
reduce_domain(Image,Region,ImageReduced)
derivate_gauss(ImageReduced,Laplace,Sigma,'laplace')
zero_crossing_sub_pix(Laplace,Edges)
disp_xld(Edges,WindowHandle)

```

Result

[zero_crossing_sub_pix](#) usually returns the value 2 (H_MSG_TRUE). If necessary, an exception is raised.

Parallelization Information

[zero_crossing_sub_pix](#) is *reentrant* and automatically *parallelized* (on *tuple* level).

Possible Predecessors

[laplace](#), [laplace_of_gauss](#), [diff_of_gauss](#), [derivate_gauss](#)

Alternatives

[zero_crossing](#)

Module

Sub-pixel operators

Chapter 14

System

14.1 Database

count_relation (: : RelationName : NumOfTuples)

Number of entries in the HALCON database.

The operator `count_relation` counts the number of entries in one of the four relations of the HALCON database. The HALCON database is organized as follows:

There are two basic relations for region-data and image-matrices. The HALCON objects region and image are constructed from elements from these two relations: a region consists of a pointer to a tuple in the region-data relation. An image consists also of a pointer to a tuple in the region-data relation (like a region) and additionally of one or more pointers to tuples in the matrix relation. If there is more than one matrix pointer, the image is called a multi-channel image.

Both regions and images are called objects. A region can be considered as the special case of an iconic object having no image matrices. For reasons of an efficient memory management, the tuples of the region-data relation and the image-matrix relation will be used by different objects together. Therefore there may be for example more images than image matrices. Only the two lowlevel relations are of relevance to the memory consumption. Image objects (regions as well as images) consist only of references on region and matrix data and therefore only need a couple of bytes of memory.

Possible values for `RelationName`:

'image': Image matrices. One matrix may also be the component of more than one image (no redundant storage).

'region': Regions (the full and the empty region are always available). One region may of course also be the component of more than one image object (no redundant storage).

'XLD': eXtended Line Description: Contours, Polygons, parallels, lines, etc. XLD data types don't have gray values and are stored with subpixel accuracy.

'object': Iconic objects. Composed of a region (called region) and optionally image matrices (called image).

'tuple': In the compact mode, tuples of iconic objects are stored as a surrogate in this relation. Instead of working with the individual object keys, only this tuple key is used. It depends on the host language, whether the objects are passed individually (Prolog and C++) or as tuples (C, Smalltalk, Lisp, OPS-5).

Certain database objects will be created already by the operator `reset_obj_db` and therefore have to be available all the time (the undefined gray value component, the objects 'full' (FULL_REGION in HALCON/C) and 'empty' (EMPTY_REGION in HALCON/C) as well as the herein included empty and full region). By calling `get_channel_info`, the operator therefore appears correspondingly also as 'creator' of the full and empty region. The procedure can be used for example to check the completeness of the `clear_obj` operation.

Parameter

- ▷ **RelationName** (input_control) string $\leadsto$ *string*
 Relation of interest of the HALCON database.
Default Value : 'object'
List of values : RelationName $\in$ {'image', 'region', 'XLD', 'object', 'tuple'}
- ▷ **NumOfTuples** (output_control) integer $\leadsto$ *integer*
 Number of tuples in the relation.

Example

```

reset_obj_db(512,512,3)
count_relation('image',I1)
count_relation('region',R1)
count_relation('XLD',X1)
count_relation('object',O1)
count_relation('tuple',T1)
read_image(X,'monkey')
count_relation('image',I2)
count_relation('region',R2)
count_relation('XLD',X2)
count_relation('object',O2)
count_relation('tuple',T2)

/*
Result:  I1 = 1  (undefined image)
         R1 = 2  (full and empty region)
         X1 = 0  (no XLD data)
         O1 = 2  (full and empty objects)
         T1 = 0  (always 0 in the normal mode )

         I2 = 2  (additionally the image 'monkey')
         R2 = 2  (read_image uses the full region)
         X2 = 0  (no XLD data)
         O2 = 3  (additionally the image object X)
         T2 = 0.

*/

```

Result

If the parameter is correct, the operator `count_relation` returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Parallelization Information

`count_relation` is *reentrant* and processed *without parallelization*.

Possible Predecessors

`reset_obj_db`

See also

`clear_obj`

Module

System

get_modules (: : : UsedModules, ModuleKey)

Query of used modules and the module key.

`get_modules` returns the module numbers of all operators used up to this point. Each operator belongs to one module (maximum 32). Each module has a name, which is returned in `UsedModules`. Based on the used modules, a key is generated that is needed for the licence manager. `get_modules` is normally called at the end of a programm to check the used modules.

Parameter
▷ UsedModules (output_control) string-array $\leadsto$ <i>string</i> Names of used modules. ▷ ModuleKey (output_control) integer $\leadsto$ <i>integer</i> Key for licence manager.
Parallelization Information
get_modules is <i>reentrant</i> and processed <i>without</i> parallelization.
Module
Operators not requiring licensing

reset_obj_db (: : DefaultImageWidth, DefaultImageHeight,
DefaultChannels :)

Initialization of the HALCON system.

The operator **reset_obj_db** initializes the HALCON system. With this procedure the four relations (grayvalue data, region data, iconic objects and object tuples) which are necessary for image processing with HALCON will be installed (see also **count_relation**). In case the relations already exist, all tuples in the relations will be deallocated!

The parameters **DefaultImageWidth** and **DefaultImageHeight** provide the initial values for the global maximum image size. If the first created object is an image, (e.g. **read_image**), the set values will be overruled in Standard-HALCON by the size of this picture. Instead of this, in Parallel HALCON the set values will only be changed, if they are smaller than the size of the created object. If on the other hand the first object to be created is a region, both in Standard- and in Parallel HALCON the values will only be adjusted in case the new image is larger than the set values. This is not only the case for the first image which is created or read: the global image size will always be enlarged, if larger images are created.

The global image size is of consequence for the opening of windows (**open_window**) and the clipping of regions. Whenever the clip mode is activated, regions will be clipped according to the global image size (**set_system** ('clip_region', 'true')). This can lead to problems if images of various sizes are used. In this case only the fact that a region is smaller or of the same size as the largest image can be guaranteed.

The parameter **DefaultChannels** returns the most frequent number of channels of an image object. This value can be set to 0 if for the most part regions are used. If more channels than those having been set at the initialization are necessary for one image, the number will be enlarged dynamically for this image. If less channels than those having been set at the initialization are necessary for the image, the superfluous channels will be set as undefined. For the user this will seem as if they were non existent, however, memory is allocated unnecessarily.

The parameter values can be queried using the operator **get_system**.

Attention

If the operator **reset_obj_db** is not called at the beginning of a HALCON session, HALCON will be initialized automatically by the operator **reset_obj_db(128,128,0)**. In case the operator **reset_obj_db** is called again, all image objects in the database will be deallocated.

Parameter
▷ DefaultImageWidth (input_control) integer $\leadsto$ <i>integer</i> Default image width (in pixels). Default Value : 128 Suggested values : DefaultImageWidth $\in$ {64, 128, 256, 512, 525, 1024} ▷ DefaultImageHeight (input_control) integer $\leadsto$ <i>integer</i> Default image height (in pixels). Default Value : 128 Suggested values : DefaultImageHeight $\in$ {64, 128, 256, 512, 768, 1024} ▷ DefaultChannels (input_control) integer $\leadsto$ <i>integer</i> Usual number of channels by which the system constant 'max_channels' is limited. Default Value : 0 Suggested values : DefaultChannels $\in$ {0, 1, 2, 3, 4, 5, 6, 7}

Result

The operator `reset_obj_db` returns the value 2 (H_MSG_TRUE) if the parameter values are correct. Otherwise an exception will be raised.

Parallelization Information

`reset_obj_db` is *reentrant* and processed *without* parallelization.

See also

`get_channel_info`, `count_relation`

Module

Operators not requiring licensing

14.2 Error-Handling

get_check (: : : Check)

State of the HALCON control modes.

Executing the operator `get_check` the user can inquire what kind of control modes are currently activated and which are not. `Check` gives the tuple containing the names of the control modes (see also `set_check`) which are preceded by a tilde (~, e.g. '~data'), if the corresponding control is deactivated.

Parameter

- ▷ **Check** (output_control) string-array $\leadsto$ *string*
 Tuple of the currently activated control modes.

Result

`get_check` always returns the value 2 (H_MSG_TRUE).

Parallelization Information

`get_check` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`set_check`

See also

`set_check`

Module

System

get_error_text (: : ErrorNumber : ErrorText)

Inquiry after the error text of a HALCON error number.

The operator `get_error_text` returns the error text for the corresponding HALCON error number. This is indeed the same text which will be given during an exception. The operator `get_error_text` is especially useful if the error treatment is programmed by the users themselves (see also `set_check(:,:,~give_error':)`).

Attention

Unknown error numbers will trigger a standard message.

Parameter

- ▷ **ErrorNumber** (input_control) integer $\leadsto$ *integer*
 Number of the HALCON error.
Restriction : $(1 \leq \text{ErrorNumber}) \wedge (\text{ErrorNumber} \leq 36000)$
- ▷ **ErrorText** (output_control) string $\leadsto$ *string*
 Corresponding error text.

Example

```

Herror    err;
char      message[MAX_STRING];

set_check("~give_error");
err = send_region(region, socket_id);
set_check("give_error");
if (err != MESS_TRUE) {
  get_error_text((long)err, message);
  fprintf(stderr, "my error message: %s\n", message);
  exit(1);
}

```

Result

The operator [get_error_text](#) always returns the value 2 (H_MSG_TRUE).

Parallelization Information

[get_error_text](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[set_check](#)

See also

[set_check](#)

Module

System

get_spy (: : Class : Value)

Current configuration of the HALCON debugging-tool.

The operator [get_spy](#) returns the current configuration of spy, the HALCON debugging tool. The available control modes (possible choices for [Class](#)) as well as the corresponding tuning possibilities (possible values for [Value](#)) can be called up by using the operator [query_spy](#). You will find a more detailed description under [set_spy](#).

Parameter

- ▷ **Class** (input_control) string $\leadsto$ string
Control mode
Default Value : 'mode'
List of values : **Class** $\in$ { 'mode', 'procedure', 'input_control', 'output_control', 'parameter_values', 'input_gray_window', 'output_gray_window', 'input_region_window', 'db', 'output_region_window', 'halt', 'timeout', 'button_window', 'button_window', 'button_click', 'button_notify', 'log_file', 'error', 'internal' }
- ▷ **Value** (output_control) string $\leadsto$ string / integer / real
State of the control mode.

Result

The operator [get_spy](#) returns the value 2 (H_MSG_TRUE) if the parameter [Class](#) is correct. Otherwise an exception is raised.

Parallelization Information

[get_spy](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[reset_obj_db](#)

See also

[set_spy](#), [query_spy](#)

Module

System

query_spy (: : : Classes, Values)

Inquiring for possible settings of the HALCON debugging tool.

The operator **query_spy** returns all possible settings of spy, the HALCON debugging tool, i.e. all the available control modes (**Classes**) as well as the corresponding possible ways of setting (**Values**). For a more detailed description of spy see operator **set_spy**.

Attention

The values of **Values** cannot be used as direct input for **set_spy**, as they are transmitted as a symbolic constant.

Parameter

- ▷ **Classes** (output_control) string-array $\leadsto$ string
Available control modes (see also **set_spy**).
- ▷ **Values** (output_control) string-array $\leadsto$ string
Corresponding state of the control modes.

Result

query_spy always returns the value 2 (H_MSG_TRUE).

Parallelization Information

query_spy is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

reset_obj_db

See also

set_spy, **get_spy**

Module

System

set_check (: : Check :)

Activating and deactivating of HALCON control modes.

With the help of the operator **set_check** different control modes of the HALCON system can be activated or deactivated. If a certain control mode is activated, parameters etc. will be checked at runtime. Whenever an inconsistency is hereby detected, the program will be interrupted by an exception.

It is recommendable to activate the control modes during the development of a program and to deactivate them only after a successfully concluded testrun. For if the control mode is deactivated and an error occurs, the system may react in an unpredictable way.

The HALCON system provides various possible control modes which can be activated and deactivated independently. By calling the operator **set_check** with the name (**Check**) of the desired control mode, this control mode is activated; the control mode is deactivated by passing its name prefixed with a tilde (~, z.B. '~data').

Available control modes:

'color': If this control mode is activated, only colors may be used which are supported by the display for the currently active window. Otherwise an error message is displayed.

In case of deactivated control mode and non existent colors, the *nearest color* is used (see also **set_color**, **set_gray**, **set_rgb**).

'text': If this control mode is activated, it will check the coordinates during the setting of the text cursor as well as during the display of strings (**write_string**) to the effect whether a part of a sign would lie outside the windowframe (a fact which is not forbidden in principle by the system).

If the control mode is deactivated, the text will be clipped at the windowframe.

'parameter': (For HALCON/PRO only)

If this control mode is activated, output parameter may not be instantiated by calling a procedure.

Otherwise a normal unification mechanism is used.

'data': (For program development)

Checks the consistency of image objects (regions and grayvalue components).

- 'interface':** If this control mode is activated, the interface between the host language and the HALCON procedures will be checked in course (e.g. typifying and counting of the values).
- 'database':** This is a consistency check of the database (e.g. checks whether an object which shall be canceled does indeed exist or not.)
- 'give_error':** Determines whether errors shall trigger exceptions or not. If this control modes is deactivated, the application program must provide a suitable error treatment itself. Please note that errors which are not reported usually lead to undefined output parameters which may cause an unpredictable reaction of the program.
- 'father':** If this control mode is activated by calling the operator `open_window`, HALCON allows only the usage of the number of another HALCON window as the "father" of the new window; otherwise it allows also other X Window IDs.
- 'region':** For program development)
Checks the consistency of chords (this may lead to a notable speed reduction of routines).
- 'clear':** Normally, if a list of objects shall be canceled by using `clear_obj`, an exception will be raised, in case individual objects do not or no longer exist. If the 'clear' mode is activated, such objects will be ignored.
- 'all':** Activates all control modes.
- 'none':** Deactivates all control modes.
- 'default':** Original setting.

Parameter

- ▷ **Check** (input_control) string(-array) $\leadsto$ string
Desired control mode.
Default Value : 'default'
List of values : Check $\in$ {'color', 'text', 'database', 'data', 'interface', 'give_error', 'father', 'region', 'clear', 'all', 'none', 'default'}

Result

The operator `set_check` returns the value 2 (H_MSG_TRUE), if the parameters are correct. Otherwise an exception will be raised.

Parallelization Information

`set_check` is *reentrant* and processed *without* parallelization.

See also

`get_check`, `set_color`, `set_rgb`, `set_hsi`, `write_string`

Module

System

set_spy (: : Class, Value :)

Control of the HALCON Debugging Tools.

The operator `set_spy` is the HALCON debugging tool. This tool allows the flexible control of the input and output data of HALCON-operators - in graphical as well as in textual form. The datacontrol is activated by using

`set_spy(::'mode', 'on':),`

and deactivated by using

`set_spy(::'mode', 'off':).`

The debugging tool can further be activated with the help of the environment variable HALCONSPY. The definition of this variable corresponds to calling up 'mode' and 'on'.

The following control modes can be tuned (in any desired combination of course) with the help of **Class/Value**:

Class Meaning / Value

- 'operator'** When a routine is called, its name and the names of its parameters will be given (in TRIAS notation).
Value: 'on' or 'off'
default: 'off'
- 'input_control'** When a routine is called, the names and values of the input control parameters will be given.
Value: 'on' or 'off'
default: 'off'
- 'output_control'** When a routine is called, the names and values of the output control parameters are given.
Value: 'on' or 'off'
default: 'off'
- 'parameter_values'** Additional information on 'input_control' and 'output_control': indicates how many values per parameter shall be displayed at most (maximum tuple length of the output).
Value: tuple length (integer)
default: 4
- 'db'** Information concerning the 4 relations in the HALCON-database. This is especially valuable in looking for forgotten [clear_obj](#).
Value: 'on' or 'off'
default: 'off'
- 'input_gray_window'** Any reading access of the gray-value component of an (input) image object will cause the gray-value component to be shown in the indicated window (Window-ID; 'none' will deactivate this control).
Value: Window-ID (integer) or 'none'
default: 'none'
- 'output_gray_window'** As soon as the gray-value component of an (output) image object is set, spy will show this gray-value component in the indicated window (Window-ID; 'none' will deactivate this control).
Value: Window-ID (integer) or 'none'
default: 'none'
- 'input_region_window'** Any reading access of the region of an (input) iconic object will cause this region to be shown in the indicated (Window-ID; 'none' will deactivate this control).
Value: Window-ID (integer) or 'none'
default: 'none'
- 'output_region_window'** As soon as the region of an (output) iconic object is set, spy will show this region in the indicated window (Window-ID; 'none' will deactivate this control).
Value: Window-ID (integer) or 'none'
default: 'none'
- 'time'** Processing time of the operator
Value: 'on' or 'off'
default: 'off'
- 'halt'** Determines whether there is a halt after every individual action ('multiple') or only at the end of each operator ('single'). The parameter is only effective if the halt has been activated by 'timeout' or 'button_window'.
Value: 'single' or 'multiple'
default: 'multiple'
- 'timeout'** After every output there will be a halt of the indicated number of seconds.
Value: seconds (real)
default 0.0
- 'button_window'** Alternative to 'timeout': after every output spy waits until the cursor indicates ('button_click' = 'false') or clicks into ('button_click' = 'true') the indicated window. (Window-ID; 'none' will deactivate this control).
Value: Window-ID (integer) or 'none'
default: 'none'
- 'button_click'** Additional option for 'button_window': determines whether or not a mouse-click has to be waited for after an output.
Value: 'on' or 'off'
default: 'off'
- 'button_notify'** If 'button_notify' is activated, spy generates a beep after every output. This is useful in combination with 'button_window'.

Value: 'on' or 'off'
default: 'off'

'log_file' Spy can hereby divert the text output into a file having been opened with open_file.

Value: a file handle (see [open_file](#))

'error' If 'error' is activated and an internal error occurs, spy will show the internal procedures (file/line) concerned.

Value: 'on' or 'off'
default: 'off'

'internal' If 'internal' is activated, spy will display the internal procedures and their parameters (file/line) while an HALCON-operator is processed.

Value: 'on' or 'off'
default: 'off'

Parameter

- ▷ **Class** (input_control) string $\leadsto$ string
Control mode
Default Value : 'mode'
List of values : Class $\in$ {'mode', 'operator', 'input_control', 'output_control', 'parameter_values', 'input_gray_window', 'db', 'time', 'output_gray_window', 'output_region_window', 'input_region_window', 'halt', 'timeout', 'button_window', 'button_click', 'button_notify', 'log_file', 'error', 'internal'}
- ▷ **Value** (input_control) string $\leadsto$ string / integer / real
State of the control mode to be set.
Default Value : 'on'
Suggested values : Value $\in$ {'on', 'off', 1, 2, 3, 4, 5, 10, 50, 0.0, 1.0, 2.0, 5.0, 10.0}

Example

```
/* init spy: Setting of the wished control modi */
set_spy("mode", "on");
set_spy("operator", "on");
set_spy("input_control", "on");
set_spy("output_control", "on");
/* calling of program section, that will be examined */
set_spy("mode", "off");
```

Result

The operator [set_spy](#) returns the value 2 (H_MSG_TRUE) if the parameters are correct. Otherwise an exception is raised.

Parallelization Information

[set_spy](#) is reentrant, local, and processed without parallelization.

Possible Predecessors

[reset_obj_db](#)

See also

[get_spy](#), [query_spy](#)

Module

System

14.3 Information

disp_info (: : ProcName, Machine, DispProgram, Extension :)

Display the manual information of a procedure on the screen.

The operator [disp_info](#) together with the previewer [DispProgram](#) shows the manual entry of the indicated procedure on the screen. The individual files ([ProcName.Extension](#)) are located in the HALCON-Directory "'doc/ps/reference/LANGUAGE/*'", whereby the value for LANGUAGE will be determined by the

currently used host language. The directory can also be generated with the help of the operator `set_system (::'reference_dir','Pfad':)`.

Parameter

- ▷ **ProcName** (input_control) proc_name $\leadsto$ string
Name of the sought procedure.
Default Value : 'read_image'
- ▷ **Machine** (input_control) string $\leadsto$ string
Name of the computer to which the data shall be transmitted or empty string.
Default Value : ''
- ▷ **DispProgram** (input_control) string $\leadsto$ string
Name of the program which shall display the help text.
Default Value : 'ghostview'
Suggested values : DispProgram $\in$ {'gs', 'ghostview'}
- ▷ **Extension** (input_control) string $\leadsto$ string
Extension of the helpfile.
Default Value : 'ps'

Result

If the parameter values are correct and the file and the program are available, the operator `disp_info` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

`disp_info` is reentrant and processed without parallelization.

See also

`open_window`

Module

Image / region / XLD management

`get_chapter_info ( : : Chapter : Info )`

Get information concerning the chapters on procedures.

The operator `get_chapter_info` gives information concerning the chapters on procedures. If instead of **Chapter** the empty string is transmitted, the routine will provide in **Info** the names of all chapters. If on the other hand a certain chapter or a chapter and its subchapter(s) are indicated (by a tuple of names), the corresponding subchapters or - in case there are no further subchapters - the names of the corresponding procedures will be given. The organization of the chapters on procedures is the same as the organization of chapters and subchapters in the HALCON-manual. Please note: The chapters on procedures respectively the subchapters concerning an individual procedure can be called by using the operator `get_operator_info (::<Name>, 'chapter', Info:)`. The Online-texts will be taken from the files english.hlp, english.sta, english.num and english.idx, which will be searched by HALCON in the currently used directory or the directory 'help_dir' (see also `get_system` and `set_system`).

Parameter

- ▷ **Chapter** (input_control) string(-array) $\leadsto$ string
Procedure class or subclass of interest.
Default Value : ''
- ▷ **Info** (output_control) string-array $\leadsto$ string
Procedure classes (**Chapter** = '') or procedure subclasses respectively procedures.

Result

If the parameter values are correct and the helpfile is available, the operator `get_chapter_info` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

`get_chapter_info` is processed completely exclusively without parallelization.

Possible Predecessors

`get_system`, `set_system`

See also

[get_operator_info](#), [get_system](#), [set_system](#)

Module

Image / region / XLD management

get_keywords (: : ProcName : Keywords)

Get keywords which are assigned to procedures.

The operator [get_keywords](#) returns all the keywords in the online-texts corresponding to those procedures which have the indicated substring [ProcName](#) in their name. If instead of [ProcName](#) the empty string is transmitted, the operator [get_keywords](#) returns all keywords. The keywords of an individual procedure can also be called by using the operator [get_operator_info](#). The online-texts will be taken from the files *english.hlp*, *english.sta*, *english.num*, *english.key* and *english.idx*, which are searched by HALCON in the currently used directory and in the directory 'help_dir' (see also [get_system](#) and [set_system](#)).

Parameter

▷ **ProcName** (input_control) *proc_name* $\leadsto$ *string*
 Substring in the names of those procedures for which keywords are needed.
Default Value : 'get_keywords'

▷ **Keywords** (output_control) *string-array* $\leadsto$ *string*
 Keywords for the procedures.

Result

The operator [get_keywords](#) returns the value 2 (H_MSG_TRUE) if the parameters are correct and the helpfiles are available. Otherwise an exception handling is raised.

Parallelization Information

[get_keywords](#) is processed *completely exclusively* without parallelization.

Possible Predecessors

[get_chapter_info](#)

Alternatives

[get_operator_info](#)

See also

[get_operator_name](#), [search_operator](#), [get_param_info](#)

Module

Image / region / XLD management

get_operator_info (: : ProcName, Slot : Information)

Get information concerning a HALCON-procedure.

With the help of the operator [get_operator_info](#) the online-texts concerning a certain procedure can be called (see also [get_operator_name](#)). The form of information available for all procedures ([Slot](#)) can be called using the operator [query_operator_info](#). For the time being the following slots are available:

'short': Short description of the procedure.

'abstract': Description of the procedure.

'procedure_class': Name(s) of the chapter(s) in the procedure hierarchy (chapter, subchapter in the HALCON manual).

'functionality': Functionality is equivalent to the object class to which the procedure can be assigned.

'keywords': Keywords of the procedure (optional).

'example': Example for the use of the procedure (optional). The operator 'example.LANGUAGE' (LANGUAGE $\in$ {c,c++,smalltalk,trias}) calls up examples for a certain language if available. If the language is not indicated or if no example is available in this language, the TRIAS-example will be returned.

- 'complexity'**: Complexity of the procedure (optional).
- 'effect'**: Not in use so far.
- 'alternatives'**: Alternative procedures (optional).
- 'see_also'**: Procedures containing further information (optional).
- 'predecessor'**: Possible and sensible predecessor
- 'successor'**: Possible and sensible successor
- 'result_state'**: Return value of the procedure (TRUE, FALSE, FAIL, VOID or EXCEPTION).
- 'attention'**: Restrictions and advice concerning the correct use of the procedure (optional).
- 'parameter'**: Names of the parameter of the procedure (see also [get_param_info](#)).
- 'references'**: Literary references (optional).
- 'module'**: The module to which the operator is assigned.
- 'html_path'**: The directory where the HTML documentation of the operator resides.
- 'parameter_relations'**: Assurances concerning the parameters (optional).

The texts will be taken from the files english.hlp, english.sta, english.key, english.num und english.idx which will be searched by HALCON in the currently used directory or in the directory 'help_dir' (respectively 'user_help_dir') (see also [get_system](#) and [set_system](#)). By adding '.latex' after the slotname, the text of slots containing textual information can be made available in L^AT_EX notation.

<i>Parameter</i>	
▷ ProcName (input_control)	proc_name $\leadsto$ string
Name of the procedure on which more information is needed.	
Default Value : 'get_operator_info'	
▷ Slot (input_control)	string $\leadsto$ string
Desired information.	
Default Value : 'abstract'	
List of values : Slot $\in$ {'short', 'abstract', 'procedure_class', 'functionality', 'effect', 'complexity', 'predecessor', 'successor', 'alternatives', 'see_also', 'keywords', 'example', 'attention', 'result_state', 'return_value', 'references', 'sourcefiles', 'deffile', 'module', 'html_path'}	
▷ Information (output_control)	string(-array) $\leadsto$ string
Information (empty if no information is available)	

Result

The operator [get_operator_info](#) returns the value 2 (H_MSG_TRUE) if the parameters are correct and the helpfiles are available. Otherwise an exception handling is raised.

Parallelization Information

[get_operator_info](#) is processed *completely exclusively* without parallelization.

Possible Predecessors

[get_keywords](#), [search_operator](#), [get_operator_name](#), [query_operator_info](#),
[query_param_info](#), [get_param_info](#)

Possible Successors

[get_param_names](#), [get_param_num](#), [get_param_types](#)

Alternatives

[get_param_names](#)

See also

[query_operator_info](#), [get_param_info](#), [get_operator_name](#), [get_param_num](#),
[get_param_types](#)

Module

Image / region / XLD management

get_operator_name (: : Pattern : ProcNames)

Get procedures with the given string as a substring of their name.

The operator `get_operator_name` takes a string (**Pattern**) as input and searches all HALCON-procedures having this string as a substring in their name. If an empty string is entered, the names of all procedures available will be returned.

Parameter	
▷ Pattern (input_control)	string $\leadsto$ string Substring of the sought names (empty $\leq$ all names). Default Value : 'info'
▷ ProcNames (output_control)	string-array $\leadsto$ string Detected procedure names.
Result	
The operator <code>get_operator_name</code> returns the value 2 (H_MSG_TRUE) if the helpfiles are available. Otherwise an exception handling is raised.	
Parallelization Information	
<code>get_operator_name</code> is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.	
Possible Successors	
<code>get_operator_info</code> , <code>get_param_names</code> , <code>get_param_num</code> , <code>get_param_types</code>	
Alternatives	
<code>search_operator</code>	
See also	
<code>get_operator_info</code> , <code>get_param_names</code> , <code>get_param_num</code> , <code>get_param_types</code>	
Module	
Image / region / XLD management	

get_param_info (: : ProcName, ParamName, Slot : Information)

Get information concerning the procedure parameters.

The operator `get_param_info` is used for calling up the online-texts assigned to a parameter of an indicated procedure. The form of information available for each parameter (**Slot**), can be called up by using the operator `query_param_info`. At the moment the following slots are available:

'description': Description of the parameter.

'description.latex': Description of the parameter in $\text{\LaTeX}$ notation.

'parameter.class': Parameter classes: 'input_object', 'output_object', 'input_control' oder 'output_control'.

'type.list': Permitted type(s) of data for parameter values (for control parameters only). Value: 'real', 'integer' oder 'string'.

'default.type': Default-type for parameter values (for control parameters only). This type of parameter is the one HALCON/C uses in the "simple mode". If 'none' is indicated, the "tuple mode" must be used. Value: 'real', 'integer', 'string' oder 'none'.

'sem.type': Semantic type of the parameter. This is important to allow the assignment of the parameters to object classes in object-oriented languages (C++, Smalltalk). If more than one parameter belongs semantically to one type, this fact is indicated as well. So far the the following objects are supported:

```
object, image, region, xld,
xld_cont, xld_para, xld_poly, xld_ext_para, xld_mod_para,
integer, real, number, string,
channel, grayval, window,
histogram, distribution,
point(.x, .y), extent(.x, .y),
angle(.rad oder .deg),
circle(.center.x, .center.y, .radius),
arc(.center.x, .center.y, .angle.rad, .begin.x, .begin.y),
ellipse(.center.x, .center.y, .angle.rad, .radius1, .radius2),
line(.begin.x, .begin.y, .end.x, .end.y)
```

```
rectangle(.origin.x, .origin.y, .corner.x, .corner.y
bzw. .extent.x, .extent.y),
polygon(.x, .y), contour(.x, .y),
coordinates(.x, .y), chord(.x1, .x2, .y),
chain(.begin.x, .begin.y, .code).
```

'default_value': Default-value for the parameter (for input-control parameters only). It is the question of mere information only (the parameter value must be transmitted explicitly, even if the default-value is used). This entry serves only as a notice, a point of departure for own experiments. The values have been selected so that they normally do not cause any errors but generate something that makes sense.

'multi_value': 'true', if more than one value is permitted in this parameter position, otherwise 'false'.

'multichannel': 'true', in case the input image object may be multichannel.

'mixed_type': For control parameters exclusively and only if value tuples ('multivalue'-'true') and various types of data are permitted for the parameter values ('type_list' having more than one value). In this case [Slot](#) indicates, whether values of various types may be mixed in one tuple ('true' or 'false').

'values': Selection of values (optional).

'value_list': In case a parameter can take only a limited number of values, this fact will be indicated explicitly (optional).

'valuemin': Minimum value of a value interval.

'valuemax': Maximum value of a value interval.

'valuefunction': Function describing the course of the values for a series of tests (lin, log, quadr, ...).

'steprec': Recommended step width for the parameter values in a series of tests.

'stepmin': Minimum step width of the parameter values in a series of tests.

'valuenumber': Expression describing the number of parameters as such or in relation to other parameters.

'assertion': Expression describing the parameter values as such or in relation to other parameters.

The online-texts will be taken from the files english.hlp, english.sta, english.key, english.num and english.idx which will be searched by HALCON in the currently used directory or the directory 'help_dir' (see also [get_system](#) and [set_system](#)).

<i>Parameter</i>	
▷ ProcName (input_control)	proc_name $\leadsto$ string
Name of the procedure on whose parameter more information is needed.	
Default Value : 'get_param_info'	
▷ ParamName (input_control)	string $\leadsto$ string
Name of the parameter on which more information is needed.	
Default Value : 'Slot'	
▷ Slot (input_control)	string $\leadsto$ string
Desired information.	
Default Value : 'description'	
List of values : Slot $\in$ {'description', 'type_list', 'default_type', 'sem_type', 'default_value', 'values', 'value_list', 'valuemin', 'valuemax', 'valuefunction', 'valuenumber', 'assertion', 'steprec', 'stepmin', 'mixed_type', 'multivalue', 'multichannel'}	
▷ Information (output_control)	string(-array) $\leadsto$ string
Information (empty in case there is no information available).	

Result

The operator [get_param_info](#) returns the value 2 (H_MSG_TRUE) if the parameters are correct and the help-files are available. Otherwise an exception handling is raised.

Parallelization Information

[get_param_info](#) is processed *completely exclusively* without parallelization.

Possible Predecessors

[get_keywords](#), [search_operator](#)

Alternatives

[get_param_names](#), [get_param_num](#), [get_param_types](#)

See also

[query_param_info](#), [get_operator_info](#), [get_operator_name](#)

Module

Image / region / XLD management

get_param_names (: : ProcName : InpObjPar, OutpObjPar, InpCtrlPar, OutpCtrlPar)

Get the names of the parameters of a HALCON-procedure.

For the HALCON-procedure indicated in [ProcName](#) the operator [get_param_names](#) returns the names of the input objects, the output objects, of the input control parameters and the output control parameters.

Parameter

- ▷ **ProcName** (input_control) proc_name $\leadsto$ *string*
Name of the procedure.
Default Value : 'get_param_names'
- ▷ **InpObjPar** (output_control) string-array $\leadsto$ *string*
Names of the input objects.
- ▷ **OutpObjPar** (output_control) string-array $\leadsto$ *string*
Names of the output objects.
- ▷ **InpCtrlPar** (output_control) string-array $\leadsto$ *string*
Names of the input control parameters.
- ▷ **OutpCtrlPar** (output_control) string-array $\leadsto$ *string*
Names of the output control parameters.

Result

The operator [get_param_names](#) returns the value 2 (H_MSG_TRUE) if the name of the procedure exists and the helpfiles are available. Otherwise an exception handling is raised.

Parallelization Information

[get_param_names](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[get_keywords](#), [search_operator](#), [get_operator_name](#), [get_operator_info](#)

Possible Successors

[get_param_num](#), [get_param_types](#)

Alternatives

[get_operator_info](#), [get_param_info](#)

See also

[get_param_num](#), [get_param_types](#), [get_operator_name](#)

Module

Image / region / XLD management

get_param_num (: : ProcName : CName, InpObjPar, OutpObjPar, InpCtrlPar, OutpCtrlPar, Type)

Get number of the different parameter classes of a HALCON-procedure.

The operator [get_param_num](#) returns the number of the input and output object parameters, as well as the input and output control parameters for the indicated HALCON-procedure. Further, you will receive the name of the C-function ([CName](#)) called by the procedure. The output parameter [Type](#) indicates, whether the procedure is a system procedure or an user procedure.

Parameter	
▷ ProcName (input_control)	proc_name $\leadsto$ string
Name of the procedure.	
Default Value : 'get_param_num'	
▷ CName (output_control)	string $\leadsto$ string
Name of the called C-function.	
▷ InpObjPar (output_control)	integer $\leadsto$ integer
Number of the input object parameters.	
▷ OutpObjPar (output_control)	integer $\leadsto$ integer
Number of the output object parameters.	
▷ InpCtrlPar (output_control)	integer $\leadsto$ integer
Number of the input control parameters.	
▷ OutpCtrlPar (output_control)	integer $\leadsto$ integer
Number of the output control parameters.	
▷ Type (output_control)	string $\leadsto$ string
System procedure or user procedure.	
Suggested values : Type $\in$ {'system', 'user'}	
Result	
The operator <code>get_param_num</code> returns the value 2 (H_MSG_TRUE) if the name of the procedure exists. Otherwise an exception handling is raised.	
Parallelization Information	
<code>get_param_num</code> is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Predecessors	
<code>get_keywords</code> , <code>search_operator</code> , <code>get_operator_name</code> , <code>get_operator_info</code>	
Possible Successors	
<code>get_param_types</code>	
Alternatives	
<code>get_operator_info</code> , <code>get_param_info</code>	
See also	
<code>get_param_names</code> , <code>get_param_types</code> , <code>get_operator_name</code>	
Module	
Image / region / XLD management	

get_param_types (: : ProcName : InpCtrlParType, OutpCtrlParType)

Get default data type for the control parameters of a HALCON-procedure.

The operator `get_param_types` returns the default data type for each input and output control parameter. The default type of a parameter is the type used in "simple mode" in HALCON/C. This concerns parameters which allow more than one type as for example `write_string`. Hereby the types of input parameters are combined in the variable `InpCtrlParType`, whereas the types of output parameters are combined in the variable `OutpCtrlParType`. The following types are possible:

'integer': an integer.

'integer tuple': an integer or a tuple of integers.

'real': a floating point number.

'real tuple': a floating point number or a tuple of floating point numbers.

'string': a string.

'string tuple': a string or a tuple of strings.

'no_default': individual value of which the type cannot be determined.

'no_default tuple': individual value or tuple of values of which the type cannot be determined.

'default': individual value of unknown type, whereby the systems assumes it to be an 'integer'.

Parameter
▷ ProcName (input_control) proc_name $\leadsto$ string Name of the procedure. Default Value : 'get_param_types' ▷ InpCtrlParType (output_control) string-array $\leadsto$ string Default type of the input control parameters. ▷ OutpCtrlParType (output_control) string-array $\leadsto$ string Default type of the output control parameters.
Result
The operator get_param_types returns the value 2 (H_MSG_TRUE) if the indicated procedure name exists. Otherwise an exception handling is raised.
Parallelization Information
get_param_types is <i>reentrant</i> and processed <i>without</i> parallelization.
Possible Predecessors
get_keywords , search_operator , get_operator_name , get_operator_info
Alternatives
get_param_info
See also
get_param_names , get_param_num , get_operator_info , get_operator_name
Module
Image / region / XLD management

query_operator_info (: : : Slots)

Query slots concerning information with relation to the operator [get_operator_info](#).

The operator [query_operator_info](#) returns the names of those online texts (**Slots**) which are available online for each procedure. The information itself can be called up using

[get_operator_info](#)(: : <ProcName>, <Slot> : <Information>).

Parameter
▷ Slots (output_control) string-array $\leadsto$ string Slotnames of the operator get_operator_info.
Result
The operator query_operator_info always returns the value 2 (H_MSG_TRUE).
Parallelization Information
query_operator_info is <i>local</i> and processed <i>completely exclusively</i> without parallelization.
Possible Successors
get_operator_info
See also
get_operator_info
Module
Image / region / XLD management

query_param_info (: : : Slots)

Query slots of the online-information concerning the operator [get_param_info](#).

The operator [query_param_info](#) returns the names of those pieces of information (**Slots**) which are available online for each parameter (online texts). The online texts themselves can be called up using

[get_param_info](#)(: : <Procedurname>, <Parametername>, <Slot> : <Information>).

Parameter
▷ Slots (output_control) string-array $\leadsto$ <i>string</i> Slotnames for the operator get_param_info .
Result
query_param_info always returns the value 2 (H_MSG_TRUE).
Parallelization Information
query_param_info is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.
Possible Successors
get_param_info
See also
get_param_info
Module
Image / region / XLD management

search_operator (: : Keyword : ProcNames)

Search names of all procedures assigned to one keyword.

The operator [search_operator](#) returns the names of all procedures whose online-texts include the keyword [Keyword](#) (see also [get_operator_info](#)). All available keywords are called by using the operator [get_keywords](#)(::':': <keywords>). The online-texts are taken from the files english.hlp, english.sta, english.key, english.num and Halcon.idx, which are searched by HALCON in the currently used directory or the directory 'help_dir' (see also [get_system](#) and [get_system](#)).

Parameter
▷ Keyword (input_control) string $\leadsto$ <i>string</i> Keyword for which corresponding procedures are searched. Default Value : 'Information'
▷ ProcNames (output_control) string-array $\leadsto$ <i>string</i> Procedures whose slot 'keyword' contains the keyword.
Result
The operator search_operator returns the value 2 (H_MSG_TRUE) if the parameters are correct and the helpfiles are available. Otherwise an exception handling is raised.
Parallelization Information
search_operator is processed <i>completely exclusively</i> without parallelization.
Possible Predecessors
get_keywords
See also
get_keywords , get_operator_info , get_param_info
Module
Image / region / XLD management

14.4 Operating-System

count_seconds (: : : Seconds)

Elapsed processing time since the last call of [count_seconds](#).

The operator [count_seconds](#) helps to measure the runtime. The first call of the procedure normally returns a zero. Each further call returns the processing time which has elapsed in the meantime in seconds. The method by which the time is measured can be set with [set_system](#)('clock_mode', ...).

Attention
The time measurement is not exact and depends on the load of the computer.

Parameter	
▷ Seconds (output_control)	real $\leadsto$ real
Processtime since the program start.	
Example	
<pre>count_seconds(:::Start) > /* program segment to be measured */ count_seconds(:::End) > eval(::End - Start:RunTime).</pre>	
Result	
The operator count_seconds always returns the value 2 (H_MSG_TRUE).	
Parallelization Information	
count_seconds is <i>reentrant</i> and processed <i>without</i> parallelization.	
See also	
set_system	
Module	
System	

system_call (: : Command :)

Executes a system command.

The operator **system_call** executes the system command specified by the string pointed to by **Command** (C-procedure "system"). If the string is empty, an interactive shell will be started ('csh -i').

Parameter	
▷ Command (input_control)	string $\leadsto$ string
Command to be called by the system.	
Default Value : 'ls'	
Result	
If the entered operator can be executed by the system, the operator system_call returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.	
Parallelization Information	
system_call is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Predecessors	
count_seconds	
See also	
wait_seconds , count_seconds	
Module	
System	

wait_seconds (: : Seconds :)

Delaying the execution of the program.

The operator **wait_seconds** delays the execution by the number of seconds indicated in **Seconds**.

Parameter	
▷ Seconds (input_control)	real $\leadsto$ real
Number of seconds by which the execution of the program will be delayed.	
Default Value : 10	
Restriction : Seconds $\geq$ 0	

	<i>Result</i>	
The operator <code>wait_seconds</code> always returns the value 2 (H_MSG_TRUE).		
	<i>Parallelization Information</i>	
<code>wait_seconds</code> is <i>reentrant</i> and processed <i>without</i> parallelization.		
	<i>Possible Successors</i>	
<code>system_call</code>		
	<i>See also</i>	
<code>system_call</code> , <code>count_seconds</code>		
	<i>Module</i>	
System		

14.5 Parallelization

check_par_hw_potential (: : AllInPars :)

Check hardware regarding its potential for parallel processing.

`check_par_hw_potential` is necessary for an efficient automatic parallelization, which is used by HALCON to better utilize multiprocessor hardware in order to speed up the processing of operators. As the parallelization of operators is done automatically, there is no need for the user to explicitly prepare or change programs for their parallelization. Thus, all HALCON-based programs can be used unchanged on multiprocessor hardware and nevertheless utilize the potential of parallel hardware. `check_par_hw_potential` checks a given hardware with respect to a parallel processing of HALCON operators. At this, it examines every operator, which can be sped up in principle by an automatic parallelization. Each examined operator is processed several times - both sequentially and in parallel - with a changing set of input parameter values/images. The latter helps to evaluate dependencies between an operator's input parameter characteristics (e.g. the size of an input image) and the efficiency of its parallel processing. At this, `AllInPars` is used in the following way: In the normal case, i.e. if `AllInPars` contains the default value 0 ("false"), only those input parameters are examined which are supposed to show influence on the processing time. Other parameters are not examined so that the whole process is sped up. However, in some rare cases, the internal implementation of a HALCON operator might change from one HALCON release to another. Then, a parameter which did not show any direct influence on the processing time in former releases, may now show such an influence. In this case it is necessary to set `AllInPars` to 1 ("true") in order to force the examination of all input parameters. If this happens, the HALCON release notes will most likely contain an appropriate note about this fact. Overall, `check_par_hw_potential` performs several test loops and collects a lot of hardware-specific informations, which enable HALCON to optimize the automatic parallelization for a given hardware. The hardware information is stored so that it can be used again in future HALCON sessions. Thus, it is sufficient, to start `check_par_hw_potential` once on each multiprocessor machine that is used for parallel processing. Of course, it should be started again, if the hardware of the machine changes, for example, by installing a new cpu, or if the operating system of the machine changes, or if the machine gets a new host name. The latter is necessary, because HALCON identifies the machine-specific parallelization information by the machine's host name. If the same multiprocessor machine is used with different operating systems, such as Windows and Linux, it is necessary to start `check_par_hw_potential` once for each operating system in order to correctly measure the rather strong influence of the operating system on the potential of exploiting multiprocessor hardware. Under Windows, HALCON stores the parallelization knowledge, which belongs to a specific machine, in the machine's registry. At this, it uses a machine-specific registry key, which can be used by different users simultaneously. In the normal case, this key can be written or changed by any user under Windows NT. However, under Windows 2000 the key may only be changed by users with administrator privileges or by users which at least belong to the "power user" group. For all other users `check_par_hw_potential` shows no effect (but does not return an error). Under Linux/UNIX the parallelization information is stored in a file in the HALCON installation directory (\$HALCONROOT). Again this means that `check_par_hw_potential` must be called by users with the appropriate privileges, here by users which have write access to the HALCON directory. If HALCON is used within a network under Linux/UNIX, the denoted file contains the information about every computer in the network for which the hardware check has been successfully completed.

Attention

During its test loops `check_par_hw_potential` has to start every examined operator several times. Thus, the processing of `check_par_hw_potential` can take rather a long time. `check_par_hw_potential`

bases on the automatic parallelization of operators which is exclusively supported by Parallel HALCON. Thus, `check_par_hw_potential` always returns an appropriate error, if it used with a non-parallel HALCON version. `check_par_hw_potential` must be called by users with the appropriate privileges for storing the parallelization information permanently (see the operator's description above for more details about this subject).

Parameter
▷ AllInpPars (input_control) integer $\leadsto$ integer Check every input parameter? Default Value : 0 List of values : AllInpPars $\in$ {0, 1}
Result
<code>check_par_hw_potential</code> returns 2 (H_MSG_TRUE) if all parameters are correct.
Parallelization Information
<code>check_par_hw_potential</code> is <i>local</i> and processed <i>completely exclusively</i> without parallelization.
Possible Successors
<code>store_par_knowledge</code>
See also
<code>store_par_knowledge</code> , <code>load_par_knowledge</code>
Module
System

load_par_knowledge (: : FileName :)

Load knowledge about automatic parallelization from file.

`load_par_knowledge` supports the automatic parallelization of HALCON operators, which is used to better utilize multiprocessor hardware in order to speed up the processing of operators. To parallelize the processing of operators automatically HALCON needs some specific knowledge about the used hardware. This hardware-specific knowledge can be obtained by using the operator `check_par_hw_potential`. In the normal case, HALCON stores this knowledge in a specific file in the HALCON installation directory (Linux/UNIX) or within the “registry” (Windows). This enables HALCON to use the knowledge again later on. With `load_par_knowledge` it is possible to load this knowledge explicitly from an ASCII file. At this, `FileName` denotes the name of this file (incl. path and file extension). The file must conform to a specific syntax and must have been stored beforehand by using `store_par_knowledge`. While reading the file `load_par_knowledge` checks whether its content was written for the currently used computer and whether the contained parallelization information regards the currently used HALCON version (and revision). If this is the case, `load_par_knowledge` adopts the information so that it will also be used with further HALCON sessions. Otherwise, the information is ignored and `load_par_knowledge` returns an appropriate error message.

Parameter
▷ FileName (input_control) filename.named $\leadsto$ string Name of parallelization knowledge file. Default Value : ”
Result
<code>load_par_knowledge</code> returns 2 (H_MSG_TRUE) if all parameters are correct.
Parallelization Information
<code>load_par_knowledge</code> is <i>local</i> and processed <i>completely exclusively</i> without parallelization.
Possible Predecessors
<code>store_par_knowledge</code>
See also
<code>store_par_knowledge</code> , <code>check_par_hw_potential</code>
Module
System

store_par_knowledge (: : FileName :)

Store knowledge about automatic parallelization in file.

store_par_knowledge supports the automatic parallelization of HALCON operators, which is used to better utilize multiprocessor hardware in order to speed up the processing of operators. To parallelize the processing of operators automatically HALCON needs some specific knowledge about the used hardware. This hardware-specific knowledge can be obtained by calling the operator **check_par_hw_potential**. There, HALCON stores the knowledge in a specific file in the HALCON installation directory (Linux/UNIX) or within the “registry” (Windows). This enables HALCON to use the knowledge again later on. With **store_par_knowledge** it is possible to store this knowledge explicitly as an ASCII file. At this, **FileName** denotes the name of this file (incl. path and file extension). The stored knowledge can be read again later on by using **load_par_knowledge**.

Parameter

- ▷ **FileName** (input_control) filename.named ~> *string*
 Name of parallelization knowledge file.
Default Value : ”

Result

store_par_knowledge returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

store_par_knowledge is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

check_par_hw_potential

Possible Successors

load_par_knowledge

See also

load_par_knowledge, **check_par_hw_potential**

Module

System

14.6 Parameters

get_system (: : Query : Information)

Information concerning the currently used HALCON system parameter.

The operator **get_system** returns information concerning the currently activated HALCON system parameters. Some of these parameters can be changed dynamically by using the operator **set_system**. They are marked by a + in the list below. By passing the string ‘?’ as the parameter **Query**, the names of all system parameters are provided with **Information**.

The following system parameters can be queried:

Versions

- ’version’**: HALCON version number, e.g.: 6.0
- ’last_update’**: Date of creation of the HALCON library
- ’revision’**: Revision number of the HALCON library, e.g.: 1

Upper Limits

- ’max_contour_length’**: Maximum number of contour respectively polygone control points of a region.
- ’max_images’**: Maximum total of images.
- ’max_channels’**: Maximum number of channels of an image.
- ’max_obj_per_par’**: Maximum number of image objects which may be used during one call up per parameter
- ’max_inp_obj_par’**: Maximum number of input parameters.
- ’max_outp_obj_par’**: Maximum number of output parameters.

- 'max_inp_ctrl_par'**: Maximum number of input control parameters.
- 'max_outp_ctrl_par'**: Maximum number of output control parameters.
- 'max_window'**: Maximum number of windows.
- 'max_window_types'**: Maximum number of window systems.
- 'max_proc'**: Maximum number of HALCON procedures (system defined + user defined).

Graphic

- +flush_graphic'**: Determines, whether the flush operation is called or not after each HALCON procedure.
- +int2_bits'**: Number of significant bits of int2 images. This number is used when scaling the gray values.
If the value is -1 the gray values will be automatically scaled (default).
- +int_zooming'**: Determines if the zooming of images is done with integer arithmetic or with floating point arithmetic.
- +draw_mode'**: (no description available)
- +ignore_colormap'**: (no description available)
- +backing_store'**: Storage of the window contents in case of overlaps.
- +icon_name'**: (no description available)
- +window_name'**: (no description available)
- +default_font'**: Name of the font to set at opening the window.
- +single_lut'**: (no description available)
- +update_lut'**: (no description available)
- +x_package'**: Number of bytes which are sent to the X server during each transfer of data.
- +num_gray_4'**: Number of colors reserved under X Windows concerning the output of graylevels ([disp_channel](#)) on a machine with 4 bitplanes (16 colors).
- +num_gray_6'**: Number of colors reserved under X Window concerning the output of graylevels ([disp_channel](#)) on a machine with 6 bitplanes (64 colors).
- +num_gray_8'**: Number of colors reserved under X Window concerning the output of graylevels ([disp_channel](#)) on a machine with 8 bitplanes (256 colors).
- +num_gray_percentage'**: HALCON reserves a certain amount of the available colors under X Window for the representation of graylevels ([disp_image](#)). This shall interfere with other X applications as little as possible. However, if HALCON does not succeed in reserving a minimum percentage of 'num_gray_percentage' of the necessary colors on the X server, a certain amount of the lookup-table will be claimed for the HALCON graylevels regardless of the consequences for other applications.
This may result in undesired shifts of color when switching between HALCON windows and windows of other applications, or if (outside HALCON) a window-dump is generated. The number of the real graylevels to be reserved depends on the number of available bitplanes on the output machine (see also 'num_gray_*'). Naturally no colors will be reserved on monochrome machines - the graylevels will instead be dithered when displayed. If graylevel displays are used, only different shades of gray will be applied ('black', 'white', 'gray', etc.). Machines with 2 or 3 bitplanes will be considered monochrome machines, machines with 5 (7) bitplanes like machines with 4 (6) bitplanes, and machines having more than 8 bitplanes like machines with 8 bitplanes. A special case are machines providing a 24 bit display (true color machines). Naturally no colors are reserved for the display of graylevels in this case.
Note: Before the first window on a machine with x bitplanes is opened, num_gray_x indicates the number of colors which have to be reserved for the display of graylevels, afterwards, however, it will indicate the number of colors which actually have been reserved.
- +num_graphic_percentage'**: (no description available)
- +num_graphic_2'**: Number of the HALCON graphic colors reserved under X Window (for [disp_region](#) etc.) on a machine with 2 bitplanes (4 colors).
- +num_graphic_4'**: Number of the HALCON graphic colors reserved under X Window (for [disp_region](#) etc.) on a machine with 4 bitplanes (16 colors).
- +num_graphic_6'**: Number of the HALCON graphic colors reserved under X Window (for [disp_region](#) etc.) on a machine with 6 bitplanes (64 colors).
- +num_graphic_8'**: Number of the HALCON graphic colors reserved under X Window (for [disp_region](#) etc.) on a machine with 8 bitplanes (256 colors).

Image Processing

- +neighborhood'**: Using the 4 or 8 neighborhood.

- +’only_lines’:** (no description available)
- +’init_new_image’:** Initialization of images before applying grayvalue transformations.
- +’no_object_result’:** Behavior for an empty object lists.
- +’empty_region_result’:** Reaction of procedures concerning input objects with empty regions which actually are not useful for such objects (e.g. certain region features, segmentation, etc.). Possible return values:
 - ’true’:** the error will be ignored if possible
 - ’false’:** the procedure returns FALSE
 - ’fail’:** the procedure returns FAIL
 - ’void’:** the procedure returns VOID
 - ’exception’:** an exception is raised
- +’store_empty_region’:** Storing of objects with empty regions.
- +’clip_region’:** Clipping of output regions so that they fit the global image size.
- ’width’:** Global maximum image width - in Standard-HALCON this value contains the maximum image width of all HALCON image objects which are currently stored in memory. In Parallel HALCON this value contains the maximum image width of all HALCON image objects which are or were in memory since the start of the current HALCON session (this also includes objects which may be deleted meanwhile).
- ’height’:** Global maximum image height - in Standard-HALCON this value contains the maximum image height of all HALCON image objects which are currently stored in memory. In Parallel HALCON this value contains the maximum image height of all HALCON image objects which are or were in memory since the start of the current HALCON session (this also includes objects which may be deleted meanwhile).
- ’obj_images’:** Current number of grayvalue components per image object.
- +’current_runlength_number’:** Currently used number of chords which can be used for the encoding of regions.

Parallelization

- +’parallelize_operators’:** Determines whether Parallel HALCON uses an automatic parallelization to speed up the processing of operators on multiprocessor machines.
- +’reentrant’:** Denotes whether Parallel HALCON currently supports reentrancy (default case), or whether this feature has been switched off. Reentrancy is necessary for the automatic parallelization of Parallel HALCON and for calling and processing multiple HALCON operators in parallel within multithreaded applications.
- ’processor_num’:** Returns the number of processors which Parallel HALCON has found on the hardware it is running on. This also indicates the number of processors which is used by Parallel HALCON for the automatic parallelization of operators.

File

- +’flush_file’:** Buffering of file output.
- +’ocr_trainf_version’:** This parameter returns the file format used for writing an OCR training file. The operators `write_ocr_trainf`, `write_ocr_trainf_image` and `concat_ocr_trainf` write training data in ASCII format for version number 1 or in binary format for version number 2.

Directories

- +’image_dir’:** Path which will be searched for the image file after the default directory (see also: `read_image`).
- +’lut_dir’:** Path for the default directory for color tables (see also: `set_lut`).
- +’reference_dir’:** Path for the default directory concerning the postscript HALCON documentation.
- +’help_dir’:** Path for the default help directory for the online help files:
 - {german,english}. {hlp,sta,idx,num,key}.

Other

- +’do_low_error’:** Flag, if low level error should be printed.
- ’num_proc’:** Total number of the available HALCON procedures (’num_sys_proc’ + ’num_user_proc’).
- ’num_sys_proc’:** Number of the system procedures (supported procedures).
- ’num_user_proc’:** Number of the user defined procedures (see also ’Extension Packages’ manual).

- 'byte_order'**: Byte order of the processor ('msb_first' or 'lsb_first').
- 'operating_system'**: Name of the operating system of the computer on which the HALCONprocess is being executed.
- 'operating_system_version'**: Version number of the operating system of the computer on which the HALCON process is being executed.
- + 'clock_mode'** Method used for measuring the time in `count_seconds` ('processor_time', 'elapsed_time', or 'performance_counter').
- + 'max_connection'** Maximum number of regions returned by `connection`.
- + 'alloctmp_single_block'**: Flag for kind of temporary memory management.
- + 'alloctmp_max_blocksize'**: Maximum size of memory blocks to be allocated within temporary memory management. (No effect, if 'alloctmp_max_blocksize' == -1 or 'alloctmp_single_block' == 'true')
- + 'extern_alloc_func'**: Pointer to external function for memory allocation of result images.
- + 'extern_free_func'**: Pointer to external function for memory deallocation of result images.
- 'temp_mem'**: Amount of temporary memory used by the last operator. This feature is only supported by Standard-HALCON. Instead of this, Parallel HALCON returns the amount of temporary memory used by `get_system` itself.
- + 'language'**: Language used for error messages ('english' or 'german').

Parameter

- ▷ **Query** (input_control) string(-array) $\leadsto$ string
Desired system parameter.
Default Value : 'width'
List of values : Query $\in$ {'?', 'version', 'last_update', 'revision', 'max_images', 'max_channels', 'max_obj_per_par', 'max_inp_obj_par', 'max_outp_obj_par', 'max_inp_ctrl_par', 'max_outp_ctrl_par', 'max_window', 'max_window_types', 'max_proc', 'flush_graphic', 'int2_bits', 'int_zooming', 'draw_mode', 'ignore_colormap', 'backing_store', 'icon_name', 'window_name', 'default_font', 'single_lut', 'update_lut', 'x_package', 'num_gray_4', 'num_gray_6', 'num_gray_8', 'num_gray_percentage', 'num_graphic_2', 'num_graphic_4', 'num_graphic_6', 'num_graphic_8', 'num_graphic_percentage', 'neighborhood', 'only_lines', 'init_new_image', 'no_object_result', 'empty_region_result', 'store_empty_region', 'clip_region', 'width', 'height', 'obj_images', 'current_runlength_number', 'flush_file', 'ocr_trainf_version', 'image_dir', 'lut_dir', 'reference_dir', 'help_dir', 'num_proc', 'num_sys_proc', 'num_user_proc', 'temp_mem', 'alloctmp_single_block', 'alloctmp_max_blocksize', 'extern_alloc_func', 'extern_free_func', 'do_low_error', 'reentrant', 'parallelize_operators', 'processor_num', 'clock_mode', 'max_connection', 'byte_order', 'operating_system', 'operating_system_version', 'language'}
- ▷ **Information** (output_control) integer(-array) $\leadsto$ integer / real / string
Current value of the system parameter.

Result

The operator `get_system` returns the value 2 (H_MSG_TRUE) if the parameters are correct. Otherwise an exception is raised.

Parallelization Information

`get_system` is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

`reset_obj_db`

Possible Successors

`set_system`

See also

`set_system`

Module

System

set_system (: : Systemparameter, Value :)

Setting of HALCON system parameters.

The operator `set_system` allows to change different system parameters with relation to the runlength.

Available system parameters:

- 'neighborhood':** This parameter is used with all procedures which examine neighborhood relations: `connection`, `get_region_contour`, `get_region_chain`, `get_region_polygon`, `get_region_thickness`, `boundary`, `paint_region`, `disp_region`, `fill_up`, `contlength`, `shape_histo_all`.
Value: 4 or 8
default: 8
- 'default_font':** Whenever a window is opened, a font will be set for the text output, whereby the 'default_font' will be used. If the preset font cannot be found, another fontname can be set before opening the window.
Value: Filename of the fonts
default: fixed
- 'single_lut':** (no description available) **default:** 'false'
- 'update_lut'** Determines whether the HALCON color tables are adapted according to their environment or not.
Value: 'true' or 'false'
default: 'false'
- 'image_dir':** Image files (e.g. `read_image` and `read_sequence`) will be looked for in the currently used directory and in 'image_dir' (if no absolute paths are indicated). More than one directory name can be indicated (searchpaths), separated by semicolons (Windows NT) or colons (Unix). The path can also be determined using the environment variable HALCONIMAGES.
Value: Name of the filepath
default: '\$HALCONROOT/images' bzw. '%HALCONROOT%/images'
- 'lut_dir':** Color tables (`set_lut`) which are realized as an ASCII-file will be looked for in the currently used directory and in 'lut_dir' (if no absolute paths are indicated). If HALCONROOT is set, HALCON will search the color tables in the sub-directory "'lut'".
Value: Name of the filepath
default: '\$HALCONROOT/lut' bzw. '%HALCONROOT%/lut'
- 'reference_dir':** The HTML and postscript sources of the HALCON documentation (especially the HALCON manual itself) will be looked for in the currently used directory or in 'reference_dir'. This system parameter is necessary for example using the operator `disp_info`. This parameter can also be set by the environment variable HALCONROOT before initializing HALCON. In this case the variable must indicate the directory above the help directories (that is the HALCON-Homedirectory): e.g.: '/usr/local/halcon'
Value: Name of the filepath
default: '\$HALCONROOT/doc/ps/reference/c' bzw. '%HALCONROOT%/doc/ps/reference/c'
- 'help_dir':** The online text files german or english.hlp, .sta, .key .num and .idx will be looked for in the currently used directory or in 'help_dir'. This system parameter is necessary for instance using the operators `get_operator_info` and `get_param_info`. This parameter can also be set by the environment variable HALCONROOT before initializing HALCON. In this case the variable must indicate the directory above the help directories (that is the HALCON-Homedirectory): e.g.: '/usr/local/halcon'
Value: Name of the filepath
default: '\$HALCONROOT/help' bzw. '%HALCONROOT%/help'
- 'only_lines':** (no description available) **default:** 'true'
- 'init_new_image':** Determines whether new images shall be set to 0 before using filters. This is not necessary if always the whole image is filtered or if the data of not filtered image areas are unimportant.
Value: 'true' or 'false'
default: 'true'
- 'no_object_result':** Determines how operations processing iconic objects shall react if the object tuple is empty (= no objects). Available values for **Value:**
 'true': the error will be ignored
 'false': the procedure returns FALSE
 'fail': the procedure returns FAIL
 'void': the procedure returns VOID
 'exception': an exception is raised
default: 'true'

- 'empty_region_result':** Controls the reaction of procedures concerning input objects with empty regions which actually are not useful for such objects (e.g. certain region features, segmentation, etc.). Available values for **Value**:
 'true': the error will be ignored if possible
 'false': the procedure returns FALSE
 'fail': the procedure returns FAIL
 'void': the procedure returns VOID
 'exception': an exception is raised
 default: 'true'
- 'store_empty_region':** Quite a number of operations will lead to the creation of objects with an empty region (= no image points) (e.g. [intersection](#), [threshold](#), etc.). This parameter determines whether the object with an empty region will be returned as a result ('true') or whether it will be ignored ('false') that is no result will be returned.
Value: 'true' or 'false'
 default: 'true'
- 'backing_store':** Determines whether the window content will be refreshed in case of overlapping of the windows. Some implementations of X Window are faulty; in order to avoid these errors, the storing of contents can be deactivated. It may be recommendable in some cases to deactivate the security mechanism, if e.g. performance / memory is what matters.
Value: true or false
 default: true
- 'flush_graphic':** After each HALCON procedure which creates a graphic output, a flush operation will be executed in order to display the data immediately on screen. This is not necessary with all programs (e.g. if everything is done with the help of the mouse). In this case 'flush_graphic' can be set to 'false' to improve the runlength.
Value: 'true' or 'false'
 default: 'true'
- 'flush_file':** This parameter determines whether the output into a file (also to the terminal) shall be buffered or not. If the output is to be buffered, in general the data will be displayed on the terminal only after entering the operator [fnew_line](#).
Value: 'true' or 'false'
 default: 'true'
- 'ocr_trainf_version'** This parameter determines the format that is used for writing an OCR training file. The operators [write_ocr_trainf](#), [write_ocr_trainf_image](#) and [concat_ocr_trainf](#) write training data in ASCII format for version number 1 or in binary format for version number 2. The binary version is faster in reading and writing data and stores training files more packed. The ASCII format is compatible to older HALCON releases.
Value: 1, 2
 default: 2
- 'x_package':** The output of image data via the network may cause errors owing to the heavy load on the computer or on the network. In order to avoid this, the data are transmitted in small packages. If the computer is used locally, these units can be enlarged at will. This can lead to a notably improved output performance.
Value: package size (in bytes)
 default: 20480
- 'int2_bits':** Number of significant bits of int2 images. This number is used when scaling the gray values. If the values is -1 the gray values will be automatically scaled (default).
Value: -1 or 9..15
 default: -1
- 'num_gray_4':** Number of colors to be reserved under X Window to allow the output of graylevels disp_channel on a machine with 4 bitplanes (16 colors).
 Attention! This value may only be changed before the first window has been opened on the machine.
Value: 2 - 12
 default: 8
- 'num_gray_6':** Number of colors to be reserved under X Window to allow the output of graylevels disp_channel on a machine with 6 bitplanes (64 colors).
 Attention! This value may only be changed before the first window has been opened on the machine.

Value: 2 - 62
default: 50

'num_gray_8': Number of colors to be reserved under X Window to allow the output of graylevels `disp_channel` on a machine with 8 bitplanes (256 colors).

Attention! This value may only be changed before the first window has been opened on the machine.

Value: 2 - 254
default: 140

'num_gray_percentage': Under X Window HALCON reserves a part of the available colors for the representation of gray values (`disp_channel`). This shall interfere with other X applications as little as possible. However, if HALCON does not succeed in reserving a minimum percentage of 'num_gray_percentage' of the necessary colors on the X server, a certain amount of the lookup table will be claimed for the HALCON graylevels regardless of the consequences. This may result in undesired shifts of color when switching between HALCON windows and windows of other applications, or (outside HALCON) if a window-dump is generated. The number of the real graylevels to be reserved depends on the number of available bitplanes on the outputmachine (see also 'num_gray_*'. Naturally no colors will be reserved on monochrome machines - the graylevels will instead be dithered when displayed. If graylevel-displays are used, only different shades of gray will be applied ('black', 'white', 'gray', etc.). Machines with 2 or 3 bitplanes will be considered monochrome machines, machines with 5 (7) bitplanes like machines with 4 (6) bitplanes, and machines having more than 8 bitplanes like machines with 8 bitplanes. A special case are machines providing a 24 bit display (true color machines). Naturally no colors are reserved for the display of graylevels in this case.

Note: This value may only be changed before the first window has been opened on the machine. For before opening the first window on a machine with x bitplanes, `num_gray_x` indicates the number of colors which have to be reserved for the display of graylevels, afterwards, however, it will indicate the number of colors which actually have been reserved.

Value: 0 - 100
default: 30

'num_graphic_percentage': (no description available) default: 60

'draw_mode': (no description available) default: 'complement'

'int_zooming': (no description available) default: 'true'

'ignore_colormap': (no description available) default: 'false'

'icon_name': (no description available) default: 'default'

'num_graphic_2': Number of the graphic colors to be reserved by HALCON under X Window (concerning the operators `disp_region` etc.) on a machine with 2 bitplanes (4 colors).

Attention: This value may only be changed before the first window has been opened on the machine.

Value: 0 - 2
default: 2

'num_graphic_4': Number of the graphic colors to be reserved by HALCON under X Window (concerning the operators `disp_region` etc.) on a machine with 4 bitplanes (16 colors).

Attention: This value may only be changed before the first window has been opened on the machine.

Value: 0 - 14
default: 5

'num_graphic_6': Number of the graphic colors to be reserved by HALCON under X Window (concerning the operators `disp_region` etc.) on a machine with 6 bitplanes (64 colors).

Attention: This value may only be changed before the first window has been opened on the machine.

Value: 0 - 62
default: 10

'num_graphic_8': Number of the graphic colors to be reserved by HALCON under X Window (concerning the operators `disp_region` etc.) on a machine with 8 bitplanes (256 colors).

Attention: This value may only be changed before the first window has been opened on the machine.

Value: 0 - 64
default: 20

'graphic_colors' HALCON reserves the first `num_graphic_x` colors from this list of color names as graphic colors. As a default HALCON uses this same list which is also returned by using `query_all_colors`. However, the list can be changed individually: hereby a tuple of color names will be returned as value. It is recommendable that such a tuple always includes the colors 'black' and 'white', and optionally also 'red', 'green'

and 'blue'. If 'default' is set as [Value](#), HALCON returns to the initial setting. Note: On graylevel machines not the first x colors will be reserved, but the first x shades of gray from the list.

Attention: This value may only be changed before the first window has been opened on the machine.

Value: Tuples of X Window color names

default: see also `query_all_colors`

'current_runlength_number': Regions will be stored internally in a certain runlengthcode. This parameter can determine the maximum number of chords which may be used for representing a region. Please note that some procedures raise the number on their own if necessary.

The value can be enlarged as well as reduced.

Value: maximum number of chords

default: 50000

'clip_region': Determines whether the regions of iconic objects of the HALCON database will be clipped to the currently used image size or not. This is the case for example in procedures like [gen_circle](#), [gen_rectangle1](#) or [dilation1](#).

See also: [reset_obj_db](#)

Value: 'true' or 'false'

default: 'true'

'do_low_error' Determines whether the HALCON should print low level error or not.

Value: 'true' or 'false'

default: 'false'

'reentrant' Determines whether HALCON must be reentrant for being used within a parallel programming environment (e.g. a multithreaded application). This parameter is only of importance for Parallel HALCON, which can process several operators concurrently. Thus, the parameter is ignored by the sequentially working HALCON-Version. If it is set to 'true', Parallel HALCON internally uses synchronization mechanisms to protect shared data objects from concurrent accesses. Though this is inevitable with any effectively parallel working application, it may cause undesired overhead, if used within an application which works purely sequentially. The latter case can be signalled by setting 'reentrant' to 'false'. This switches off all internal synchronization mechanisms and thus reduces overhead. Of course, Parallel HALCON then is no longer thread-safe, which causes another side-effect: Parallel HALCON will then no longer use the internal parallelization of operators, because this needs reentrancy. Setting 'reentrant' to 'true' resets Parallel HALCON to its default state, i.e. it is reentrant (and thread-safe) and it uses the automatic parallelization to speed up the processing of operators on multiprocessor machines.

Value: 'true' or 'false'

default: Parallel HALCON: 'true', otherwise: 'false'

'parallelize_operators' Determines whether Parallel HALCON uses an automatic parallelization to speed up the processing of operators on multiprocessor machines. This feature can be switched off by setting 'parallelize_operators' to 'false'. Even then Parallel HALCON will remain reentrant (and thread-safe), unless the parameter 'reentrant' is changed via [set_system](#) accordingly. Changing 'parallelize_operators' can be helpful, for example, if HALCON-operators are called by a multithreaded application, which also does the scheduling and load-balancing of operators and data by itself. Then it may be undesired that HALCON performs additional parallelization steps, which may disturb the application's scheduling and load-balancing concepts. The parameter 'parallelize_operators' is only supported by Parallel HALCON and thus ignored by the sequentially working HALCON-Version.

Value: 'true' or 'false'

default: Parallel HALCON: 'true', otherwise: 'false'

'clock_mode' Determines the method used for measuring the current time in [count_seconds](#). For **Value**=`'processor_time'`, the time used by the HALCON process is measured. For **Value**=`'elapsed_time'`, the true elapsed time is measured. For **Value**=`'performance_counter'`, under Windows the time is measured by using a performance counter, which results in a much higher accuracy than the system clock, which is used for **Value**=`'processor_time'`. On Unix systems, the modes `'processor_time'` and `'performance_counter'` are identical, because no performance counters are available on Unix systems.

'max_connection' Determines the maximum number of regions returned by [connection](#). For **Value**=0, all regions are returned.

'alloctmp_single_block' Kind of temporary memory management. **Value:** 'true' or 'false'

default: 'false'

'alloctmp_max_blocksize' Maximum size of memory blocks to be allocated within temporary memory management. (No effect, if 'alloctmp_max_blocksize' == -1 or 'alloctmp_single_block' == 'true') **Value:** -1 or 0

default: -1

'extern_alloc_funct' Pointer to external function for memory allocation of result images. default: 0

'extern_free_funct' Pointer to external function for memory deallocation of result images. default: 0

'language' Language used for error messages. **Value:** 'english' or 'german'. default: 'english'

Parameter

- ▷ **Systemparameter** (input_control) string(-array) $\leadsto$ string
Name of the system parameter to be changed.
Default Value : 'image_dir'
List of values : Systemparameter $\in \{ \text{'neighborhood', 'default_font', 'single_lut', 'update_lut', 'image_dir', 'lut_dir', 'reference_dir', 'help_dir', 'only_lines', 'init_new_image', 'no_object_result', 'empty_region_result', 'store_empty_region', 'ocr_trainf_version', 'backing_store', 'flush_graphic', 'flush_file', 'x_package', 'int2_bits', 'icon_name', 'num_gray_4', 'num_gray_6', 'num_gray_8', 'num_gray_percentage', 'num_graphic_2', 'num_graphic_4', 'num_graphic_6', 'num_graphic_8', 'num_graphic_percentage', 'draw_mode', 'int_zooming', 'ignore_colormap', 'graphic_colors', 'current_runlength_number', 'clip_region', 'update_lut', 'do_low_error', 'reentrant', 'parallelize_operators', 'clock_mode', 'max_connection', 'alloctmp_single_block', 'alloctmp_max_blocksize', 'extern_alloc_funct', 'extern_free_funct', 'language'} \}$
- ▷ **Value** (input_control) string(-array) $\leadsto$ string / integer / real
New value of the system parameter.
Default Value : 'true'
Suggested values : Value $\in \{ \text{'true', 'false', 0, 4, 8, 100, 140, 255} \}$

Result

The operator **set_system** returns the value 2 (H.MSG.TRUE) if the parameters are correct. Otherwise an exception will be raised.

Parallelization Information

set_system is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

reset_obj_db, **get_system**, **set_check**

See also

get_system, **set_check**, **count_seconds**

Module

System

14.7 Serial

clear_serial (:: SerialHandle, Channel :)

Clear the buffer of a serial connection.

clear_serial discards data written to the serial device referred to by **SerialHandle**, but not transmitted (**Channel** = 'output'), or data received, but not read (**Channel** = 'input'), or performs both these operations at once (**Channel** = 'in_out').

Parameter

- ▷ **SerialHandle** (input_control) serial_id $\leadsto$ integer
Serial interface handle.
- ▷ **Channel** (input_control) string $\leadsto$ string
Buffer to be cleared.
Default Value : 'input'
List of values : Channel $\in \{ \text{'input', 'output', 'in_out'} \}$

Result

If the parameters are correct and the buffers of the serial device could be cleared, the operator **clear_serial** returns the value 2 (H.MSG.TRUE). Otherwise an exception is raised.

Parallelization Information

clear_serial is *reentrant* and processed *without* parallelization.

	Possible Predecessors
open_serial	
	Possible Successors
read_serial , write_serial	
	See also
read_serial	
	Module
System	

close_all_serials (: : :)

Close all serial devices.

[close_all_serials](#) closes all serial devices that have been opened with [open_serial](#).

	Result
close_all_serials	returns always 2 (H_MSG_TRUE).
	Parallelization Information
close_all_serials	is <i>reentrant</i> and processed <i>without</i> parallelization.
	Possible Predecessors
open_serial	
	Alternatives
close_serial	
	See also
open_serial , close_file	
	Module
System	

close_serial (: : SerialHandle :)

Close a serial device.

[close_serial](#) closes a serial device that was opened with [open_serial](#).

	Parameter
▷ SerialHandle (input_control)	serial_id $\leadsto$ integer
	Serial interface handle.
	Result
If the parameters are correct and the device could be closed, the operator close_serial returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.	
	Parallelization Information
close_serial	is <i>reentrant</i> and processed <i>without</i> parallelization.
	Possible Predecessors
open_serial	
	See also
open_serial , close_file	
	Module
System	

```
get_serial_param ( :: SerialHandle : BaudRate, DataBits, FlowControl,
Parity, StopBits, TotalTimeOut, InterCharTimeOut )
```

Get the parameters of a serial device.

[get_serial_param](#) returns the current parameter settings of the serial device passed in [SerialHandle](#). For a description of the parameters of a serial device, see [set_serial_param](#).

Parameter

- ▷ **SerialHandle** (input_control) serial_id ~> integer
Serial interface handle.
- ▷ **BaudRate** (output_control) integer ~> integer
Speed of the serial interface.
- ▷ **DataBits** (output_control) integer ~> integer
Number of data bits of the serial interface.
- ▷ **FlowControl** (output_control) string ~> string
Type of flow control of the serial interface.
- ▷ **Parity** (output_control) string ~> string
Parity of the serial interface.
- ▷ **StopBits** (output_control) integer ~> integer
Number of stop bits of the serial interface.
- ▷ **TotalTimeOut** (output_control) integer ~> integer
Total timeout of the serial interface in ms.
- ▷ **InterCharTimeOut** (output_control) integer ~> integer
Inter-character timeout of the serial interface in ms.

Result

If the parameters are correct and the parameters of the device could be read, the operator [get_serial_param](#) returns the value 2 (H_MSG.TRUE). Otherwise an exception is raised.

Parallelization Information

[get_serial_param](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[open_serial](#)

Possible Successors

[get_serial_param](#), [read_serial](#), [write_serial](#)

See also

[set_serial_param](#)

Module

System

```
open_serial ( :: PortName : SerialHandle )
```

Open a serial device.

[open_serial](#) opens a serial device. The name of the device is determined by the parameter [PortName](#) and is operating system specific. On Windows NT machines, 'COM1'-'COM4' is typically used, while on Unix systems the serial devices usually are named '/dev/tty*'. The parameters of the serial device, e.g., its speed or number of data bits, are set to the system default values for the respective device after the device has been opened. They can be set or changed by calling [set_serial_param](#).

Parameter

- ▷ **PortName** (input_control) filename.named ~> string
Name of the serial port.
Default Value : "COM1"
Suggested values : PortName ∈ {"COM1", "COM2", "COM3", "COM4", "/dev/ttya", "/dev/ttyb",
"/dev/tty00", "/dev/tty01", "/dev/ttyd1", "/dev/ttyd2", "/dev/cua0", "/dev/cua1"}

- ▷ **SerialHandle** (output_control) serial_id $\leadsto$ integer
Serial interface handle.

Result

If the parameters are correct and the device could be opened, the operator `open_serial` returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Parallelization Information

`open_serial` is *reentrant* and processed *without* parallelization.

Possible Successors

`set_serial_param`, `read_serial`, `write_serial`, `close_serial`

See also

`set_serial_param`, `get_serial_param`, `open_file`

Module

System

read_serial (: : SerialHandle, NumCharacters : Data)

Read from a serial device.

`read_serial` tries to read `NumCharacters` from the serial device given in `SerialHandle`. The read characters are returned in `Data` as a tuple of integers. This allows to read NUL characters, which would otherwise be interpreted as the end of a string. If the timeout of the serial device has been set to a value greater than 0 with `set_serial_param`, `read_serial` waits at most as long for the arrival of the first character as indicated by the timeout. Otherwise, the operator returns immediately. In any case, the number of characters available at the time of return are passed back to the caller, i.e., fewer characters than requested can be returned. This can be checked by the length of the tuple `Data`.

Parameter

- ▷ **SerialHandle** (input_control) serial_id $\leadsto$ integer
Serial interface handle.
- ▷ **NumCharacters** (input_control) integer $\leadsto$ integer
Number of characters to read.
Default Value : 1
Suggested values : NumCharacters $\in$ {1, 2, 3, 4, 5, 10, 20, 40, 100}
- ▷ **Data** (output_control) integer(-array) $\leadsto$ integer
Read characters (as tuple of integers).

Result

If the parameters are correct and the read from the device was successful, the operator `read_serial` returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Parallelization Information

`read_serial` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`open_serial`

See also

`write_serial`

Module

System

set_serial_param (: : SerialHandle, BaudRate, DataBits, FlowControl, Parity, StopBits, TotalTimeout, InterCharTimeout :)

Set the parameters of a serial device.

`set_serial_param` can be used to set the parameters of a serial device. The parameter `BaudRate` determines the input and output speed of the device. It should be noted that not all devices support all possible speeds. The

number of sent and received data bits is set with `DataBits`. The parameter `FlowControl` determines if and what kind of data flow control should be used. In the latter case, a choice between software control (`'xon_xoff'`) and hardware control (`'cts_rts'`, `'dtr_dsr'`) can be made. If and what kind of parity check of the transmitted data should be performed can be determined by `Parity`. The number of stop bits sent is set with `StopBits`. Finally, two timeout for reading from the serial device can be set. The parameter `TotalTimeout` determines the maximum time, which may pass in `read_serial` until the first character arrives, independent of the actual number of characters requested. The parameter `InterCharTimeout` determines the time which may pass between the reading of individual characters, if multiple characters are requested with `read_serial`. If one of the timeouts is set to `-1`, a read waits an arbitrary amount of time for the arrival of characters. Thus, on Windows NT systems, a total timeout of `TotalTimeout + nInterCharTimeout` results if `n` characters are to be read. On Unix systems, only one of the two timeouts can be set. Thus, if both timeouts are passed larger than `-1`, only the total timeout is used. The unit of both timeouts is milliseconds. For each parameter, the current values can be left in effect by passing `'unchanged'`.

Parameter	
▷ SerialHandle (input_control)	serial_id $\leadsto$ integer Serial interface handle.
▷ BaudRate (input_control)	integer $\leadsto$ integer / string Speed of the serial interface. Default Value : <code>'unchanged'</code> List of values : <code>BaudRate</code> $\in$ {50, 75, 110, 134, 150, 200, 300, 600, 1200, 1800, 2400, 4800, 9600, 19200, 38400, 57600, 76800, 115200, 153600, 230400, 307200, 460800, <code>'unchanged'</code> }
▷ DataBits (input_control)	integer $\leadsto$ integer / string Number of data bits of the serial interface. Default Value : <code>'unchanged'</code> List of values : <code>DataBits</code> $\in$ {5, 6, 7, 8, <code>'unchanged'</code> }
▷ FlowControl (input_control)	string $\leadsto$ string Type of flow control of the serial interface. Default Value : <code>'unchanged'</code> List of values : <code>FlowControl</code> $\in$ { <code>'none'</code> , <code>'xon_xoff'</code> , <code>'cts_rts'</code> , <code>'dtr_dsr'</code> , <code>'unchanged'</code> }
▷ Parity (input_control)	string $\leadsto$ string Parity of the serial interface. Default Value : <code>'unchanged'</code> List of values : <code>Parity</code> $\in$ { <code>'none'</code> , <code>'odd'</code> , <code>'even'</code> , <code>'unchanged'</code> }
▷ StopBits (input_control)	integer $\leadsto$ integer / string Number of stop bits of the serial interface. Default Value : <code>'unchanged'</code> List of values : <code>StopBits</code> $\in$ {1, 2, <code>'unchanged'</code> }
▷ TotalTimeout (input_control)	integer $\leadsto$ integer / string Total timeout of the serial interface in ms. Default Value : <code>'unchanged'</code> Suggested values : <code>TotalTimeout</code> $\in$ {-1, 0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, <code>'unchanged'</code> }
▷ InterCharTimeout (input_control)	integer $\leadsto$ integer / string Inter-character timeout of the serial interface in ms. Default Value : <code>'unchanged'</code> Suggested values : <code>InterCharTimeout</code> $\in$ {-1, 0, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, <code>'unchanged'</code> }

Result
If the parameters are correct and the parameters of the device could be set, the operator `set_serial_param` returns the value 2 (H.MSG_TRUE). Otherwise an exception is raised.

Parallelization Information
`set_serial_param` is *reentrant* and processed *without parallelization*.

Possible Predecessors
`open_serial`, `get_serial_param`

Possible Successors
`read_serial`, `write_serial`

See also

[get_serial_param](#)

Module

System

write_serial (: : SerialHandle, Data :)

Write to a serial connection.

write_serial writes the characters given in **Data** to the serial device given by **SerialHandle**. The data to be written is passed as a tuple of integers. This allows to write NUL characters, which would otherwise be interpreted as the end of a string. **write_serial** always waits until all data has been transmitted, i.e., a timeout for writing cannot be set.

Parameter

▷ **SerialHandle** (input_control) serial_id $\leadsto$ integer
Serial interface handle.

▷ **Data** (input_control) integer(-array) $\leadsto$ integer
Characters to write (as tuple of integers).

Result

If the parameters are correct and the write to the device was successful, the operator **write_serial** returns the value 2 (H_MSG_TRUE). Otherwise an exception is raised.

Parallelization Information

write_serial is *reentrant* and processed *without* parallelization.

Possible Predecessors

[open_serial](#)

See also

[read_serial](#)

Module

System

14.8 Sockets

close_socket (: : Socket :)

Close a socket.

close_socket closes a socket that was previously opened with [open_socket_accept](#), [open_socket_connect](#), or [socket_accept_connect](#). For a detailed example, see [open_socket_accept](#).

Parameter

▷ **Socket** (input_control) socket_id $\leadsto$ integer
Socket number.

Parallelization Information

close_socket is *reentrant* and processed *without* parallelization.

See also

[open_socket_accept](#), [open_socket_connect](#), [socket_accept_connect](#)

Module

System

get_next_socket_data_type (: : Socket : DataType)

Determine the HALCON data type of the next socket data.

[get_next_socket_data_type](#) returns the data type of the next data that are present on the socket [Socket](#) and returns it in [DataType](#). The possible values for [DataType](#) are:

- 'no_data': No data are present.
- 'no_halcon_data': Some data are present, but they are not HALCON data.
- 'tuple': The next data is a tuple.
- 'region': The next data is a region object.
- 'image': The next data is an image object.
- 'xld_cont': The next data is an XLD contour.
- 'xld_poly': The next data is an XLD polygon.
- 'xld_para': The next data is an XLD parallel.
- 'xld_mod_para': The next data is a modified XLD parallel.
- 'xld_ext_para': The next data is an extended XLD parallel.

Parameter

- ▷ **Socket** (input_control) socket_id $\leadsto$ integer
Socket number.
- ▷ **DataType** (output_control) string $\leadsto$ string
Data type of next HALCON data.

Parallelization Information

[get_next_socket_data_type](#) is *reentrant* and processed *without* parallelization.

See also

[send_image](#), [receive_image](#), [send_region](#), [receive_region](#), [send_tuple](#), [receive_tuple](#)

Module

System

get_socket_timeout (: : Socket : Timeout)

Get the timeout of a socket.

[get_socket_timeout](#) returns the timeout for the socket connection that is passed in [Socket](#). For a description of possible values of [Timeout](#) see [set_socket_timeout](#).

Parameter

- ▷ **Socket** (input_control) socket_id $\leadsto$ integer
Socket number.
- ▷ **Timeout** (output_control) number $\leadsto$ real / string
Socket timeout.

Parallelization Information

[get_socket_timeout](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[open_socket_accept](#), [open_socket_connect](#), [socket_accept_connect](#)

See also

[set_socket_timeout](#)

Module

System

open_socket_accept (: : Port : AcceptingSocket)

Open a socket that accepts connection requests.

open_socket_accept opens a socket that accepts incoming connection requests by other HALCON processes. This operator is the necessary first step in the establishment of a communication channel between two HALCON processes. The socket listens for incoming connection requests on the port number given by **Port**. The accepting socket is returned in **AcceptingSocket**. **open_socket_accept** returns immediately without waiting for a request from another process, done by calling **open_socket_connect** in the other process. This allows multiple other processes to connect to the particular HALCON process that calls **open_socket_accept**. To accept an incoming connection request, **socket_accept_connect** must be called after another process has called **open_socket_connect**.

Parameter

- ▷ **Port** (input_control) integer $\leadsto$ integer
 Port number.
Default Value : 3000
Typical range of values : $1024 \leq \text{Port} \leq 65535$
Minimum Increment : 1
Recommended Increment : 1
- ▷ **AcceptingSocket** (output_control) socket_id $\leadsto$ integer
 Socket number.

Example

```

/* Process 1 */
dev_set_colored (12)
open_socket_accept (3000, AcceptingSocket)
/* Busy wait for an incoming connection */
dev_error_var (Error, 1)
dev_set_check ('~give_error')
OpenStatus := 5
while (OpenStatus # 2)
    socket_accept_connect (AcceptingSocket, 'false', Socket)
    OpenStatus := Error
    wait_seconds (0.2)
endwhile
dev_set_check ('give_error')
/* Connection established */
receive_image (Image, Socket)
threshold (Image, Region, 0, 63)
send_region (Region, Socket)
receive_region (ConnectedRegions, Socket)
area_center (ConnectedRegions, Area, Row, Column)
send_tuple (Socket, Area)
send_tuple (Socket, Row)
send_tuple (Socket, Column)
close_socket (Socket)
close_socket (AcceptingSocket)

/* Process 2 */
dev_set_colored (12)
open_socket_connect ('localhost', 3000, Socket)
read_image (Image, 'fabrik')
send_image (Image, Socket)
receive_region (Region, Socket)
connection (Region, ConnectedRegions)
send_region (ConnectedRegions, Socket)
receive_tuple (Socket, Area)
receive_tuple (Socket, Row)

```

receive_tuple (Socket, Column)
close_socket (Socket)

Parallelization Information

[open_socket_accept](#) is *reentrant* and processed *without parallelization*.

Possible Successors

[socket_accept_connect](#)

See also

[open_socket_connect](#), [close_socket](#), [send_image](#), [receive_image](#), [send_region](#),
[receive_region](#), [send_tuple](#), [receive_tuple](#)

Module

System

open_socket_connect (: : HostName, Port : Socket)

Open a socket to an existing socket.

[open_socket_connect](#) opens a connection to an accepting socket on the computer [HostName](#), which listens on port [Port](#). The listening socket in the other HALCON process must have been created earlier with the operator [open_socket_accept](#). The socket thus created is returned in [Socket](#). To establish the connection, the HALCON process, in which the accepting socket resides, must call [socket_accept_connect](#). For a detailed example, see [open_socket_accept](#).

Parameter

- ▷ **HostName** (input_control) string $\leadsto$ *string*
Hostname of the computer to connect to.
Default Value : 'localhost'
- ▷ **Port** (input_control) integer $\leadsto$ *integer*
Port number.
Default Value : 3000
Typical range of values : $1024 \leq \text{Port} \leq 65535$
Minimum Increment : 1
Recommended Increment : 1
- ▷ **Socket** (output_control) socket_id $\leadsto$ *integer*
Socket number.

Parallelization Information

[open_socket_connect](#) is *reentrant* and processed *without parallelization*.

Possible Successors

[send_image](#), [receive_image](#), [send_region](#), [receive_region](#), [send_tuple](#), [receive_tuple](#)

See also

[open_socket_accept](#), [socket_accept_connect](#), [close_socket](#)

Module

System

receive_image (: Image : Socket :)

Receive an image over a socket connection.

[receive_image](#) reads an image object that was sent over the socket connection determined by [Socket](#) by another HALCON process using the operator [send_image](#). If no image has been sent, the HALCON process calling [receive_image](#) blocks until enough data arrives. For a detailed example, see [open_socket_accept](#).

Parameter
▷ Image (output_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Received image. ▷ Socket (input_control) socket_id $\leadsto$ <i>integer</i> Socket number.
Parallelization Information
receive_image is <i>reentrant</i> and processed <i>without</i> parallelization.
Possible Predecessors
open_socket_connect , socket_accept_connect
See also
send_image , send_region , receive_region , send_tuple , receive_tuple , get_next_socket_data_type
Module
System

receive_region (: Region : Socket :)

Receive regions over a socket connection.

[receive_region](#) reads a region object that was sent over the socket connection determined by [Socket](#) by another HALCON process using the operator [send_region](#). If no regions have been sent, the HALCON process calling [receive_region](#) blocks until enough data arrives. For a detailed example, see [open_socket_accept](#).

Parameter
▷ Region (output_object) region(-array) $\leadsto$ <i>Hobject</i> Received regions. ▷ Socket (input_control) socket_id $\leadsto$ <i>integer</i> Socket number.
Parallelization Information
receive_region is <i>reentrant</i> and processed <i>without</i> parallelization.
Possible Predecessors
open_socket_connect , socket_accept_connect
See also
send_region , send_image , receive_image , send_tuple , receive_tuple , get_next_socket_data_type
Module
System

receive_tuple (: : Socket : Tuple)

Receive a tuple over a socket connection.

[receive_tuple](#) reads a tuple that was sent over the socket connection determined by [Socket](#) by another HALCON process using the operator [send_tuple](#). If no tuple has been sent, the HALCON process calling [receive_tuple](#) blocks until enough data arrives. For a detailed example, see [open_socket_accept](#).

Parameter
▷ Socket (input_control) socket_id $\leadsto$ <i>integer</i> Socket number. ▷ Tuple (output_control) string $\leadsto$ <i>string</i> Received tuple.

<i>Parallelization Information</i>
receive_tuple is <i>reentrant</i> and processed <i>without parallelization</i> .
<i>Possible Predecessors</i>
open_socket_connect , socket_accept_connect
<i>See also</i>
send_tuple , send_image , receive_image , send_region , receive_region , get_next_socket_data_type
<i>Module</i>
System

receive_xld (: XLD : Socket :)

Receive an XLD object over a socket connection.

[receive_xld](#) reads an XLD object that was sent over the socket connection determined by [Socket](#) by another HALCON process using the operator [send_xld](#). If no XLD object has been sent, the HALCON process calling [receive_xld](#) blocks until enough data arrives. For a detailed example, see [send_xld](#).

<i>Parameter</i>
▷ XLD (output_object) xld(-array) $\leadsto$ <i>Hobject</i> Received XLD object.
▷ Socket (input_control) socket_id $\leadsto$ <i>integer</i> Socket number.

<i>Parallelization Information</i>
receive_xld is <i>reentrant</i> and processed <i>without parallelization</i> .
<i>Possible Predecessors</i>
open_socket_connect , socket_accept_connect
<i>See also</i>
send_xld , send_image , receive_image , send_region , receive_region , send_tuple , receive_tuple , get_next_socket_data_type
<i>Module</i>
System

send_image (Image : : Socket :)

Send an image over a socket connection.

[send_image](#) sends an image object over the socket connection determined by [Socket](#). The receiving HALCON process must call [receive_image](#) to read the image from the socket. For a detailed example, see [open_socket_accept](#).

<i>Parameter</i>
▷ Image (input_object) image(-array) $\leadsto$ <i>Hobject</i> : byte / direction / cyclic / int1 / int2 / uint2 / int4 / real / complex / dvf Image to be sent.
▷ Socket (input_control) socket_id $\leadsto$ <i>integer</i> Socket number.

<i>Parallelization Information</i>
send_image is <i>reentrant</i> and processed <i>without parallelization</i> .
<i>Possible Predecessors</i>
open_socket_connect , socket_accept_connect
<i>See also</i>
receive_image , send_region , receive_region , send_tuple , receive_tuple , get_next_socket_data_type

Module

System

send_region (Region : : Socket :)

Send regions over a socket connection.

send_region sends a region object over the socket connection determined by **Socket**. The receiving HAL-CON process must call **receive_region** to read the regions from the socket. For a detailed example, see **open_socket_accept**.

Parameter

- ▷ **Region** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be sent.
- ▷ **Socket** (input_control) socket_id $\leadsto$ *integer*
Socket number.

Parallelization Information

send_region is *reentrant* and processed *without* parallelization.

Possible Predecessors

open_socket_connect, **socket_accept_connect**

See also

receive_region, **send_image**, **receive_image**, **send_tuple**, **receive_tuple**,
get_next_socket_data_type

Module

System

send_tuple (: : Socket, Tuple :)

Send a tuple over a socket connection.

send_tuple sends a tuple over the socket connection determined by **Socket**. The receiving HAL-CON process must call **receive_tuple** to read the tuple from the socket. For a detailed example, see **open_socket_accept**.

Parameter

- ▷ **Socket** (input_control) socket_id $\leadsto$ *integer*
Socket number.
- ▷ **Tuple** (input_control) string $\leadsto$ *string*
Tuple to be sent.

Parallelization Information

send_tuple is *reentrant* and processed *without* parallelization.

Possible Predecessors

open_socket_connect, **socket_accept_connect**

See also

receive_tuple, **send_image**, **receive_image**, **send_region**, **receive_region**,
get_next_socket_data_type

Module

System

send_xld (XLD : : Socket :)

Send an XLD object over a socket connection.

[send_xld](#) sends an XLD object over the socket connection determined by [Socket](#). The receiving HALCON process must call [receive_xld](#) to read the XLD object from the socket.

Parameter

- ▷ **XLD** (input_object) xld(-array) $\leadsto$ *Hobject*
XLD object to be sent.
- ▷ **Socket** (input_control) socket_id $\leadsto$ *integer*
Socket number.

Example

```

/* Process 1 */
dev_set_colored (12)
open_socket_accept (3000, AcceptingSocket)
socket_accept_connect (AcceptingSocket, 'true', Socket)
receive_image (Image, Socket)
edges_sub_pix (Image, Edges, 'canny', 1.5, 20, 40)
send_xld (Edges, Socket)
receive_xld (Polygons, Socket)
split_contours_xld (Polygons, Contours, 'polygon', 1, 5)
gen_parallel_xld (Polygons, Parallels, 10, 30, 0.15, 'true')
send_xld (Parallels, Socket)
receive_xld (ModParallels, Socket)
receive_xld (ExtParallels, Socket)
stop ()
close_socket (Socket)
close_socket (AcceptingSocket)

/* Process 2 */
dev_set_colored (12)
open_socket_connect ('localhost', 3000, Socket)
read_image (Image, 'mreut')
send_image (Image, Socket)
receive_xld (Edges, Socket)
gen_polygons_xld (Edges, Polygons, 'ramer', 2)
send_xld (Polygons, Socket)
split_contours_xld (Polygons, Contours, 'polygon', 1, 5)
receive_xld (Parallels, Socket)
mod_parallel_xld (Parallels, Image, ModParallels, ExtParallels,
                  0.4, 160, 220, 10)
send_xld (ModParallels, Socket)
send_xld (ExtParallels, Socket)
stop ()
close_socket (Socket)

```

Parallelization Information

[send_xld](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[open_socket_connect](#), [socket_accept_connect](#)

See also

[receive_xld](#), [send_image](#), [receive_image](#), [send_region](#), [receive_region](#), [send_tuple](#),
[receive_tuple](#), [get_next_socket_data_type](#)

Module

System

set_socket_timeout (: : Socket, Timeout :)

Set the timeout of a socket.

set_socket_timeout sets the timeout for the socket connection that is passed in **Socket**. The **Timeout** is only used for reading data from the socket and for calls to **socket_accept_connect**. If problems during the transmission of the data cause a timeout, the underlying protocol cannot synchronize itself with the data any longer. Therefore, in these cases, the only possibility to put the system into a consistent state is to close both sockets and to open them anew. It should be noted that sometimes while reading data no error message will be returned if the sending socket is closed while the receiving socket is waiting for data. In these cases, empty data are returned (either objects or tuples).

The timeout is given in seconds as a floating point number. It can also be set to *'infinite'*, causing the read calls to wait indefinitely.

Parameter	
▷ Socket (input_control)	socket_id $\leadsto$ integer Socket number.
▷ Timeout (input_control)	number $\leadsto$ real / integer / string Socket timeout. Default Value : 'infinite' Suggested values : Timeout $\in$ {'infinite', 0, 1, 2, 3, 4, 5, 10, 30, 60}
Parallelization Information	
set_socket_timeout is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Predecessors	
open_socket_accept , open_socket_connect , socket_accept_connect	
Possible Successors	
socket_accept_connect , receive_image , receive_region , receive_xld	
See also	
get_socket_timeout	
Module	
System	

socket_accept_connect (: : AcceptingSocket, Wait : Socket)

Accept a connection request on a listening socket.

socket_accept_connect accepts an incoming connection request, generated by **open_socket_connect** in another HALCON process, on the listening socket **AcceptingSocket**. The listening socket must have been created earlier with **open_socket_accept**. If **Wait**=*'true'*, **socket_accept_connect** waits until a connection request from another HALCON process arrives. If **Wait**=*'false'*, **socket_accept_connect** returns with the error FAIL, if currently there are no connection requests from other HALCON processes. The result of **socket_accept_connect** is another socket **Socket**, which is used for a two-way communication with another HALCON process. After this connection has been established, data can be exchanged between the two processes by calling the appropriate send or receive operators. For a detailed example, see **open_socket_accept**.

Parameter	
▷ AcceptingSocket (input_control)	socket_id $\leadsto$ integer Socket number of the accepting socket.
▷ Wait (input_control)	string $\leadsto$ string Should the operator wait until a connection request arrives? List of values : Wait $\in$ {'true', 'false'}
▷ Socket (output_control)	socket_id $\leadsto$ integer Socket number.
Parallelization Information	
socket_accept_connect is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Predecessors	
open_socket_accept	
Possible Successors	
send_image , receive_image , send_region , receive_region , send_tuple , receive_tuple	

	<i>See also</i>	
open_socket_connect , close_socket		
	<i>Module</i>	
System		

Chapter 15

Tools

15.1 Affine-Transformations

affine_trans_point_2d (: : HomMat2D, Px, Py : Qx, Qy)

Apply an arbitrary affine 2D transformation to points.

affine_trans_point_2d applies an arbitrary affine 2D transformation, i.e., scaling, rotation, translation, and slant (skewing), to the input points (**Px**,**Py**) and returns the resulting points in (**Qx**,**Qy**). The affine transformation is described by the homogeneous transformation matrix given in **HomMat2D**. This corresponds to the following equation (input and output points as homogeneous vectors):

$$\begin{pmatrix} Qx \\ Qy \\ 1 \end{pmatrix} = \text{HomMat2D} \cdot \begin{pmatrix} Px \\ Py \\ 1 \end{pmatrix}$$

If the points to transform are specified in standard image coordinates, their *row* coordinates must be passed in **Px** and their *column* coordinates in **Py**. This is necessary to obtain a right-handed coordinate system for the image. In particular, this assures that rotations are performed in the correct direction. Note that the (*x,y*) order of the matrices quite naturally corresponds to the usual (row,column) order for coordinates in the image.

The transformation matrix can be created using the operators **hom_mat2d_identity**, **hom_mat2d_rotate**, **hom_mat2d_translate**, etc., or can be the result of operators like **vector_angle_to_rigid**.

For example, if **HomMat2D** corresponds to a rigid transformation, i.e., if it consists of a rotation and a translation, the points are transformed as follows:

$$\begin{pmatrix} Qx \\ Qy \\ 1 \end{pmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} Px \\ Py \\ 1 \end{pmatrix} = \left(\mathbf{R} \cdot \begin{pmatrix} Px \\ Py \\ 1 \end{pmatrix} + \mathbf{t} \right)$$

Parameter

- ▷ **HomMat2D** (input_control) affine2d-array $\leadsto$ *real*
Input transformation matrix.
Number of elements : 6
- ▷ **Px** (input_control) point.x(-array) $\leadsto$ *real* / integer
Input point(s) (x or row coordinate).
Default Value : 64
Suggested values : $Px \in \{0, 16, 32, 64, 128, 256, 512, 1024\}$
- ▷ **Py** (input_control) point.y(-array) $\leadsto$ *real* / integer
Input point(s) (y or column coordinate).
Default Value : 64
Suggested values : $Py \in \{0, 16, 32, 64, 128, 256, 512, 1024\}$

- ▷ **Qx** (output_control) point.x(-array) $\leadsto$ *real*
Output point(s) (x or row coordinate).
- ▷ **Qy** (output_control) point.y(-array) $\leadsto$ *real*
Output point(s) (y or column coordinate).

Result

If the parameters are valid, the operator `affine_trans_point_2d` returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Parallelization Information

`affine_trans_point_2d` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`hom_mat2d_translate`, `hom_mat2d_scale`, `hom_mat2d_rotate`, `hom_mat2d_slant`

Possible Successors

`hom_mat2d_translate`, `hom_mat2d_scale`, `hom_mat2d_rotate`, `hom_mat2d_slant`

Module

Basic operators

affine_trans_point_3d (: : HomMat3D, Px, Py, Pz : Qx, Qy, Qz)

Apply an arbitrary affine 3D transformation to points.

`affine_trans_point_3d` applies an arbitrary affine 3D transformation, i.e., scaling, rotation, and translation, to the input points (**Px,Py,Pz**) and returns the resulting points in (**Qx, Qy,Qz**). The affine transformation is described by the homogeneous transformation matrix given in `HomMat3D`. This corresponds to the following equation (input and output points as homogeneous vectors):

$$\begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \text{HomMat3D} \cdot \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix}$$

The transformation matrix can be created using the operators `hom_mat3d_identity`, `hom_mat3d_scale`, `hom_mat3d_rotate`, `hom_mat3d_translate`, etc., or be the result of `pose_to_hom_mat3d`.

For example, if `HomMat3D` corresponds to a rigid transformation, i.e., if it consists of a rotation and a translation, the points are transformed as follows:

$$\begin{pmatrix} Qx \\ Qy \\ Qz \\ 1 \end{pmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} = \left(\mathbf{R} \cdot \begin{pmatrix} Px \\ Py \\ Pz \\ 1 \end{pmatrix} + \mathbf{t} \right)$$

Parameter

- ▷ **HomMat3D** (input_control) affine3d-array $\leadsto$ *real*
Input transformation matrix.
Number of elements : 12
- ▷ **Px** (input_control) point3d.x(-array) $\leadsto$ *real* / integer
Input point(s) (x coordinate).
Default Value : 64
Suggested values : Px $\in$ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **Py** (input_control) point3d.y(-array) $\leadsto$ *real* / integer
Input point(s) (y coordinate).
Default Value : 64
Suggested values : Py $\in$ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **Pz** (input_control) point3d.z(-array) $\leadsto$ *real* / integer
Input point(s) (z coordinate).
Default Value : 64
Suggested values : Pz $\in$ {0, 16, 32, 64, 128, 256, 512, 1024}

- ▷ **Qx** (output_control) point3d.x(-array) $\leadsto$ *real*
Output point(s) (x coordinate).
- ▷ **Qy** (output_control) point3d.y(-array) $\leadsto$ *real*
Output point(s) (y coordinate).
- ▷ **Qz** (output_control) point3d.z(-array) $\leadsto$ *real*
Output point(s) (z coordinate).

Result

If the parameters are valid, the operator [affine_trans_point_3d](#) returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Parallelization Information

[affine_trans_point_3d](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[hom_mat3d_translate](#), [hom_mat3d_scale](#), [hom_mat3d_rotate](#), [pose_to_hom_mat3d](#)

Possible Successors

[hom_mat3d_translate](#), [hom_mat3d_scale](#), [hom_mat3d_rotate](#), [project_3d_point](#)

Module

Basic operators

dvf_to_hom_mat2d (*VectorField* : : : *HomMat2D*)

Approximate an affine map from a displacement vector field.

[dvf_to_hom_mat2d](#) approximates an affine map from the displacement vector field [VectorField](#). The affine map is returned in [HomMat2D](#).

If the displacement vector field has been computed from the original image I_{orig} and the second image I_{res} , the internally stored transformation matrix (see [affine_trans_image](#)) contains a map that describes how to transform the first image I_{orig} to the second image I_{res} .

Parameter

- ▷ **VectorField** (input_object) singlechannel-image $\leadsto$ *Hobject* : *dvf*
Input image.
- ▷ **HomMat2D** (output_control) affine2d-array $\leadsto$ *real*
Output transformation matrix.
Number of elements : 6

Parallelization Information

[dvf_to_hom_mat2d](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[optical_flow_match](#)

Possible Successors

[affine_trans_image](#)

Alternatives

[vector_to_hom_mat2d](#)

Module

Basic operators

hom_mat2d_compose (: : *HomMat2DLeft*,
HomMat2DRight : *HomMat2DCompose*)

Multiply two homogeneous 2D transformation matrices.

[hom_mat2d_compose](#) composes a new 2D transformation matrix by multiplying the two input matrices:

$$\text{HomMat2DCompose} = \text{HomMat2DLeft} \cdot \text{HomMat2DRight}$$

For example, if the two input matrices correspond to rigid transformations, i.e., to transformations consisting of a rotation and a translation, the resulting matrix is calculated as follows:

$$\text{HomMat2DCompose} = \begin{bmatrix} \mathbf{R}_l & \mathbf{t}_l \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R}_r & \mathbf{t}_r \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_l \cdot \mathbf{R}_r & \mathbf{R}_l + \mathbf{t}_l \cdot \mathbf{t}_r \\ 0 & 0 & 1 \end{bmatrix}$$

Parameter

- ▷ **HomMat2DLeft** (input_control) affine2d-array $\leadsto$ real
Left input transformation matrix.
Number of elements : 6
- ▷ **HomMat2DRight** (input_control) affine2d-array $\leadsto$ real
Right input transformation matrix.
Number of elements : 6
- ▷ **HomMat2DCompose** (output_control) affine2d-array $\leadsto$ real
Output transformation matrix.
Number of elements : 6

Result

If the parameters are valid, the operator `hom_mat2d_compose` returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Parallelization Information

`hom_mat2d_compose` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`hom_mat2d_compose`, `hom_mat2d_translate`, `hom_mat2d_scale`, `hom_mat2d_rotate`,
`hom_mat2d_slant`

Possible Successors

`hom_mat2d_translate`, `hom_mat2d_scale`, `hom_mat2d_rotate`, `hom_mat2d_slant`

Module

Basic operators

hom_mat2d_identity (: : : HomMat2DIdentity)

Generate the homogeneous transformation matrix of the identical 2D transformation.

`hom_mat2d_identity` generates the homogeneous transformation matrix `HomMat2DIdentity` describing the identical 2D transformation:

$$\text{HomMat2DIdentity} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is not stored because it is identical for all homogeneous matrices that describe an affine transformation. Thus, `HomMat2DIdentity` is stored as the tuple [1,0,0,0,1,0].

Parameter

- ▷ **HomMat2DIdentity** (output_control) affine2d-array $\leadsto$ real
Transformation matrix.
Number of elements : 6

Result

`hom_mat2d_identity` always returns 2 (H_MSG_TRUE).

Parallelization Information

`hom_mat2d_identity` is *reentrant* and processed *without* parallelization.

Possible Successors

`hom_mat2d_translate`, `hom_mat2d_scale`, `hom_mat2d_rotate`, `hom_mat2d_slant`

Module

Basic operators

hom_mat2d_invert (: : HomMat2D : HomMat2DInvert)

Invert a homogeneous 2D transformation matrix.

hom_mat2d_invert inverts the homogeneous 2D transformation matrix given by **HomMat2D**. The resulting matrix is returned in **HomMat2DInvert**.

Parameter

- ▷ **HomMat2D** (input_control) affine2d-array $\leadsto$ real
Input transformation matrix.
Number of elements : 6
- ▷ **HomMat2DInvert** (output_control) affine2d-array $\leadsto$ real
Output transformation matrix.
Number of elements : 6

Result

hom_mat2d_invert returns 2 (H_MSG_TRUE) if the parameters are valid and the input matrix is invertible. Otherwise, an exception is raised.

Parallelization Information

hom_mat2d_invert is *reentrant* and processed *without* parallelization.

Possible Predecessors

hom_mat2d_translate, **hom_mat2d_scale**, **hom_mat2d_rotate**, **hom_mat2d_slant**

Possible Successors

hom_mat2d_translate, **hom_mat2d_scale**, **hom_mat2d_rotate**, **hom_mat2d_slant**

Module

Basic operators

hom_mat2d_rotate (: : HomMat2D, Phi, Px, Py : HomMat2DRotate)

Add a rotation to a homogeneous 2D transformation matrix.

hom_mat2d_rotate adds a rotation by the angle **Phi** to the homogeneous 2D transformation matrix **HomMat2D** and returns the resulting matrix in **HomMat2DRotate**. The rotation is described by a 2×2 rotation matrix **R**. It is performed relative to the global (i.e., fixed) coordinate system, which corresponds to the following chain of transformation matrices:

$$\text{HomMat2DRotate} = \begin{bmatrix} \mathbf{R} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat2D} \quad \mathbf{R} = \begin{bmatrix} \cos(\text{Phi}) & -\sin(\text{Phi}) \\ \sin(\text{Phi}) & \cos(\text{Phi}) \end{bmatrix}$$

The point (**Px**,**Py**) is the fixed point of the transformation, i.e., this point remains unchanged when transformed using **HomMat2DRotate**. To obtain this behavior, first a translation is added to the input transformation matrix that moves the fixed point onto the origin of the global coordinate system. Then, the rotation is added, and finally a translation that moves the fixed point back to its original position. This corresponds to the following chain of transformations:

$$\text{HomMat2DRotate} = \begin{bmatrix} 1 & 0 & +\text{Px} \\ 0 & 1 & +\text{Py} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -\text{Px} \\ 0 & 1 & -\text{Py} \\ 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat2D}$$

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & tc \\ rd & re & tf \\ 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, tc, rd, re, tf].

Parameter

- ▷ **HomMat2D** (input_control) affine2d-array $\leadsto$ real
Input transformation matrix.
Number of elements : 6
- ▷ **Phi** (input_control) angle.rad $\leadsto$ real / integer
Rotation angle.
Default Value : 0.78
Suggested values : Phi $\in \{0.1, 0.2, 0.3, 0.4, 0.78, 1.57, 3.14\}$
Typical range of values : 0 $\leq$ Phi $\leq$ 6.28318530718
- ▷ **Px** (input_control) point.x $\leadsto$ real / integer
Fixed point of the transformation (x coordinate).
Default Value : 0
Suggested values : Px $\in \{0, 16, 32, 64, 128, 256, 512, 1024\}$
- ▷ **Py** (input_control) point.y $\leadsto$ real / integer
Fixed point of the transformation (y coordinate).
Default Value : 0
Suggested values : Py $\in \{0, 16, 32, 64, 128, 256, 512, 1024\}$
- ▷ **HomMat2DRotate** (output_control) affine2d-array $\leadsto$ real
Output transformation matrix.
Number of elements : 6

Result

If the parameters are valid, the operator `hom_mat2d_rotate` returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Parallelization Information

`hom_mat2d_rotate` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`hom_mat2d_identity`, `hom_mat2d_translate`, `hom_mat2d_scale`, `hom_mat2d_rotate`, `hom_mat2d_slant`

Possible Successors

`hom_mat2d_translate`, `hom_mat2d_scale`, `hom_mat2d_rotate`, `hom_mat2d_slant`

Module

Basic operators

hom_mat2d_scale (: : HomMat2D, Sx, Sy, Px, Py : HomMat2DScale)

Add a scaling to a homogeneous 2D transformation matrix.

`hom_mat2d_scale` adds a scaling by the scale factors `Sx` and `Sy` to the homogeneous 2D transformation matrix `HomMat2D` and returns the resulting matrix in `HomMat2DScale`. The scaling is described by a 2×2 scaling matrix `S`. It is performed relative to the global (i.e., fixed) coordinate system; this corresponds to the following chain of transformation matrices:

$$\text{HomMat2DScale} = \begin{bmatrix} \text{S} & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat2D} \quad \text{S} = \begin{bmatrix} \text{Sx} & 0 \\ 0 & \text{Sy} \end{bmatrix}$$

The point (`Px`,`Py`) is the fixed point of the transformation, i.e., this point remains unchanged when transformed using `HomMat2DScale`. To obtain this behavior, first a translation is added to the input transformation matrix that moves the fixed point onto the origin of the global coordinate system. Then, the scaling is added, and finally a translation that moves the fixed point back to its original position. This corresponds to the following chain of transformations:

$$\text{HomMat2DScale} = \begin{bmatrix} 1 & 0 & +Px \\ 0 & 1 & +Py \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} S & 0 \\ 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -Px \\ 0 & 1 & -Py \\ 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat2D}$$

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & tc \\ rd & re & tf \\ 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, tc, rd, re, tf].

Parameter

- ▷ **HomMat2D** (input_control) affine2d-array $\leadsto$ real
Input transformation matrix.
Number of elements : 6
- ▷ **Sx** (input_control) number $\leadsto$ real / integer
Scale factor along the x-axis.
Default Value : 2
Suggested values : Sx $\in$ {0.125, 0.25, 0.5, 1, 2, 4, 8, 16}
Restriction : Sx $\neq$ 0
- ▷ **Sy** (input_control) number $\leadsto$ real / integer
Scale factor along the y-axis.
Default Value : 2
Suggested values : Sy $\in$ {0.125, 0.25, 0.5, 1, 2, 4, 8, 16}
Restriction : Sy $\neq$ 0
- ▷ **Px** (input_control) point.x $\leadsto$ real / integer
Fixed point of the transformation (x coordinate).
Default Value : 0
Suggested values : Px $\in$ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **Py** (input_control) point.y $\leadsto$ real / integer
Fixed point of the transformation (y coordinate).
Default Value : 0
Suggested values : Py $\in$ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **HomMat2DScale** (output_control) affine2d-array $\leadsto$ real
Output transformation matrix.
Number of elements : 6

Result

hom_mat2d_scale returns 2 (H_MSG_TRUE) if both scale factors are not 0. If necessary, an exception is raised.

Parallelization Information

hom_mat2d_scale is *reentrant* and processed *without* parallelization.

Possible Predecessors

hom_mat2d_identity, **hom_mat2d_translate**, **hom_mat2d_scale**, **hom_mat2d_rotate**,
hom_mat2d_slant

Possible Successors

hom_mat2d_translate, **hom_mat2d_scale**, **hom_mat2d_rotate**, **hom_mat2d_slant**

Module

Basic operators

hom_mat2d_slant (: : HomMat2D, Theta, Axis, Px, Py : HomMat2DSlant)

Add a slant to a homogeneous 2D transformation matrix.

`hom_mat2d_slant` adds a slant by the angle `Theta` to the homogeneous 2D transformation matrix `HomMat2D` and returns the resulting matrix in `HomMat2DSlant`. A slant is an affine transformation in which one coordinate axis remains fixed, while the other coordinate axis is rotated counterclockwise by an angle `Theta`. The parameter `Axis` determines which coordinate axis is slanted. For `Axis = 'x'`, the x-axis is slanted and the y-axis remains fixed, while for `Axis = 'y'` the y-axis is slanted and the x-axis remains fixed. The slanting is performed relative to the global (i.e., fixed) coordinate system; this corresponds to the following chains of transformation matrices:

$$\begin{aligned} \text{Axis} = 'x' : \quad \text{HomMat2DSlant} &= \begin{bmatrix} \cos(\text{Theta}) & 0 & 0 \\ \sin(\text{Theta}) & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat2D} \\ \text{Axis} = 'y' : \quad \text{HomMat2DSlant} &= \begin{bmatrix} 1 & -\sin(\text{Theta}) & 0 \\ 0 & \cos(\text{Theta}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat2D} \end{aligned}$$

The point (P_x, P_y) is the fixed point of the transformation, i.e., this point remains unchanged when transformed using `HomMat2DSlant`. To obtain this behavior, first a translation is added to the input transformation matrix that moves the fixed point onto the origin of the global coordinate system. Then, the slant is added, and finally a translation that moves the fixed point back to its original position. This corresponds to the following chain of transformations for `Axis = 'x'`:

$$\text{HomMat2DSlant} = \begin{bmatrix} 1 & 0 & +P_x \\ 0 & 1 & +P_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(\text{Theta}) & 0 & 0 \\ \sin(\text{Theta}) & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & -P_x \\ 0 & 1 & -P_y \\ 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat2D}$$

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} r_a & r_b & t_c \\ r_d & r_e & t_f \\ 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple `[ra, rb, tc, rd, re, tf]`.

Parameter

- ▷ **HomMat2D** (input_control) affine2d-array $\leadsto$ real
Input transformation matrix.
Number of elements : 6
- ▷ **Theta** (input_control) angle.rad $\leadsto$ real / integer
Slant angle.
Default Value : 0.78
Suggested values : $\text{Theta} \in \{0.1, 0.2, 0.3, 0.4, 0.78, 1.57, 3.14\}$
Typical range of values : $0 \leq \text{Theta} \leq 6.28318530718$
- ▷ **Axis** (input_control) string $\leadsto$ string
Coordinate axis that is slanted.
Default Value : 'x'
List of values : $\text{Axis} \in \{'x', 'y'\}$
- ▷ **Px** (input_control) point.x $\leadsto$ real / integer
Fixed point of the transformation (x coordinate).
Default Value : 0
Suggested values : $P_x \in \{0, 16, 32, 64, 128, 256, 512, 1024\}$
- ▷ **Py** (input_control) point.y $\leadsto$ real / integer
Fixed point of the transformation (y coordinate).
Default Value : 0
Suggested values : $P_y \in \{0, 16, 32, 64, 128, 256, 512, 1024\}$
- ▷ **HomMat2DSlant** (output_control) affine2d-array $\leadsto$ real
Output transformation matrix.
Number of elements : 6

Result

If the parameters are valid, the operator `hom_mat2d_slant` returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Parallelization Information

`hom_mat2d_slant` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`hom_mat2d_identity`, `hom_mat2d_translate`, `hom_mat2d_scale`, `hom_mat2d_rotate`, `hom_mat2d_slant`

Possible Successors

`hom_mat2d_translate`, `hom_mat2d_scale`, `hom_mat2d_rotate`, `hom_mat2d_slant`

Module

Basic operators

hom_mat2d_to_affine_par (: : HomMat2D : Sx, Sy, Phi, Theta, Tx, Ty)

Compute the affine transformation parameters from a homogeneous 2D transformation matrix.

`hom_mat2d_to_affine_par` computes the affine transformation parameters corresponding to the homogeneous 2D transformation matrix `HomMat2D`. The parameters `Sx` and `Sy` determine how the transformation scales the original x- and y-axes, respectively. The two scaling factors are always positive. The angle `Theta` describes whether the transformed coordinate axes are orthogonal (`Theta` = 0) or slanted. If $|\text{Theta}| > \pi/2$, the transformation contains a reflection. The angle `Phi` determines the rotation of the transformed x-axis with respect to the original x-axis. The parameters `Tx` and `Ty` determine the translation of the two coordinate systems. The matrix `HomMat2D` can be constructed from the six transformation parameters by the following operator sequence:

```
hom_mat2d_identity (HomMat2DIdentity)
hom_mat2d_scale (HomMat2DIdentity, Sx, Sy, 0, 0, HomMat2DScale)
hom_mat2d_slant (HomMat2DScale, Theta, 'y', 0, 0, HomMat2DSlant)
hom_mat2d_rotate (HomMat2DSlant, Phi, 0, 0, HomMat2DRotate)
hom_mat2d_translate (HomMat2DRotate, Tx, Ty, HomMat2D)
```

This is equivalent to the following chain of transformation matrices:

$$\text{HomMat2D} = \begin{bmatrix} 1 & 0 & +\text{Tx} \\ 0 & 1 & +\text{Ty} \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(\text{Phi}) & -\sin(\text{Phi}) & 0 \\ \sin(\text{Phi}) & \cos(\text{Phi}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & -\sin(\text{Theta}) & 0 \\ 0 & \cos(\text{Theta}) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \text{Sx} & 0 & 0 \\ 0 & \text{Sy} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Parameter

- ▷ **HomMat2D** (input_control) affine2d-array $\leadsto$ real
Input transformation matrix.
Number of elements : 6
- ▷ **Sx** (output_control) real $\leadsto$ real
Scaling factor along the x direction.
- ▷ **Sy** (output_control) real $\leadsto$ real
Scaling factor along the y direction.
- ▷ **Phi** (output_control) angle.rad $\leadsto$ real
Rotation angle.
- ▷ **Theta** (output_control) angle.rad $\leadsto$ real
Slant angle.
- ▷ **Tx** (output_control) point.x $\leadsto$ real
Translation along the x direction.
- ▷ **Ty** (output_control) point.y $\leadsto$ real
Translation along the y direction.

Result

If the matrix `HomMat2D` is non-degenerate, `hom_mat2d_to_affine_par` returns 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Parallelization Information

`hom_mat2d_to_affine_par` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`vector_to_hom_mat2d`, `vector_to_rigid`, `vector_to_similarity`

Possible Successors

`hom_mat2d_translate`, `hom_mat2d_scale`, `hom_mat2d_rotate`, `hom_mat2d_slant`

Module

Basic operators

hom_mat2d_translate (: : HomMat2D, Tx, Ty : HomMat2DTranslate)

Add a translation to a homogeneous 2D transformation matrix.

`hom_mat2d_translate` adds a translation by the vector $\mathbf{t} = (Tx, Ty)$ to the homogeneous 2D transformation matrix `HomMat2D` and returns the resulting matrix in `HomMat2DTranslate`. The translation is performed relative to the global (i.e., fixed) coordinate system; this corresponds to the following chain of transformation matrices:

$$\text{HomMat2DTranslate} = \begin{bmatrix} 1 & 0 & \mathbf{t} \\ 0 & 1 & \\ 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat2D} \quad \mathbf{t} = \begin{pmatrix} Tx \\ Ty \end{pmatrix}$$

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & tc \\ rd & re & tf \\ 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple `[ra, rb, tc, rd, re, tf]`.

Parameter

- ▷ **HomMat2D** (input_control) affine2d-array $\leadsto$ *real*
Input transformation matrix.
Number of elements : 6
- ▷ **Tx** (input_control) point.x $\leadsto$ *real* / integer
Translation along the x-axis.
Default Value : 64
Suggested values : $Tx \in \{0, 16, 32, 64, 128, 256, 512, 1024\}$
- ▷ **Ty** (input_control) point.y $\leadsto$ *real* / integer
Translation along the y-axis.
Default Value : 64
Suggested values : $Ty \in \{0, 16, 32, 64, 128, 256, 512, 1024\}$
- ▷ **HomMat2DTranslate** (output_control) affine2d-array $\leadsto$ *real*
Output transformation matrix.
Number of elements : 6

Result

If the parameters are valid, the operator `hom_mat2d_translate` returns 2 (`H_MSG_TRUE`). If necessary, an exception is raised.

Parallelization Information

`hom_mat2d_translate` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`hom_mat2d_identity`, `hom_mat2d_translate`, `hom_mat2d_scale`, `hom_mat2d_rotate`, `hom_mat2d_slant`

Possible Successors

`hom_mat2d_translate`, `hom_mat2d_scale`, `hom_mat2d_rotate`, `hom_mat2d_slant`

Module

Basic operators

```
hom_mat3d_compose ( : : HomMat3DLeft,
HomMat3DRight : HomMat3DCompose )
```

Multiply two homogeneous 3D transformation matrices.

[hom_mat3d_compose](#) composes a new 3D transformation matrix by multiplying the two input matrices:

$$\text{HomMat3DCompose} = \text{HomMat3DLeft} \cdot \text{HomMat3DRight}$$

For example, if the two input matrices correspond to rigid transformations, i.e., to transformations consisting of a rotation and a translation, the resulting matrix is calculated as follows:

$$\text{HomMat3DCompose} = \begin{bmatrix} \mathbf{R}_l & \mathbf{t}_l \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R}_r & \mathbf{t}_r \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}_l \cdot \mathbf{R}_r & \mathbf{R}_l \cdot \mathbf{t}_r + \mathbf{t}_l \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Parameter

- ▷ **HomMat3DLeft** (input_control) affine3d-array $\leadsto$ real
Left input transformation matrix.
Number of elements : 12
- ▷ **HomMat3DRight** (input_control) affine3d-array $\leadsto$ real
Right input transformation matrix.
Number of elements : 12
- ▷ **HomMat3DCompose** (output_control) affine3d-array $\leadsto$ real
Output transformation matrix.
Number of elements : 12

Result

If the parameters are valid, the operator [hom_mat3d_compose](#) returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Parallelization Information

[hom_mat3d_compose](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[hom_mat3d_compose](#), [hom_mat3d_translate](#), [hom_mat3d_scale](#), [hom_mat3d_rotate](#),
[pose_to_hom_mat3d](#)

Possible Successors

[hom_mat3d_translate](#), [hom_mat3d_scale](#), [hom_mat3d_rotate](#)

See also

[affine_trans_point_3d](#), [hom_mat3d_identity](#), [hom_mat3d_rotate](#), [hom_mat3d_translate](#),
[pose_to_hom_mat3d](#), [hom_mat3d_to_pose](#)

Module

Basic operators

```
hom_mat3d_identity ( : : : HomMat3DIdentity )
```

Generate the homogeneous transformation matrix of the identical 3D transformation.

[hom_mat3d_identity](#) generates the homogeneous transformation matrix [HomMat3DIdentity](#) describing the identical 3D transformation:

$$\text{HomMat3DIdentity} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is not stored because it is identical for all homogeneous matrices that describe an affine transformation. Thus, **HomMat3DIdentity** is stored as the tuple [1,0,0,0,0,1,0,0,0,0,1,0].

Parameter

- ▷ **HomMat3DIdentity** (output_control) affine3d-array $\leadsto$ real
Transformation matrix.
Number of elements : 12

Result

hom_mat3d_identity always returns 2 (H_MSG_TRUE).

Parallelization Information

hom_mat3d_identity is *reentrant* and processed *without* parallelization.

Possible Successors

hom_mat3d_translate, **hom_mat3d_scale**, **hom_mat3d_rotate**

Alternatives

pose_to_hom_mat3d

Module

Basic operators

hom_mat3d_invert (: : HomMat3D : HomMat3DInvert)

Invert a homogeneous 3D transformation matrix.

hom_mat3d_invert inverts the homogeneous 3D transformation matrix given by **HomMat3D**. The resulting matrix is returned in **HomMat3DInvert**.

Parameter

- ▷ **HomMat3D** (input_control) affine3d-array $\leadsto$ real
Input transformation matrix.
Number of elements : 12
- ▷ **HomMat3DInvert** (output_control) affine3d-array $\leadsto$ real
Output transformation matrix.
Number of elements : 12

Result

hom_mat3d_invert returns 2 (H_MSG_TRUE) if the parameters are valid and the input matrix is invertible. Otherwise, an exception is raised.

Parallelization Information

hom_mat3d_invert is *reentrant* and processed *without* parallelization.

Possible Predecessors

hom_mat3d_translate, **hom_mat3d_scale**, **hom_mat3d_rotate**, **pose_to_hom_mat3d**

Possible Successors

hom_mat3d_translate, **hom_mat3d_scale**, **hom_mat3d_rotate**, **hom_mat3d_to_pose**

See also

affine_trans_point_3d, **hom_mat3d_identity**, **hom_mat3d_rotate**, **hom_mat3d_translate**, **pose_to_hom_mat3d**, **hom_mat3d_to_pose**, **hom_mat3d_compose**

Module

Basic operators

hom_mat3d_rotate (: : HomMat3D, Phi, Axis, Px, Py, Pz : HomMat3DRotate)

Add a rotation to a homogeneous 3D transformation matrix.

`hom_mat3d_rotate` adds a rotation around the `Axis`-axis by the angle `Phi` to the homogeneous 3D transformation matrix `HomMat3D` and returns the resulting matrix in `HomMat3DRotate`. The rotation is described by a 3×3 rotation matrix $\mathbf{R}$. It is performed relative to the global (i.e., fixed) coordinate system; this corresponds to the following chain of transformation matrices:

`Axis` = 'x':

$$\text{HomMat3DRotate} = \begin{bmatrix} & 0 \\ \mathbf{R}_x & 0 \\ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat3D} \quad \mathbf{R}_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\text{Phi}) & -\sin(\text{Phi}) \\ 0 & \sin(\text{Phi}) & \cos(\text{Phi}) \end{bmatrix}$$

`Axis` = 'y':

$$\text{HomMat3DRotate} = \begin{bmatrix} & 0 \\ \mathbf{R}_y & 0 \\ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat3D} \quad \mathbf{R}_y = \begin{bmatrix} \cos(\text{Phi}) & 0 & \sin(\text{Phi}) \\ 0 & 1 & 0 \\ -\sin(\text{Phi}) & 0 & \cos(\text{Phi}) \end{bmatrix}$$

`Axis` = 'z':

$$\text{HomMat3DRotate} = \begin{bmatrix} & 0 \\ \mathbf{R}_z & 0 \\ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat3D} \quad \mathbf{R}_z = \begin{bmatrix} \cos(\text{Phi}) & -\sin(\text{Phi}) & 0 \\ \sin(\text{Phi}) & \cos(\text{Phi}) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

The point `(Px,Py,Pz)` is the fixed point of the transformation, i.e., this point remains unchanged when transformed using `HomMat3DRotate`. To obtain this behavior, first a translation is added to the input transformation matrix that moves the fixed point onto the origin of the global coordinate system. Then, the rotation is added, and finally a translation that moves the fixed point back to its original position. This corresponds to the following chain of transformations:

$$\text{HomMat3DRotate} = \begin{bmatrix} 1 & 0 & 0 & +\text{Px} \\ 0 & 1 & 0 & +\text{Py} \\ 0 & 0 & 1 & +\text{Pz} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} & 0 \\ \mathbf{R} & 0 \\ & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -\text{Px} \\ 0 & 1 & 0 & -\text{Py} \\ 0 & 0 & 1 & -\text{Pz} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat3D}$$

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & rc & td \\ re & rf & rg & th \\ ri & rj & rk & tl \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple `[ra, rb, rc, td, re, rf, rg, th, ri, rj, rk, tl]`.

Parameter

- ▷ **HomMat3D** (input_control) affine3d-array $\leadsto$ real
Input transformation matrix.
Number of elements : 12
- ▷ **Phi** (input_control) angle.rad $\leadsto$ real / integer
Rotation angle.
Default Value : 0.78
Suggested values : `Phi` $\in$ {0.1, 0.2, 0.3, 0.4, 0.78, 1.57, 3.14}
Typical range of values : $0 \leq \text{Phi} \leq 6.28318530718$
- ▷ **Axis** (input_control) string $\leadsto$ string
Axis, to be rotated around.
Default Value : "x"
Suggested values : `Axis` $\in$ {"x", "y", "z"}

- ▷ **Px** (input_control) point3d.x $\leadsto$ real / integer
Fixed point of the transformation (x coordinate).
Default Value : 0
Suggested values : Px $\in$ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **Py** (input_control) point3d.y $\leadsto$ real / integer
Fixed point of the transformation (y coordinate).
Default Value : 0
Suggested values : Py $\in$ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **Pz** (input_control) point3d.z $\leadsto$ real / integer
Fixed point of the transformation (z coordinate).
Default Value : 0
Suggested values : Pz $\in$ {0, 16, 32, 64, 128, 256, 512, 1024}
- ▷ **HomMat3DRotate** (output_control) affine3d-array $\leadsto$ real
Output transformation matrix.
Number of elements : 12

Result

If the parameters are valid, the operator `hom_mat3d_rotate` returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Parallelization Information

`hom_mat3d_rotate` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`hom_mat3d_identity`, `hom_mat3d_translate`, `hom_mat3d_scale`, `hom_mat3d_rotate`

Possible Successors

`hom_mat3d_translate`, `hom_mat3d_scale`, `hom_mat3d_rotate`

See also

`hom_mat3d_invert`, `hom_mat3d_identity`, `hom_mat3d_rotate`, `hom_mat3d_translate`,
`pose_to_hom_mat3d`, `hom_mat3d_to_pose`, `hom_mat3d_compose`

Module

Basic operators

hom_mat3d_scale (: : HomMat3D, Sx, Sy, Sz, Px, Py,
Pz : HomMat3DScale)

Add a scaling to a homogeneous 3D transformation matrix.

`hom_mat3d_scale` adds a scaling by the scale factors `Sx`, `Sy`, and `Sz` to the homogeneous 3D transformation matrix `HomMat3D` and returns the resulting matrix in `HomMat3DScale`. The scaling is described by a 3×3 scaling matrix `S`. It is performed relative to the global (i.e., fixed) coordinate system; this corresponds to the following chain of transformation matrices:

$$\text{HomMat3DScale} = \begin{bmatrix} & 0 \\ \mathbf{S} & 0 \\ & 0 \\ 0\ 0\ 0 & 1 \end{bmatrix} \cdot \text{HomMat3D} \quad \mathbf{S} = \begin{bmatrix} \text{Sx} & 0 & 0 \\ 0 & \text{Sy} & 0 \\ 0 & 0 & \text{Sz} \end{bmatrix}$$

The point (`Px`,`Py`,`Pz`) is the fixed point of the transformation, i.e., this point remains unchanged when transformed using `HomMat3DScale`. To obtain this behavior, first a translation is added to the input transformation matrix that moves the fixed point onto the origin of the global coordinate system. Then, the scaling is added, and finally a translation that moves the fixed point back to its original position. This corresponds to the following chain of transformations:

$$\text{HomMat3DScale} = \begin{bmatrix} 1 & 0 & 0 & +\text{Px} \\ 0 & 1 & 0 & +\text{Py} \\ 0 & 0 & 1 & +\text{Pz} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} & 0 \\ \mathbf{S} & 0 \\ & 0 \\ 0\ 0\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -\text{Px} \\ 0 & 1 & 0 & -\text{Py} \\ 0 & 0 & 1 & -\text{Pz} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat3D}$$

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & rc & td \\ re & rf & rg & th \\ ri & rj & rk & tl \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, rc, td, re, rf, rg, th, ri, rj, rk, tl].

Parameter

- ▷ **HomMat3D** (input_control) affine3d-array $\leadsto$ real
Input transformation matrix.
Number of elements : 12
- ▷ **Sx** (input_control) number $\leadsto$ real / integer
Scale factor along the x-axis.
Default Value : 2
Suggested values : Sx $\in \{0.125, 0.25, 0.5, 1, 2, 4, 8, 112\}$
Restriction : Sx $\neq 0$
- ▷ **Sy** (input_control) number $\leadsto$ real / integer
Scale factor along the y-axis.
Default Value : 2
Suggested values : Sy $\in \{0.125, 0.25, 0.5, 1, 2, 4, 8, 112\}$
Restriction : Sy $\neq 0$
- ▷ **Sz** (input_control) number $\leadsto$ real / integer
Scale factor along the z-axis.
Default Value : 2
Suggested values : Sz $\in \{0.125, 0.25, 0.5, 1, 2, 4, 8, 112\}$
Restriction : Sz $\neq 0$
- ▷ **Px** (input_control) point3d.x $\leadsto$ real / integer
Fixed point of the transformation (x coordinate).
Default Value : 0
Suggested values : Px $\in \{0, 16, 32, 64, 128, 256, 512, 1024\}$
- ▷ **Py** (input_control) point3d.y $\leadsto$ real / integer
Fixed point of the transformation (y coordinate).
Default Value : 0
Suggested values : Py $\in \{0, 16, 32, 64, 128, 256, 512, 1024\}$
- ▷ **Pz** (input_control) point3d.z $\leadsto$ real / integer
Fixed point of the transformation (z coordinate).
Default Value : 0
Suggested values : Pz $\in \{0, 16, 32, 64, 128, 256, 512, 1024\}$
- ▷ **HomMat3DScale** (output_control) affine3d-array $\leadsto$ real
Output transformation matrix.
Number of elements : 12

Result

[hom_mat3d_scale](#) returns 2 (H_MSG_TRUE) if all three scale factors are not 0. If necessary, an exception is raised.

Parallelization Information

[hom_mat3d_scale](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[hom_mat3d_identity](#), [hom_mat3d_translate](#), [hom_mat3d_scale](#), [hom_mat3d_rotate](#)

Possible Successors

[hom_mat3d_translate](#), [hom_mat3d_scale](#), [hom_mat3d_rotate](#)

Module

Basic operators

hom_mat3d_translate (: : HomMat3D, Tx, Ty, Tz : HomMat3DTranslate)

Add a translation to a homogeneous 3D transformation matrix.

hom_mat3d_translate adds a translation by the vector $\mathbf{t} = (Tx, Ty, Tz)$ to the homogeneous 3D transformation matrix **HomMat3D** and returns the resulting matrix in **HomMat3DTranslate**. The translation is performed relative to the global (i.e., fixed) coordinate system; this corresponds to the following chain of transformation matrices:

$$\text{HomMat3DTranslate} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \text{HomMat3D} \quad \mathbf{t} = \begin{pmatrix} Tx \\ Ty \\ Tz \end{pmatrix}$$

Attention

Note that homogeneous matrices are stored row-by-row as a tuple; the last row is not stored because it is identical for all homogeneous matrices that describe an affine transformation. For example, the homogeneous matrix

$$\begin{bmatrix} ra & rb & rc & td \\ re & rf & rg & th \\ ri & rj & rk & tl \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

is stored as the tuple [ra, rb, rc, td, re, rf, rg, th, ri, rj, rk, tl].

Parameter

- ▷ **HomMat3D** (input_control) affine3d-array $\leadsto$ *real*
Input transformation matrix.
Number of elements : 12
- ▷ **Tx** (input_control) point3d.x $\leadsto$ *real* / integer
Translation along the x-axis.
Default Value : 64
Suggested values : Tx $\in \{0, 16, 32, 64, 128, 256, 512, 1024\}$
- ▷ **Ty** (input_control) point3d.y $\leadsto$ *real* / integer
Translation along the y-axis.
Default Value : 64
Suggested values : Ty $\in \{0, 16, 32, 64, 128, 256, 512, 1024\}$
- ▷ **Tz** (input_control) point3d.z $\leadsto$ *real* / integer
Translation along the z-axis.
Default Value : 64
Suggested values : Tz $\in \{0, 16, 32, 64, 128, 256, 512, 1024\}$
- ▷ **HomMat3DTranslate** (output_control) affine3d-array $\leadsto$ *real*
Output transformation matrix.
Number of elements : 12

Result

If the parameters are valid, the operator **hom_mat3d_translate** returns 2 (H_MSG_TRUE). If necessary, an exception is raised.

Parallelization Information

hom_mat3d_translate is *reentrant* and processed *without* parallelization.

Possible Predecessors

hom_mat3d_identity, **hom_mat3d_translate**, **hom_mat3d_scale**, **hom_mat3d_rotate**

Possible Successors

hom_mat3d_translate, **hom_mat3d_scale**, **hom_mat3d_rotate**

See also

hom_mat3d_invert, **hom_mat3d_identity**, **hom_mat3d_rotate**, **hom_mat3d_translate**, **pose_to_hom_mat3d**, **hom_mat3d_to_pose**, **hom_mat3d_compose**

Module

Basic operators

vector_angle_to_rigid (: : Row1, Column1, Angle1, Row2, Column2, Angle2 : HomMat2D)

Compute a rigid affine transformation from points and angles.

vector_angle_to_rigid computes a rigid affine transformation, i.e., a transformation consisting of a rotation and a translation, from a point correspondence and two corresponding angles and returns it in form of the homogeneous transformation matrix **HomMat2D**. The matrix consists of 2 components: a rotation matrix **R** and a translation vector **t** (also see **hom_mat2d_rotate** and **hom_mat2d_translate**):

$$\text{HomMat2D} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \mathbf{t} \\ 0 & 1 & \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R} & 0 \\ 0 & 1 \end{bmatrix} = \mathbf{H}(\mathbf{t}) \cdot \mathbf{H}(\mathbf{R})$$

The coordinates of the original point are passed in (**Row1,Column1**), while the corresponding angle is passed in **Angle1**. The coordinates of the transformed point are passed in (**Row2,Column2**), while the corresponding angle is passed in **Angle2**. The following equation describes the transformation of the point using homogeneous vectors:

$$\begin{pmatrix} \text{Row2} \\ \text{Column2} \\ 1 \end{pmatrix} = \text{HomMat2D} \cdot \begin{pmatrix} \text{Row1} \\ \text{Column1} \\ 1 \end{pmatrix}$$

In particular, the operator **vector_angle_to_rigid** is useful to construct a rigid affine transformation from the results of the matching operators (e.g., **find_shape_model** or **best_match_rot_mg**), which transforms a reference image to the current image or (if the parameters are passed in reverse order) from the current image to the reference image.

HomMat2D can be used directly with operators that transform data using affine transformations, e.g., **affine_trans_image**.

Parameter

- ▷ **Row1** (input_control) point.y $\leadsto$ real / integer
Row coordinate of the original point.
 - ▷ **Column1** (input_control) point.x $\leadsto$ real / integer
Column coordinate of the original point.
 - ▷ **Angle1** (input_control) angle.rad $\leadsto$ real / integer
Angle of the original point.
 - ▷ **Row2** (input_control) point.y $\leadsto$ real / integer
Row coordinate of the transformed point.
 - ▷ **Column2** (input_control) point.x $\leadsto$ real / integer
Column coordinate of the transformed point.
 - ▷ **Angle2** (input_control) angle.rad $\leadsto$ real / integer
Angle of the transformed point.
 - ▷ **HomMat2D** (output_control) affine2d-array $\leadsto$ real
Output transformation matrix.
- Number of elements : 6**

Example

```
draw_rectangle2 (WindowID, RowTempl, ColumnTempl, PhiTempl, Length1, Length2)
gen_rectangle2 (Rectangle, RowTempl, ColumnTempl, PhiTempl, Length1, Length2)
reduce_domain (ImageTempl, Rectangle, ImageReduced)
create_template_rot (ImageReduced, 4, 0, rad(360), rad(1), 'sort',
                    'original', TemplateID)
while (true)
    best_match_rot_mg (Image, TemplateID, 0, rad(360), 30, 'true', 4, Row,
                      Column, Angle, ErrMatch)
    if (ErrMatch<255)
        vector_angle_to_rigid (Row, Column, Angle, RowTempl,
```

```

        ColumnTempl, 0, HomMat2D)
    affine_trans_image (Image, ImageAffinTrans, HomMat2D, 'constant',
        'false')
endif
endwhile
clear_template (TemplateID)

```

Parallelization Information

[vector_angle_to_rigid](#) is *reentrant* and processed *without parallelization*.

Possible Predecessors

[best_match_rot_mg](#), [best_match_rot](#)

Possible Successors

[hom_mat2d_invert](#), [affine_trans_image](#), [affine_trans_region](#),
[affine_trans_contour_xld](#), [affine_trans_polygon_xld](#), [affine_trans_point_2d](#)

Alternatives

[vector_to_rigid](#)

See also

[dvf_to_hom_mat2d](#)

Module

Basic operators

vector_to_hom_mat2d (: : Rows1, Columns1, Rows2, Columns2 : HomMat2D)

Approximate an affine transformation from point correspondences.

[vector_to_hom_mat2d](#) approximates an affine transformation from at least three point correspondences and returns it in form of the homogeneous transformation matrix [HomMat2D](#) (siehe [hom_mat2d_to_affine_par](#) for the content of the homogeneous transformation matrix).

The point correspondences are passed in the tuples ([Rows1](#),[Columns1](#)) and ([Rows2](#),[Columns2](#)), where corresponding points must be at the same index positions in the tuples. If more than three point correspondences are passed the transformation is overdetermined. In this case, the returned transformation is the transformation that minimizes the distances between the input points ([Rows1](#),[Columns1](#)) and the transformed points ([Rows2](#),[Columns2](#)), as described in the following equation (points in form of homogeneous vectors):

$$\sum_i \left\| \begin{pmatrix} \text{Rows2}[i] \\ \text{Columns2}[i] \\ 1 \end{pmatrix} - \text{HomMat2D} \cdot \begin{pmatrix} \text{Rows1}[i] \\ \text{Columns1}[i] \\ 1 \end{pmatrix} \right\|^2 = \text{minimal}$$

[HomMat2D](#) can be used directly with operators that transform data using affine transformations, e.g., [affine_trans_image](#).

Parameter

- ▷ **Rows1** (input_control) point.y-array $\leadsto$ *real*
Row coordinates of the starting points.
 - ▷ **Columns1** (input_control) point.x-array $\leadsto$ *real*
Column coordinates of the starting points.
 - ▷ **Rows2** (input_control) point.y-array $\leadsto$ *real*
Row coordinates of the end points.
 - ▷ **Columns2** (input_control) point.x-array $\leadsto$ *real*
Column coordinates of the end points.
 - ▷ **HomMat2D** (output_control) affine2d-array $\leadsto$ *real*
Output transformation matrix.
- Number of elements : 6**

Parallelization Information

[vector_to_hom_mat2d](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[affine_trans_image](#)

Alternatives

[dvf_to_hom_mat2d](#)

See also

[affine_trans_image](#), [optical_flow_match](#)

Module

Basic operators

vector_to_rigid (: : Rows1, Columns1, Rows2, Columns2 : HomMat2D)

Approximate a rigid affine transformation from point correspondences.

[vector_to_rigid](#) approximates a rigid affine transformation, i.e., a transformation consisting of a rotation and a translation, from at least two point correspondences and returns it in form of the homogeneous transformation matrix [HomMat2D](#). The matrix consists of 2 components: a rotation matrix **R** and a translation vector **t** (also see [hom_mat2d_rotate](#) and [hom_mat2d_translate](#)):

$$\text{HomMat2D} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \mathbf{t} \\ 0 & 1 & \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R} & 0 \\ 0 & 1 \end{bmatrix} = \mathbf{H}(\mathbf{t}) \cdot \mathbf{H}(\mathbf{R})$$

The point correspondences are passed in the tuples ([Rows1](#), [Columns1](#)) and ([Rows2](#), [Columns2](#)), where corresponding points must be at the same index positions in the tuples. The transformation is always overdetermined. Therefore, the returned transformation is the transformation that minimizes the distances between the original points ([Rows1](#), [Columns1](#)) and the transformed points ([Rows2](#), [Columns2](#)), as described in the following equation (points in form of homogeneous vectors):

$$\sum_i \left\| \begin{pmatrix} \text{Rows2}[i] \\ \text{Columns2}[i] \\ 1 \end{pmatrix} - \text{HomMat2D} \cdot \begin{pmatrix} \text{Rows1}[i] \\ \text{Columns1}[i] \\ 1 \end{pmatrix} \right\|^2 = \text{minimum}$$

[HomMat2D](#) can be used directly with operators that transform data using affine transformations, e.g., [affine_trans_image](#).

Parameter

- ▷ **Rows1** (input_control) point.y-array $\leadsto$ *real*
Row coordinates of the original points.
- ▷ **Columns1** (input_control) point.x-array $\leadsto$ *real*
Column coordinates of the original points.
- ▷ **Rows2** (input_control) point.y-array $\leadsto$ *real*
Row coordinates of the transformed points.
- ▷ **Columns2** (input_control) point.x-array $\leadsto$ *real*
Column coordinates of the transformed points.
- ▷ **HomMat2D** (output_control) affine2d-array $\leadsto$ *real*
Output transformation matrix.
Number of elements : 6

Parallelization Information

[vector_to_rigid](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[affine_trans_image](#), [affine_trans_region](#), [affine_trans_contour_xld](#),
[affine_trans_polygon_xld](#), [affine_trans_point_2d](#)

Alternatives

[vector_to_hom_mat2d](#), [vector_to_similarity](#)

See also

[dvf_to_hom_mat2d](#)

Module

Basic operators

vector_to_similarity (: : Rows1, Columns1, Rows2,
Columns2 : HomMat2D)

Approximate an similarity transformation from point correspondences.

[vector_to_similarity](#) approximates a similarity transformation, i.e., a transformation consisting of a scaling, a rotation, and a translation, from at least two point correspondences and returns it in form of the homogeneous transformation matrix [HomMat2D](#). The matrix consists of 3 components: a scaling matrix **S**, a rotation matrix **R**, and a translation vector **t** (also see [hom_mat2d_scale](#), [hom_mat2d_rotate](#), and [hom_mat2d_translate](#)):

$$\text{HomMat2D} = \begin{bmatrix} \mathbf{R} \cdot \mathbf{S} & \mathbf{t} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & \mathbf{t} \\ 0 & 1 & \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R} & 0 \\ 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{S} & 0 \\ 0 & 1 \end{bmatrix} = \mathbf{H}(\mathbf{t}) \cdot \mathbf{H}(\mathbf{R}) \cdot \mathbf{H}(\mathbf{S})$$

The point correspondences are passed in the tuples ([Rows1](#), [Columns1](#)) and ([Rows2](#), [Columns2](#)), where corresponding points must be at the same index positions in the tuples. If more than two point correspondences are passed the transformation is overdetermined. In this case, the returned transformation is the transformation that minimizes the distances between the original points ([Rows1](#), [Columns1](#)) and the transformed points ([Rows2](#), [Columns2](#)), as described in the following equation (points as homogeneous vectors):

$$\sum_i \left\| \begin{pmatrix} \text{Rows2}[i] \\ \text{Columns2}[i] \\ 1 \end{pmatrix} - \text{HomMat2D} \cdot \begin{pmatrix} \text{Rows1}[i] \\ \text{Columns1}[i] \\ 1 \end{pmatrix} \right\|^2 = \text{minimum}$$

[HomMat2D](#) can be used directly with operators that transform data using affine transformations, e.g., [affine_trans_image](#).

Parameter

- ▷ **Rows1** (input_control) point.y-array $\leadsto$ *real*
Row coordinates of the original points.
 - ▷ **Columns1** (input_control) point.x-array $\leadsto$ *real*
Column coordinates of the original points.
 - ▷ **Rows2** (input_control) point.y-array $\leadsto$ *real*
Row coordinates of the transformed points.
 - ▷ **Columns2** (input_control) point.x-array $\leadsto$ *real*
Column coordinates of the transformed points.
 - ▷ **HomMat2D** (output_control) affine2d-array $\leadsto$ *real*
Output transformation matrix.
- Number of elements : 6**

Parallelization Information

[vector_to_similarity](#) is *reentrant* and processed *without parallelization*.

Possible Successors

[affine_trans_image](#), [affine_trans_region](#), [affine_trans_contour_xld](#),
[affine_trans_polygon_xld](#), [affine_trans_point_2d](#)

Alternatives

[vector_to_hom_mat2d](#), [vector_to_rigid](#)

See also

[dvf_to_hom_mat2d](#)

Module

Basic operators

15.2 Background-Estimator

close_all_bg_esti (: : :)

Delete all background estimation data sets.

`close_all_bg_esti` deletes the background estimation data sets and releases all used memory.

Attention

Since all estimators are closed by `close_all_bg_esti`, all handles become invalid.

Result

If it is possible to close the background estimation data sets the operator `close_all_bg_esti` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

`close_all_bg_esti` is *local* and processed *completely exclusively* without parallelization.

Alternatives

`close_bg_esti`

See also

`create_bg_esti`

Module

Background estimation

close_bg_esti (: : BgEstiHandle :)

Delete the background estimation data set.

`close_bg_esti` deletes the background estimation data set and releases all used memory.

Parameter

- ▷ **BgEstiHandle** (input_control) bg_estimation $\leadsto$ integer
ID of the BgEsti data set.

Example

```
/* read Init-Image: */
read_image(InitImage, 'Init_Image')
/* initialize BgEsti-Dataset with
   fixed gains and threshold adaption: */
create_bg_esti(InitImage, 0.7, 0.7, 'fixed', 0.002, 0.02,
               'on', 7, 10, 3.25, 15.0, BgEstiHandle)
/* read the next image in sequence: */
read_image(Image1, 'Image_1')
/* estimate the Background: */
run_bg_esti(Image1, Region1, BgEstiHandle)
/* display the foreground region: */
disp_region(Region1, WindowHandle)
/* read the next image in sequence: */
read_image(Image2, 'Image_2')
/* estimate the Background: */
run_bg_esti(Image2, Region2, BgEstiHandle)
/* display the foreground region: */
disp_region(Region2, WindowHandle)
/* etc. */
/* - end of background estimation - */
/* close the dataset: */
close_bg_est(BgEstiHandle).
```

Result

`close_bg_esti` returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

`close_bg_esti` is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

`run_bg_esti`

See also

`create_bg_esti`

Module

Background estimation

create_bg_esti (InitializeImage : : Syspar1, Syspar2, GainMode, Gain1, Gain2, AdaptMode, MinDiff, StatNum, ConfidenceC, TimeC : BgEstiHandle)

Generate and initialize a data set for the background estimation.

`create_bg_esti` creates a new data set for the background estimation and initializes it with the appropriate parameters. The estimated background image is part of this data set. The newly created set automatically becomes the current set.

`InitializeImage` is used as an initial prediction for the background image. For a good prediction an image of the observed scene without moving objects should be passed in `InitializeImage`. That way the foreground adaptation rate can be held low. If there is no empty scene image available, a homogenous gray image can be used instead. In that case the adaptation rate for the foreground image must be raised, because initially most of the image will be detected as foreground. The initialization image must be of type *byte* or *real*. Because of processing single-channel images, data sets must be created for every channel. Size and region of `InitializeImage` determines size and region for all background estimations (`run_bg_esti`) that are performed with this data set.

`Syspar1` and `Syspar2` are the parameters of the Kalman system matrix. The system matrix describes the system of the gray value changes according to Kalman filter theory. The background estimator implements a different system for each pixel.

`GainMode` defines whether a fixed Kalman gain should be used for the estimation or whether the gain should adapt itself depending on the difference between estimation and actual value. If `GainMode` is set to *'fixed'*, then `Gain1` is used as Kalman gain for pixels predicted as foreground and `Gain2` as gain for pixels predicted as background. `Gain1` should be smaller than `Gain2`, because adaptation of the foreground should be slower than adaptation of the background. Both `Gain1` and `Gain2` should be smaller than 1.0.

If `GainMode` is set to *'frame'*, then tables for foreground and background estimation are computed containing Kalman gains for all the 256 possible grayvalue changes. `Gain1` and `Gain2` then denote the number of frames necessary to adapt the difference between estimated value and actual value. So with a fixed time for adaptation (i.e. number of frames) the needed Kalman gain grows with the grayvalue difference. `Gain1` should therefore be larger than `Gain2`. Different gains for different grayvalue differences are useful if the background estimator is used for generating an 'empty' scene assuming that there are always moving objects in the observed area. In that case the adaptation time for foreground adaptation (`Gain1`) must not be too big. `Gain1` and `Gain2` should be bigger than 1.0.

`AdaptMode` denotes, whether the foreground/background decision threshold applied to the grayvalue difference between estimation and actual value is fixed or whether it adapts itself depending on the grayvalue deviation of the background pixels.

If `AdaptMode` is set to *'off'*, the parameter `MinDiff` denotes a fixed threshold. The parameters `StatNum`, `ConfidenceC` and `TimeC` are meaningless in this case.

If `AdaptMode` is set to *'on'*, then `MinDiff` is interpreted as a base threshold. For each pixel an offset is added to this threshold depending on the statistical evaluation of the pixel value over time. `StatNum` holds the number of data sets (past frames) that are used for computing the grayvalue variance (FIR-Filter). `ConfidenceC` is used to determine the confidence interval.

The confidence interval determines the values of the background statistics if background pixels are hidden by a foreground object and thus are detected as foreground. According to the student t-distribution the confidence constant is 4.30 (3.25, 2.82, 2.26) for a confidence interval of 99,8% (99,0%, 98,0%, 95,0%). `TimeC` holds a

time constant for the exp-function that raises the threshold in case of a foreground estimation of the pixel. That means, the threshold is raised in regions where movement is detected in the foreground. That way larger changes in illumination are tolerated if the background becomes visible again. The main reason for increasing this tolerance is the impossibility for a prediction of illumination changes while the background is hidden. Therefore no adaptation of the estimated background image is possible.

Attention

If **GainMode** was set to 'frame', the run-time can be extremely long for large values of **Gain1** or **Gain2**, because the values for the gains' table are determined by a simple binary search.

Parameter

- ▷ **InitializeImage** (input_object) image $\leadsto$ Hobject : byte / real
initialization image.
- ▷ **Syspar1** (input_control) real $\leadsto$ real
1. system matrix parameter.
Default Value : 0.7
Suggested values : Syspar1 $\in$ {0.65, 0.7, 0.75}
Typical range of values : $0.05 \leq \text{Syspar1} \leq 1.0$
Recommended Increment : 0.05
- ▷ **Syspar2** (input_control) real $\leadsto$ real
2. system matrix parameter.
Default Value : 0.7
Suggested values : Syspar2 $\in$ {0.65, 0.7, 0.75}
Typical range of values : $0.05 \leq \text{Syspar2} \leq 1.0$
Recommended Increment : 0.05
- ▷ **GainMode** (input_control) string $\leadsto$ string
Gain type.
Default Value : 'fixed'
List of values : GainMode $\in$ {'fixed', 'frame'}
- ▷ **Gain1** (input_control) real $\leadsto$ real
Kalman gain / foreground adaptation time.
Default Value : 0.002
Suggested values : Gain1 $\in$ {10.0, 20.0, 50.0, 0.1, 0.05, 0.01, 0.005, 0.001}
Restriction : $0.0 \leq \text{Gain1}$
- ▷ **Gain2** (input_control) real $\leadsto$ real
Kalman gain / background adaptation time.
Default Value : 0.02
Suggested values : Gain2 $\in$ {2.0, 4.0, 8.0, 0.5, 0.1, 0.05, 0.01}
Restriction : $0.0 \leq \text{Gain2}$
- ▷ **AdaptMode** (input_control) string $\leadsto$ string
Threshold adaptation.
Default Value : 'on'
List of values : AdaptMode $\in$ {'on', 'off'}
- ▷ **MinDiff** (input_control) real $\leadsto$ real
Foreground/background threshold.
Default Value : 7.0
Suggested values : MinDiff $\in$ {3.0, 5.0, 7.0, 9.0, 11.0}
Recommended Increment : 0.2
- ▷ **StatNum** (input_control) integer $\leadsto$ integer
Number of statistic data sets.
Default Value : 10
Suggested values : StatNum $\in$ {5, 10, 20, 30}
Typical range of values : $1 \leq \text{StatNum}$
Recommended Increment : 5

- ▷ **ConfidenceC** (input_control) real $\leadsto$ real
Confidence constant.
Default Value : 3.25
Suggested values : ConfidenceC $\in \{4.30, 3.25, 2.82, 2.62\}$
Recommended Increment : 0.01
Restriction : $0.0 < \text{ConfidenceC}$
- ▷ **TimeC** (input_control) real $\leadsto$ real
Constant for decay time.
Default Value : 15.0
Suggested values : TimeC $\in \{10.0, 15.0, 20.0\}$
Recommended Increment : 5.0
Restriction : $0.0 < \text{TimeC}$
- ▷ **BgEstiHandle** (output_control) bg_estimation $\leadsto$ integer
ID of the BgEsti data set.

Example

```
/* read Init-Image: */
read_image(InitImage, 'Init_Image')
/* initialize 1. BgEsti-Dataset with
   fixed gains and threshold adaption: */
create_bg_esti(InitImage, 0.7, 0.7, 'fixed', 0.002, 0.02,
               'on', 7.0, 10, 3.25, 15.0, BgEstiHandle1)
/* initialize 2. BgEsti-Dataset with
   frame orientated gains and fixed threshold */
create_bg_esti(InitImage, 0.7, 0.7, 'frame', 30.0, 4.0,
               'off', 9.0, 10, 3.25, 15.0, BgEstiHandle2).
```

Result

`create_bg_esti` returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

`create_bg_esti` is *local* and processed *completely exclusively* without parallelization.

Possible Successors

`run_bg_esti`

See also

`set_bg_esti_params`, `close_bg_esti`

Module

Background estimation

get_bg_esti_params (: : BgEstiHandle : Syspar1, Syspar2, GainMode, Gain1, Gain2, AdaptMode, MinDiff, StatNum, ConfidenceC, TimeC)

Return the parameters of the data set.

`get_bg_esti_params` returns the parameters of the data set. The returned parameters are the same as in `create_bg_esti` and `set_bg_esti_params` (see these for an explanation).

Parameter

- ▷ **BgEstiHandle** (input_control) bg_estimation $\leadsto$ integer
ID of the BgEsti data set.
- ▷ **Syspar1** (output_control) real $\leadsto$ real
1. system matrix parameter.
- ▷ **Syspar2** (output_control) real $\leadsto$ real
2. system matrix parameter.
- ▷ **GainMode** (output_control) string $\leadsto$ string
Gain type.

- ▷ **Gain1** (output_control) real $\leadsto$ real
Kalman gain / foreground adaptation time.
- ▷ **Gain2** (output_control) real $\leadsto$ real
Kalman gain / background adaptation time.
- ▷ **AdaptMode** (output_control) string $\leadsto$ string
Threshold adaptation.
- ▷ **MinDiff** (output_control) real $\leadsto$ real
Foreground / background threshold.
- ▷ **StatNum** (output_control) integer $\leadsto$ integer
Number of statistic data sets.
- ▷ **ConfidenceC** (output_control) real $\leadsto$ real
Confidence constant.
- ▷ **TimeC** (output_control) real $\leadsto$ real
Constant for decay time.

Example

```

/* read Init-Image:*/
read_image(InitImage,'Init_Image')
/* initialize BgEsti-Dataset with
   fixed gains and threshold adaption: */
create_bg_esti(InitImage,0.7,0.7,'fixed',0.002,0.02,
               'on',7.0,10,3.25,15.0,BgEstiHandle)
/* read the next image in sequence: */
read_image(Image1,'Image_1')
/* estimate the Background: */
run_bg_esti(Image1,Region1,BgEstiHandle)
/* display the foreground region: */
disp_region(Region1,WindowHandle)
/* read the next image in sequence: */
read_image(Image2,'Image_2')
/* estimate the Background: */
run_bg_esti(Image2,Region2,BgEstiHandle)
/* display the foreground region: */
disp_region(Region2,WindowHandle)
/* etc. */
/* change only the gain parameter in dataset: */
get_bg_esti_params(BgEstiHandle,par1,par2,par3,par4,
                  par5,par6,par7,par8,par9,par10)
set_bg_esti_params(BgEstiHandle,par1,par2,par3,0.004,
                  0.08,par6,par7,par8,par9,par10)
/* read the next image in sequence: */
read_image(Image3,'Image_3')
/* estimate the Background: */
run_bg_esti(Image3,Region3,BgEstiHandle)
/* display the foreground region: */
disp_region(Region3,WindowHandle)
/* etc. */

```

Result

[get_bg_esti_params](#) returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

[get_bg_esti_params](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[create_bg_esti](#)

Possible Successors

[run_bg_esti](#)

	See also
set_bg_esti_params	
	Module
Background estimation	

give_bg_esti (: BackgroundImage : BgEstiHandle :)

Return the estimated background image.

[give_bg_esti](#) returns the estimated background image of the current BgEsti data set. The background image has the same type and size as the initialization image passed in [create_bg_esti](#).

	Parameter
▷ BackgroundImage (output_object) image $\leadsto$ Hobject : byte / real	
Estimated background image of the current data set.	
▷ BgEstiHandle (input_control) bg_estimation $\leadsto$ integer	
ID of the BgEsti data set.	

	Example

```
/* read Init-Image: */
read_image(InitImage, 'Init_Image')
/* initialize BgEsti-Dataset with
   fixed gains and threshold adaption: */
create_bg_esti(InitImage, 0.7, 0.7, 'fixed', 0.002, 0.02,
               'on', 7, 10, 3.25, 15.0, BgEstiHandle)
/* read the next image in sequence: */
read_image(Image1, 'Image_1')
/* estimate the Background: */
run_bg_esti(Image1, Region1, BgEstiHandle)
/* give the background image from the aktive dataset: */
give_bg_esti(BgImage, BgEstiHandle)
/* display the background image: */
disp_image(BgImage, WindowHandle).
```

	Result
give_bg_esti returns 2 (H_MSG_TRUE) if all parameters are correct.	

	Parallelization Information
give_bg_esti is <i>local</i> and processed <i>completely exclusively</i> without parallelization.	

	Possible Predecessors
run_bg_esti	

	Possible Successors
run_bg_esti , create_bg_esti , update_bg_esti	

	See also
run_bg_esti , update_bg_esti , create_bg_esti	

	Module
Background estimation	

run_bg_esti (PresentImage : ForegroundRegion : BgEstiHandle :)

Estimate the background and return the foreground region.

[run_bg_esti](#) adapts the background image stored in the BgEsti data set using a Kalman filter on each pixel and returns a region of the foreground (detected moving objects).

For every pixel an estimation of its grayvalue is computed using the values of the current data set and its stored background image and the current image ([PresentImage](#)). By comparison to the threshold (fixed or adaptive, see [create_bg_esti](#)) the pixels are classified as either foreground or background.

The background estimation processes only single-channel images. Therefore the background has to be adapted separately for every channel.

The background estimation should be used on half- or even quarter-sized images. For this, the input images (and the initialization image!) has to be reduced using [zoom_image_factor](#). The advantage is a shorter run-time on one hand and a low-band filtering on the other. The filtering eliminates high frequency noise and results in a more reliable estimation. As a result the threshold (see [create_bg_esti](#)) can be lowered. The foreground region returned by [run_bg_esti](#) then has to be enlarged again for further processing.

Attention

The passed image ([PresentImage](#)) must have the same type and size as the background image of the current data set (initialized with [create_bg_esti](#)).

Parameter

- ▷ **PresentImage** (input_object) image $\leadsto$ *Hobject*: byte / real
Current image.
- ▷ **ForegroundRegion** (output_object) region $\leadsto$ *Hobject*
Region of the detected foreground.
- ▷ **BgEstiHandle** (input_control) bg_estimation $\leadsto$ *integer*
ID of the BgEsti data set.

Example

```
/* read Init-Image: */
read_image(InitImage, 'Init_Image')
/* initialize BgEsti-Dataset with
   fixed gains and threshold adaption */
create_bg_esti(InitImage, 0.7, 0.7, 'fixed', 0.002, 0.02,
               'on', 7, 10, 3.25, 15.0, BgEstiHandle)
/* read the next image in sequence: */
read_image(Image1, 'Image_1')
/* estimate the Background: */
run_bg_esti(Image1, Region1, BgEstiHandle)
/* display the foreground region: */
disp_region(Region1, WindowHandle)
/* read the next image in sequence: */
read_image(Image2, 'Image_2')
/* estimate the Background: */
run_bg_esti(Image2, Region2, BgEstiHandle)
/* display the foreground region: */
disp_region(Region2, WindowHandle)
/* etc. */
```

Result

[run_bg_esti](#) returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

[run_bg_esti](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[create_bg_esti](#), [update_bg_esti](#)

Possible Successors

[run_bg_esti](#), [give_bg_esti](#), [update_bg_esti](#)

See also

[set_bg_esti_params](#), [create_bg_esti](#), [update_bg_esti](#), [give_bg_esti](#)

Module

Background estimation

```
set_bg_esti_params ( : : BgEstiHandle, Syspar1, Syspar2, GainMode,
Gain1, Gain2, AdaptMode, MinDiff, StatNum, ConfidenceC, TimeC : )
```

Change the parameters of the data set.

`set_bg_esti_params` is used to change the parameters of the data set. The parameters passed by `set_bg_esti_params` are the same as in `create_bg_esti` (see there for an explanation).

The image format cannot be changed! To do this, a new data set with an initialization image of the appropriate format has to be created.

To exchange the background image completely, use `update_bg_esti`. The current image then has to be passed for both the input image and the update region.

Attention

If `GainMode` was set to `'frame'`, the run-time can be extremely long for large values of `Gain1` or `Gain2`, because the values for the gains' table are determined by a simple binary search.

Parameter

- ▷ **BgEstiHandle** (input_control) bg_estimation $\leadsto$ integer
ID of the BgEsti data set.
- ▷ **Syspar1** (input_control) real $\leadsto$ real
1. system matrix parameter.
Default Value : 0.7
Suggested values : Syspar1 $\in$ {0.65, 0.7, 0.75}
Typical range of values : $0.05 \leq \text{Syspar1} \leq 1.0$
Recommended Increment : 0.05
- ▷ **Syspar2** (input_control) real $\leadsto$ real
2. system matrix parameter.
Default Value : 0.7
Suggested values : Syspar2 $\in$ {0.65, 0.7, 0.75}
Typical range of values : $0.05 \leq \text{Syspar2} \leq 1.0$
Recommended Increment : 0.05
- ▷ **GainMode** (input_control) string $\leadsto$ string
Gain type.
Default Value : 'fixed'
List of values : GainMode $\in$ {'fixed', 'frame'}
- ▷ **Gain1** (input_control) real $\leadsto$ real
Kalman gain / foreground adaptation time.
Default Value : 0.002
Suggested values : Gain1 $\in$ {10.0, 20.0, 50.0, 0.1, 0.05, 0.01, 0.005, 0.001}
Restriction : $0.0 \leq \text{Gain1}$
- ▷ **Gain2** (input_control) real $\leadsto$ real
Kalman gain / background adaptation time.
Default Value : 0.02
Suggested values : Gain2 $\in$ {2.0, 4.0, 8.0, 0.5, 0.1, 0.05, 0.01}
Restriction : $0.0 \leq \text{Gain2}$
- ▷ **AdaptMode** (input_control) string $\leadsto$ string
Threshold adaptation.
Default Value : 'on'
List of values : AdaptMode $\in$ {'on', 'off'}
- ▷ **MinDiff** (input_control) real $\leadsto$ real
Foreground/background threshold.
Default Value : 7.0
Suggested values : MinDiff $\in$ {3.0, 5.0, 7.0, 9.0, 11.0}
Recommended Increment : 0.2

- ▷ **StatNum** (input_control) integer $\leadsto$ integer
 Number of statistic data sets.
Default Value : 10
Suggested values : StatNum $\in \{5, 10, 20, 30\}$
Typical range of values : $1 \leq \text{StatNum}$
Recommended Increment : 5
- ▷ **ConfidenceC** (input_control) real $\leadsto$ real
 Confidence constant.
Default Value : 3.25
Suggested values : ConfidenceC $\in \{4.30, 3.25, 2.82, 2.62\}$
Recommended Increment : 0.01
Restriction : $0.0 < \text{ConfidenceC}$
- ▷ **TimeC** (input_control) real $\leadsto$ real
 Constant for decay time.
Default Value : 15.0
Suggested values : TimeC $\in \{10.0, 15.0, 20.0\}$
Recommended Increment : 5.0
Restriction : $0.0 < \text{TimeC}$

Example

```
/* read Init-Image:*/
read_image(InitImage, 'Init_Image')
/* initialize BgEsti-Dataset with
   fixed gains and threshold adaption: */
create_bg_esti(InitImage, 0.7, 0.7, 'fixed', 0.002, 0.02,
               'on', 7.0, 10, 3.25, 15.0, BgEstiHandle)
/* read the next image in sequence: */
read_image(Image1, 'Image_1')
/* estimate the Background: */
run_bg_esti(Image1, Region1, BgEstiHandle)
/* display the foreground region: */
disp_region(Region1, WindowHandle)
/* read the next image in sequence: */
read_image(Image2, 'Image_2')
/* estimate the Background: */
run_bg_esti(Image2, Region2, BgEstiHandle)
/* display the foreground region: */
disp_region(Region2, WindowHandle)
/* etc. */
/* change parameter in dataset: */
set_bg_esti_params(BgEstiHandle, 0.7, 0.7, 'fixed',
                  0.004, 0.08, 'on', 9.0, 10, 3.25, 20.0)
/* read the next image in sequence: */
read_image(Image3, 'Image_3')
/* estimate the Background: */
run_bg_esti(Image3, Region3, BgEstiHandle)
/* display the foreground region: */
disp_region(Region3, WindowHandle)
/* etc. */
```

Result

[set_bg_esti_params](#) returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

[set_bg_esti_params](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[create_bg_esti](#)

Possible Successors

[run_bg_esti](#)

See also

[update_bg_esti](#)

Module

Background estimation

update_bg_esti (PresentImage, UpDateRegion : : BgEstiHandle :)

Change the estimated background image.

[update_bg_esti](#) overwrites the image stored in the current BgEsti data set with the grayvalues of [PresentImage](#) within the bounds of [UpDateRegion](#). This can be used for a "hard" adaptation: Image regions with a sudden change in (known) background can be adapted very fast this way.

Attention

The passed image ([PresentImage](#)) must have the same type and size as the background image of the current data set (initialized with [create_bg_esti](#)).

Parameter

- ▷ **PresentImage** (input_object) image $\leadsto$ *Hobject* : byte / real
Current image.
- ▷ **UpDateRegion** (input_object) region $\leadsto$ *Hobject*
Region describing areas to change.
- ▷ **BgEstiHandle** (input_control) bg_estimation $\leadsto$ *integer*
ID of the BgEsti data set.

Example

```
/* read Init-Image: */
read_image(InitImage, 'Init_Image')
/* initialize BgEsti-Dataset with
   fixed gains and threshold adaption */
create_bg_esti(InitImage, 0.7, 0.7, 'fixed', 0.002, 0.02,
               'on', 7, 10, 3.25, 15.0, BgEstiHandle)
/* read the next image in sequence: */
read_image(Image1, 'Image_1')
/* estimate the Background: */
run_bg_esti(Image1, Region1, BgEstiHandle)
/* use the Region and the information of a knowledge base */
/* to calculate the UpDateRegion */
update_bg_esti(Image1, UpdateRegion, BgEstiHandle)
/* then read the next image in sequence: */
read_image(Image2, 'Image_2')
/* estimate the Background: */
run_bg_esti(Image2, Region2, BgEstiHandle)
/* etc. */
```

Result

[update_bg_esti](#) returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

[update_bg_esti](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[run_bg_esti](#)

Possible Successors

[run_bg_esti](#)

See also

[run_bg_esti](#), [give_bg_esti](#)

Module

Background estimation

15.3 Barcode

decode_1d_bar_code (: : BarCodeElements, BarCodeDescr : Characters, Reference, IsCorrect)

Decoding of a sequence of elements of a bar code.

decode_1d_bar_code decodes a sequence of elements which have been extracted by **find_1d_bar_code** or **get_1d_bar_code** into a sequence of characters. As input the widths of the elements (in pixels) are used. The discrete form as it is returned from **discrete_1d_bar_code** can optionally be used. Otherwise **decode_1d_bar_code** creates the discrete form automatically.

The result of **decode_1d_bar_code** is a sequence of **Characters** and the corresponding reference numbers (**Reference**). In addition a parity check is applied, the result of which is returned in **IsCorrect**. If all characters are used to store user data (i.e. no parity is used) the value of this parameter has to be ignored.

Attention

For bar codes of type Pharmacode the reading direction cannot be determined from the bar code data because the bar code does not specify extra characters (like start, stop, or parity check characters) that enable the determination of a reading direction, and hence the bar code can always be decoded in both directions. Therefore, for PharmaCodes the results of decoding the code in both possible reading directions are returned as two values in **Characters** as well as **Reference**. The decision which Element of **Characters** and **Reference** contains the correct reading direction must be made by the caller based on the orientation of the bar code (as returned in the parameter Orientation in **find_1d_bar_code** and **find_1d_bar_code_region**). The respective first element of **Characters** and **Reference** corresponds to the standard reading direction in the orientation given by Orientation, while the second element corresponds to the opposite reading direction. If, for example, the orientation is 0, the first element corresponds to the reading direction from right to left. If the orientation is $\pi/2$, the first element corresponds to the reading direction from top to bottom.

Parameter

- ▷ **BarCodeElements** (input_control) number-array $\leadsto$ *real*
Widths of the elements of the bar code.
 - ▷ **BarCodeDescr** (input_control) barcode_1d-array $\leadsto$ *string* / integer / real
Description of a bar code class.
 - ▷ **Characters** (output_control) string-array $\leadsto$ *string*
Decoded characters in standard interpretation.
 - ▷ **Reference** (output_control) integer-array $\leadsto$ *integer*
Decoded characters as numbers.
 - ▷ **IsCorrect** (output_control) integer $\leadsto$ *integer*
Information whether the bar code is correct.
- List of values** : IsCorrect $\in \{0, 1\}$

Example

```
HTuple    empty;    // empty list of values
HTuple    BarCodeDescr;
HTuple    BarcodeFound, Elements, Orientation;
HTuple    Characters, Reference, IsCorrect;
HObject    Image, CodeRegion;

gen_1d_bar_code_descr("EAN 13", 13, 13, &BarCodeDescr);
find_1d_bar_code(Image, &CodeRegion, BarCodeDescr, empty, empty,
                 &BarcodeFound, &Elements, &Orientation);
if (BarcodeFound[0].l)
{
    decode_1d_bar_code(Elements, BarCodeDescr,
                      &Characters, &Reference, &IsCorrect);
    if (IsCorrect[0].l)
        for (int i=0; i<Characters.Num(); i++)
        {
            char *value = Characters[i];
```

```
}
}
```

Result

The operator `decode_1d_bar_code` returns the value 2 (H_MSG.TRUE) if the transferred bar code description is correct and the element list can be decoded.

Parallelization Information

`decode_1d_bar_code` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`find_1d_bar_code`, `get_1d_bar_code`

Module

Barcode reader

```
decode_2d_bar_code ( : : BarCodeDescr, BarCodeDimension,
  BarCodeData : SymbolCharacters, CorrSymbolData, DecodedData,
  DecodingError, StructuredAppend )
```

Decode 2D bar code data.

`decode_2d_bar_code` decodes (binary) data of a 2D bar code that has been extracted with the help of `get_2d_bar_code` or `get_2d_bar_code_pos`. The parameter `BarCodeData` contains the data values, `BarCodeDimension` describes the size of the data matrix (see operator `get_2d_bar_code`), and `BarCodeDescr` describes the 2D bar code class (see operator `gen_2d_bar_code_descr`).

First, the binary data are converted into a stream of 8-bit characters (`SymbolCharacters`). This stream consists of the actual, possibly erroneous data and of additional characters used by the subsequent error correcting routine. If all errors could be corrected, the resulting, still undecoded data is returned in the parameter `CorrSymbolData`. The parameter `DecodingError` contains the number of corrected errors, or a negative number if the correction failed.

Then, the data is decoded and returned as a tuple of ASCII characters in the parameter `DecodedData`. If the symbol is part of a group of symbols (ECC 200: "Structured Append"), the parameter `StructuredAppend` contains the number (place) of the symbol in the group, the overall number of symbols in the group, and a group identifier. If the symbol is not part of a group, the first two entries of `StructuredAppend` contain a 1.

Parameter

- ▷ **BarCodeDescr** (input_control) barcode_2d-array $\rightsquigarrow$ string / integer / real
Description of the bar code class.
- ▷ **BarCodeDimension** (input_control) integer-array $\rightsquigarrow$ integer
Tuple with the dimension of the examined symbol. In the case of ECC 200: width, height, symbol code.
- ▷ **BarCodeData** (input_control) integer-array $\rightsquigarrow$ integer
Tuple with the data values of the examined symbol.
- ▷ **SymbolCharacters** (output_control) string-array $\rightsquigarrow$ string
Data and error codewords of the symbol.
- ▷ **CorrSymbolData** (output_control) integer-array $\rightsquigarrow$ integer
Corrected data codewords of the symbol.
- ▷ **DecodedData** (output_control) integer-array $\rightsquigarrow$ integer
Decoded data characters as numbers.
- ▷ **DecodingError** (output_control) integer $\rightsquigarrow$ integer
Number of errors during the decoding process.
- ▷ **StructuredAppend** (output_control) integer-array $\rightsquigarrow$ integer
If the symbol belongs to a group ("structured append"): position in the group, total symbol number, group ("file") id.

Result

The return value signals incorrect parameters as well as a failure in the decoding procedure. The error code 8812, e.g., signals that the decoded data stream contained an invalid data word. Up to now, user-defined control words can not be handled by the operator. If such control words are detected in the decoded data stream, the error code 8813 is returned.

Parallelization Information

[decode_2d_bar_code](#) is reentrant and processed without parallelization.

Possible Predecessors

[get_2d_bar_code](#)

Module

Barcode reader

discrete_1d_bar_code (: : Elements, BarCodeDescr : DisrecteBarCode)

Generate a discrete bar code from the elements widths.

[discrete_1d_bar_code](#) converts the list of element widths (output from [find_1d_bar_code](#) or [get_1d_bar_code](#)) into a discrete bar code. Thus every element is then represented by its number of modules (1,2,..) and no longer as its width in pixels.

This operator is used if the bar code type is not available so that [decode_1d_bar_code](#) cannot be applied, thus the user wants to find the bar code with the help of HALCON operators and then himself decode the bar code. To create the bar code description the operator [gen_1d_bar_code_descr_gen](#) is used and with [find_1d_bar_code](#) the element widths are extracted. Then [discrete_1d_bar_code](#) is used to create the list of the multiple of the modules.

Parameter

- ▷ **Elements** (input_control) number-array $\leadsto$ real
List of elements widths of the bar code.
- ▷ **BarCodeDescr** (input_control) barcode_1d-array $\leadsto$ string / integer / real
Description of a bar code class.
- ▷ **DisrecteBarCode** (output_control) number-array $\leadsto$ integer
Widths of elements as multiple of modules.

Example

```
HTuple    empty;    // empty list of values
HTuple    BarCodeDescr;
HTuple    BarcodeFound,Elements,Orientation;
HTuple    DiscreteBarCode;
Hobject    Image,CodeRegion;
```

```
gen_1d_bar_code_descr_gen(20,40,2,empty,empty,-1.0,"false",&BarCodeDescr);
find_1d_bar_code(Image,&CodeRegion,BarCodeDescr,empty,empty,
                 &BarcodeFound,&Elements,&Orientation);
if (BarcodeFound[0].l)
{
    discrete_1d_bar_code(Elements,BarCodeDescr,&DiscreteBarCode);
    for (int i=0; i<DiscreteBarCode.Num(); i++)
    {
        int NumModules = DiscreteBarCode[i];
    }
}
```

Result

The operator [discrete_1d_bar_code](#) returns the value 2 (H_MSG_TRUE) if the transferred bar code description is correct and the the element list can be discretized.

Parallelization Information

[discrete_1d_bar_code](#) is reentrant and processed without parallelization.

Possible Predecessors

[find_1d_bar_code](#), [get_1d_bar_code](#)

See also

[decode_1d_bar_code](#)

```
find_1d_bar_code ( Image : CodeRegion : BarCodeDescr, GenericName,
GenericValue : BarcodeFound, BarCodeElements, Orientation )
```

Look for one bar code in an image.

find_1d_bar_code looks for a bar code in an image. As input it needs the description of the bar code as generated by **gen_1d_bar_code_descr** or **gen_1d_bar_code_descr_gen**. With this description the kind of bar code to be extracted is specified.

The result of the operator are the widths of the elements and the bar code region with its orientation. If a bar code was found **BarcodeFound** returns the value 1. Otherwise it returns 0.

To control the internal image processing the parameters **GenericName** and **GenericValue** are used. This is done by passing the names of the control parameters to be changed in **GenericName** as a list of strings. In **GenericValue** the values are passed at the corresponding index positions.

Normally none of the values have to be changed because the operator can adapt automatically to changing situations. Only in the case of difficult capturing conditions or with special bar codes a change should be applied. Here is the list of all parameters which can be changed:

'amplitude_sobel' Minimum amplitude for edge extraction: The first step of the search for the bar code is based on the sobel filter. In contrast to thresholding this has the advantage of being independent of different illumination conditions. In the case of a low contrast (difference between bright and dark elements) this value can be chosen lower to find all parts of the bar code. If the contrast is very good the value can be set to a higher value. This results in a better runtime because less pixels have to be evaluated. The effect of this parameter can be checked with the operator **sobel_dir** with edge type 'sum_abs'.

Default value: 70

Used for: Search for bar code region

'min_size_element' During the analysis of elements (or parts of them) this parameter is used to eliminate very small objects which do not belong to the bar code. You have to be aware that with a low image quality elements can be divided into smaller pieces. The parameter has to be larger than the smallest piece. With good image quality the value can be larger to reduce runtime. In any case the value may not be larger than the length of one element (in pixels). The area calculated includes only the border of the elements and is thus smaller. If the value is too large parts of the bar code may be missing.

Default value: 30

Used for: Search for bar code region

'max_size_element' Similar to the minimal size this parameter is used to suppress large objects. You have to know that with a low resolution multiple elements can be extracted as one intermediate region. The value has thus to be larger than the size of this region. This can be the size of many elements. The area internally calculated includes only the border of the elements. Furthermore the value should be smaller than the total size of the bar code.

Default value: 15000

Used for: Search for bar code region

'angle_range' This parameter has the same function as the parameter 'sum_angles' but here it is used during the initial search for elements of the bar code. It has influence on the estimation of the orientation of the elements. For low quality images this value has to be large. A small value results in a reduced runtime. The value is given in degrees. There is no need to change the parameter.

Default value: 24

Used for: Search for bar code region

'correct_angle' This parameter is used to decide if a region can be part of a bar code. For this the orientation of all boundary points are calculated. If enough points have the same orientation the region is accepted for further processing. Increasing this value reduces the amount of accepted regions and improves speed. For low image quality this value has to be low.

Default value: 0.5 (corresponds 50 percent)

Used for: Search for bar code region

'dilation_factor' After the initial step of finding elements these are combined into a bar code region ([CodeRegion](#)). This is implemented using the closing operator. The mask size is automatically estimated. If this estimation is wrong it can be corrected using this parameter. If the bar code elements have large gaps the value has to be larger than one. If the bar code is very close to other objects and is wrongly connected to this object the value can be smaller than 1. The parameter is independent of the image quality and runtime.

Default value: 1

Used for: Search for bar code region

'sum_angles' This parameter is used for the final estimation of the orientation of the bar code region (value of [Orientation](#)). For this at first the orientation of the boundaries of all elements are calculated. The most frequent orientation is calculated. Using the neighboring orientations the mean orientation is calculated. The parameter determines the amount of neighboring angles used for summing up. The value is given in degrees.

Default value: 40

Used for: Search for bar code region

'min_area_bar_code' This value is used for the final decision if the region is a bar code. The parameter specifies the minimum area of the bar code. This area used here consists only of the boundary pixels of the elements and is thus smaller than the area of [CodeRegion](#). If areas were found which are too small the value has to be increased. The parameter can also be used if small areas with the same orientation as the bar code elements were found. The parameter is independent of the image quality and runtime.

Default value: 1000

Used for: Search for bar code region

'sigma_project' For the extraction of the element widths the gray values of the bar code region are projected in the direction of the elements. The data calculated this way is smoothed to reduce noise. For bar codes with large gaps between the elements this value can be increased if extra elements were detected. The value should be larger than 0.5.

Default value: 0.7

Used for: Extraction of element widths

'amplitude_project' For the calculation of the element widths from the gray projections this parameter controls the minimum amplitude (gray value difference) between dark and light elements. With a lot of noise (e.g. scratches) this value can be increased. For low contrast images the value has to be low.

Default value: 3

Used for: Extraction of element widths

'width_project' The gray value projections are calculated only in a part of the bar code region along several scanlines (see below). This parameter specifies the half width of the area that is used to calculate the gray value projections. If the orientation of the region is not estimated correctly this value must be small. If there are distortions (e.g., scratches) this value can be increased. For very narrow bar codes the value can be set to

a small value.

Default value: 5

Used for: Extraction of element widths

'add_length_project' Because the length of the bar code is sometimes too short to get elements at the beginning and at the end the region is extended based on this parameter. If the bar code lies near to another object resulting in a connection of both areas this value can be decreased.

Default value: 5

Used for: Extraction of element widths

'interpolation_project' With this parameter the type of interpolation used for the gray projection is determined. With a value of 1 a bilinear interpolation is used. With 0 no interpolation is used. In this case the runtime is shorter but it might result in a wrong extraction of the elements.

Default value: 1

Used for: Extraction of element widths

'num_scanlines' This value determines the maximum number of scanlines in which the bar code is searched. This value should be chosen relatively large if it is expected that bar codes with severe disturbances must be read. If *num_scanlines* is set to a value greater than 1, the default value of *width_project* is 5, otherwise 25. This choice of default values for *width_project* enables a robust extraction of the bar code even if only a single scanline is used.

Default value: 9

Used for: Extraction of element widths

'max_extra_elements' With this parameter, elements extracted in the vicinity of the bar code that are parallel to the bars of the bar code can be eliminated. Hence, this parameter enables the correct decoding of the bar code if there are extraneous bars close to the bar code. The value of *max_extra_elements* is used for both ends of the bar code separately, so that a total of $2 * \text{max\_extra\_elements}$ extraneous elements (bars and spaces) can be eliminated. Hence, on each side of the bar code up to $\text{max\_extra\_elements}/2$ extraneous bars can be eliminated.

Default value: 6

Used for: Extraction of element widths

Please note that these parameters should normally not be changed.

	Parameter	
▷	Image (input_object)	image $\leadsto$ <i>Hobject</i> : byte / uint2 Image with bar code inside.
▷	CodeRegion (output_object)	region $\leadsto$ <i>Hobject</i> Region of bar code.
▷	BarCodeDescr (input_control)	barcode_1d-array $\leadsto$ string / integer / real Description of a bar code class.
▷	GenericName (input_control)	string(-array) $\leadsto$ string Names of optional control parameters. Default Value : '' List of values : <i>GenericName</i> $\in$ { 'amplitude_sobel', 'min_size_element', 'max_size_element', 'angle_range', 'correct_angle', 'dilation_factor', 'sum_angles', 'sigma_project', 'amplitude_project', 'width_project', 'add_length_project', 'interpolation_project', 'num_scanlines', 'max_extra_elements' }
▷	GenericValue (input_control)	number(-array) $\leadsto$ real / integer Values of optional control parameters. Default Value : ''

- ▷ **BarcodeFound** (output_control) integer $\leadsto$ integer
Information whether the bar code was found.
List of values : BarcodeFound $\in \{0, 1\}$
- ▷ **BarCodeElements** (output_control) number-array $\leadsto$ real
Widths of elements.
- ▷ **Orientation** (output_control) angle.rad $\leadsto$ real
Orientation of bar code.

Example

```
HTuple    empty;    // empty list of values
HTuple    BarCodeDescr;
HTuple    BarcodeFound, Elements, Orientation;
HTuple    Characters, Reference, IsCorrect;
HObject    Image, CodeRegion;

gen_1d_bar_code_descr("EAN 13", 13, 13, &BarCodeDescr);
find_1d_bar_code(Image, &CodeRegion, BarCodeDescr, empty, empty,
                 &BarcodeFound, &Elements, &Orientation);
if (BarcodeFound[0].l)
{
    decode_1d_bar_code(Elements, BarCodeDescr,
                      &Characters, &Reference, &IsCorrect);
    if (IsCorrect[0].l)
        for (int i=0; i<Characters.Num(); i++)
        {
            char *value = Characters[i];
        }
}
```

Result

The operator [find_1d_bar_code](#) returns the value 2 (H_MSG_TRUE) if the parameters are correct.

Parallelization Information

[find_1d_bar_code](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_1d_bar_code_descr](#), [gen_1d_bar_code_descr_gen](#)

Possible Successors

[decode_1d_bar_code](#), [discrete_1d_bar_code](#)

Alternatives

[find_1d_bar_code_region](#)

See also

[sobel_dir](#), [gray_projections](#), [gen_measure_rectangle2](#)

Module

Barcode reader

find_1d_bar_code_region (Image : CodeRegion : BarCodeDescr,
GenericName, GenericValue : Orientation)

Look for multiple bar code regions in an image.

[find_1d_bar_code_region](#) looks for multiple barcodes in the image. In contrast to [find_1d_bar_code](#) this operator is used if an image contains more than one bar code. Here only the regions but not the widths of the elements are extracted. For every region the orientation in radians is calculated.

The control of the image processing is identical to [find_1d_bar_code](#). The description of the parameters [GenericName](#) and [GenericValue](#) can be found at this operator.

Parameter

- ▷ **Image** (input_object) image $\leadsto$ *Hobject*: byte / uint2
Image with bar codes inside.
- ▷ **CodeRegion** (output_object) region(-array) $\leadsto$ *Hobject*
Regions of bar codes.
- ▷ **BarCodeDescr** (input_control) barcode_1d-array $\leadsto$ *string* / integer / real
Description of a bar code class.
- ▷ **GenericName** (input_control) string(-array) $\leadsto$ *string*
Names of optional parameters.
Default Value : '[]'
List of values : GenericName $\in$ { 'amplitude_sobel', 'min_size_element', 'max_size_element',
'angle_range', 'correct_angle', 'dilation_factor', 'sum_angles' }
- ▷ **GenericValue** (input_control) number(-array) $\leadsto$ *real* / integer / string
Values of optional parameters.
Default Value : '[]'
- ▷ **Orientation** (output_control) real(-array) $\leadsto$ *real*
Orientation of bar code.

Example

```

HTuple   empty;    // empty list of values
HTuple   BarCodeDescr;
HTuple   Orientations, Elements;
HTuple   Characters, Reference, IsCorrect;
Hobject   Image, CodeRegions, CodeRegion, GrayRegion;
long      num;

gen_1d_bar_code_descr("code 39", 4, 15, &BarCodeDescr);
find_1d_bar_code_region(Image, &CodeRegion, BarCodeDescr, empty, empty,
                        &Orientations);
count_obj(CodeRegions, &num);
for (long i=0; i<num; i++)
{
    select_obj(CodeRegions, &CodeRegion, i);
    reduce_domain(Image, CodeRegion, GrayRegion)\:
    get_1d_bar_code(GrayRegion, BarCodeDescr, empty, empty, Orientations[i],
                    &Elements);
    decode_1d_bar_code(Elements, BarCodeDescr,
                       &Characters, &Reference, &IsCorrect);
}

```

Result

The operator [find_1d_bar_code_region](#) returns the value 2 (H_MSG_TRUE) if the parameters are correct and at least one bar code is found.

Parallelization Information

[find_1d_bar_code_region](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_1d_bar_code_descr](#), [gen_1d_bar_code_descr_gen](#)

Possible Successors

[get_1d_bar_code](#), [count_obj](#), [select_obj](#), [reduce_domain](#)

Alternatives

[find_1d_bar_code](#)

See also

[sobel_dir](#)

Module

Barcode reader

find_2d_bar_code (Image : CodeRegion : BarCodeDescr, GenParamNames, GenParamValues : CodeRegDescr)

Find regions that might contain a 2D bar code.

find_2d_bar_code searches in the image **Image** for regions that might contain a 2D bar code. Candidate regions are returned in the array of regions **CodeRegion**. Whether such a region really contains a (readable) 2D bar code can only be determined with the help of the operator **get_2d_bar_code** (or **get_2d_bar_code_pos**).

Besides regions that might contain a 2D bar code, the operator passes further, internal information about the regions to the operator for extracting the data from a symbol (**get_2d_bar_code** or **get_2d_bar_code_pos**). This information is stored in a region descriptor and returned in the parameter **CodeRegDescr**.

Depending on the method for printing the bar code ('mode'), different image processing steps are performed. Together with other information about the 2D bar code, the actual printing method is part of a descriptor created with the help of the operator **gen_2d_bar_code_descr** and passed in the parameter **BarCodeDescr**.

In the case of difficult conditions, additional control parameters can be passed to the operator with the help of the (optional) generic parameters **GenParamNames** and **GenParamValues**, in the form of descriptor-value pairs. One group of parameters describes the appearance of the symbols in the actual images, e.g., the expected size of data elements in pixels. The operator will then check candidate regions against the specified criteria. Suitable values can be determined from the real images, e.g., by manually selecting a region with a bar code and then applying the corresponding operator. If the specification of such parameters does not lead to successfully extracted regions, the user can influence the underlying image processing operators directly with the help of a second group of parameters. As the image processing steps taken depend on the printing method, different parameters are to be applied.

- Parameter that will be passed on in **CodeRegDescr**:

'module_width' Mean size of a module (data element) in pixels.

From this quantity, a set of other parameters will be derived. A module should have a size of 4 to 16 pixels in order to be recognized successfully.

GenParamValues: > 0
default: 10

- Parameters for the printing method '**printed**':

The following parameters are used to limit the set of candidate regions with the help of region features. To prevent the use of a certain feature, the corresponding parameter must be set to -1.

'anisometry_max' Maximum anisometry (see operator **eccentricity**).

GenParamValues: > 1 (-1: feature not used)
default: 1.45

'compactness_min' Minimum compactness (see operator **compactness**).

GenParamValues: > 1 (-1: feature not used)
default: 1.2

'compactness_max' Maximum compactness.

GenParamValues: ≥ 'compactness_min' (-1: feature not used)
default: 3.0

'circularity_max' Maximum circularity (see operator **circularity**).

GenParamValues: 0...1.0 (-1: feature not used)
default: 0.8

'circularity_min' Minimum circularity.

GenParamValues: ≤ 'circularity_max' (-1: feature not used)
default: 0.45

'deviation_min' Minimum deviation of gray values (see operator **intensity**).

GenParamValues: > 1.0 (-1: feature not used)
default: 20.0

The following parameters are automatically derived from the size of a module, `'module_width'`, and from parameters of the bar code descriptor `BarcodeDescr` (see operator `gen_2d_bar_code_descr`). However, they can be adjusted individually, too.

ATTENTION: If you want to set `'module_width'` together with other parameters, make sure set `'module_width'` BEFORE the others!

'mean_mask_size_1' Mask size for the first smoothing of the image.

GenParamValues: ≥ 3
default: $2.5 * 'module\_width'$

'mean_mask_size_2' Mask size for the second smoothing of the image.

GenParamValues: ≥ 5
default: $10.0 * 'module\_width'$

'area_min' Minimum size of the symbol in pixels squared.

GenParamValues: > 0
default: $0.25 * 'module\_width'^2 * 'columns\_min' * 'rows\_min'$

'area_max' Maximum size of the symbol in pixels squared.

GenParamValues: $\geq 'area\_min'$
default: $4.00 * 'module\_width'^2 * 'columns\_min' * 'rows\_min'$

'closing_mask_rad' Mask size for calling the operator `closing_circle` with a test region..

GenParamValues: > 0
default: `'module_width'`

- Parameters for the printing method `'engraved_darkfield'`:

'compactness_min' Minimum compactness (see operator `compactness`).

GenParamValues: > 1.0 (-1: feature not used)
default: 1.2

'compactness_max' .

GenParamValues: $\geq 'compactness\_min'$ (-1: feature not used)
default: 3.0

'edge_thresh' Maximum compactness.

GenParamValues: $0 \dots 255$
default: 120

'region_rect2_rel' Relation between the areas of the region and the smallest surrounding rectangle.

GenParamValues: < 1.0
default: 0.7

'median_mask_rad' Mask size for the initial median filtering (see operator `median_image`).

GenParamValues: > 0
default: $0.1 * 'module\_width' + 0.5$

- Parameters for the printing method `'engraved_lightfield'`:

'anisometry_max' Maximum anisometry (see operator `eccentricity`).

GenParamValues: > 1 (-1: feature not used)
default: 1.45

'compactness_min' Minimum compactness (see operator `compactness`).

GenParamValues: > 1 (-1: feature not used)
default: 1.2

'compactness_max' Maximum compactness.

GenParamValues: $\geq 'compactness\_min'$ (-1: feature not used)
default: 3.0

- 'circularity_max'** Maximum circularity (see operator [circularity](#)).
- GenParamValues:** 0...1.0 (-1: feature not used)
- default:** 0.8
- 'circularity_min'** Minimum circularity.
- GenParamValues:** $\leq$ 'circularity_max' (-1: feature not used)
- default:** 0.45
- 'deviation_min'** Minimum deviation of gray values (see operator [intensity](#)).
- GenParamValues:** > 1.0 (-1: feature not used)
- default:** 20.0
- 'mean_mask_size'** Mask size for smoothing the image.
- GenParamValues:** ≥ 3
- default:** $10 * 'module\_width'$
- 'area_min'** Minimum size of the symbol in pixels squared.
- GenParamValues:** > 0
- default:** $0.25 * 'module\_width'^2 * 'columns\_min' * 'rows\_min'$
- 'area_max'** Maximum size of the symbol in pixels squared.
- GenParamValues:** $\geq 'area\_min'$
- default:** $4.00 * 'module\_width'^2 * 'columns\_min' * 'rows\_min'$
- 'closing_mask_rad'** Mask size for calling the operator [closing_circle](#) with a test region..
- GenParamValues:** > 0
- default:** 'module_width'
- 'opening_mask_rad'** Mask size for calling the operator [opening_circle](#) with a test region..
- GenParamValues:** > 0
- default:** $1.5 * 'module\_width'$

Parameter

- ▷ **Image** (input_object) image $\leadsto$ *Hobject*
Image of one or more bar codes.
- ▷ **CodeRegion** (output_object) region(-array) $\leadsto$ *Hobject*
Regions that might contain a bar code.
- ▷ **BarCodeDescr** (input_control) barcode_2d-array $\leadsto$ *string / integer / real*
Description of a 2D bar code class to look for.
- ▷ **GenParamNames** (input_control) string(-array) $\leadsto$ *string*
List of names of (optional) generic parameters controlling the image processing.
Default Value : '[]'
List of values : GenParamNames $\in$ {'module_width', 'anisometry_max', 'compactness_min', 'compactness_max', 'circularity_min', 'circularity_max', 'deviation_min', 'mean_mask_size', 'mean_mask_size_1', 'mean_mask_size_2', 'area_min', 'area_max', 'closing_mask_rad', 'opening_mask_rad', 'compactness_min', 'compactness_max', 'edge_thresh', 'region_rect2_rel', 'median_mask_rad'}
- ▷ **GenParamValues** (input_control) number(-array) $\leadsto$ *real / integer / string*
List of values of generic parameters controlling the image processing.
Default Value : '[]'
List of values : GenParamValues $\in$ {1, 1.1, 1.2, 1.3, 1.4, 1.5, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20}
- ▷ **CodeRegDescr** (output_control) string-array $\leadsto$ *string / integer / real*
Additional parameters describing the bar code regions. They can be used for extracting the data (see [decode_2d_bar_code](#)).

Result

The operator [find_2d_bar_code](#) returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Parallelization Information

[find_2d_bar_code](#) is *reentrant* and processed *without* parallelization.

	Possible Predecessors
gen_2d_bar_code_descr	
	Possible Successors
get_2d_bar_code , get_2d_bar_code_pos	
	Module
Barcode reader	

gen_1d_bar_code_descr (: : CodeName, MinCharacters,
MaxCharacters : BarCodeDescr)

Generate a description of a 1D bar code.

[gen_1d_bar_code_descr](#) generates a description of a one dimensional bar code. This description is used for the search ([find_1d_bar_code](#) or [find_1d_bar_code_region](#)) and the decoding ([decode_1d_bar_code](#)) of the bar code. [gen_1d_bar_code_descr](#) is therefore the first operator in a program sequence for bar code processing. [gen_1d_bar_code_descr](#) has only to be called once at the beginning of a program. The descriptor can be used multiple times. In the case of different types of bar codes the operator has to be called once for each type.

You have to be aware that this description contains only basic informations about the bar code. Thus an arbitrary description can be used to *extract* almost every type of bar code. On the other hand a specific description is important for the *decoding* of a bar code type.

	Parameter
▷ CodeName (input_control) string $\leadsto$ <i>string</i> Name of bar code. Default Value : "EAN 13" List of values : CodeName $\in$ {"2/5 Industrial", "2/5 Interleaved", "Code 39", "Codabar", "Code 128", "Code 93", "EAN 13", "EAN 13 Add-On 2", "EAN 13 Add-On 5", "EAN 8", "EAN 8 Add-On 2", "EAN 8 Add-On 5", "UPC-A", "UPC-A Add-On 2", "UPC-A Add-On 5", "UPC-E", "UPC-E Add-On 2", "UPC-E Add-On 5", "PharmaCode"}	
▷ MinCharacters (input_control) integer $\leadsto$ <i>integer</i> Minimum number of characters (if not fixed). Default Value : 6 Suggested values : MinCharacters $\in$ {-1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 16, 18, 20, 25, 30}	
▷ MaxCharacters (input_control) integer $\leadsto$ <i>integer</i> Maximum number of characters (if not fixed). Default Value : 10 Suggested values : MaxCharacters $\in$ {-1, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 14, 16, 18, 20, 25, 30, 35, 40, 50}	
▷ BarCodeDescr (output_control) barcode_1d-array $\leadsto$ <i>string</i> / <i>integer</i> / <i>real</i> Description of a bar code class.	

	Example

```

HTuple   empty;    // empty list of values
HTuple   BarCodeDescr;
HTuple   BarcodeFound, Elements, Orientation;
HTuple   Characters, Reference, IsCorrect;
HObject   Image, CodeRegion;

gen_1d_bar_code_descr("EAN 13", 13, 13, &BarCodeDescr);
find_1d_bar_code(Image, &CodeRegion, BarCodeDescr, empty, empty,
                 &BarcodeFound, &Elements, &Orientation);
if (BarcodeFound[0].l)
{
    decode_1d_bar_code(Elements, BarCodeDescr,
                      &Characters, &Reference, &IsCorrect);
    if (IsCorrect[0].l)
        for (int i=0; i<Characters.Num(); i++)

```



```

    {
        char *value = Characters[i];
    }
}

```

Result

The operator `gen_1d_bar_code_descr` returns the value 2 (H_MSG_TRUE) if the transferred bar code name is correct.

Parallelization Information

`gen_1d_bar_code_descr` is *reentrant* and processed *without* parallelization.

Possible Successors

`find_1d_bar_code`, `find_1d_bar_code_region`

Alternatives

`gen_1d_bar_code_descr_gen`

Module

Barcode reader

```

gen_1d_bar_code_descr_gen ( : : MinCodeLength, MaxCodeLength,
    ElementSizes, StartElement, StopElement, MaxSizeRatio,
    DiscreteCode : BarCodeDescr )

```

Generate a generic description of a 1D bar code.

`gen_1d_bar_code_descr_gen` generates a generic description of a one dimensional bar code. This operator is used if the bar code description cannot be generated by `gen_1d_bar_code_descr` but nonetheless has to be found by using the operators `find_1d_bar_code` or `find_1d_bar_code_region`. Note that in this case the bar code cannot be decoded by `decode_1d_bar_code`. Thus in this case the decoding has to be done by the user.

The values for `DiscreteCode` and `ElementSizes` have to be set correctly in any case. The other values are not so important for the extraction because they can be estimated by the system automatically.

Parameter

- ▷ **MinCodeLength** (input_control) integer $\rightsquigarrow$ integer
Minimum length of the code in modules (including start and stop elements).
Default Value : 30
List of values : MinCodeLength $\in \{-1, 30, 60, 90, 110, 130, 150, 200\}$
- ▷ **MaxCodeLength** (input_control) integer $\rightsquigarrow$ integer
Maximum length of the code in modules (including start and stop elements).
Default Value : 30
List of values : MaxCodeLength $\in \{-1, 30, 60, 90, 110, 130, 150, 200\}$
- ▷ **ElementSizes** (input_control) integer $\rightsquigarrow$ integer
Number of different element sizes.
Default Value : 2
List of values : ElementSizes $\in \{1, 2, 3, 4, 5, 6, 7\}$
- ▷ **StartElement** (input_control) integer(-array) $\rightsquigarrow$ integer
List of elements of the start sequence. The width of an element is given as the number of modules. Gaps are given as negative values.
Default Value : '[1,-1]'
- ▷ **StopElement** (input_control) integer(-array) $\rightsquigarrow$ integer
List of elements of the stop sequence. The width of an element is given as the number of modules. Gaps are given as negative values.
Default Value : '[1,-1]'
- ▷ **MaxSizeRatio** (input_control) number $\rightsquigarrow$ real
Maximum ratio length to height.
Default Value : 2.5
List of values : MaxSizeRatio $\in \{-1, 2, 3, 4, 5, 6\}$

- ▷ **DiscreteCode** (input_control) string $\leadsto$ string
Discrete code (ignore white elements).
Default Value : 'false'
List of values : DiscreteCode $\in$ {'true', 'false'}
- ▷ **BarCodeDescr** (output_control) barcode_1d-array $\leadsto$ string / integer / real
Description of a bar code class.

Example

```
HTuple   empty;    // empty list of values
HTuple   BarCodeDescr;
HTuple   BarcodeFound, Elements, Orientation;
HTuple   DiscreteBarCode;
Hobject   Image, CodeRegion;

gen_1d_bar_code_descr_gen(20, 40, 2, empty, empty, -1.0, "false", &BarCodeDescr);
find_1d_bar_code(Image, &CodeRegion, BarCodeDescr, empty, empty,
                  &BarcodeFound, &Elements, &Orientation);
if (BarcodeFound[0].l)
{
    discrete_1d_bar_code(Elements, BarCodeDescr, &DiscreteBarCode);
    for (int i=0; i<DiscreteBarCode.Num(); i++)
    {
        int NumModules = DiscreteBarCode[i];
    }
}
```

Result

The operator [gen_1d_bar_code_descr_gen](#) returns the value 2 (H_MSG_TRUE) if the transferred values are correct.

Parallelization Information

[gen_1d_bar_code_descr_gen](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[find_1d_bar_code](#), [find_1d_bar_code_region](#)

Alternatives

[gen_1d_bar_code_descr](#)

Module

Barcode reader

gen_2d_bar_code_descr (: : CodeType, GenParamNames,
GenParamValues : BarCodeDescr)

Generate a generic description of a 2D bar code class.

[gen_2d_bar_code_descr](#) creates a generic description of a 2D bar code. This description consists of the type of the bar code and further characteristic features of the code.

The parameter [CodeType](#) specifies the type of the 2D bar code. Currently, the following types are supported:

'Data Matrix ECC 200' Data Matrix type ECC 200.

The bar code can be described more closely using the parameters [GenParamNames](#) and [GenParamValues](#), which form pairs of descriptors and values. The following parameters describe the size and the form of the symbols:

'columns' Exact number of columns of the symbol.

GenParamValues: 10, 12, 14, 16, 18, 20, 22, 24, 26, 32, 36, 40,
44, 48, 52, 64, 72, 80, 88, 96, 104, 120, 132, 144
default: -

'columns_min' Minimum number of columns.

GenParamValues: 10...144 (even)
default: 10

'columns_max' Maximum number of columns.

GenParamValues: 10...144 (even)

'rows' Exact number of rows of the symbol.

GenParamValues: 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 32, 36, 40,
 44, 48, 52, 64, 72, 80, 88, 96, 104, 120, 132, 144
default: -

'rows_min' Minimum number of rows.

GenParamValues: 8...144 (even)
default: 8

'rows_max' Maximum number of rows.

GenParamValues: 8...144 (even)
default: 144

'shape' Form of the symbol.

GenParamValues: 'square', 'rectangle', 'any'
default: 'any'

The following parameters describe the printing method and the appearance of the bar code:

'foreground' Appearance of bits corresponding to a logical 1 (foreground).

2D bar codes can be printed dark on light or vice versa. If the appearance of bits corresponding to a logical 1 can be specified in advance, computing time will be significantly reduced, especially when looking for regions that might contain a bar code (operator [find_2d_bar_code](#)).

GenParamValues: 'any': unknown appearance
 'dark','black': symbols are printed dark on light
 'light','white': symbols are printed light on dark
default: 'any'

'mode' Method used for printing the 2D bar code.

By specifying this parameter, image processing can be optimized.

GenParamValues: 'printed': Standard black-and-white printing. Modules appear as square regions; neighboring modules with the same value form one region.
 'engraved_darkfield': Modules with the value 1 are engraved into the surface, e.g. by a laser. Using dark field lighting these modules appear as dark round dots with a white corona; neighboring modules with the value 1 are therefore separated by gaps.
 'engraved_lightfield': Similar to 'engraved_darkfield', but using light field lighting. Modules with the value 1 therefore appear as light dots with a dark corona; neighboring modules with the value 1 are again separated by gaps.
default: 'printed'

Parameter

- ▷ **CodeType** (input_control) string $\leadsto$ string
 Type of the 2D bar code.
Default Value : "Data Matrix ECC 200"
List of values : CodeType $\in$ {"Data Matrix ECC 200"}

- ▷ **GenParamNames** (input_control) string-array $\leadsto$ *string*
List of names of generic parameters describing the 2D bar code class.
Default Value : '[]'
List of values : GenParamNames $\in$ {'columns', 'columns_min', 'columns_max', 'rows', 'rows_min', 'rows_max', 'foreground', 'shape', 'mode'}
- ▷ **GenParamValues** (input_control) integer-array $\leadsto$ *integer / string / real*
List of values of the generic parameters describing the 2D bar code class.
Default Value : '[]'
List of values : GenParamValues $\in$ {8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 32, 36, 40, 44, 48, 52, 64, 72, 80, 88, 96, 104, 120, 132, 144, 'dark', 'light', 'square', 'rectangle', 'printed', 'engraved_darkfield', 'engraved_lightfield', 'any'}
- ▷ **BarCodeDescr** (output_control) barcode_2d-array $\leadsto$ *string / integer / real*
Description of the 2D bar code class.

Example

```
gen_2d_bar_code_descr ('Data Matrix ECC 200', ['mode', 'foreground'],
                     ['printed', 'dark'], BarCodeDescr)
```

Result

The operator [gen_2d_bar_code_descr](#) returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Parallelization Information

[gen_2d_bar_code_descr](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[find_2d_bar_code](#)

Module

Barcode reader

```
get_1d_bar_code ( BarCodeRegion : : BarCodeDescr, GenericName,
                  GenericValue, Orientation : BarCodeElements )
```

Extract the widths of the elements inside a bar code region.

[get_1d_bar_code](#) extracts the widths of the elements of a bar code inside the specified region.

The control of the processing is identical to [find_1d_bar_code](#). The description of the parameters [GenericName](#) and [GenericValue](#) can be found at this operator.

Parameter

- ▷ **BarCodeRegion** (input_object) image $\leadsto$ *Hobject : byte / uint2*
Region of bar code.
- ▷ **BarCodeDescr** (input_control) barcode_1d-array $\leadsto$ *string / integer / real*
Description of a bar code class.
- ▷ **GenericName** (input_control) string(-array) $\leadsto$ *string*
Names of optional parameters.
Default Value : '[]'
List of values : GenericName $\in$ {'sigma_project', 'amplitude_project', 'width_project', 'add_length_project', 'interpolation_project', 'num_scanlines', 'max_extra_elements'}
- ▷ **GenericValue** (input_control) number(-array) $\leadsto$ *real / integer*
Values of optional parameters.
Default Value : '[]'
- ▷ **Orientation** (input_control) angle.rad $\leadsto$ *real*
Orientation of bar code.
- ▷ **BarCodeElements** (output_control) number-array $\leadsto$ *real*
Widths of elements.

Example

```

HTuple    empty;    // empty list of values
HTuple    BarCodeDescr;
HTuple    Orientations, Elements;
HTuple    Characters, Reference, IsCorrect;
Hobject    Image, CodeRegions, CodeRegion, GrayRegion;
long      num;

gen_1d_bar_code_descr("code 39", 4, 15, &BarCodeDescr);
find_1d_bar_code_region(Image, &CodeRegion, BarCodeDescr, empty, empty,
                        &Orientations);
count_obj(CodeRegions, &num);
for (long i=0; i<num; i++)
{
    select_obj(CodeRegions, &CodeRegion, i);
    reduce_domain(Image, CodeRegion, GrayRegion)\:
    get_1d_bar_code(GrayRegion, BarCodeDescr, empty, empty, Orientations[i],
                    &Elements);
    decode_1d_bar_code(Elements, BarCodeDescr,
                       &Characters, &Reference, &IsCorrect);
}

```

Result

The operator [get_1d_bar_code](#) returns the value 2 (H_MSG_TRUE) if the parameters are correct.

Parallelization Information

[get_1d_bar_code](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[find_1d_bar_code_region](#), [select_obj](#), [reduce_domain](#)

Possible Successors

[decode_1d_bar_code](#)

Alternatives

[find_1d_bar_code](#)

Module

Barcode reader

get_2d_bar_code (BarCodeRegion, Image : : BarCodeDescr, CodeRegDescr, GenParamNames, GenParamValues : BarCodeDimension, BarCodeData)

Extract the values of the data elements (in ECC 200: “modules”) inside a bar code region (“Data Matrix symbol”).

[get_2d_bar_code](#) extracts the 2D bar code from a candidate symbol region given in [BarCodeRegion](#) and the corresponding image given in [Image](#). [BarCodeDescr](#) contains a descriptor of the expected bar code class, which is created with the help of the operator [gen_2d_bar_code_descr](#). [CodeRegDescr](#) contains further information regarding the candidate region. This descriptor is created by the operator [find_2d_bar_code](#).

If [BarCodeRegion](#) contains a 2D bar code, its data, i.e. the bits of the matrix, are written line by line into a tuple and returned in the parameter [BarCodeData](#). Positive values stand for a logical one, negative values for a logical zero. A value of zero signals that the data element could not be classified unambiguously.

The size of the data field is returned in the parameter [BarCodeDimension](#). The first two entries specify the width and the height of the data field, the third contains a so-called symbol index which also describes its size. The size of the data field is not to be confused with the size of the symbol which in turn encompasses additional finder and alignment patterns.

If no 2D bar code could be extracted the operator returns a corresponding error code.

In difficult cases, the user can influence image processing with the help of the (optional) generic parameters [GenParamNames](#) and [GenParamValues](#). The width of a module in pixels (*'module_width'*) does not

need to be specified, it is part of the region descriptor [CodeRegDescr](#). As in the case of the operator [find_2d_bar_code](#), for the different printing methods different image processing steps are applied, leading to different parameters.

As a short overview, image processing falls into the following parts: First, the region [BarCodeRegion](#) is divided into a dark and a light region by applying a thresholding operator. Because there must be a “silent” area around each 2D bar code, this area can be examined to determine whether the bar code is printed black-on-white or vice versa. With this information, the dark and the light region can be assigned to the logical data values. Next, [BarCodeRegion](#) is approximated by a rectangle. Then, the special finder patterns that each symbol must contain are searched for in the border areas of the rectangle. From the finder pattern, the size of the data matrix and the size and the position of the modules (data elements) can be determined. Finally, the value of the modules is determined.

- Parameters controlling the classification of modules as being dark or light:

'module_rad_ratio' Part of the module that is to be used for its classification.

This parameter controls which part of the module area (in percent, measured from the center) is used to determine its value. With a 'module_rad_ratio' of 1.0, the whole area of the module will be examined. With very small 'module_rad_ratio', only one point in the middle will be examined.

GenParamValues: $0 < \text{'module\_rad\_ratio'} \leq 1$
 default: printing method '*printed*', '*engraved_lightfield*': 0.3
 printing method '*engraved_darkfield*': 0.7

'use_grayvals' Method for the classification of modules (Selectable only for the printing methods '*printed*' and '*engraved_lightfield*'; for the printing method '*engraved_darkfield*', the gray-value-based method is always used!).

The parameter 'use_grayvals' controls by which method the modules are classified. If 'use_grayvals' is set to 'no', the module's value is determined by examining whether the part of the module specified by 'module_rad_ratio' is part of the region that has been assigned the value 1. If more than half of the module is part of this region, the module is interpreted as a logical one.

In the case 'use_grayvals' is set to 'yes', the gray values inside 'module_rad_ratio' are examined. For this, the known border parts of the symbol are used to create classifiers with the features “minimum gray value”, “maximum gray value”, and “range of gray values”. A feature will be automatically excluded if it cannot separate the known dark and light regions. It can also be excluded manually with the help of the parameters below. The value of the modules is then determined by applying the trained classifiers. In case of ambiguous results, the majority decision is adopted.

If none of the gray value features can separate the test regions, the operator switches automatically to the region-based method in case the printing method is set to '*printed*'. This can be detected by examining the data values returned in [BarCodeData](#): The region-based method yields multiples of 4, the gray-value-based method values between -3 and +3. If separation fails when the printing method is '*engraved_darkfield*', an error code is returned. Note, that classification performance can be enhanced by choosing a suitable value for 'module_rad_ratio'.

GenParamValues: 'yes', 'no'
 default: 'yes'

'use_grayval_min' Use the feature “gray value minimum” for classifying module values.

GenParamValues: 'yes', 'no'
 default: 'yes'

'use_grayval_max' Use the feature “gray value maximum” for classifying module values.

GenParamValues: 'yes', 'no'
 default: '*printed*', '*engraved_lightfield*': 'yes'
 '*engraved_darkfield*': 'no'

'use_grayval_range' Use the feature “gray value range” for classifying module values.

GenParamValues: 'yes', 'no'
 default: 'yes'

- Parameters for the printing methods '*printed*' and '*engraved_lightfield*':

'enlarge_region_rad' Enlarge the symbol region.

GenParamValues: > 0 (≤ 0 : enlargement prohibited)
 default: 2.5

'thresh_percent' Area of the histogram in which to look for an optimal threshold.

The threshold for separating dark and light regions will be searched for in the middle 'thresh_percent' percent of the gray value histogram. If 'thresh_percent' is set to 0, a threshold exactly in the middle of the histogram is selected. If 'thresh_percent' is set to 100, the whole histogram will be searched. Note, that the threshold influences not only the size of the data region, but also the position of the surrounding rectangle and with it, slightly, the position of the modules.

GenParamValues: 0 ... 100
 default: 50

'thresh_step_width' Step width for the search of the optimal threshold.

The histogram area specified in the parameter 'thresh_percent' will be stepped through using 'thresh_step_width' until an optimal separation of dark and light regions is reached. The optimum is declared to be found when the number of individual regions is at its maximum. The smaller 'thresh_step_width' is, the better a threshold can be determined, however at the cost of rising computing time, of course.

GenParamValues: > 0
 default: 10

'smooth_cont' Parameter for approximating the region by a polygon (rectangle).

The parameter 'smooth_cont' controls to which degree the contour is smoothed before approximating it with a polygon (see operator [segment_contours_xld](#)).

GenParamValues: 0,3,5,7,9,...
 default: 5

'max_line_dist1', 'max_line_dist2' Distance between the contour and its approximation.

The parameters 'max_line_dist1' and 'max_line_dist2' control the maximal distance between the contour and its approximation (see operator [segment_contours_xld](#)).

GenParamValues: ≥ 0
 default: 4

'min_cont_len' Minimum length of the contour elements in pixel.

Contour elements shorter than 'min_cont_len' will be ignored when approximating the region by a rectangle. In the case of small symbols or low resolution, it can be helpful to decrease this parameter.

GenParamValues: > 0
 default: 50

'border_width_min' Minimum width of the border search area.

The border area of the symbol region must contain the so-called finder patterns (ECC 200: two lines with the value 1, two with alternating values). To find these patterns, a search region is constructed along the sides of the rectangle. Starting with 'border_width_min', the width of the search region is increased until the finder patterns have been detected, or until 'border_width_max' has been reached.

GenParamValues: > 0
 default: 2

'border_width_max' Maximum width of the border search area.

GenParamValues: $\geq$ 'border_width_min'
 default: 5

'border_width' Exact width of the border search area.

Alternatively, the exact width of this search area can be specified in the parameter 'border_width'. This automatically sets 'border_width_min' and 'border_width_max'.

GenParamValues: > 0
 default: -

Parameters for the printing method 'engraved_darkfield':

'median_mask_rad' Mask size for the initial median filtering (see operator [median_image](#)).

GenParamValues: 3,5,7,...,501
default: 'module_width'*0.1 + 0.5

'gray_erosion_size' Mask size for the subsequent gray value erosion (see operator [gray_erosion_rect](#)).

GenParamValues: 3,5,7,...,511
default: 'module_width'*0.18 + 3.5

'measure_sigma' Parameter for the extraction of linear edge segment pairs (see operator [measure_pairs](#)).

GenParamValues: 0.4...100
default: 'module_width' < 20: 0.7
'module_width' ≥ 20: 1.4

'measure_thresh' Parameter for the extraction of linear edge segment pairs (see operator [measure_pairs](#)).

GenParamValues: 1...255
default: 2.0

Attention

The operator [get_2d_bar_code](#) returns error codes to signal that the candidate region did not contain a valid 2D bar code. As the region candidates extracted by the operator [find_2d_bar_code](#) can perfectly well encompass regions without a bar code, every call to [get_2d_bar_code](#) should be surrounded by error handling code (see example).

Parameter

- ▷ **BarCodeRegion** (input_object) region $\leadsto$ *Hobject*
Region that might contain a 2D bar code.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte
Corresponding image.
- ▷ **BarCodeDescr** (input_control) barcode_2d-array $\leadsto$ *string* / integer / real
Description of the bar code class.
- ▷ **CodeRegDescr** (input_control) string-array $\leadsto$ *string* / integer / real
Additional parameters describing the bar code region. They can be used for extracting the data
- ▷ **GenParamNames** (input_control) string-array $\leadsto$ *string*
List of names of (optional) generic control parameters.
Default Value : ''
List of values : GenParamNames $\in$ {'module_rad_ratio', 'use_grayvals', 'use_grayval_min',
'use_grayval_max', 'use_grayval_range', 'enlarge_region_rad', 'thresh_percent', 'thresh_step_width',
'smooth_cont', 'max_line_dist1', 'max_line_dist2', 'min_cont_len', 'border_width_min', 'border_width_max',
'border_width', 'median_mask_rad', 'gray_erosion_size', 'measure_sigma', 'measure_thresh'}
- ▷ **GenParamValues** (input_control) number-array $\leadsto$ *real* / integer
List of values of the generic parameters.
Default Value : ''
Suggested values : GenParamValues $\in$ {0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 'yes', 'no', -1, 1.5,
2.5, 3.5, 4.5, 5.5, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50}
- ▷ **BarCodeDimension** (output_control) integer-array $\leadsto$ *integer*
Tuple with the dimension of the extracted symbol. In the case of ECC 200: data field width, height, symbol
index.
- ▷ **BarCodeData** (output_control) integer-array $\leadsto$ *integer*
Tuple with the data values of the extracted symbol. value > 0: logical 1, value < 0: logical 0, value =
0: module could not be classified.

Example

```
dev_error_var (ErrorCode, 1)
gen_2d_bar_code_descr ('Data Matrix ECC 200', ['mode'], ['printed'],
                     BarCodeDescr)
```



```

read_image (Image, 'bar2d.tif')
find_2d_bar_code (Image, CodeRegion, BarCodeDescr,
                  ['module_width'], [10], CodeRegDescr)
SymbolNum := |CodeRegion|
for i := 1 to SymbolNum by 1
  ObjectSelected := CodeRegion[i]
  dev_set_check ('~give_error')
  get_2d_bar_code (ObjectSelected, Image, BarCodeDescr, CodeRegDescr,
                  [], [], BarCodeDimension, BarCodeData)

  err := ErrorCode
  dev_set_check ('give_error')
  if (err = 2)
    decode_2d_bar_code (BarCodeDescr, BarCodeDimension, BarCodeData,
                        SymbolCharacters, CorrSymbolData, DecodedData,
                        DecodingError, StructuredAppend)

    if (|DecodedData|)
      tuple_chrt (DecodedData, String)
    endif
  endif
endif
endfor

```

Result

The return value can signal incorrect parameters as well as the failure to extract a 2D bar code. Such a failure can be due to different causes: If no surrounding rectangle was found, the operator returns the error code 8808. Error code 8807 signals that the rectangle does not lie completely inside the image. If no finder pattern was detected inside the border search area, the error code 8810 is returned. If the two continuous lines forming the 'L' of the finder pattern could not be found, the error code 8809 is returned. In the case of the printing method '*engraved_darkfield*', the error code 8811 signals that none of the gray value features can separate the test regions, i.e. that modules cannot be classified.

Parallelization Information

[get_2d_bar_code](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[find_2d_bar_code](#)

Possible Successors

[decode_2d_bar_code](#)

Alternatives

[get_2d_bar_code_pos](#)

Module

Barcode reader

```

get_2d_bar_code_pos ( BarCodeRegion, Image : : BarCodeDescr,
  CodeRegDescr, GenParamNames, GenParamValues : BarCodeDimension,
  BarCodeData, DataElementRow, DataElementCol )

```

Extract the data values of the elements (in ECC 200: “modules”) inside a bar code region (“Data Matrix symbol”) and their positions in the image.

[get_2d_bar_code_pos](#) extracts the 2D bar code data from a candidate symbol region given in [BarCodeRegion](#) and the corresponding image given in [Image](#) elements. In contrast to [get_2d_bar_code](#), this operator also returns the positions of the data elements in the image in the parameters [DataElementRow](#) and [DataElementCol](#). This can be used to check the result of the operator, e.g., by displaying the data in the real image. For a description of the other parameters, see [get_2d_bar_code](#).

Attention

The operator [get_2d_bar_code_pos](#) returns error codes to signal that the candidate region did not contain a valid 2D bar code. As the region candidates extracted by the operator [find_2d_bar_code](#) can perfectly well encompass regions without a bar code, every call to [get_2d_bar_code_pos](#) should be surrounded by error handling code (see example).

Parameter

- ▷ **BarCodeRegion** (input_object) region $\leadsto$ *Hobject*
Region that might contain a bar code.
 - ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte
Corresponding image.
 - ▷ **BarCodeDescr** (input_control) barcode_2d-array $\leadsto$ *string* / integer / real
Description of the 2D bar code class.
 - ▷ **CodeRegDescr** (input_control) string-array $\leadsto$ *string* / integer / real
Additional parameters describing the bar code region. They can be used for extracting the data.
 - ▷ **GenParamNames** (input_control) string-array $\leadsto$ *string*
List of names of (optional) generic control parameters.
Default Value : '[]'
List of values : GenParamNames $\in$ { 'module_rad_ratio', 'use_grayvals', 'use_grayval_min',
'use_grayval_max', 'use_grayval_range', 'enlarge_region_rad', 'thresh_percent', 'thresh_step_width',
'smooth_cont', 'max_line_dist1', 'max_line_dist2', 'min_cont_len', 'border_width_min', 'border_width_max',
'border_width', 'median_mask_rad', 'gray_erosion_size', 'measure_sigma', 'measure_thresh' }
 - ▷ **GenParamValues** (input_control) number-array $\leadsto$ *real* / integer
List of values of the generic parameters.
Default Value : '[]'
Suggested values : GenParamValues $\in$ { 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 'yes', 'no', -1, 1.5,
2.5, 3.5, 4.5, 5.5, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30, 40, 50 }
 - ▷ **BarCodeDimension** (output_control) integer-array $\leadsto$ *integer*
Tuple with the dimension of the extracted symbol. In the case of ECC 200: data field width, height, symbol
index.
 - ▷ **BarCodeData** (output_control) integer-array $\leadsto$ *integer*
Tuple with the data values of the extracted symbol. value > 0: logical 1, value < 0: logical 0, value =
0: module could not be classified.
 - ▷ **DataElementRow** (output_control) real-array $\leadsto$ *real*
Tuple with the row positions of the data elements of the extracted symbol in the image.
 - ▷ **DataElementCol** (output_control) real-array $\leadsto$ *real*
Tuple with the column positions of the data elements of the extracted symbol in the image.
-

Example

```

dev_set_check ('~give_error')
get_2d_bar_code_pos (ObjectSelected, Image, BarCodeDescr,
                    CodeRegDescr, [], [], BarCodeDimension, BarCodeData,
                    DataElementRow, DataElementCol)

err := ErrorCode
dev_set_check ('give_error')
if (err = 2)
    idx := 0
    dev_update_pc ('off')
    dev_update_time ('off')
    dev_update_var ('off')
    for m := 0 to BarCodeDimension[0]-1 by 1
        for n := 0 to BarCodeDimension[1]-1 by 1
            if (BarCodeData[idx]>0)
                set_color (WindowHandle, 'green')
            else
                if (BarCodeData[idx]<0)
                    set_color (WindowHandle, 'red')
                else
                    set_color (WindowHandle, 'blue')
                endif
            endif
            disp_circle (WindowHandle, DataElementRow[idx],
                        DataElementCol[idx], 1)
        endfor
    endfor

```

```

        idx := idx+1
    endfor
endfor
dev_update_pc ('on')
dev_update_time ('on')
dev_update_var ('on')
decode_2d_bar_code (BarCodeDescr, BarCodeDimension, BarCodeData,
                    SymbolCharacters, CorrSymbolData, DecodedData,
                    DecodingError, StructuredAppend)
endif

```

Result

The return value can signal incorrect parameters as well as the failure to extract a 2D bar code. Such a failure can be due to different causes: If no surrounding rectangle was found, the operator returns the error code 8808. Error code 8807 signals that the rectangle does not lie completely inside the image. If no finder pattern was detected inside the border search area, the error code 8810 is returned. If the two continuous lines forming the 'L' of the finder pattern could not be found, the error code 8809 is returned. In the case of the printing method *'engraved_darkfield'*, the error code 8811 signals that none of the gray value features can separate the test regions, i.e., that modules cannot be classified.

Parallelization Information

[get_2d_bar_code_pos](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[find_2d_bar_code](#)

Possible Successors

[decode_2d_bar_code](#)

Alternatives

[get_2d_bar_code](#)

See also

[gen_2d_bar_code_descr](#)

Module

Barcode reader

15.4 Calibration

caltab_points (: : CalTabDescrFile : X, Y, Z)

Read the mark center points from the calibration plate description file.

[caltab_points](#) reads the mark center points from the calibration plate description file [CalTabDescrFile](#) (see [create_caltab](#)) and returns their coordinates in [X](#), [Y](#) and [Z](#). The mark center points are 3D coordinates in the calibration plate coordinate system and describe the 3D model of the calibration plate. The calibration plate coordinate system is located in the middle of the surface of the calibration plate, its z-axis points into the calibration plate, its x-axis to the right, and its y-axis downwards.

The mark center points are typically used as input parameters for the operator [camera_calibration](#). This operator projects the model points into the image, minimizes the distance between the projected points and the observed 2D coordinates in the image (see [find_marks_and_pose](#)), and from this computes the exact values for the interior and exterior camera parameters.

Parameter

- ▷ **CalTabDescrFile** (input_control) string $\leadsto$ *string*
File name of the calibration plate description.
Default Value : 'caltab.descr'
- ▷ **X** (output_control) real-array $\leadsto$ *real*
X coordinates of the mark center points in the coordinate system of the calibration plate.
- ▷ **Y** (output_control) real-array $\leadsto$ *real*
Y coordinates of the mark center points in the coordinate system of the calibration plate.

- ▷ **Z** (output_control) real-array $\leadsto$ real
 Z coordinates of the mark center points in the coordinate system of the calibration plate.

Example

```
* read_image(Image1, 'calib-01')
* find calibration pattern
find_caltab(Image1, Caltab1, 'caltab.descr', 3, 112, 5)
* find calibration marks and start poses
StartCamPar := [0.008, 0.0, 0.000011, 0.000011, 384, 288, 768, 576]
find_marks_and_pose(Image1, Caltab1, 'caltab.descr', StartCamPar,
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord1, CCoord1,
                    StartPose1)
* read 3D positions of calibration marks
caltab_points('caltab.descr', NX, NY, NZ) >
* camera calibration
camera_calibration(NX, NY, NZ, RCoord1, CCoord1, StartCamPar,
                  StartPose1, 'all', CamParam, FinalPose, Errors)
* visualize calibration result
disp_image(Image1, WindowHandle)
set_color(WindowHandle, 'red')
disp_caltab('caltab.descr', CamParam, FinalPose, 1.0)
```

Result

caltab_points returns 2 (H_MSG_TRUE) if all parameter values are correct and the file **CalTabDescrFile** has been read successfully. If necessary, an exception handling is raised.

Parallelization Information

caltab_points is *reentrant* and processed *without* parallelization.

Possible Successors

camera_calibration

See also

find_caltab, **find_marks_and_pose**, **camera_calibration**, **disp_caltab**, **sim_caltab**, **project_3d_point**, **get_line_of_sight**, **create_caltab**

Module

Camera calibration

camera_calibration (: : NX, NY, NZ, NRow, NCol, StartCamParam,
 NStartPose, EstimateParams : CamParam, NFinalPose, Errors)

Determine all camera parameters by a simultaneous minimization process.

camera_calibration performs the calibration of a camera. For this, known 3D model points (with coordinates **NX**, **NY**, **NZ**) are projected in the image and the sum of the squared distance between these projections and the corresponding image points (with coordinates **NRow**, **NCol**) is minimized.

If the minimization converges, the exact interior (**CamParam**) and exterior (**NFinalPose**) camera parameters are determined by this minimization algorithm. The parameters **StartCamParam** and **NStartPose** are used as initial values for the minimization process. Since this algorithm simultaneously handles correspondences between image and model points from different images, it is also called **multi-image calibration**.

In general, camera calibration means the exact determination of the parameters that model the (optical) projection of any 3D world point $\mathbf{P}^w$ into a (sub-)pixel $[r,c]$ in the image. This is important, if the original 3D pose of an object has to be computed using an image (e.g., measuring of industrial parts).

Used 3D camera model

The projection consists of multiple steps: First, the point $\mathbf{p}^w$ is transformed from world into camera coordinates (points as homogeneous vectors, compare **affine_trans_point_3d**):

$$\begin{pmatrix} \mathbf{p}^c \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{pmatrix} \mathbf{p}^w \\ 1 \end{pmatrix}$$

Then, the point is projected into the image plane, i.e., onto the sensor chip. If the underlying camera model is a **pinhole camera with radial distortions**, i.e., if the focal length passed in [StartCamParam](#) is greater than 0, the projection is described by the following equations:

$$u = \text{Focus} \cdot \frac{x}{z} \quad \text{and} \quad v = \text{Focus} \cdot \frac{y}{z}$$

In contrast, if the focal length is passed as 0 in [StartCamParam](#), the camera model of a **telecentric camera with radial distortions** is used, i.e., it is assumed that the optics of the lens of the camera performs a parallel projection. In this case, the corresponding equations are:

$$u = x \quad \text{and} \quad v = y$$

The following equations compensate for radial distortion:

$$\tilde{u} = \frac{2u}{1 + \sqrt{1 - 4\kappa(u^2 + v^2)}} \quad \text{and} \quad \tilde{v} = \frac{2v}{1 + \sqrt{1 - 4\kappa(u^2 + v^2)}}$$

Finally, the point is transformed from the image plane coordinate system into the image coordinate system, i.e., the pixel coordinate system:

$$c = \frac{\tilde{u}}{S_x} + C_x \quad \text{and} \quad r = \frac{\tilde{v}}{S_y} + C_y$$

Camera parameters

The total of 14 camera parameters can be divided into the interior and exterior camera parameters:

Interior camera parameters: These parameters describe the characteristics of the used camera, especially the dimension of the sensor itself and the projection properties of the used combination of lens, camera, and frame grabber. The camera model (as described above) contains the following 8 parameters:

Focus: Focal length of the lens. 0 for telecentric lenses.

Kappa (κ): Distortion coefficient to model the pillow- or barrel-shaped distortions caused by the lens.

Sx: Scale factor, corresponds to the horizontal distance between two neighboring cells on the sensor. Attention: This value increases, if the image is subsampled!

Sy: Scale factor, corresponds to the vertical distance between two neighboring cells on the sensor. Since in most cases the image signal is sampled line-synchronously, this value is determined by the dimension of the sensor and needn't be estimated for pinhole cameras by the calibration process. Attention: This value increases, if the image is subsampled!

Cx: Column coordinate of the image center point (center of the radial distortion).

Cy: Row coordinate of the image center point (center of the radial distortion).

ImageWidth: Width of the sampled image. Attention: This value decreases, if the image is subsampled!

ImageHeight: Height of the sampled image. Attention: This value decreases, if the image is subsampled!

Exterior camera parameters: These 6 parameters describe the 3D pose, i.e., the position and orientation, of the world coordinate system relative to the camera coordinate system. Three parameters describe the translation, three the rotation. See [create_pose](#) for more information about 3D poses. Note that [camera_calibration](#) operates with all types of 3D poses for [NStartPose](#).

When using the standard calibration plate, the world coordinate system is defined by the coordinate system of the calibration plate, which is located in the middle of the surface of the calibration plate, its z-axis pointing into the calibration plate, its x-axis to the right, and its y-axis downwards.

Additional information about the calibration process

The use of `camera_calibration` leads to some questions, which are dealt with in the following sections:

How to generate a appropriate calibration plate? The simplest method to determine the interior parameters of a camera is the use of the standard calibration plate as generated by the operator `create_caltab`. You can obtain high-precision calibration plates in various sizes and materials from your local distributor. In case of small distances between object and lens it may be sufficient to print the calibration pattern by a laser printer and to mount it on a cardboard. Otherwise – especially by using a wide-angle lens – it is possible to print the PostScript file on a large ink-jet printer and to mount it on a aluminum plate. It is very important that the mark coordinates in the calibration plate description file correspond to the real ones on the calibration plate with high accuracy. Thus, the calibration plate description file has to be modified in accordance with the measurement of the calibration plate!

How to take a set of suitable images? If you use the standard calibration plate, you can proceed in the following way: With the combination of lens (fixed distance!), camera, and frame grabber to be calibrated a set of images of the calibration plate has to be taken, see `open_framegrabber` and `grab_image`. The following items have to be considered:

- At least a total of 10 to 20 images should be taken into account.
- The calibration plate has to be completely visible (incl. border!).
- Reflections etc. on the calibration plate should be avoided.
- Within the set of images the calibration plate should appear in different positions and orientations: Once left in the image, once right, once (left and right) at the bottom, once (left or right) at the top, from different distances etc. At this, the calibration plate should be rotated around its x- and/or y-axis, so the perspective distortions of the calibration pattern are clearly visible. Thus, the exterior camera parameters (camera pose with regard of the calibration plate) should be set to a large variety of different values!
- The calibration plate should fill at least a quarter of the whole image to ensure the robust detection of the marks.

How to extract the calibration marks in the images? If a standard calibration plate is used, you can use the operators `find_caltab` and `find_marks_and_pose` to determine the coordinates of the calibration marks in each image and to compute a rough estimate for the exterior camera parameters. The concatenation of these values can directly be used as initial values for the exterior camera parameters (`NStartPose`) in `camera_calibration`.

Obviously, images in which the segmentation of the calibration plate (`find_caltab`) has failed or the calibration marks haven't been determined successfully by `find_marks_and_pose` should not be used.

How to find suitable initial values for the interior camera parameters? The operators `find_marks_and_pose` (determination of initial values for the exterior camera parameters) and `camera_calibration` require initial values for the interior camera parameters. These parameters can be provided by a appropriate text file (see `read_cam_par`), which can be generated by `write_cam_par` or can be edited manually.

The following should be considered for the initial values of the single parameters:

Focus: The initial value is the nominal focal length of the the used lens, e.g., 0.008 m.

Kappa: Use 0.0 as initial value.

Sx: The initial value for the horizontal distance between two neighboring cells depends on the dimension of the used chip of the camera (see technical specifications of the camera). Generally, common chips are either 1/3"-Chips (e.g., SONY XC-73, SONY XC-777), 1/2"-Chips (e.g., SONY XC-999, Panasonic WV-CD50), or 2/3"-Chips (e.g., SONY DXC-151, SONY XC-77). Notice: The value of Sx increases if the image is subsampled! Appropriate initial values are:

	Full image (768*576)	Subsampling (384*288)
1/3"-Chip	0.0000055 m	0.0000110 m
1/2"-Chip	0.0000086 m	0.0000172 m
2/3"-Chip	0.0000110 m	0.0000220 m

The value for Sx is calibrated, since the video signal of a camera normally isn't sampled pixel-synchronously.

Sy: Since most off-the-shelf cameras have quadratic pixels, the same values for Sy are valid as for Sx. In contrast to Sx the value for Sy will **not** be calibrated for pinhole cameras, because the video signal of a camera normally is sampled line-synchronously. Thus, the initial value is equal to the final value. Appropriate initial values are:

	Full image (768*576)	Subsampling (384*288)
1/3"-Chip	0.0000055 m	0.0000110 m
1/2"-Chip	0.0000086 m	0.0000172 m
2/3"-Chip	0.0000110 m	0.0000220 m

Cx and Cy: Initial values for the coordinates of the image center is the half image width and half image height. Notice: The values of Cx and Cy decrease if the image is subsampled! Appropriate initial values are:

	Full image (768*576)	Subsampling (384*288)
Cx	384.0	192.0
Cy	288.0	144.0

ImageWidth and ImageHeight: These two parameters are determined by the the used frame grabber and therefore are not calibrated. Appropriate initial values are, for example:

	Full image (768*576)	Subsampling (384*288)
ImageWidth	768	384
ImageHeight	576	288

Which camera parameters have to be estimated? The input parameter `EstimateParams` is used to select which camera parameters to estimate. Usually this parameter is set to `'all'`, i.e., all 6 exterior camera parameters (translation and rotation) and all interior camera parameters are determined. If the interior camera parameters already have been determined (e.g., by a previous call to `camera_calibration`) it is often desired to only determine the pose of world coordinate system in camera coordinates (i.e., the exterior camera parameters). In this case, `EstimateParams` can be set to `'pose'`. This has the same effect as `EstimateParams = ['alpha','beta','gamma','transx','transy','transz']`. Otherwise, `EstimateParams` contains a tuple of strings indicating the combination of parameters to estimate.

What is the order within the individual parameters? The length of the tuple `NStartPose` corresponds to the number of calibration images, e.g., using 15 images leads to a length of the tuple `NStartPose` equal to $15 \cdot 7 = 105$ (15 times the 7 exterior camera parameters). The first 7 values correspond to the pose of the calibration plate in the first image, the next 7 values to the pose in the second image, etc.

This fixed number of calibration images has to be considered within the tuples with the coordinates of the 3D model marks and the extracted 2D marks. If 15 images are used, the length of the tuples `NRow` and `NCol` is 15 times the length of the tuples with the coordinates of the 3D model marks (`NX`, `NY`, and `NZ`). If every image consists 49 marks, the length of the tuples `NRow` and `NCol` is $15 \cdot 49 = 735$, while the length of the tuples `NX`, `NY`, and `NZ` is 49. The order of the values in `NRow` and `NCol` is “image after image”, i.e., using 49 marks the first 3D model point corresponds to the 1st, 50th, 99th, 148th, 197th, 246th, etc. extracted 2D mark.

The 3D model points can be read from a calibration plate description file using the operator `caltab.points`. Initial values for the poses of the calibration plate can be determined by applying `find_marks_and_pose` for each image. The tuple `NStartPose` is set by the concatenation of all these poses.

What is the meaning of the output parameters? If the camera calibration process is finished successfully, i.e., the minimization process has converged, the output parameters `CamParam` and `NFinalPose` contain the computed exact values for the interior and exterior camera parameters. The length of the tuple `NFinalPose` corresponds to the length of the tuple `NStartPose`.

The representation types of `NFinalPose` correspond to the representation type of the first tuple of `NStartPose` (see `create_pose`). You can convert the representation type by `convert_pose_type`.

The computed average errors (`Errors`) give an impression of the accuracy of the calibration. The error values (deviations in x and y coordinates) are measured in pixels.

Must I use a planar calibration object? No. The operator `camera_calibration` is designed in a way that the input tuples `NX`, `NY`, `NZ`, `NRow`, and `NCol` can contain any 3D/2D correspondences, see the above paragraph explaining the order of the single parameters.

Thus, it makes no difference how the required 3D model marks and the corresponding extracted 2D marks are determined. On one hand, it is possible to use a 3D calibration pattern, on the other hand, you also can use any characteristic points (natural landmarks) with known position in the world. By setting `EstimateParams` to `'pose'`, it is thus possible to compute the pose of an object in camera coordinates! For this, at least three

3D/2D-correspondences are necessary as input. `NStartPose` can, e.g., be generated directly as shown in the program example for `create_pose`.

Attention

The minimization process of the calibration depends on the initial values of the interior (`StartCamParam`) and exterior (`NStartPose`) camera parameters. The computed average errors `Errors` give an impression of the accuracy of the calibration. The errors (deviations in x and y coordinates) are measured in pixels.

Parameter

- ▷ **NX** (input_control) real-array $\leadsto$ real
Ordered tuple with all x coordinates of the calibration marks (in meters).
- ▷ **NY** (input_control) real-array $\leadsto$ real
Ordered tuple with all y coordinates of the calibration marks (in meters).
- ▷ **NZ** (input_control) real-array $\leadsto$ real
Ordered tuple with all z coordinates of the calibration marks (in meters).
- ▷ **NRow** (input_control) real-array $\leadsto$ real
Ordered tuple with all row coordinates of the extracted calibration marks (in pixels).
- ▷ **NCol** (input_control) real-array $\leadsto$ real
Ordered tuple with all column coordinates of the extracted calibration marks (in pixels).
- ▷ **StartCamParam** (input_control) real-array $\leadsto$ real / integer
Initial values for the interior camera parameters.
- ▷ **NStartPose** (input_control) pose-array $\leadsto$ real / integer
Ordered tuple with all initial values for the exterior camera parameters.
- ▷ **EstimateParams** (input_control) string $\leadsto$ string / integer
Camera parameters to be estimated.
Default Value : 'all'
List of values : EstimateParams $\in$ {'all', 'pose', 'alpha', 'beta', 'gamma', 'transx', 'transy', 'transz', 'focus', 'kappa', 'cx', 'cy', 'sx', 'sy'}
- ▷ **CamParam** (output_control) number-array $\leadsto$ real / integer
Interior camera parameters.
- ▷ **NFinalPose** (output_control) pose-array $\leadsto$ real / integer
Ordered tuple with all exterior camera parameters.
- ▷ **Errors** (output_control) real-array $\leadsto$ real
Average error distances in pixels.

Example

```
* read calibration images
read_image(Image1, 'calib-01')
read_image(Image2, 'calib-02')
read_image(Image3, 'calib-03')
* find calibration pattern
find_caltab(Image1, Caltab1, 'caltab.descr', 3, 112, 5)
find_caltab(Image2, Caltab2, 'caltab.descr', 3, 112, 5)
find_caltab(Image3, Caltab3, 'caltab.descr', 3, 112, 5)
* find calibration marks and start poses
find_marks_and_pose(Image1, Caltab1, 'caltab.descr',
                    [0.008, 0.0, 0.000011, 0.000011, 384, 288, 768, 576],
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord1, CCoord1,
                    StartPose1)
find_marks_and_pose(Image2, Caltab2, 'caltab.descr',
                    [0.008, 0.0, 0.000011, 0.000011, 384, 288, 768, 576],
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord2, CCoord2,
                    StartPose2)
find_marks_and_pose(Image3, Caltab3, 'caltab.descr',
                    [0.008, 0.0, 0.000011, 0.000011, 384, 288, 768, 576],
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord3, CCoord3,
                    StartPose3)
```



```

* read 3D positions of calibration marks
caltab_points('caltab.descr', NX, NY, NZ)
* camera calibration
camera_calibration(NX, NY, NZ, [RCoord1, RCoord2, RCoord3],
                        [CCoord1, CCoord2, CCoord3],
                        [0.008, 0.0, 0.000011, 0.000011, 384, 288, 768, 576],
                        [StartPose1, StartPose2, StartPose3], 'all',
                        CamParam, NFinalPose, Errors)
* write interior camera parameters to file
write_cam_par(CamParam, 'campar.dat')

```

Result

[camera_calibration](#) returns 2 (H_MSG.TRUE) if all parameter values are correct and the desired camera parameters have been determined by the minimization algorithm. If necessary, an exception handling is raised.

Parallelization Information

[camera_calibration](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[find_marks_and_pose](#), [caltab_points](#), [read_cam_par](#)

Possible Successors

[write_pose](#), [pose_to_hom_mat3d](#), [disp_caltab](#), [sim_caltab](#)

See also

[find_caltab](#), [find_marks_and_pose](#), [disp_caltab](#), [sim_caltab](#), [write_cam_par](#),
[read_cam_par](#), [create_pose](#), [convert_pose_type](#), [write_pose](#), [read_pose](#),
[pose_to_hom_mat3d](#), [hom_mat3d_to_pose](#), [caltab_points](#), [create_caltab](#)

Module

Camera calibration

change_radial_distortion_cam_par (: : Mode, CamParIn,
 Kappa : CamParOut)

Determine new camera parameters in accordance to the specified radial distortion.

[change_radial_distortion_cam_par](#) modifies the interior camera parameters in accordance to the specified radial distortion [Kappa](#). Via [Mode](#) one of the following modes can be selected:

- **'fixed'**: Only [Kappa](#) is modified, the other interior camera parameters remain unchanged. In general, this leads to a change of the visible part of the scene.
- **'fullsize'**: The scale factors S_x and S_y and the image center point $[C_x, C_y]^T$ are modified in order to preserve the visible part of the scene. Thus, all points visible in the original image are also visible in the modified (rectified) image. In general, this leads to undefined pixels in the modified image.
- **'adaptive'**: A trade-off between the other modes: The visible part of the scene is slightly reduced to prevent undefined pixels in the modified image. Similarly to 'fullsize', the scale factors and the image center point are modified.

In all modes the radial distortion coefficient κ in [CamParOut](#) is set to [Kappa](#). The transformation of a pixel in the modified image into the image plane using [CamParOut](#) results in the same point as the transformation of a pixel in the original image via [CamParIn](#).

Parameter

- ▷ **Mode** (input_control) string $\leadsto$ string
 Mode
Default Value : 'adaptive'
Suggested values : Mode $\in$ {'fullsize', 'adaptive', 'fixed'}
- ▷ **CamParIn** (input_control) real-array $\leadsto$ real
 Interior camera parameters (original).

- ▷ **Kappa** (input_control) real $\leadsto$ real
Desired radial distortion.
Default Value : 0.0
- ▷ **CamParOut** (output_control) real-array $\leadsto$ real
Interior camera parameters (modified).

Result

[change_radial_distortion_cam_par](#) returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception handling is raised.

Parallelization Information

[change_radial_distortion_cam_par](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[camera_calibration](#), [read_cam_par](#)

Possible Successors

[change_radial_distortion_image](#), [change_radial_distortion_contours_xld](#)

See also

[camera_calibration](#), [read_cam_par](#), [change_radial_distortion_image](#),
[change_radial_distortion_contours_xld](#)

Module

Camera calibration

change_radial_distortion_contours_xld (
Contours : ContoursRectified : CamParIn, CamParOut :)

Change the radial distortion of contours.

[change_radial_distortion_contours_xld](#) changes the radial distortion of the input contours **Contours** in accordance to the interior camera parameters [CamParIn](#) and [CamParOut](#). Each subpixel of an input contour is transformed into the image plane using [CamParIn](#) and subsequently projected into a subpixel of the corresponding contour in **ContoursRectified** using [CamParOut](#).

If [CamParOut](#) was computed via [change_radial_distortion_cam_par](#), the contours **ContoursRectified** are equivalent to **Contours** obtained with a lens with a modified radial distortion. If κ is 0 the contours are rectified. A subsequent pose estimation (determination of the exterior camera parameters) is not affected by this operation.

Parameter

- ▷ **Contours** (input_object) xld_cont(-array) $\leadsto$ Hobject
Original contours.
- ▷ **ContoursRectified** (output_object) xld_cont(-array) $\leadsto$ Hobject
Resulting contours with modified radial distortion.
- ▷ **CamParIn** (input_control) real-array $\leadsto$ real
Interior camera parameter for [Contours](#).
- ▷ **CamParOut** (input_control) real-array $\leadsto$ real
Interior camera parameter for [ContoursRectified](#).

Parallelization Information

[change_radial_distortion_contours_xld](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[change_radial_distortion_cam_par](#), [gen_contours_skeleton_xld](#), [edges_sub_pix](#),
[smooth_contours_xld](#)

Possible Successors

[gen_polygons_xld](#), [smooth_contours_xld](#)

See also

[change_radial_distortion_cam_par](#), [camera_calibration](#), [read_cam_par](#),
[change_radial_distortion_image](#)

Module

Camera calibration

change_radial_distortion_image (Image, Region : ImageRectified : CamParIn, CamParOut :)

Change the radial distortion of an image.

[change_radial_distortion_image](#) changes the radial distortion of the input image [Image](#) in accordance to the interior camera parameters [CamParIn](#) and [CamParOut](#). The image size remains unchanged. Each pixel of the output image is transformed into the image plane using [CamParOut](#) and subsequently projected into a subpixel of [Image](#) using [CamParIn](#). The resulting gray value is determined by bilinear interpolation. If the subpixel is outside of [Image](#), the corresponding pixel in [ImageRectified](#) is set to 'black' and eliminated from the image domain.

If [CamParOut](#) was computed via [change_radial_distortion_cam_par](#), [ImageRectified](#) is equivalent to [Image](#) obtained with a lens with a modified radial distortion. If κ is 0 the image is rectified. A subsequent pose estimation (determination of the exterior camera parameters) is not affected by this operation.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / uint2
Original image.
- ▷ **Region** (input_object) region $\leadsto$ *Hobject*
Region of interest in [Image](#).
- ▷ **ImageRectified** (output_object) (multichannel-)image(-array) $\leadsto$ *Hobject* : byte / uint2
Resulting image with modified radial distortion.
- ▷ **CamParIn** (input_control) real-array $\leadsto$ *real*
Interior camera parameter for [Image](#).
- ▷ **CamParOut** (input_control) real-array $\leadsto$ *real*
Interior camera parameter for [Image](#).

Result

[change_radial_distortion_image](#) returns 2 (H_MSG_TRUE) if all parameter values are correct and the input is not empty. If necessary, an exception handling is raised.

Parallelization Information

[change_radial_distortion_image](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[change_radial_distortion_cam_par](#), [read_image](#), [grab_image](#)

Possible Successors

[edges_image](#), [threshold](#)

See also

[change_radial_distortion_cam_par](#), [camera_calibration](#), [read_cam_par](#),
[change_radial_distortion_contours_xld](#)

Module

Camera calibration

contour_to_world_plane_xld (Contours : ContoursTrans : CamParam, WorldPose, Scale :)

Transform an XLD contour into the plane $z=0$ of a world coordinate system.

The operator [contour_to_world_plane_xld](#) transforms contour points given in [Contours](#) into the plane $z=0$ in a world coordinate system and returns the 3D contour points in [ContoursTrans](#). The world coordinate system is chosen by passing its 3D pose relative to the camera coordinate system in [WorldPose](#). In [CamParam](#) you must pass the interior camera parameters (see [write_cam_par](#) for the sequence of the parameters and the underlying camera model).

In many cases [CamParam](#) and [WorldPose](#) are the result of calibrating the camera with the operator [camera_calibration](#). See below for an example.

With the parameter [Scale](#) you can scale the resulting 3D coordinates. The parameter [Scale](#) must be specified as the ratio *desired unit/original unit*. The original unit is determined by the coordinates of the calibration object.

If the original unit is meters (which is the case if you use the standard calibration plate), you can set the desired unit directly by selecting 'm', 'cm', 'mm' or ' μ m' for the parameter [Scale](#).

Internally, the operator first computes the line of sight between the projection center and the image point in the camera coordinate system, taking into account the radial distortions. The line of sight is then transformed into the world coordinate system specified in [WorldPose](#). By intersecting the plane $z=0$ with the line of sight the 3D coordinates of the transformed contour [ContoursTrans](#) are obtained.

Parameter

- ▷ **Contours** (input_object) xld_cont(-array) $\leadsto$ *Hobject*
Input XLD contours to be transformed in image coordinates.
- ▷ **ContoursTrans** (output_object) xld_cont(-array) $\leadsto$ *Hobject*
Transformed XLD contours in world coordinates.
- ▷ **CamParam** (input_control) number-array $\leadsto$ *real* / integer
Interior camera parameters.
Number of elements : 8
- ▷ **WorldPose** (input_control) pose-array $\leadsto$ *real* / integer
3D pose of the world coordinate system in camera coordinates.
Number of elements : 7
- ▷ **Scale** (input_control) number $\leadsto$ *string* / integer / real
Scale oder dimension
Default Value : 'm'
Suggested values : $\text{Scale} \in \{ 'm', 'cm', 'mm', 'microns', '\mu m', 1.0, 0.01, 0.001, '1.0e-6', 0.0254, 0.3048, 0.9144 \}$

Example

```
* perform camera calibration (with standard calibration plate)
camera_calibration(NX, NY, NZ, NRow, NCol, StartCamParam, NStartPose, 'all',
                  FinalCamParam, NFinalPose, Errors)
* world coordinate system is defined by calibration plate in first image
FinalPose1 := NFinalPose[0:6]
* compensate thickness of plate
set_origin_pose(FinalPose1, 0, 0, 0.0006, WorldPose)
* transform contours into world coordinate system (unit mm)
contour_to_world_plane_xld(Contours, ContoursTrans,
                          FinalCamParam, WorldPose, 'mm')
```

Result

[contour_to_world_plane_xld](#) returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception handling is raised.

Parallelization Information

[contour_to_world_plane_xld](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[create_pose](#), [hom_mat3d_to_pose](#), [camera_calibration](#), [hand_eye_calibration](#),
[set_origin_pose](#)

See also

[image_points_to_world_plane](#)

Module

Camera calibration

convert_pose_type (: : PoseIn, OrderOfTransform, OrderOfRotation,
ViewOfTransform : PoseOut)

Change the representation type of a 3D pose.

[convert_pose_type](#) converts the 3D pose [PoseIn](#) into a 3D pose [PoseOut](#) with a different representation type. See [create_pose](#) for details about 3D poses, their representation types, and the meaning of the parameters [OrderOfTransform](#), [OrderOfRotation](#), and [ViewOfTransform](#).

Note that `convert_pose_type` only changes the representation of a 3D pose, but not the rigid transformation described by the pose.

Parameter	
▷ PoseIn (input_control)	pose-array $\leadsto$ real / integer Original 3D pose. Number of elements : 7
▷ OrderOfTransform (input_control)	string $\leadsto$ string Order of rotation and translation. Default Value : "Rp+T" Suggested values : OrderOfTransform $\in$ {"Rp+T", "R(p-T)"}
▷ OrderOfRotation (input_control)	string $\leadsto$ string Meaning of the rotation values. Default Value : "gba" Suggested values : OrderOfRotation $\in$ {"gba", "abg", "rodriguez"}
▷ ViewOfTransform (input_control)	string $\leadsto$ string View of transformation. Default Value : "point" Suggested values : ViewOfTransform $\in$ {"point", "coordinate_system"}
▷ PoseOut (output_control)	pose-array $\leadsto$ real / integer 3D transformation. Number of elements : 7

Example

```
* get pose (exterior camera parameters):
read_pose ('campose.dat', Pose)
* convert pose to a pose with desired semantic
convert_pose_type (Pose, 'Rp+T', 'abg', 'point', Pose2)
```

Result

`convert_pose_type` returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception handling is raised.

Parallelization Information

`convert_pose_type` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`create_pose`, `hom_mat3d_to_pose`, `camera_calibration`, `hand_eye_calibration`

Possible Successors

`write_pose`

See also

`create_pose`, `get_pose_type`, `write_pose`, `read_pose`

Module

Camera calibration

create_caltab (: : Width, CalTabDescrFile, CalTabFile :)

Generate a calibration plate description file and a corresponding PostScript file.

`create_caltab` generates the description of a standard calibration plate for HALCON. This calibration plate consists of 49 black circular marks on a white plane, which are surrounded by a black frame. The parameter `Width` sets the width (equal to the height) of the whole calibration plate in meters. Using a width of 0.8 m, the distance between two neighboring marks becomes 10 cm, and the mark radius and the frame width are set to 2.5 cm. The calibration plate coordinate system is located in the middle of the surface of the calibration plate, its z-axis points into the calibration plate, its x-axis to the right, and its y-axis downwards.

The file `CalTabDescrFile` contains the calibration plate description, e.g., the number of rows and columns of the calibration plate, the geometry of the surrounding frame (see `find_caltab`), and the coordinates and

the radius of all calibration plate marks given in the calibration plate coordinate system. A file generated by `create_caltab` looks like the following (comments are marked by a '#' at the beginning of a line):

```
#
# Description of the standard calibration plate
# used for the camera calibration in HALCON
#

# 7 rows X 7 columns
# Distance between mark centers [meter]: 0.1

# Number of marks per row
r 7

# Number of marks per column
c 7

# Quadratic frame (with outer and inner border) around calibration plate
w 0.025
o -0.41 0.41 0.41 -0.41
i -0.4 0.4 0.4 -0.4

# calibration marks: x y radius [Meter]

# calibration marks at y = -0.3 m
-0.3 -0.3 0.025
-0.2 -0.3 0.025
-0.1 -0.3 0.025
0 -0.3 0.025
0.1 -0.3 0.025
0.2 -0.3 0.025
0.3 -0.3 0.025

# calibration marks at y = -0.2 m
-0.3 -0.2 0.025
-0.2 -0.2 0.025
-0.1 -0.2 0.025
0 -0.2 0.025
0.1 -0.2 0.025
0.2 -0.2 0.025
0.3 -0.2 0.025

# calibration marks at y = -0.1 m
-0.3 -0.1 0.025
-0.2 -0.1 0.025
-0.1 -0.1 0.025
0 -0.1 0.025
0.1 -0.1 0.025
0.2 -0.1 0.025
0.3 -0.1 0.025

# calibration marks at y = 0 m
-0.3 0 0.025
-0.2 0 0.025
-0.1 0 0.025
0 0 0.025
0.1 0 0.025
0.2 0 0.025
0.3 0 0.025
```

```

# calibration marks at y = 0.1 m
-0.3 0.1 0.025
-0.2 0.1 0.025
-0.1 0.1 0.025
0 0.1 0.025
0.1 0.1 0.025
0.2 0.1 0.025
0.3 0.1 0.025

# calibration marks at y = 0.2 m
-0.3 0.2 0.025
-0.2 0.2 0.025
-0.1 0.2 0.025
0 0.2 0.025
0.1 0.2 0.025
0.2 0.2 0.025
0.3 0.2 0.025

# calibration marks at y = 0.3 m
-0.3 0.3 0.025
-0.2 0.3 0.025
-0.1 0.3 0.025
0 0.3 0.025
0.1 0.3 0.025
0.2 0.3 0.025
0.3 0.3 0.025

```

The file [CalTabFile](#) contains the corresponding PostScript description of the calibration plate.

Attention

Depending on the accuracy of the used output device (e.g., laser printer), the printed calibration plate may not match the values in the calibration plate description file [CalTabDescrFile](#) exactly. Thus, the coordinates of the calibration marks in the calibration plate description file may have to be corrected!

Parameter

- ▷ **Width** (input_control) real $\leadsto$ *real*
 Width of the calibration plate in meters.
Default Value : 0.8
Suggested values : $\text{Width} \in \{1.2, 0.8, 0.6, 0.4, 0.2, 0.1\}$
Recommended Increment : 0.1
Restriction : $0.0 < \text{Width}$
- ▷ **CalTabDescrFile** (input_control) string $\leadsto$ *string*
 File name of the calibration plate description.
Default Value : 'caltab.descr'
- ▷ **CalTabFile** (input_control) string $\leadsto$ *string*
 File name of the PostScript file.
Default Value : 'caltab.ps'

Example

```

* create calibration plate with width = 80 cm
create_caltab(0.8, 'caltab.descr', 'caltab.ps')

```

Result

[create_caltab](#) returns 2 (H_MSG_TRUE) if all parameter values are correct and both files have been written successfully. If necessary, an exception handling is raised.

Parallelization Information

[create_caltab](#) is processed *completely exclusively* without parallelization.

Possible Successors

[read_cam_par](#), [caltab_points](#)

See also

[find_caltab](#), [find_marks_and_pose](#), [camera_calibration](#), [disp_caltab](#), [sim_caltab](#)

Module

Camera calibration

create_pose (: : TransX, TransY, TransZ, RotX, RotY, RotZ,
OrderOfTransform, OrderOfRotation, ViewOfTransform : Pose)

Create a 3D pose.

create_pose creates the 3D pose **Pose**. A pose describes a rigid 3D transformation, i.e., a transformation consisting of an arbitrary translation and rotation, with 6 parameters: **TransX**, **TransY**, and **TransZ** specify the translation along the x-, y-, and z-axis, respectively, while **RotX**, **RotY**, and **RotZ** describe the rotation.

3D poses are typically used in two ways: First, to describe the position and orientation of one coordinate system relative to another (e.g., the pose of a part's coordinate system relative to the camera coordinate system - in short: the pose of the part relative to the camera) and secondly, to describe how coordinates can be transformed between two coordinate systems (e.g., to transform points from part coordinates into camera coordinates).

Representation of orientation (rotation)

A 3D rotation around an arbitrary axis can be represented by 3 parameters in multiple ways. HALCON lets you choose between three of them with the parameter **OrderOfRotation**: If you pass the value 'gba', the rotation is described by the following chain of rotations around the three axes (see [hom_mat3d_rotate](#) for the content for the rotation matrices $\mathbf{R}_x$, $\mathbf{R}_y$, and $\mathbf{R}_z$):

$$\mathbf{R}_{gba} = \mathbf{R}_x(\text{RotX}) \cdot \mathbf{R}_y(\text{RotY}) \cdot \mathbf{R}_z(\text{RotZ})$$

Please note that you can “read” this chain in two ways: If you start from the right, the rotations are always performed relative to the global (i.e., fixed or “old”) coordinate system. Thus, $\mathbf{R}_{gba}$ can be read as follows: First rotate around the z-axis, then around the “old” y-axis, and finally around the “old” x-axis. In contrast, if you read from the left to the right, the rotations are performed relative to the local (i.e., “new”) coordinate system. Then, $\mathbf{R}_{gba}$ corresponds to the following: First rotate around the x-axis, then around the “new” y-axis, and finally around the “new(est)” z-axis.

Reading $\mathbf{R}_{gba}$ from right to left corresponds to the following sequence of operator calls:

```
hom_mat3d_identity (HomMat3DIdent)
hom_mat3d_rotate (HomMat3DIdent, RotZ, 'z', 0, 0, 0, HomMat3DRotZ)
hom_mat3d_rotate (HomMat3DRotZ, RotY, 'y', 0, 0, 0, HomMat3DRotYZ)
hom_mat3d_rotate (HomMat3DRotYZ, RotX, 'x', 0, 0, 0, HomMat3DXYZ)
```

When passing 'abg' in **OrderOfRotation**, the rotation corresponds to the following chain:

$$\mathbf{R}_{abg} = \mathbf{R}_z(\text{RotZ}) \cdot \mathbf{R}_y(\text{RotY}) \cdot \mathbf{R}_x(\text{RotX})$$

If you pass 'rodriguez' in **OrderOfRotation**, the rotation parameters **RotX**, **RotY**, and **RotZ** are interpreted as the x-, y-, and z-component of the so-called Rodriguez rotation vector. The direction of the vector defines the (arbitrary) axis of rotation. The length of the vector usually defines the rotation angle with positive orientation. Here, a variation of the Rodriguez vector is used, where the length of the vector defines the tangent of half the rotation angle:

$$\mathbf{R}_{rodriguez} = \text{rotate around } \begin{pmatrix} \text{RotX} \\ \text{RotY} \\ \text{RotZ} \end{pmatrix} \text{ by } 2 \cdot \arctan(\sqrt{\text{RotX}^2 + \text{RotY}^2 + \text{RotZ}^2})$$

Corresponding homogeneous transformation matrix

You can obtain the homogeneous transformation matrix corresponding to a pose with the operator [pose_to_hom_mat3d](#). In the standard definition, this is the following homogeneous transformation matrix, which can be split into two separate matrices, one for the translation ($\mathbf{H}(\mathbf{t})$) and one for the rotation ($\mathbf{H}(\mathbf{R})$):

$$\begin{aligned}\mathbf{H}_{pose} &= \begin{bmatrix} \mathbf{R} & \mathbf{t} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} \mathbf{R}(\text{RotX, RotY, RotZ}) & \begin{matrix} \text{TransX} \\ \text{TransY} \\ \text{TransZ} \end{matrix} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \\ &= \begin{bmatrix} 1 & 0 & 0 & \text{TransX} \\ 0 & 1 & 0 & \text{TransY} \\ 0 & 0 & 1 & \text{TransZ} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R}(\text{RotX, RotY, RotZ}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \mathbf{H}(\mathbf{t}) \cdot \mathbf{H}(\mathbf{R})\end{aligned}$$

Transformation of coordinates

The following equation describes how a point can be transformed from coordinate system 1 into coordinate system 2 with a pose, or more exactly, with the corresponding homogeneous transformation matrix ${}^2\mathbf{H}_1$ (input and output points as homogeneous vectors, see also [affine_trans_point_3d](#)). Note that to transform points from coordinate system 1 into system 2, you use the transformation matrix that describes the pose of coordinate system 1 relative to system 2.

$$\begin{pmatrix} \mathbf{p}^2 \\ 1 \end{pmatrix} = {}^2\mathbf{H}_1 \cdot \begin{pmatrix} \mathbf{p}^1 \\ 1 \end{pmatrix} = \begin{pmatrix} \mathbf{R}(\text{RotX, RotY, RotZ}) \cdot \mathbf{p}^1 + \begin{pmatrix} \text{TransX} \\ \text{TransY} \\ \text{TransZ} \end{pmatrix} \\ 1 \end{pmatrix}$$

This corresponds to the following operator calls:

```
pose_to_hom_mat3d(PoseOf1In2, HomMat3DFrom1In2)
affine_trans_point_3d(HomMat3DFrom1In2, P1X, P1Y, P1Z, P2X, P2Y, P2Z)
```

Non-standard pose definitions

So far, we described the standard pose definition. To create such poses, you select the (default) values 'Rp+T' for the parameter [OrderOfTransform](#) and 'point' for [ViewOfTransform](#). By specifying other values for these parameters, you can create non-standard poses types, which we describe briefly below. Please note that these representation types are only supported for backwards compatibility; we strongly recommend to use the standard types.

If you select 'R(p-T)' for [OrderOfTransform](#), the created pose corresponds to the following chain of transformations, i.e., the sequence of rotation and translation is reversed and the translation is negated:

$$\mathbf{H}_{R(p-T)} = \begin{bmatrix} \mathbf{R}(\text{RotX, RotY, RotZ}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 & -\text{TransX} \\ 0 & 1 & 0 & -\text{TransY} \\ 0 & 0 & 1 & -\text{TransZ} \\ 0 & 0 & 0 & 1 \end{bmatrix} = \mathbf{H}(\mathbf{R}) \cdot \mathbf{H}(-\mathbf{t})$$

If you select 'coordinate_system' for [ViewOfTransform](#), the sequence of transformations remains constant, but the rotation angles are negated. Please note that, contrary to its name, this is not equivalent to transforming a coordinate system!

$$\mathbf{H}_{coordinate\_system} = \begin{bmatrix} 1 & 0 & 0 & \text{TransX} \\ 0 & 1 & 0 & \text{TransY} \\ 0 & 0 & 1 & \text{TransZ} \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \mathbf{R}(-\text{RotX, -RotY, -RotZ}) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Returned data structure

The created 3D pose is returned in **Pose**, which is a tuple of length seven. The first three elements hold the translation parameters **TransX**, **TransY**, and **TransZ**, followed by the rotation parameters **RotX**, **RotY**, and **RotZ**. The last element codes the representation type of the pose, which you selected with the parameters **OrderOfTransform**, **OrderOfRotation**, and **ViewOfTransform**. The following table lists the possible combinations. As already noted, we recommend to use only the representation types with **OrderOfTransform** = 'Rp+T' and **ViewOfTransform** = 'point' (codes 0, 2, and 4).

OrderOfTransform	OrderOfRotation	ViewOfTransform	Code
'Rp+T'	'gba'	'point'	0
'Rp+T'	'abg'	'point'	2
'Rp+T'	'rodriguez'	'point'	4
'Rp+T'	'gba'	'coordinate_system'	1
'Rp+T'	'abg'	'coordinate_system'	3
'Rp+T'	'rodriguez'	'coordinate_system'	5
'R(p-T)'	'gba'	'point'	8
'R(p-T)'	'abg'	'point'	10
'R(p-T)'	'rodriguez'	'point'	12
'R(p-T)'	'gba'	'coordinate_system'	9
'R(p-T)'	'abg'	'coordinate_system'	11
'R(p-T)'	'rodriguez'	'coordinate_system'	13

You can convert poses into other representation types using **convert_pose_type** and query the type using **get_pose_type**.

Parameter	
▷ TransX (input_control)	real $\leadsto$ real
Translation along the x-axis (in [m]).	
Default Value : 0.1	
Suggested values : $\text{TransX} \in \{-1.0, -0.75, -0.5, -0.25, -0.2, -0.1, -0.5, -0.25, -0.125, -0.01, 0, 0.01, 0.125, 0.25, 0.5, 0.1, 0.2, 0.25, 0.5, 0.75, 1.0\}$	
▷ TransY (input_control)	real $\leadsto$ real
Translation along the y-axis (in [m]).	
Default Value : 0.1	
Suggested values : $\text{TransY} \in \{-1.0, -0.75, -0.5, -0.25, -0.2, -0.1, -0.5, -0.25, -0.125, -0.01, 0, 0.01, 0.125, 0.25, 0.5, 0.1, 0.2, 0.25, 0.5, 0.75, 1.0\}$	
▷ TransZ (input_control)	real $\leadsto$ real
Translation along the z-axis (in [m]).	
Default Value : 0.1	
Suggested values : $\text{TransZ} \in \{-1.0, -0.75, -0.5, -0.25, -0.2, -0.1, -0.5, -0.25, -0.125, -0.01, 0, 0.01, 0.125, 0.25, 0.5, 0.1, 0.2, 0.25, 0.5, 0.75, 1.0\}$	
▷ RotX (input_control)	real $\leadsto$ real
Rotation around x-axis or x component of the Rodriguez vector (in [°] or without unit).	
Default Value : 90	
Suggested values : $\text{RotX} \in \{90, 180, 270\}$	
Typical range of values : $0 \leq \text{RotX} \leq 360$	
▷ RotY (input_control)	real $\leadsto$ real
Rotation around y-axis or y component of the Rodriguez vector (in [°] or without unit).	
Default Value : 90	
Suggested values : $\text{RotY} \in \{90, 180, 270\}$	
Typical range of values : $0 \leq \text{RotY} \leq 360$	
▷ RotZ (input_control)	real $\leadsto$ real
Rotation around z-axis or z component of the Rodriguez vector (in [°] or without unit).	
Default Value : 90	
Suggested values : $\text{RotZ} \in \{90, 180, 270\}$	
Typical range of values : $0 \leq \text{RotZ} \leq 360$	

- ▷ **OrderOfTransform** (input_control) string $\leadsto$ string
Order of rotation and translation.
Default Value : "Rp+T"
Suggested values : OrderOfTransform $\in$ {"Rp+T", "R(p-T)"}
- ▷ **OrderOfRotation** (input_control) string $\leadsto$ string
Meaning of the rotation values.
Default Value : "gba"
Suggested values : OrderOfRotation $\in$ {"gba", "abg", "rodriguez"}
- ▷ **ViewOfTransform** (input_control) string $\leadsto$ string
View of transformation.
Default Value : "point"
Suggested values : ViewOfTransform $\in$ {"point", "coordinate_system"}
- ▷ **Pose** (output_control) pose-array $\leadsto$ real / integer
3D pose.
Number of elements : 7

Example

```
* goal: calibration with non-standard calibration object
* read start values for interior camera parameters
read_cam_par('campar.dat', CamParam)
* (read 3D world points [WorldPointsX,WorldPointsY,WorldPointsZ],
*  extract corresponding 2D image points [PixelsRow,PixelsColumn])
* task: create starting value for the exterior camera parameters, i.e., the
*       pose of the calibration object in the calibration images
* first image: calibration object placed at a distance of 0.5 and 0.1
*              'below' the camera coordinate system
*              orientation 'read from left to right': rotated 30 degrees
*              around the optical axis of the camera (z-axis),
*              then tilted 10 degrees around the new y-axis
create_pose(0.1, 0.0, 0.5, 30, 10, 0, 'Rp+T', 'abg', 'point', StartPose1)
* (accumulate all poses in StartPoses = [StartPose1, StartPose2, ...])
* perform the calibration
camera_calibration(WorldPointsX, WorldPointsY, WorldPointsZ,
                  PixelsRow, PixelsColumn, CamParam, StartPoses, 'pose',
                  FinalCamParam, FinalPoses, Errors)
```

Result

[create_pose](#) returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception handling is raised.

Parallelization Information

[create_pose](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[pose_to_hom_mat3d](#), [write_pose](#), [camera_calibration](#), [hand_eye_calibration](#)

Alternatives

[read_pose](#), [hom_mat3d_to_pose](#)

See also

[hom_mat3d_rotate](#), [hom_mat3d_translate](#), [convert_pose_type](#), [get_pose_type](#),
[hom_mat3d_to_pose](#), [pose_to_hom_mat3d](#), [write_pose](#), [read_pose](#)

Module

Camera calibration

disp_caltab (: : WindowHandle, CalTabDescrFile, CamParam, CaltabPose,
ScaleFac :)

Project and visualize the 3D model of the calibration plate in the image.

`disp_caltab` is used to visualize the calibration marks and the connecting lines between the marks of the used calibration plate (`CalTabDescrFile`) in the window specified by `WindowHandle`. For this, the 3D model of the calibration plate is projected into the image plane using the interior (`CamParam`) and exterior camera parameters (`CalTabPose`, i.e., the pose of the calibration plate in camera coordinates). The underlying camera model (pinhole or telecentric camera with radial distortion) is described in `write_cam_par` and `camera_calibration`.

Typically, `disp_caltab` is used to verify the result of the camera calibration (see `camera_calibration`) by superimposing it onto the original image. The current line width can be set by `set_line_width`, the current color can be set by `set_color`.

The parameter `ScaleFac` influences the number of supporting points to approximate the elliptic contours of the calibration marks. You should increase the number of supporting points, if the image part in the output window is displayed with magnification (see `set_part`).

Parameter

- ▷ **WindowHandle** (input_control) window $\leadsto$ integer
Window in which the calibration plate should be visualized.
- ▷ **CalTabDescrFile** (input_control) string $\leadsto$ string
File name of the calibration plate description.
Default Value : 'caltab.descr'
- ▷ **CamParam** (input_control) number-array $\leadsto$ real / integer
Interior camera parameters.
Number of elements : 8
- ▷ **CalTabPose** (input_control) pose-array $\leadsto$ real / integer
Exterior camera parameters (3D pose of the calibration plate in camera coordinates).
Number of elements : 7
- ▷ **ScaleFac** (input_control) real $\leadsto$ real
Scaling factor for the visualization.
Default Value : 1.0
Suggested values : `ScaleFac` $\in$ {0.5, 1.0, 2.0, 3.0}
Recommended Increment : 0.05
Restriction : `0.0 < ScaleFac`

Example

```
* read calibration image
read_image(Image1, 'calib-01')
* find calibration pattern
find_caltab(Image1, Caltab1, 'caltab.descr', 3, 112, 5)
* find calibration marks and start poses
StartCamPar := [Focus, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]
find_marks_and_pose(Image1, Caltab1, 'caltab.descr', StartCamPar,
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord1, CCoord1,
                    StartPose1)
* read 3D positions of calibration marks
caltab_points('caltab.descr', NX, NY, NZ)
* camera calibration
camera_calibration(NX, NY, NZ, RCoord1, CCoord1, StartCamPar,
                  StartPose1, 11, CamParam, FinalPose, Errors)
* visualize calibration result
disp_image(Image1, WindowHandle)
set_color(WindowHandle, 'red')
disp_caltab('caltab.descr', CamParam, FinalPose, 1.0)
```

Result

`disp_caltab` returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception handling is raised.

Parallelization Information

`disp_caltab` is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[camera_calibration](#), [read_cam_par](#), [read_pose](#)

See also

[find_marks_and_pose](#), [camera_calibration](#), [sim_caltab](#), [write_cam_par](#), [read_cam_par](#), [create_pose](#), [write_pose](#), [read_pose](#), [project_3d_point](#), [get_line_of_sight](#)

Module

Camera calibration

find_caltab (Image : Caltab : CalTabDescrFile, SizeGauss, MarkThresh, MinDiamMarks :)*Segment the standard calibration plate region in the image.*

[find_caltab](#) is used to determine the region of a plane calibration plate with circular marks in the input image [Image](#). First the input image is smoothed (see [gauss_image](#)); the size of the used filter mask is given by [SizeGauss](#). Afterwards, a thresholding operator (see [threshold](#)) with minimum gray value [MarkThresh](#) and maximum gray value 255 is applied. Among the extracted connected regions the most convex region with almost correct number of holes (corresponding to the dark marks of the calibration plate) is selected. Holes with a diameter smaller than the expected size of the marks [MinDiamMarks](#) are eliminated to reduce the impact of noise. The number of marks is read from the calibration plate description file [CalTabDescrFile](#). The complete explanation of this file can be found within the description of [create_caltab](#).

Parameter

- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte / uint2
Input image.
- ▷ **Caltab** (output_object) region $\leadsto$ *Hobject*
Output region.
- ▷ **CalTabDescrFile** (input_control) string $\leadsto$ *string*
File name of the calibration plate description.
Default Value : 'caltab.descr'
- ▷ **SizeGauss** (input_control) integer $\leadsto$ *integer*
Filter size of the Gaussian.
Default Value : 3
List of values : SizeGauss $\in$ {0, 3, 5, 7, 9, 11}
- ▷ **MarkThresh** (input_control) integer $\leadsto$ *integer*
Threshold value for mark extraction.
Default Value : 112
List of values : MarkThresh $\in$ {48, 64, 80, 96, 112, 128, 144, 160}
- ▷ **MinDiamMarks** (input_control) integer $\leadsto$ *integer*
Expected minimal diameter of the marks on the calibration plate.
Default Value : 5
List of values : MinDiamMarks $\in$ {3, 5, 9, 15, 30, 50, 70}

Example

```
* read calibration image
read_image(Image, 'calib-01')
* find calibration pattern
find_caltab(Image, Caltab, 'caltab.descr', 3, 112, 5)
```

Result

[find_caltab](#) returns 2 (H_MSG_TRUE) if all parameter values are correct and an image region is found. The behavior in case of empty input (no image given) can be set via [set_system](#) (`::'no_object_result', <Result>:`) and the behavior in case of an empty result region via [set_system](#) (`::'store_empty_region', <true/false>:`). If necessary, an exception handling is raised.

Parallelization Information

[find_caltab](#) is *reentrant* and processed *without* parallelization.

	Possible Predecessors
read_image	
	Possible Successors
find_marks_and_pose	
	See also
find_marks_and_pose , camera_calibration , disp_caltab , sim_caltab , caltab_points , create_caltab	
	Module
Camera calibration	

```
find_marks_and_pose ( Image, CalTabRegion : : CalTabDescrFile,
StartCamParam, StartThresh, DeltaThresh, MinThresh, Alpha, MinContLength,
MaxDiamMarks : RCoord, CCoord, StartPose )
```

Extract the 2D calibration marks from the image and calculate initial values for the exterior camera parameters.

[find_marks_and_pose](#) is used to determine the necessary input data for the subsequent camera calibration (see [camera_calibration](#)): First, the 2D center points [[RCoord](#),[CCoord](#)] of the calibration marks within the region [CalTabRegion](#) of the input image [Image](#) are extracted and ordered. Secondly, a rough estimate for the exterior camera parameters ([StartPose](#)) is computed, i.e., the 3D pose (= position and orientation) of the calibration plate relative to the camera coordinate system (see [create_pose](#) for more information about 3D poses).

In the input image [Image](#) an edge detector is applied (see [edges_image](#), mode 'lanser2') to the region [CalTabRegion](#), which can be found by applying the operator [find_caltab](#). The filter parameter for this edge detection can be tuned via [Alpha](#). In the edge image closed contours are searched for: The number of closed contours must correspond to the number of calibration marks as described in the calibration plate description file [CalTabDescrFile](#) and the contours have to be elliptically shaped. Contours shorter than [MinContLength](#) are discarded, just as contours enclosing regions with a diameter larger than [MaxDiamMarks](#) (e.g., the border of the calibration plate).

For the detection of contours a threshold operator is applied to amplitude of the edge detector. All points with a high amplitude (i.e., borders of marks) are selected.

First, the threshold value is set to [StartThresh](#). If the search for the closed contours or the successive pose estimate fails, this threshold value is successively decreased by [DeltaThresh](#) down to a minimum value of [MinThresh](#).

Each of the found contours is refined with subpixel accuracy (see [edges_sub_pix](#)) and subsequently approximated by an ellipse. The center points of these ellipses represent a good approximation of the desired 2D image coordinates [[RCoord](#),[CCoord](#)] of the calibration mark center points. The order of the values within these two tuples is in row-major order beginning at the upper left in the image. This order must correspond to the order of the 3D coordinates of the calibration marks in the calibration plate description file [CalTabDescrFile](#), since this fixes the correspondences between extracted image marks and known model marks!

Based on the ellipse parameters for each calibration mark, a rough estimate for the exterior camera parameters is finally computed. For this purpose the fixed correspondences between extracted image marks and known model marks are used. The estimate [StartPose](#) describes the pose of the calibration plate in the camera coordinate system as required by the operator [camera_calibration](#).

	Parameter
▷ Image (input_object)	image $\leadsto$ <i>Hobject</i> : byte / uint2 Input image.
▷ CalTabRegion (input_object)	region $\leadsto$ <i>Hobject</i> Region of the calibration plate.
▷ CalTabDescrFile (input_control)	string $\leadsto$ <i>string</i> File name of the calibration plate description. Default Value : 'caltab.descr'
▷ StartCamParam (input_control)	number-array $\leadsto$ <i>real</i> / integer Initial values for the interior camera parameters.

- ▷ **StartThresh** (input_control) number $\leadsto$ integer
Initial threshold value for contour detection.
Default Value : 128
List of values : StartThresh $\in \{80, 96, 112, 128, 144, 160\}$
- ▷ **DeltaThresh** (input_control) number $\leadsto$ integer
Loop value for successive reduction of [StartThresh](#).
Default Value : 10
List of values : DeltaThresh $\in \{6, 8, 10, 12, 14, 16, 18, 20, 22\}$
- ▷ **MinThresh** (input_control) number $\leadsto$ integer
Minimum threshold for contour detection.
Default Value : 18
List of values : MinThresh $\in \{8, 10, 12, 14, 16, 18, 20, 22\}$
- ▷ **Alpha** (input_control) real $\leadsto$ real
Filter parameter for contour detection, see [edges_image](#).
Default Value : 0.9
Suggested values : Alpha $\in \{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1\}$
Typical range of values : $0.2 \leq \text{Alpha} \leq 2.0$
Restriction : Alpha > 0.0
- ▷ **MinContLength** (input_control) real $\leadsto$ real
Minimum length of the contours of the marks.
Default Value : 15.0
Suggested values : MinContLength $\in \{10.0, 15.0, 20.0, 30.0, 40.0, 100.0\}$
Restriction : MinContLength > 0.0
- ▷ **MaxDiamMarks** (input_control) real $\leadsto$ real
Maximum expected diameter of the marks.
Default Value : 100.0
Suggested values : MaxDiamMarks $\in \{50.0, 100.0, 150.0, 200.0, 300.0\}$
Restriction : MaxDiamMarks > 0.0
- ▷ **RCoord** (output_control) real-array $\leadsto$ real
Tuple with row coordinates of the detected marks.
- ▷ **CCoord** (output_control) real-array $\leadsto$ real
Tuple with column coordinates of the detected marks.
- ▷ **StartPose** (output_control) pose-array $\leadsto$ real / integer
Estimation for the exterior camera parameters.
Number of elements : 7

Example

```
* read calibration image
read_image(Image, 'calib-01')
* find calibration pattern
find_caltab(Image, Caltab1, 'caltab.descr', 3, 112, 5)
* find calibration marks and start pose
find_marks_and_pose(Image, Caltab, 'caltab.descr' ,
                    [0.008, 0.0, 0.000011, 0.000011, 384, 288, 768, 576],
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord, CCoord, StartPose)
```

Result

[find_marks_and_pose](#) returns 2 (H_MSG.TRUE) if all parameter values are correct and an estimation for the exterior camera parameters has been determined successfully. If necessary, an exception handling is raised.

Parallelization Information

[find_marks_and_pose](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[find_caltab](#)

Possible Successors

[camera_calibration](#)

See also

[find_caltab](#), [camera_calibration](#), [disp_caltab](#), [sim_caltab](#), [read_cam_par](#), [read_pose](#),

[create_pose](#), [pose_to_hom_mat3d](#), [caltab_points](#), [create_caltab](#), [edges_sub_pix](#),
[edges_image](#)

Module

Camera calibration

gen_image_to_world_plane_map (: Map : CamParam, WorldPose, WidthIn, HeightIn, WidthMapped, HeightMapped, Scale, Interpolation :)

Generate a projection map that describes the mapping between the image plane and a the plane $z=0$ of a world coordinate system.

[gen_image_to_world_plane_map](#) generates a projection map [Map](#), which describes the mapping between the image plane and the plane $z=0$ (plane of measurements) in a world coordinate system. This map can be used to rectify an image with the operator [map_image](#). The rectified image shows neither radial nor perspective distortions; it corresponds to an image acquired by a distortion-free camera that looks perpendicularly onto the plane of measurements. The world coordinate system is chosen by passing its 3D pose relative to the camera coordinate system in [WorldPose](#). In [CamParam](#) you must pass the interior camera parameters (see [write_cam_par](#) for the sequence of the parameters and the underlying camera model).

In many cases [CamParam](#) and [WorldPose](#) are the result of calibrating the camera with the operator [camera_calibration](#). See below for an example.

The size of the images to be mapped can be specified by the parameters [WidthIn](#) and [HeightIn](#). The pixel position of the upper left corner of the output image is determined by the origin of the world coordinate system. The size of the output image can be chosen by the parameters [WidthMapped](#), [HeightMapped](#), and [Scale](#). [WidthMapped](#) and [HeightMapped](#) must be given in pixels.

With the parameter [Scale](#) you can specify the size of a pixel in the transformed image. There are two typical scenarios: First, you can scale the image such that pixel coordinates in the transformed image directly correspond to metric units, e.g., that one pixel corresponds to one micron. This is useful if you want to perform measurements in the transformed image, which will then directly result in metric results. The second scenario is to scale the image such that its content appears in a size similar to the original image. This is useful, e.g., if you want to perform shape-based matching in the transformed image.

[Scale](#) must be specified as the ratio *desired pixel size/original unit*. A pixel size of $1\mu\text{m}$ means that a pixel in the transformed image corresponds to the area $1\mu\text{m} \times 1\mu\text{m}$ in the plane of measurements. The original unit is determined by the coordinates of the calibration object. If the original unit is meters (which is the case if you use the standard calibration plate), you can use the parameter values 'm', 'cm', 'mm', 'microns', or ' μm ' to directly set the unit of pixel coordinates in the transformed image.

The parameter [Interpolation](#) specifies whether bilinear interpolation ('bilinear') should be applied between the pixels in the input image or whether the gray value of the nearest neighboring pixel ('none') should be used.

The mapping function is stored in the output image [Map](#). [Map](#) has the same size as the resulting images after the mapping. If no interpolation is chosen, [Map](#) consists of one image containing one channel, in which for each pixel of the resulting image the linearized coordinate of the pixel of the input image is stored that is the nearest neighbor to the transformed coordinates. If bilinear interpolation is chosen, [Map](#) consists of one image containing five channels. In the first channel for each pixel in the resulting image the linearized coordinates of the pixel in the input image is stored that is in the upper left position relative to the transformed coordinates. The four other channels contain the weights of the four neighboring pixels of the transformed coordinates, which are used for the bilinear interpolation, in the following order:

2	3
4	5

The second channel, for example, contains the weights of the pixels that lie to the upper left relative to the transformed coordinates. If several images have to be mapped using the same camera parameters, [gen_image_to_world_plane_map](#) in combination with [map_image](#) is much more efficient than the operator [image_to_world_plane](#) because the mapping function needs to be computed only once.

Parameter

- ▷ **Map** (output_object) (multichannel-)image $\rightsquigarrow$ *Object* : int4 / uint2
 Image containing the mapping data.

- ▷ **CamParam** (input_control) number-array $\leadsto$ *real* / integer
Interior camera parameters.
Number of elements : 8
- ▷ **WorldPose** (input_control) pose-array $\leadsto$ *real* / integer
3D pose of the world coordinate system in camera coordinates.
Number of elements : 7
- ▷ **WidthIn** (input_control) extent.x $\leadsto$ *integer*
Width of the images to be transformed.
- ▷ **HeightIn** (input_control) extent.y $\leadsto$ *integer*
Height of the images to be transformed.
- ▷ **WidthMapped** (input_control) extent.x $\leadsto$ *integer*
Width of the resulting mapped images in pixels.
- ▷ **HeightMapped** (input_control) extent.y $\leadsto$ *integer*
Height of the resulting mapped images in pixels.
- ▷ **Scale** (input_control) number $\leadsto$ *string* / integer / real
Scale or unit.
Default Value : 'm'
Suggested values : Scale $\in$ {'m', 'cm', 'mm', 'microns', ' μ m', 1.0, 0.01, 0.001, '1.0e-6', 0.0254, 0.3048, 0.9144}
- ▷ **Interpolation** (input_control) string $\leadsto$ *string*
Type of interpolation.
Default Value : 'bilinear'
List of values : Interpolation $\in$ {'none', 'bilinear'}

Example

```
* perform camera calibration (with standard calibration plate)
camera_calibration(NX, NY, NZ, NRow, NCol, StartCamParam, NStartPose, 'all',
                  FinalCamParam, NFinalPose, Errors)
* world coordinate system is defined by calibration plate in first image
FinalPose1 := NFinalPose[0:6]
* compensate thickness of plate
set_origin_pose(FinalPose1, 0, 0, 0.000073, WorldPose)
* goal: rectify images
* first determine parameters such that the entire image content is visible
* -> transform image boundary into world plane, determine smallest
*     rectangle around it
get_image_pointer1(Image, Pointer, Type, Width, Height)
gen_rectangle1 (ImageRect, 0, 0, Height-1, Width-1)
gen_contour_region_xld (ImageRect, ImageBorder, 'border')
contour_to_world_plane_xld(ImageBorder, ImageBorderWCS, FinalCamParam,
                          WorldPose, 1)
smallest_rectangle1_xld (ImageBorderWCS, MinY, MinX, MaxY, MaxX)
* -> move the pose to the upper left corner of the surrounding rectangle
set_origin_pose(WorldPose, MinX, MinY, 0, PoseForEntireImage)
* -> determine the scaling factor such that the center pixel has the same
*     size in the original and in the rectified image
*     method: transform corner points of the pixel into the world
*             coordinate system, compute their distances, and use their
*             mean as the scaling factor
image_points_to_world_plane(FinalCamParam, PoseForEntireImage,
                           [Height/2, Height/2, Height/2+1],
                           [Width/2, Width/2+1, Width/2],
                           1, WorldPixelX, WorldPixelY)
distance_pp(WorldPixelY[0], WorldPixelX[0], WorldPixelY[1], WorldPixelX[1],
            WorldLength1)
distance_pp(WorldPixelY[0], WorldPixelX[0], WorldPixelY[2], WorldPixelX[2],
            WorldLength2)
ScaleForSimilarPixelSize := (WorldLength1+WorldLength2)/2
```

```

* -> determine output image size such that entire input image fits into it
ExtentX := MaxX-MinX
ExtentY := MaxY-MinY
WidthRectifiedImage := ExtentX/ScaleForSimilarPixelSize
HeightRectifiedImage := ExtentY/ScaleForSimilarPixelSize
* create mapping with the determined parameters
gen_image_to_world_plane_map(Map, FinalCamParam, PoseForEntireImage,
                             Width, Height,
                             WidthRectifiedImage, HeightRectifiedImage,
                             ScaleForSimilarPixelSize, 'bilinear')
* transform grabbed images with the created map
while(1)
    grab_image_async(Image, FGHandle, -1)
    map_image(Image, Map, RectifiedImage)
endwhile

```

Result

[gen_image_to_world_plane_map](#) returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception handling is raised.

Parallelization Information

[gen_image_to_world_plane_map](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[create_pose](#), [hom_mat3d_to_pose](#), [camera_calibration](#), [hand_eye_calibration](#), [set_origin_pose](#)

Possible Successors

[map_image](#)

Alternatives

[image_to_world_plane](#)

See also

[map_image](#), [contour_to_world_plane_xld](#), [image_points_to_world_plane](#)

Module

Camera calibration

gen_radial_distortion_map (: Map : CamParIn, CamParOut, Interpolation :)

Generate a projection map that describes the mapping of images corresponding to a changing radial distortion.

[gen_radial_distortion_map](#) computes the mapping of images corresponding to a changing radial distortion in accordance to the interior camera parameters [CamParIn](#) and [CamParOut](#), which can be obtained, e.g., using the operator [camera_calibration](#). [CamParIn](#) and [CamParOut](#) contain the old and the new camera parameters including the old and the new radial distortion, respectively (also see [write_cam_par](#) for the sequence of the parameters and the underlying camera model). Each pixel of the potential output image is transformed into the image plane using [CamParOut](#) and subsequently projected into a subpixel position of the potential input image using [CamParIn](#).

The mapping function is stored in the output image [Map](#). The size of [Map](#) is given by the camera parameters and therefore defines the size of the images to be mapped using [map_image](#). If no interpolation is chosen ([Interpolation](#) = 'none'), [Map](#) consists of one image containing one channel, in which for each pixel of the output image the linearized coordinate of the pixel of the input image is stored that is the nearest neighbor to the transformed coordinates. If bilinear interpolation is chosen ([Interpolation](#) = 'bilinear'), [Map](#) consists of one image containing five channels. In the first channel for each pixel in the resulting image the linearized coordinate of the pixel in the input image is stored that is in the upper left position relative to the transformed coordinates. The four other channels contain the weights of the four neighboring pixels of the transformed coordinates, which are used for the bilinear interpolation, in the following order:

2	3
4	5

The second channel, for example, contains the weights of the pixels that lie to the upper left relative to the transformed coordinates.

If `CamParOut` was computed via `change_radial_distortion_cam_par`, the mapping describes the effect of a lens with a modified radial distortion. If κ is 0, the mapping corresponds to a rectification.

If several images have to be mapped using the same camera parameters, `gen_radial_distortion_map` in combination with `map_image` is much more efficient than the operator `change_radial_distortion_image` because the transformation must be computed only once.

Parameter	
▷ Map (output_object)	(multichannel-)image $\leadsto$ <i>Hobject</i> : int4 / uint2 Image containing the mapping data.
▷ CamParIn (input_control)	number-array $\leadsto$ <i>real</i> / integer Old camera parameters. Number of elements : 8
▷ CamParOut (input_control)	number-array $\leadsto$ <i>real</i> / integer New camera parameters. Number of elements : 8
▷ Interpolation (input_control)	string $\leadsto$ <i>string</i> Type of interpolation. Default Value : 'bilinear' List of values : Interpolation $\in$ {'none', 'bilinear'}
Result	
<code>gen_radial_distortion_map</code> returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception handling is raised.	
Parallelization Information	
<code>gen_radial_distortion_map</code> is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Predecessors	
<code>change_radial_distortion_cam_par</code> , <code>camera_calibration</code> , <code>hand_eye_calibration</code>	
Possible Successors	
<code>map_image</code>	
Alternatives	
<code>change_radial_distortion_image</code>	
See also	
<code>change_radial_distortion_contours_xld</code>	
Module	
Camera calibration	

get_line_of_sight (: : Row, Column, CamParam : PX, PY, PZ, QX, QY, QZ)

Compute the line of sight corresponding to a point in the image.

`get_line_of_sight` computes the line of sight corresponding to a pixel (Row, Column) in the image. The line of sight is a (straight) line in the camera coordinate system, which is described by two points (PX,PY,PZ) and (QX,QY,QZ) on the line. A pinhole or telecentric camera model with radial distortions described by the interior camera parameters `CamParam` is used (see `write_cam_par`). If a pinhole camera is used, the second point lies on the focal plane, i.e., the output parameter `QZ` is equivalent to the focal length of the camera. The equation of the line of sight is given by

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \begin{pmatrix} PX \\ PY \\ PZ \end{pmatrix} + l \cdot \begin{pmatrix} QX - PX \\ QY - PY \\ QZ - PZ \end{pmatrix}$$

The advantage of representing the line of sight as two points is that it is easier to transform the line in 3D. To do so, all that is necessary is to apply the operator `affine_trans_point_3d` to the two points.

Parameter

- ▷ **Row** (input_control) real-array $\leadsto$ real
Row coordinate of the pixel.
- ▷ **Column** (input_control) real-array $\leadsto$ real
Column coordinate of the pixel.
- ▷ **CamParam** (input_control) number-array $\leadsto$ real / integer
Interior camera parameters.
Number of elements : 8
- ▷ **PX** (output_control) real-array $\leadsto$ real
X coordinate of the first point on the line of sight in the camera coordinate system
- ▷ **PY** (output_control) real-array $\leadsto$ real
Y coordinate of the first point on the line of sight in the camera coordinate system
- ▷ **PZ** (output_control) real-array $\leadsto$ real
Z coordinate of the first point on the line of sight in the camera coordinate system
- ▷ **QX** (output_control) real-array $\leadsto$ real
X coordinate of the second point on the line of sight in the camera coordinate system
- ▷ **QY** (output_control) real-array $\leadsto$ real
Y coordinate of the second point on the line of sight in the camera coordinate system
- ▷ **QZ** (output_control) real-array $\leadsto$ real
Z coordinate of the second point on the line of sight in the camera coordinate system

Example

```
* get interior camera parameters
read_cam_par('campar.dat', CamParam)
* inverse projection
get_line_of_sight([50, 100], [100, 200], CamParam, PX, PY, PZ, QX, QY, QZ)
```

Result

[get_line_of_sight](#) returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception handling is raised.

Parallelization Information

[get_line_of_sight](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[read_cam_par](#), [camera_calibration](#)

Possible Successors

[affine_trans_point_3d](#)

See also

[camera_calibration](#), [disp_caltab](#), [read_cam_par](#), [project_3d_point](#), [affine_trans_point_3d](#)

Module

Camera calibration

get_pose_type (:: Pose : OrderOfTransform, OrderOfRotation, ViewOfTransform)

Get the representation type of a 3D pose.

With [get_pose_type](#), the representation type of the 3D pose [Pose](#) can be queried. See [create_pose](#) for details about 3D poses, their representation types, and the meaning of the parameters [OrderOfTransform](#), [OrderOfRotation](#), and [ViewOfTransform](#).

Parameter	
▷ Pose (input_control)	pose-array $\leadsto$ real / integer 3D pose. Number of elements : 7
▷ OrderOfTransform (output_control)	string $\leadsto$ string Order of rotation and translation.
▷ OrderOfRotation (output_control)	string $\leadsto$ string Meaning of the rotation values.
▷ ViewOfTransform (output_control)	string $\leadsto$ string View of transformation.
Result	
create_pose returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception handling is raised.	
Parallelization Information	
get_pose_type is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Predecessors	
create_pose , hom_mat3d_to_pose , camera_calibration , hand_eye_calibration	
Possible Successors	
convert_pose_type	
See also	
create_pose , convert_pose_type , write_pose , read_pose	
Module	

Camera calibration

hand_eye_calibration (: : NX, NY, NZ, NRow, NCol, MPointsOfImage, MRelPoses, BaseStartPose, CamStartPose, CamParam, ToEstimate, StopCriterion, MaxIterations, MinError : BaseFinalPose, CamFinalPose, NumErrors)

Perform a hand-eye calibration.

The operator **hand_eye_calibration** determines the 3D pose of a robot (“hand”) relative to a camera (“eye”). With this information, the results of image processing can be transformed into the coordinate system of the robot, which can then, e.g., grasp an inspected part. There are two possible configurations of robot-camera (hand-eye) systems: The camera can be mounted on the robot or be stationary and observe the robot. Note that the term robot is used in place for a mechanism that moves objects. Thus, you can use **hand_eye_calibration** to calibrate many different systems, from pan-tilt heads to multi-axis manipulators.

A hand-eye calibration is performed similarly to the calibration of the external camera parameters (see **camera_calibration**): You acquire a set of images of a calibration object, determine correspondences between known model points and their projection in the images and pass them to **hand_eye_calibration** via the parameters **NX**, **NY**, **NZ**, **NRow**, **NCol**, and **MPointsOfImage**. If you use the standard calibration plate, the correspondences can be determined very easily with the operators **find_caltab** and **find_marks_and_pose**. Furthermore, you must specify the internal camera parameters in **CamParam**.

In contrast to the camera calibration, the calibration object is not moved manually. This task is delegated to the robot, which either moves the camera (mounted camera) or the calibration object (stationary camera). The robot’s movements are assumed to be known and therefore are also used as an input for the calibration (parameter **MRelPoses**).

The two hand-eye configurations are discussed in more detail below, followed by general information about the process of hand-eye calibration.

Moving camera (mounted on a robot)

In this configuration, the calibration object remains stationary and the camera is moved to different positions by the robot. The main idea behind the hand-eye calibration is that the information extracted from a calibration image, i.e., the pose of the calibration object relative to the camera (i.e., the external camera parameters), can be seen as a

chain of poses or homogeneous transformation matrices, from the calibration object via the base of the robot to its tool (end-effector) and finally to the camera:

Moving camera:

$${}^{cam}\mathbf{H}_{cal} = {}^{cam}\mathbf{H}_{tool} \cdot {}^{tool}\mathbf{H}_{base} \cdot {}^{base}\mathbf{H}_{cal}$$

${}^{cam}\mathbf{H}_{tool}$
 CamStartPose
CamFinalPose

${}^{tool}\mathbf{H}_{base}$
 MRelPoses

${}^{base}\mathbf{H}_{cal}$
 BaseStartPose
BaseFinalPose

From the set of calibration images, the operator `hand_eye_calibration` determines the two transformations at the ends of the chain, i.e., the pose of the robot tool in camera coordinates (${}^{cam}\mathbf{H}_{tool}$) and the pose of the calibration object in the robot base coordinate system (${}^{base}\mathbf{H}_{cal}$). In the input parameters `CamStartPose` and `BaseStartPose`, you must specify suitable starting values for these transformations, which are constant over all calibration images. `hand_eye_calibration` then returns the calibrated values in `CamFinalPose` and `BaseFinalPose`.

In contrast, the transformation in the middle of the chain, ${}^{tool}\mathbf{H}_{base}$, is known but changes for each calibration image, because it describes the pose of the robot moving the camera, or to be more exact its inverse pose (pose of the base coordinate system in robot tool coordinates). You must specify the (inverse) robot poses in the calibration images in the parameter `MRelPoses`.

Internally, `hand_eye_calibration` uses a Newton-type algorithm to minimize an error function based on normal equations. Analogously to the calibration of the camera itself (see `camera_calibration`), the hand-eye calibration becomes more robust if you use many calibration images that were acquired with different robot poses.

Stationary camera

In this configuration, the robot grasps the calibration object and moves it in front of the camera. Again, the information extracted from a calibration image, i.e., the pose of the calibration object in camera coordinates (i.e., the external camera parameters), are equal to a chain of poses or homogeneous transformation matrices, this time from the calibration object via the robot's tool to its base and finally to the camera:

Stationary camera:

$${}^{cam}\mathbf{H}_{cal} = {}^{cam}\mathbf{H}_{base} \cdot {}^{base}\mathbf{H}_{tool} \cdot {}^{tool}\mathbf{H}_{cal}$$

${}^{cam}\mathbf{H}_{base}$
 CamStartPose
CamFinalPose

${}^{base}\mathbf{H}_{tool}$
 MRelPoses

${}^{tool}\mathbf{H}_{cal}$
 BaseStartPose
BaseFinalPose

Analogously to the configuration with a moving camera, the operator `hand_eye_calibration` determines the two transformations at the ends of the chain, here the pose of the robot base coordinate system in camera coordinates (${}^{cam}\mathbf{H}_{base}$) and the pose of the calibration object relative to the robot tool (${}^{tool}\mathbf{H}_{cal}$). In the input parameters `CamStartPose` and `BaseStartPose`, you must specify suitable starting values for these transformations. `hand_eye_calibration` then returns the calibrated values in `CamFinalPose` and `BaseFinalPose`. Please note that the names of the parameters `BaseStartPose` and `BaseFinalPose` are misleading for this configuration!

The transformation in the middle of the chain, ${}^{base}\mathbf{H}_{tool}$, describes the pose of the robot moving the calibration object, i.e., the pose of the tool relative to the base coordinate system. You must specify the robot poses in the calibration images in the parameter `MRelPoses`.

Additional information about the calibration process

The following sections discuss individual questions arising from the use of `hand_eye_calibration`. They are intended to be a guideline for using the operator in an application, as well as to help understanding the operator.

How do I get 3D model points and their projections? 3D model points given in the world coordinate system (`NX`, `NY`, `NZ`) and their associated projections in the image (`NRow`, `NCol`) form the basis of the hand-eye calibration. In order to be able to perform a successful hand-eye calibration, you need images of the 3D model points that were obtained for sufficiently many different poses of the manipulator.

In principle, you can use arbitrary known points for the calibration. However, it is usually most convenient to use the standard calibration plate, e.g., the one that can be generated with `create_caltab`. In this case, you can use the operators `find_caltab` and `find_marks_and_pose` to extract the position of the calibration plate and of the calibration marks and the operator `caltab_points` to access the 3D coordinates of the calibration marks (see also the description of `camera_calibration`).

The parameter `MPointsOfImage` specifies the number of 3D model points used for each pose of the manipulator, i.e., for each image. With this, the 3D model points, which are stored in a linearized fashion in `NX`, `NY`, `NZ`, and their corresponding projections (`NRow`, `NCol`) can be associated with the corresponding pose of the manipulator (`MRelPoses`). Note that in contrast to the operator `camera_calibration` the 3D coordinates of the model points must be specified for each calibration image, not only once.

How do I acquire a suitable set of images? If a standard calibration plate is used, the following procedure should be used:

- At least 10 to 20 images from different positions should be taken in which the position of the camera with respect to the calibration plate is sufficiently different. The position of the calibration plate (moving camera: relative to the robot's tool; stationary camera: relative to the robot's base) must not be changed between images.
- In each image, the calibration plate must be completely visible (including its border).
- No reflections or other disturbances should be visible on the calibration plate.
- The set of images must show the calibration plate from very different positions of the manipulator. The calibration plate can and should be visible in different parts of the images. Furthermore, it should be slightly to moderately rotated around its x- or y-axis, in order to clearly exhibit distortions of the calibration marks. In other words, the corresponding exterior camera parameters (pose of the calibration plate in camera coordinates) should take on many different values.
- In each image, the calibration plate should fill at least one quarter of the entire image, in order to ensure the robust detection of the calibration marks.
- The interior camera parameters of the camera to be used must have been determined earlier and must be passed in `CamParam` (see `camera_calibration`). Note that changes of the image size, the focal length, the aperture, or the focus effect a change of the interior camera parameters.
- The camera must not be modified between the acquisition of the individual images, i.e., focal length, aperture, and focus must not be changed, because all calibration images use the same interior camera parameters. Please make sure that the focus is sufficient for the expected changes of the distance the camera from the calibration plate. Therefore, bright lighting conditions for the calibration plate are important, because then you can use smaller apertures, which result in larger depth of focus.

How do I obtain suitable starting values? Depending on the used hand-eye configuration, you need starting values for the following poses:

Moving camera

`BaseStartPose` = pose of the calibration object in robot base coordinates
`CamStartPose` = pose of the robot tool in camera coordinates

Stationary camera

`BaseStartPose` = pose of the calibration object in robot tool coordinates
`CamStartPose` = pose of the robot base in camera coordinates

The camera's coordinate system is oriented such that its optical axis corresponds to the z-axis, the x-axis points to the right, and the y-axis downwards. The coordinate system of the standard calibration plate is located in the middle of the surface of the calibration plate, its z-axis points into the calibration plate, its x-axis to the right, and its y-axis downwards.

For more information about creating a 3D pose please refer to the description of `create_pose`, which also contains a short example.

In fact, you need a starting value only for one of the two poses (`BaseStartPose` or `CamStartPose`). The other can be computed from one of the calibration images. This means that you can pick the pose that is easier to determine and let HALCON compute the other one for you.

The main idea is to exploit the fact that the two poses for which we need starting values are connected via the already described chain of transformations, here shown for a configuration with a moving camera:

Moving camera:

$${}^{cam}\mathbf{H}_{cal} = {}^{cam}\mathbf{H}_{tool} \cdot {}^{tool}\mathbf{H}_{base} \cdot {}^{base}\mathbf{H}_{cal}$$

`CamStartPose` `MRelPoses` `BaseStartPose`

In this configuration, it is typically easy to determine a starting value for ${}^{cam}\mathbf{H}_{tool}$ (**CamStartPose**). Thus, we solve the equation for ${}^{base}\mathbf{H}_{cal}$ (**BaseStartPose**):

$$\begin{aligned} \text{Moving camera:} \quad {}^{base}\mathbf{H}_{cal} &= ({}^{cam}\mathbf{H}_{tool} \cdot {}^{tool}\mathbf{H}_{base})^{-1} \cdot {}^{cam}\mathbf{H}_{cal} \\ &= {}^{base}\mathbf{H}_{tool} \cdot {}^{tool}\mathbf{H}_{cam} \cdot {}^{cam}\mathbf{H}_{cal} \end{aligned}$$

Thus, to compute **BaseStartPose** you need one of the robot poses (e.g., the one in the first image), your estimate for **CamStartPose**, and the pose of the calibration object in camera coordinates in the selected image. If you use the standard calibration plate, you typically already obtained its pose when applying the operator **find_marks_and_pose** to determine the projections of the marks. An example program can be found below.

For a configuration with a stationary camera, the chain of transformations is:

$$\text{Stationary camera:} \quad {}^{cam}\mathbf{H}_{cal} = {}^{cam}\mathbf{H}_{base} \cdot {}^{base}\mathbf{H}_{tool} \cdot {}^{tool}\mathbf{H}_{cal}$$

CamStartPose MRelPoses BaseStartPose

In this configuration, it is typically easier to determine a starting value for ${}^{tool}\mathbf{H}_{cal}$ (**BaseStartPose**). Thus, we solve the equation for ${}^{cam}\mathbf{H}_{base}$ (**CamStartPose**):

$$\begin{aligned} \text{Stationary camera:} \quad {}^{cam}\mathbf{H}_{base} &= {}^{cam}\mathbf{H}_{cal} \cdot ({}^{base}\mathbf{H}_{tool} \cdot {}^{tool}\mathbf{H}_{cal})^{-1} \\ &= {}^{cam}\mathbf{H}_{cal} \cdot {}^{cal}\mathbf{H}_{tool} \cdot {}^{tool}\mathbf{H}_{base} \end{aligned}$$

Thus, to compute **CamStartPose** you need one of the robot poses (e.g., the one in the first image), your estimate for **BaseStartPose**, and the pose of the calibration object in camera coordinates in the selected image. If you use the standard calibration plate, you typically already obtained its pose when applying the operator **find_marks_and_pose** to determine the projections of the marks. An example program can be found below.

How do I obtain the poses of the robot? In the parameter **MRelPoses** you must pass the poses of the robot in the calibration images (moving camera: pose of the robot base in robot tool coordinates; stationary camera: pose of the robot tool in robot base coordinates) in a linearized fashion. We recommend to create the robot poses in a separate program and save in files using **write_pose**. In the calibration program you can then read and accumulate them in a tuple as shown in the example program below. Besides, we recommend to save the pose of the robot tool in robot base coordinates independent of the hand-eye configuration. When using a moving camera, you then invert the read poses before accumulating them. This is also shown in the example program.

Via the cartesian interface of the robot, you can typically obtain the pose of the tool in base coordinates in a notation that corresponds to the pose representations with the codes 0 or 2 (**OrderOfRotation** = 'gba' or 'abg', see **create_pose**). In this case, you can directly use the pose values obtained from the robot as input for **create_pose**.

If the cartesian interface of your robot describes the orientation in a different way, e.g., with the representation ZYZ ($\mathbf{R}_z(\varphi_1) \cdot \mathbf{R}_y(\varphi_2) \cdot \mathbf{R}_z(\varphi_3)$), you can create the corresponding homogeneous transformation matrix step by step using the operators **hom_mat3d_rotate** and **hom_mat3d_translate** and then convert the matrix into a pose using **hom_mat3d_to_pose**. The following example code creates a pose from the ZYZ representation described above:

```
hom_mat3d_identity (HomMat3DIdent)
hom_mat3d_rotate (HomMat3DIdent,  $\varphi_3$ , 'z', 0, 0, 0, HomMat3DRotZ)
hom_mat3d_rotate (HomMat3DRotZ,  $\varphi_2$ , 'y', 0, 0, 0, HomMat3DRotYZ)
hom_mat3d_rotate (HomMat3DRotYZ,  $\varphi_1$ , 'z', 0, 0, 0, HomMat3DRotZYZ)
hom_mat3d_translate (HomMat3DRotZYZ, Tx, Ty, Tz, base_H_tool)
hom_mat3d_to_pose (base_H_tool, RobPose)
```

Please note that the hand-eye calibration only works if the robot poses **MRelPoses** are specified with high accuracy!

How can I exclude individual pose parameters from the estimation? `hand_eye_calibration` estimates a maximum of 12 pose parameters, i.e., 6 parameters each for the two computed poses `CamFinalPose` and `BaseFinalPose`. However, it is possible to exclude some of these pose parameters from the estimation. This means that the starting values of the poses remain unchanged and are assumed constant for the estimation of all other pose parameters. The parameter `ToEstimate` is used to determine which pose parameters should be estimated. In `ToEstimate`, a list of keywords for the parameters to be estimated is passed. The possible values are:

`BaseFinalPose`:

```
'baseTx' = translation along the x-axis
'baseTy' = translation along the y-axis
'baseTz' = translation along the z-axis
'baseRa' = rotation around the x-axis
'baseRb' = rotation around the y-axis
'baseRg' = rotation around the z-axis
```

`CamFinalPose`:

```
'camTx' = translation along the x-axis
'camTy' = translation along the y-axis
'camTz' = translation along the z-axis
'camRa' = rotation around the x-axis
'camRb' = rotation around the y-axis
'camRg' = rotation around the z-axis
```

In order to estimate all 12 pose parameters, you can pass the keyword `'all'` (or of course a tuple containing all 12 keywords listed above).

It is useful to exclude individual parameters from the estimation if some of the pose parameters have already been measured exactly.

Which terminating criteria can be used for the error minimization? The error minimization terminates either after a fixed number of iterations or if the error falls below a given minimum error. The parameter `StopCriterion` is used to choose between these two alternatives. If `'CountIterations'` is passed, the algorithm terminates after `MaxIterations` iterations.

If `StopCriterion` is passed as `'MinError'`, the algorithm runs until the error falls below the error threshold given in `MinError`. If, however, the number of iterations reaches the number given in `MaxIterations`, the algorithm terminates with an error message.

What is the order of the individual parameters? The length of the tuple `MPointsOfImage` corresponds to the number of different positions of the manipulator and thus to the number of calibration images. The parameter `MPointsOfImage` determines the number of model points used in the individual positions. If the standard calibration plate is used, this means 49 points per position (image). If for example 15 images were acquired, `MPointsOfImage` is a tuple of length 15, where all elements of the tuple have the value 49. The number of calibration images, which is determined by the length of `MPointsOfImage`, must also be taken into account for the tuples for the 3D model points and for the extracted 2D marks, respectively. Hence, for 15 calibration images with 49 model points each, the tuples `NX`, `NY`, `NZ`, `NRow`, and `NCol` must contain $15 \cdot 49 = 735$ values each. These tuples are ordered according to the image the respective points lie in, i.e., the first 49 values correspond to the 49 model points in the first image. The order of the 3D model points and the extracted 2D model points must be the same in each image.

The length of the tuple `MRelPoses` also depends on the number of calibration images. If, for example, 15 images and therefore 15 poses are used, the length of the tuple `MRelPoses` is $15 \cdot 7 = 105$ (15 times 7 pose parameters). The first seven parameters thus determine the pose of the manipulator in the first image, and so on.

What do the output parameters mean? If `StopCriterion` was set to `'CountIterations'`, the output parameters `BaseFinalPose` and `CamFinalPose` are returned even if the algorithm didn't converge. If, however, `StopCriterion` was set to `'MinError'`, the error must fall below `'MinError'` in order for output parameters to be returned.

The representation type of `BaseFinalPose` and `CamFinalPose` is the same as in the corresponding starting values. It can be changed with the operator `convert_pose_type`. The description of the different representation types and of their conversion can be found with the documentation of the operator `create_pose`.

The parameter `NumErrors` contains a list of (numerical) errors from the individual iterations of the algorithm. Based on the evolution of the errors, it can be decided whether the algorithm has converged for the

given starting values. The error values are returned as 3D deviations in meters. Thus, the last entry of the error list corresponds to an estimate of the accuracy of the returned pose parameters.

Attention

The quality of the calibration depends on the accuracy of the input parameters (position of the calibration marks, robot poses **MRelPoses**, and the starting positions **BaseStartPose**, **CamStartPose**). Based on the returned error measures **NumErrors**, it can be decided, whether the algorithm has converged. Furthermore, the accuracy of the returned pose can be estimated. The error measures are 3D differences in meters.

Parameter

- ▷ **NX** (input_control) real-array $\leadsto$ *real*
Linear list containing all the x coordinates of the calibration points (in the order of the images).
- ▷ **NY** (input_control) real-array $\leadsto$ *real*
Linear list containing all the y coordinates of the calibration points (in the order of the images).
- ▷ **NZ** (input_control) real-array $\leadsto$ *real*
Linear list containing all the z coordinates of the calibration points (in the order of the images).
- ▷ **NRow** (input_control) real-array $\leadsto$ *real*
Linear list containing all row coordinates of the calibration points (in the order of the images).
- ▷ **NCol** (input_control) real-array $\leadsto$ *real*
Linear list containing all the column coordinates of the calibration points (in the order of the images).
- ▷ **MPointsOfImage** (input_control) integer-array $\leadsto$ *integer*
Number of the calibration points for each image.
- ▷ **MRelPoses** (input_control) pose-array $\leadsto$ *real* / *integer*
Measured 3D pose of the robot for each image (moving camera: robot base in robot tool coordinates;
stationary camera: robot tool in robot base coordinates).
- ▷ **BaseStartPose** (input_control) pose-array $\leadsto$ *real* / *integer*
Starting value for the 3D pose of the calibration object in robot base coordinates (moving camera) or in robot
tool coordinates (stationary camera), respectively.
- ▷ **CamStartPose** (input_control) pose-array $\leadsto$ *real* / *integer*
Starting value for the 3D pose of the robot tool (moving camera) or robot base (stationary camera),
respectively, in camera coordinates.
- ▷ **CamParam** (input_control) number-array $\leadsto$ *real* / *integer*
Interior camera parameters.
- ▷ **ToEstimate** (input_control) string-array $\leadsto$ *string*
Parameters to be estimated (max. 12 degrees of freedom).
Default Value : "all"
List of values : **ToEstimate** $\in$ {"all", "baseTx", "baseTy", "baseTz", "baseRa", "baseRb", "baseRg",
"camTx", "camTy", "camTz", "camRa", "camRb", "camRg"}
- ▷ **StopCriterion** (input_control) string $\leadsto$ *string*
Type of stopping criterion.
Default Value : "CountIterations"
List of values : **StopCriterion** $\in$ {"CountIterations", "MinError"}
- ▷ **MaxIterations** (input_control) integer $\leadsto$ *integer*
Maximum number of iterations to be executed.
Default Value : 15
Suggested values : **MaxIterations** $\in$ {10, 15, 20, 25, 30}
- ▷ **MinError** (input_control) real $\leadsto$ *real*
Minimum error used as the stopping criterion.
Default Value : 0.0005
Suggested values : **MinError** $\in$ {0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05, 0.1}
- ▷ **BaseFinalPose** (output_control) pose-array $\leadsto$ *real* / *integer*
Computed 3D pose for the 3D pose of the calibration object in robot base coordinates (moving camera) or in
robot tool coordinates (stationary camera), respectively.
- ▷ **CamFinalPose** (output_control) pose-array $\leadsto$ *real* / *integer*
Computed 3D pose of the robot tool (moving camera) or robot base (stationary camera), respectively, in
camera coordinates.
- ▷ **NumErrors** (output_control) real(-array) $\leadsto$ *real*
Error measures for each iteration.

Example

```

read_cam_par('campar.dat', CamParam)
CalDescr := 'caltab.descr'
caltab_points(CalDescr, X, Y, Z)
* process all calibration images
for i := 0 to NumImages-1 by 1
    read_image(Image, 'calib_'+i$'02d')
    * find marks on the calibration plate in every image
    find_caltab(Image, CalPlate, CalDescr, 3, 150, 5)
    find_marks_and_pose(Image, CalPlate, CalDescr, CamParam, 128, 10,
                        RCoordTmp, CCoordTmp, StartPose)
    * accumulate 2D and 3D coordinates of the marks
    RCoord := [RCoord, RCoordTmp]
    CCoord := [CCoord, CCoordTmp]
    XCoord := [XCoord, X]
    YCoord := [YCoord, Y]
    ZCoord := [ZCoord, Z]
    NumMarker := [NumMarker, |RCoordTmp|]
    * read pose of the robot tool in robot base coordinates
    read_pose('robpose_'+i$'02d'+'.dat', RobPose)
    * moving camera? invert pose
    if (IsMovingCameraConfig='true')
        pose_to_hom_mat3d(RobPose, base_H_tool)
        hom_mat3d_invert(base_H_tool, tool_H_base)
        hom_mat3d_to_pose(tool_H_base, RobPose)
    endif
    * accumulate robot poses
    MRelPoses := [MRelPoses, RobPose]
    * store the pose of the calibration plate in the first image and the
    * corresponding pose of the robot for later use
    if (i=0)
        cam_P_cal := StartPose
        RelPose0 := RobPose
    endif
endifor
* obtain starting values: read one, compute the other
if (IsMovingCameraConfig='true')
    * mov. camera: read pose of robot tool in camera coordinates
    * compute pose of calibration plate in robot base coordinates
    read_pose('cam_P_tool.dat', CamStartPose)
    * BaseStartPose = inverse(CamStartPose * RelPose0) * cam_P_cal
    pose_to_hom_mat3d(CamStartPose, cam_H_tool)
    pose_to_hom_mat3d(RelPose0, tool_H_base)
    pose_to_hom_mat3d(StartPose, cam_H_cal)
    hom_mat3d_compose(cam_H_tool, tool_H_base, cam_H_base)
    hom_mat3d_invert(cam_H_base, base_H_cam)
    hom_mat3d_compose(base_H_cam, cam_H_cal, base_H_cal)
    hom_mat3d_to_pose(base_H_cal, BaseStartPose)
else
    * stat. camera: read pose of calibration plate in robot tool coordinates
    * compute pose of robot base in camera coordinates
    read_pose('tool_P_cal.dat', BaseStartPose)
    * CamStartPose = cam_P_cal * inverse(RelPose0 * BaseStartPose)
    pose_to_hom_mat3d(BaseStartPose, tool_H_cal)
    pose_to_hom_mat3d(RelPose0, base_H_tool)
    pose_to_hom_mat3d(StartPose, cam_H_cal)
    hom_mat3d_compose(base_H_tool, tool_H_cal, base_H_cal)
    hom_mat3d_invert(base_H_cal, cal_H_base)

```

```

    hom_mat3d_compose(cam_H_cal, cal_H_base, cam_H_base)
    hom_mat3d_to_pose(cam_H_base, CamStartPose)
endif
*
* perform hand-eye calibration
*
hand_eye_calibration(XCoord, YCoord, ZCoord, RCoord, CCoord, NumMarker,
                    MRelPoses, BaseStartPose, CamStartPose, CamParam,
                    "all", "CountIterations", 20, 0.000670,
                    BaseFinalPose, CamFinalPose, NumErrors)
*
* measure some point P in camera coordinates (cam_px, cam_py, cam_pz)
*
* transform point into robot coordinates: base_p = base_H_cam * cam_p
if (IsMovingCameraConfig='true')
    * mov. camera: base_H_cam = base_H_tool * tool_H_cam
    *               base_P_cam = RobPose * inverse(CamFinalPose)
    pose_to_hom_mat3d(CamFinalPose, cam_H_tool)
    hom_mat3d_invert(cam_H_tool, tool_H_cam)
    * obtain current robot pose RobPose from robot
    pose_to_hom_mat3d(RobPose, base_H_tool)
    hom_mat3d_compose(base_H_tool, tool_H_cam, base_H_cam)
else
    * stat. camera: base_P_cam = inverse(CamFinalPose)
    pose_to_hom_mat3d(CamFinalPose, cam_H_base)
    hom_mat3d_invert(cam_H_base, base_H_cam)
endif
affine_trans_point_3d(base_H_cam, cam_px, cam_py, cam_pz,
                     base_px, base_py, base_pz)

```

Result

[hand_eye_calibration](#) returns 2 (H_MSG_TRUE) if all parameter values are correct and the method converges with an error less than the specified minimum error (if [StopCriterion](#) = 'MinError'). If necessary, an exception handling is raised.

Parallelization Information

[hand_eye_calibration](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[find_marks_and_pose](#)

Possible Successors

[write_pose](#), [convert_pose_type](#), [pose_to_hom_mat3d](#), [disp_caltab](#), [sim_caltab](#)

See also

[find_caltab](#), [find_marks_and_pose](#), [disp_caltab](#), [sim_caltab](#), [write_cam_par](#), [read_cam_par](#), [create_pose](#), [convert_pose_type](#), [write_pose](#), [read_pose](#), [pose_to_hom_mat3d](#), [hom_mat3d_to_pose](#), [caltab_points](#), [create_caltab](#)

Module

Camera calibration

hom_mat3d_to_pose (: : HomMat3D : Pose)

Convert a homogeneous transformation matrix into a 3D pose.

[hom_mat3d_to_pose](#) converts a homogeneous transformation matrix into the corresponding 3D pose with type code 0. For details about 3D poses and the corresponding transformation matrices please refer to [create_pose](#).

A typical application of [hom_mat3d_to_pose](#) is that a 3D pose was converted into a homogeneous transformation matrix to further transform it, e.g., with [hom_mat3d_rotate](#) or [hom_mat3d_translate](#), and now must be converted back into a pose to use it as input for operators like [image_points_to_world_plane](#).

Parameter

- ▷ **HomMat3D** (input_control) affine3d-array $\leadsto$ *real*
Homogeneous transformation matrix.
Number of elements : 12
- ▷ **Pose** (output_control) pose-array $\leadsto$ *real* / integer
Equivalent 3D pose.
Number of elements : 7

Example

```
camera_calibration(WorldPointsX, WorldPointsY, WorldPointsZ,
                  PixelsRow, PixelsColumn, CamParam, StartPose, 6,
                  FinalCamParam, FinalPose, Errors)
* transform FinalPose to homogeneous transformation matrix
pose_to_hom_mat3d(FinalPose, cam_H_cal)
* rotate it 90 degree around the y-axis to obtain a world coordinate system
* whose y- and z-axis lie in the plane of the calibration plate while the
* x-axis point 'upwards': cam_H_w = cam_H_cal * RotY(90)
hom_mat3d_identity(HomMat3DIdent)
hom_mat3d_rotate(HomMat3DIdent, deg(90), 'y', 0, 0, 0,
                HomMat3DRotateY)
hom_mat3d_compose(cam_H_cal, HomMat3DRotateY, cam_H_w)
* transform back to pose
hom_mat3d_to_pose(cam_H_w, cam_P_w)
* use pose to transform an image point into the world coordinate system
image_points_to_world_plane(FinalCamParam, cam_P_w, 87, 23.5, 1,
                           w_px, w_py)
```

Result

[hom_mat3d_to_pose](#) returns 2 (H.MSG_TRUE) if all parameter values are correct. If necessary, an exception handling is raised

Parallelization Information

[hom_mat3d_to_pose](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[hom_mat3d_rotate](#), [hom_mat3d_translate](#), [hom_mat3d_invert](#)

Possible Successors

[camera_calibration](#), [write_pose](#), [disp_caltab](#), [sim_caltab](#)

See also

[create_pose](#), [camera_calibration](#), [disp_caltab](#), [sim_caltab](#), [write_pose](#), [read_pose](#),
[pose_to_hom_mat3d](#), [project_3d_point](#), [get_line_of_sight](#), [hom_mat3d_rotate](#),
[hom_mat3d_translate](#), [hom_mat3d_invert](#), [affine_trans_point_3d](#)

Module

Camera calibration

image_points_to_world_plane (: : CamParam, WorldPose, Rows, Cols,
Scale : X, Y)

Transform image points into the plane z=0 of a world coordinate system.

The operator [image_points_to_world_plane](#) transforms image points which are given in [Rows](#) and [Cols](#) into the plane z=0 in a world coordinate system and returns their 3D coordinates in [X](#) and [Y](#). The world coordinate system is chosen by passing its 3D pose relative to the camera coordinate system in [WorldPose](#). In [CamParam](#) you must pass the interior camera parameters (see [write_cam_par](#) for the sequence of the parameters and the underlying camera model).

In many cases [CamParam](#) and [WorldPose](#) are the result of calibrating the camera with the operator [camera_calibration](#). See below for an example.

With the parameter **Scale** you can scale the resulting 3D coordinates. The parameter **Scale** must be specified as the ratio *desired unit/original unit*. The original unit is determined by the coordinates of the calibration object. If the original unit is meters (which is the case if you use the standard calibration plate), you can set the desired unit directly by selecting 'm', 'cm', 'mm' or ' μ m' for the parameter **Scale**.

Internally, the operator first computes the line of sight between the projection center and the image contour points in the camera coordinate system, taking into account the radial distortions. The line of sight is then transformed into the world coordinate system specified in **WorldPose**. By intersecting the plane $z=0$ with the line of sight the 3D coordinates **X** and **Y** are obtained.

Parameter

- ▷ **CamParam** (input_control) number-array $\leadsto$ real / integer
Interior camera parameters.
Number of elements : 8
- ▷ **WorldPose** (input_control) pose-array $\leadsto$ real / integer
3D pose of the world coordinate system in camera coordinates.
Number of elements : 7
- ▷ **Rows** (input_control) coordinates.y-array $\leadsto$ real / integer
Row coordinates of the points to be transformed.
Default Value : 100.0
- ▷ **Cols** (input_control) coordinates.x-array $\leadsto$ real / integer
Column coordinates of the points to be transformed.
Default Value : 100.0
- ▷ **Scale** (input_control) number $\leadsto$ string / integer / real
Scale or dimension
Default Value : 'm'
Suggested values : $\text{Scale} \in \{\text{'m'}, \text{'cm'}, \text{'mm'}, \text{'microns'}, \text{'}\mu\text{m'}, 1.0, 0.01, 0.001, \text{'1.0e-6'}, 0.0254, 0.3048, 0.9144\}$
- ▷ **X** (output_control) coordinates.x-array $\leadsto$ real
X coordinates of the points in the world coordinate system.
- ▷ **Y** (output_control) coordinates.y-array $\leadsto$ real
Y coordinates of the points in the world coordinate system.

Example

```
* perform camera calibration (with standard calibration plate)
camera_calibration(NX, NY, NZ, NRow, NCol, StartCamParam, NStartPose, 'all',
                  FinalCamParam, NFinalPose, Errors)
* world coordinate system is defined by calibration plate in first image
FinalPose1 := NFinalPose[0:6]
* compensate thickness of plate
set_origin_pose(FinalPose1, 0, 0, 0.0006, WorldPose)
* transform image points into world coordinate system (unit mm)
image_points_to_world_plane(FinalCamParam, WorldPose, PointRows, PointCols,
                           'mm', PointXCoord, PointYCoord)
```

Result

image_points_to_world_plane returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception handling is raised.

Parallelization Information

image_points_to_world_plane is *reentrant* and processed *without* parallelization.

Possible Predecessors

create_pose, **hom_mat3d_to_pose**, **camera_calibration**, **hand_eye_calibration**,
set_origin_pose

See also

contour_to_world_plane_xld

Module

Camera calibration

image_to_world_plane (Image : ImageWorld : CamParam, WorldPose, Width, Height, Scale, Interpolation :)

Rectify an image by transforming it into the plane $z=0$ of a world coordinate system.

image_to_world_plane rectifies an image **Image** by transforming it into the plane $z=0$ (plane of measurements) in a world coordinate system. The resulting rectified image **ImageWorld** shows neither radial nor perspective distortions; it corresponds to an image acquired by a distortion-free camera that looks perpendicularly onto the plane of measurements. The world coordinate system is chosen by passing its 3D pose relative to the camera coordinate system in **WorldPose**. In **CamParam** you must pass the interior camera parameters (see **write_cam_par** for the sequence of the parameters and the underlying camera model).

In many cases **CamParam** and **WorldPose** are the result of calibrating the camera with the operator **camera_calibration**. See below for an example.

The pixel position of the upper left corner of the output image **ImageWorld** is determined by the origin of the world coordinate system. The size of the output image **ImageWorld** can be chosen by the parameters **Width**, **Height**, and **Scale**. **Width** and **Height** must be given in pixels.

With the parameter **Scale** you can specify the size of a pixel in the transformed image. There are two typical scenarios: First, you can scale the image such that pixel coordinates in the transformed image directly correspond to metric units, e.g., that one pixel corresponds to one micron. This is useful if you want to perform measurements in the transformed image, which will then directly result in metric results. The second scenario is to scale the image such that its content appears in a size similar to the original image. This is useful, e.g., if you want to perform shape-based matching in the transformed image.

Scale must be specified as the ratio *desired pixel size/original unit*. A pixel size of $1\mu\text{m}$ means that a pixel in the transformed image corresponds to the area $1\mu\text{m} \times 1\mu\text{m}$ in the plane of measurements. The original unit is determined by the coordinates of the calibration object. If the original unit is meters (which is the case if you use the standard calibration plate), you can use the parameter values 'm', 'cm', 'mm', 'microns', or ' μm ' to directly set the unit of pixel coordinates in the transformed image.

The parameter **Interpolation** specifies, whether bilinear interpolation ('bilinear') should be applied between the pixels in the input image or whether the gray value of the nearest neighboring pixel ('none') should be used.

If several images have to be rectified using the same parameters, **gen_image_to_world_plane_map** in combination with **map_image** is much more efficient than the operator **image_to_world_plane** because the mapping function needs to be computed only once.

Parameter

- ▷ **Image** (input_object) (multichannel-)image(-array) $\leadsto$ Hobject : byte / uint2
Input image.
- ▷ **ImageWorld** (output_object) (multichannel-)image(-array) $\leadsto$ Hobject : byte / uint2
Transformed image.
- ▷ **CamParam** (input_control) number-array $\leadsto$ real / integer
Interior camera parameters.
Number of elements : 8
- ▷ **WorldPose** (input_control) pose-array $\leadsto$ real / integer
3D pose of the world coordinate system in camera coordinates.
Number of elements : 7
- ▷ **Width** (input_control) extent.x $\leadsto$ integer
Width of the resulting image in pixels.
- ▷ **Height** (input_control) extent.y $\leadsto$ integer
Height of the resulting image in pixels.
- ▷ **Scale** (input_control) number $\leadsto$ string / integer / real
Scale or unit
Default Value : 'm'
Suggested values : Scale $\in$ {'m', 'cm', 'mm', 'microns', ' μm ', 1.0, 0.01, 0.001, '1.0e-6', 0.0254, 0.3048, 0.9144}
- ▷ **Interpolation** (input_control) string $\leadsto$ string
Type of interpolation.
Default Value : 'bilinear'
List of values : Interpolation $\in$ {'none', 'bilinear'}

Example

```

* perform camera calibration (with standard calibration plate)
camera_calibration(NX, NY, NZ, NRow, NCol, StartCamParam, NStartPose, 'all',
                  FinalCamParam, NFinalPose, Errors)
* world coordinate system is defined by calibration plate in first image
FinalPose1 := NFinalPose[0:6]
* compensate thickness of plate
set_origin_pose(FinalPose1, 0, 0, 0.0006, WorldPose)
* goal: rectify image
* first determine parameters such that the entire image content is visible
* and that objects have a similar size before and after the rectification
* -> transform image boundary into world plane, determine smallest
* rectangle around it
get_image_pointer1(Image, Pointer, Type, Width, Height)
gen_rectangle1 (ImageRect, 0, 0, Height-1, Width-1)
gen_contour_region_xld (ImageRect, ImageBorder, 'border')
contour_to_world_plane_xld(ImageBorder, ImageBorderWCS, FinalCamParam,
                          WorldPose, 1)
smallest_rectangle1_xld (ImageBorderWCS, MinY, MinX, MaxY, MaxX)
* -> move the pose to the upper left corner of the surrounding rectangle
set_origin_pose(WorldPose, MinX, MinY, 0, PoseForEntireImage)
* -> determine the scaling factor such that the center pixel has the same
* size in the original and in the rectified image
* method: transform corner points of the pixel into the world
* coordinate system, compute their distances, and use their
* mean as the scaling factor
image_points_to_world_plane(FinalCamParam, PoseForEntireImage,
                           [Height/2, Height/2, Height/2+1],
                           [Width/2, Width/2+1, Width/2],
                           1, WorldPixelX, WorldPixelY)
distance_pp(WorldPixelY[0], WorldPixelX[0], WorldPixelY[1], WorldPixelX[1],
            WorldLength1)
distance_pp(WorldPixelY[0], WorldPixelX[0], WorldPixelY[2], WorldPixelX[2],
            WorldLength2)
ScaleForSimilarPixelSize := (WorldLength1+WorldLength2)/2
* -> determine output image size such that entire input image fits into it
ExtentX := MaxX-MinX
ExtentY := MaxY-MinY
WidthRectifiedImage := ExtentX/ScaleForSimilarPixelSize
HeightRectifiedImage := ExtentY/ScaleForSimilarPixelSize
* transform the image with the determined parameters
image_to_world_plane(Image, RectifiedImage, FinalCamParam,
                    PoseForEntireImage, WidthRectifiedImage,
                    HeightRectifiedImage, ScaleForSimilarPixelSize,
                    'bilinear')

```

Result

[image_to_world_plane](#) returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception handling is raised.

Parallelization Information

[image_to_world_plane](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[create_pose](#), [hom_mat3d_to_pose](#), [camera_calibration](#), [hand_eye_calibration](#),
[set_origin_pose](#)

Alternatives

[gen_image_to_world_plane_map](#), [map_image](#)

See also

[contour_to_world_plane_xld](#), [image_points_to_world_plane](#)

Module

Camera calibration

pose_to_hom_mat3d (: : Pose : HomMat3D)

Convert a 3D pose into a homogeneous transformation matrix.

[pose_to_hom_mat3d](#) converts a 3D pose [Pose](#), e.g., the exterior camera parameters, into the equivalent homogeneous transformation matrix [HomMat3D](#). For details about 3D poses and the corresponding transformation matrices please refer to [create_pose](#).

A typical application of [pose_to_hom_mat3d](#) is that you want to further transform the pose, e.g., rotate or translate it using [hom_mat3d_rotate](#) or [hom_mat3d_translate](#). In case of the exterior camera parameters, this can be necessary if the calibration plate cannot be placed such that its coordinate system coincides with the desired world coordinate system.

Parameter

- ▷ **Pose** (input_control) pose-array $\leadsto$ real / integer
3D pose.
Number of elements : 7
- ▷ **HomMat3D** (output_control) affine3d-array $\leadsto$ real
Equivalent homogeneous transformation matrix.
Number of elements : 12

Example

```
* read interior camera parameters
read_cam_par('campar.dat', CamParam)
* read exterior camera parameters
read_pose('startpose.dat', StartPose)
* (read 3D world points [WorldPointsX,WorldPointsY,WorldPointsZ],
*  extract corresponding 2D image points [PixelsRow,PixelsColumn])
* calibration of exterior camera parameters:
camera_calibration(WorldPointsX, WorldPointsY, WorldPointsZ,
                  PixelsRow, PixelsColumn, CamParam, StartPose, 'pose',
                  FinalCamParam, FinalPose, Errors)
* transform FinalPose to homogeneous transformation matrix
pose_to_hom_mat3d(FinalPose, cam_H_cal)
* rotate it 90 degree around its y-axis to obtain a world coordinate system
* whose y- and z-axis lie in the plane of the calibration plate while the
* x-axis point 'upwards': cam_H_w = cam_H_cal * RotY(90)
hom_mat3d_identity(HomMat3DIdent)
hom_mat3d_rotate(HomMat3DIdent, deg(90), 'y', 0, 0, 0,
                HomMat3DRotateY)
hom_mat3d_compose(cam_H_cal, HomMat3DRotateY, cam_H_w)
```

Result

[pose_to_hom_mat3d](#) returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception handling is raised

Parallelization Information

[pose_to_hom_mat3d](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[camera_calibration](#), [read_pose](#)

Possible Successors

[affine_trans_point_3d](#), [hom_mat3d_invert](#), [hom_mat3d_translate](#), [hom_mat3d_rotate](#),
[hom_mat3d_to_pose](#)

See also

[create_pose](#), [camera_calibration](#), [write_pose](#), [read_pose](#), [hom_mat3d_to_pose](#), [project_3d_point](#), [get_line_of_sight](#), [hom_mat3d_rotate](#), [hom_mat3d_translate](#), [hom_mat3d_invert](#), [affine_trans_point_3d](#)

Module

Camera calibration

project_3d_point (: : X, Y, Z, CamParam : Row, Column)

Project 3D points into (sub-)pixel image coordinates.

[project_3d_point](#) projects one or more 3D points (with coordinates [X](#), [Y](#), and [Z](#)) into the image plane (in pixels) and returns the result in [Row](#) und [Column](#). The coordinates [X](#), [Y](#), and [Z](#) are given in the camera coordinate system, i.e., they describe the position of the points relative to the camera.

The interior camera parameters [CamParam](#) describe the projection characteristics of the camera (see [write_cam_par](#)).

Parameter

- ▷ **X** (input_control) real-array $\leadsto$ *real*
X coordinates of the 3D points to be projected in the camera coordinate system.
- ▷ **Y** (input_control) real-array $\leadsto$ *real*
Y coordinates of the 3D points to be projected in the camera coordinate system.
- ▷ **Z** (input_control) real-array $\leadsto$ *real*
Z coordinates of the 3D points to be projected in the camera coordinate system.
- ▷ **CamParam** (input_control) number-array $\leadsto$ *real* / integer
Interior camera parameters.
Number of elements : 8
- ▷ **Row** (output_control) real-array $\leadsto$ *real*
Row coordinates of the projected points (in pixels).
Default Value : 'ProjectedRow'
- ▷ **Column** (output_control) real-array $\leadsto$ *real*
Column coordinates of the projected points (in pixels).
Default Value : 'ProjectedCol'

Example

```
* read pose of the world coordinate system in camera coordinates
read_pose(worldpose.dat', WorldPose)
* convert pose into transformation matrix
pose_to_hom_mat3d(WorldPose, HomMat3D)
* transform 3D points from world into the camera coordinate system
affine_trans_point_3d([3.0, 3.2], [4.5, 4.5], [5.8, 6.2], HomMat3D, X, Y, Z)
* read interior camera parameters
read_cam_par('campar.dat', CamParam)
* project 3D points into image
project_3d_point(X, Y, Z, CamParam, Row, Column)
```

Result

[project_3d_point](#) returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception handling is raised.

Parallelization Information

[project_3d_point](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[read_cam_par](#), [affine_trans_point_3d](#)

Possible Successors

[gen_region_points](#), [gen_region_polygon](#), [disp_polygon](#)

See also

[camera_calibration](#), [disp_caltab](#), [read_cam_par](#), [get_line_of_sight](#),
[affine_trans_point_3d](#)

Module

Camera calibration

read_cam_par (: : CamParFile : CamParam)

Read the interior camera parameters from text file.

[read_cam_par](#) is used to read the interior camera parameters [CamParam](#) from a text file with name [CamParFile](#). [CamParam](#) is a tuple that contains the interior camera parameters in the following sequence (see [write_cam_par](#) for a description of the corresponding camera models):

[Focus, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

The format of the text file is a (HALCON-independent) generic parameter description. It allows to group arbitrary sets of parameters hierarchically. The description of a single parameter within a parameter group consists of the following 3 lines:

```
Name : Shortname : Actual value ;
Type : Lower bound (optional) : Upper bound (optional) ;
Description (optional) ;
```

[write_cam_par](#) expects in the file [CamParFile](#) the parameter group Camera:Parameter. This parameter group consists of the 8 parameters Focus, Sx, Sy, Cx, Cy, Kappa (κ), ImageWidth and ImageHeight. Comments are marked by a '#' at the beginning of a line. A suitable file can look like the following:

```
# INTERNAL CAMERA PARAMETERS

ParGroup: Camera: Parameter;
  "Internal camera parameters";

Focus:foc:      0.00806039;
  DOUBLE:0.0;;
  "Focal length of the lens [meter]";

Sx:sx:  1.0629e-05;
  DOUBLE:0.0;;
  "Width of a cell on the chip [meter]";

Sy:sy:  1.1e-05;
  DOUBLE:0.0;;
  "Height of a cell on the chip [meter]";

Cx:cx:  378.236;
  DOUBLE:0.0;;
  "X-coordinate of the image center [pixel]";

Cy:cy:  297.587;
  DOUBLE:0.0;;
  "Y-coordinate of the image center [pixel]";

Kappa:kappa:    -2253.5;
  DOUBLE::;
  "Radial distortion coefficient [1/(meter*meter)]";

ImageWidth:imgw:  768;
  INT:0:2048;
```

```
"Width of the used calibration images [pixel]";
```

```
ImageHeight:imgh: 576;
```

```
INT:0:2048;
```

```
"Height of the used calibration images [pixel]";
```

Parameter

- ▷ **CamParFile** (input_control) string $\leadsto$ string
File name of interior camera parameters.
Default Value : 'campar.dat'
List of values : CamParFile $\in$ {'campar.dat', 'campar.initial', 'campar.final'}
- ▷ **CamParam** (output_control) number-array $\leadsto$ real / integer
Interior camera parameters.
Number of elements : 8

Example

```
* get interior camera parameters:
read_cam_par('campar.dat', CamParam)
```

Result

[read_cam_par](#) returns 2 (H_MSG_TRUE) if all parameter values are correct and the file has been read successfully. If necessary an exception handling is raised.

Parallelization Information

[read_cam_par](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[find_marks_and_pose](#), [sim_caltab](#), [create_caltab](#), [disp_caltab](#), [camera_calibration](#)

See also

[find_caltab](#), [find_marks_and_pose](#), [camera_calibration](#), [disp_caltab](#), [sim_caltab](#), [write_cam_par](#), [write_pose](#), [read_pose](#), [project_3d_point](#), [get_line_of_sight](#)

Module

Camera calibration

read_pose (: : PoseFile : Pose)

Read a 3D pose from a text file.

[read_pose](#) is used to read the 3D pose [Pose](#) from a text file with the name [PoseFile](#).

A pose describes a rigid 3D transformation, i.e., a transformation consisting of an arbitrary translation and rotation, with 6 parameters, three for the translation, three for the rotation. With a seventh parameter different pose types can be indicated (see [create_pose](#)).

A suitable file can be generated by the operator [write_pose](#) and looks like the following:

```
# 3D POSE PARAMETERS: rotation and translation

# Used representation type:
f 0

# Rotation angles [deg] or Rodriguezvector:
r -17.8134 1.83816 0.288092

# Translation vector (x y z [m]):
t 0.280164 0.150644 1.7554
```

Parameter
▷ PoseFile (input_control) filename $\leadsto$ string File name of the exterior camera parameters. Default Value : 'campose.dat' List of values : PoseFile $\in$ {'campose.dat', 'campose.initial', 'campose.final'} ▷ Pose (output_control) pose-array $\leadsto$ real / integer 3D pose. Number of elements : 7
Result
read_pose returns 2 (H_MSG_TRUE) if all parameter values are correct and the file has been read successfully. If necessary an exception handling is raised.
Parallelization Information
read_pose is <i>reentrant</i> and processed <i>without</i> parallelization.
Possible Predecessors
read_cam_par
Possible Successors
pose_to_hom_mat3d , camera_calibration , disp_caltab , sim_caltab
See also
create_pose , find_marks_and_pose , camera_calibration , disp_caltab , sim_caltab , write_pose , pose_to_hom_mat3d , hom_mat3d_to_pose
Module
Camera calibration

set_origin_pose (: : PoseIn, DX, DY, DZ : PoseNewOrigin)

Translate the origin of a 3D pose.

set_origin_pose translates the origin of the 3D pose **PoseIn** by the vector given by **DX**, **DY**, and **DZ** and returns the result in **PoseNewOrigin**. Note that the translation is performed relative to the coordinate system of the pose itself. For example, if **PoseIn** describes the pose of an object in camera coordinates, **PoseNewOrigin** is obtained by translating the object's coordinate system by **DX** along its own x-axis (and so on for the other axes) and not along the x-axis of the camera coordinate system. This corresponds to the following chain of transformations:

$$\text{PoseNewOrigin} = \text{PoseIn} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{bmatrix} \begin{pmatrix} \text{DX} \\ \text{DY} \\ \text{DZ} \\ 1 \end{pmatrix}$$

A typical application of this operator when defining a world coordinate system by placing the standard calibration plate on the plane of measurements. In this case, the external camera parameters returned by **camera_calibration** correspond to a coordinate system that lies above the measurement plane, because the coordinate system of the calibration plate is located on its surface and the plate has a certain thickness. To correct the pose, call **set_origin_pose** with the translation vector (0,0,*D*), where *D* is the thickness of the calibration plate.

Parameter
▷ PoseIn (input_control) pose-array $\leadsto$ real / integer original 3D pose. Number of elements : 7 ▷ DX (input_control) real $\leadsto$ real translation of the origin in x-direction. Default Value : 0 ▷ DY (input_control) real $\leadsto$ real translation of the origin in y-direction. Default Value : 0

- ▷ **DZ** (input_control) real $\leadsto$ real
translation of the origin in z-direction.
Default Value : 0
- ▷ **PoseNewOrigin** (output_control) pose-array $\leadsto$ real / integer
new 3D pose after applying the translation.
Number of elements : 7

Result

[set_origin_pose](#) returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception handling is raised.

Parallelization Information

[set_origin_pose](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[create_pose](#), [hom_mat3d_to_pose](#), [camera_calibration](#), [hand_eye_calibration](#)

Possible Successors

[write_pose](#), [pose_to_hom_mat3d](#), [image_points_to_world_plane](#),
[contour_to_world_plane_xld](#)

Module

Camera calibration

sim_caltab (: SimImage : CalTabDescrFile, CamParam, CaltabPose,
GrayBackground, GrayCaltab, GrayMarks, ScaleFac :)

Simulate an image with calibration plate.

[sim_caltab](#) is used to generate a simulated calibration image. The calibration plate description is read from the file [CalTabDescrFile](#) and will be projected into the image plane using the given camera parameters (interior camera parameters [CamParam](#) and exterior camera parameters [CaltabPose](#)), see also [project_3d_point](#).

In the simulated image only the calibration plate is shown. The image background is set to the gray value [GrayBackground](#), the calibration plate background is set to [GrayCaltab](#), and the calibration marks are set to the gray value [GrayMarks](#). The parameter [ScaleFac](#) influences the number of supporting points to approximate the elliptic contours of the calibration marks, see also [disp_caltab](#). Increasing the number of supporting points causes a more accurate determination of the mark boundary, but increases the computation time, too. For each pixel of the simulated image, which touches a subpixel-boundary of this kind, the gray value is set linearly between [GrayMarks](#) and [GrayCaltab](#) dependent on the proportion Inside/Outside.

By applying the operator [sim_caltab](#) you can generate synthetic calibration images (with known camera parameters!) to test the quality of the calibration algorithm (see [camera_calibration](#)).

Parameter

- ▷ **SimImage** (output_object) image $\leadsto$ Hobject : byte
Simulated calibration image.
- ▷ **CalTabDescrFile** (input_control) string $\leadsto$ string
File name of the calibration plate description.
Default Value : 'caltab.descr'
- ▷ **CamParam** (input_control) number-array $\leadsto$ real / integer
Interior camera parameters.
Number of elements : 8
- ▷ **CaltabPose** (input_control) pose-array $\leadsto$ real / integer
Exterior camera parameters (3D pose of the calibration plate in camera coordinates).
Number of elements : 7
- ▷ **GrayBackground** (input_control) integer $\leadsto$ integer
Gray value of image background.
Default Value : 128
Suggested values : GrayBackground $\in \{0, 32, 64, 96, 128, 160\}$
Restriction : $(0 \leq \text{GrayBackground}) \leq 255$

- ▷ **GrayCaltab** (input_control) integer $\leadsto$ integer
Gray value of calibration plate.
Default Value : 224
Suggested values : GrayCaltab $\in \{144, 160, 176, 192, 208, 224, 240\}$
Restriction : $(0 \leq \text{GrayCaltab}) \leq 255$
- ▷ **GrayMarks** (input_control) integer $\leadsto$ integer
Gray value of calibration marks.
Default Value : 80
Suggested values : GrayMarks $\in \{16, 32, 48, 64, 80, 96, 112\}$
Restriction : $(0 \leq \text{GrayMarks}) \leq 255$
- ▷ **ScaleFac** (input_control) real $\leadsto$ real
Scaling factor to reduce oversampling.
Default Value : 1.0
Suggested values : ScaleFac $\in \{1.0, 0.5, 0.25, 0.125\}$
Recommended Increment : 0.05
Restriction : $1.0 \geq \text{ScaleFac}$

Example

```
* read calibration image
read_image(Image1, 'calib-01')
* find calibration pattern
find_caltab(Image1, Caltab1, 'caltab.descr', 3, 112, 5)
* find calibration marks and initial pose
StartCamPar := [Focus, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]
find_marks_and_pose(Image1, Caltab1, 'caltab.descr', StartCamPar,
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord1, CCoord1,
                    StartPose1)
* read 3D positions of calibration marks
caltab_points('caltab.descr', NX, NY, NZ)
* camera calibration
camera_calibration(NX, NY, NZ, RCoord1, CCoord1, StartCamPar,
                  StartPose1, 11, CamParam, FinalPose, Errors)
* simulate calibration image
sim_caltab(Image1Sim, 'caltab.descr', CamParam, FinalPose, 128, 224, 80, 1)
```

Result

[sim_caltab](#) returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception handling is raised.

Parallelization Information

[sim_caltab](#) is processed under *mutual exclusion* against itself and without parallelization.

Possible Predecessors

[camera_calibration](#), [find_marks_and_pose](#), [read_pose](#), [read_cam_par](#), [hom_mat3d_to_pose](#)

Possible Successors

[find_caltab](#)

See also

[find_caltab](#), [find_marks_and_pose](#), [camera_calibration](#), [disp_caltab](#), [create_pose](#), [hom_mat3d_to_pose](#), [project_3d_point](#), [create_caltab](#)

Module

Camera calibration

write_cam_par (: : CamParam, CamParFile :)

Write the interior camera parameters to text file.

[write_cam_par](#) is used to write the interior camera parameters [CamParam](#) to a text file with name [CamParFile](#). [CamParam](#) is a tuple that contains the interior camera parameters in the following sequence:

[Focus, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]

The interior camera parameters describe the projection process of the used combination of camera, lens, and frame grabber; they can be determined calibrating the camera, see [camera.calibration](#).

The projection of a point $\mathbf{p}^c$ given in camera coordinates into a (sub-)pixel $[r, c]$ in the image consists of multiple steps: First, the point is projected into the image plane, i.e., onto the sensor chip. If the underlying camera model is a **pinhole camera with radial distortions**, i.e., if the focal length passed in [CamParam](#) is greater than 0, the projection is described by the following equations:

$$\mathbf{p}^c = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

$$u = \text{Focus} \cdot \frac{x}{z} \quad \text{and} \quad v = \text{Focus} \cdot \frac{y}{z}$$

In contrast, if the focal length is passed as 0 in [CamParam](#), the camera model of a **telecentric camera with radial distortions** is used, i.e., it is assumed that the optics of the lens of the camera performs a parallel projection. In this case, the corresponding equations are:

$$\mathbf{p}^c = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

$$u = x \quad \text{and} \quad v = y$$

The following equations compensate for radial distortion:

$$\tilde{u} = \frac{2u}{1 + \sqrt{1 - 4\kappa(u^2 + v^2)}} \quad \text{and} \quad \tilde{v} = \frac{2v}{1 + \sqrt{1 - 4\kappa(u^2 + v^2)}}$$

Finally, the point is transformed from the image plane coordinate system into the image coordinate system, i.e., the pixel coordinate system:

$$c = \frac{\tilde{u}}{S_x} + C_x \quad \text{and} \quad r = \frac{\tilde{v}}{S_y} + C_y$$

The format of the text file is a (HALCON-independent) generic parameter description. It allows to group arbitrary sets of parameters hierarchically. The description of a single parameter within a parameter group consists of the following 3 lines:

```
Name : Shortname : Actual value ;
Type : Lower bound (optional) : Upper bound (optional) ;
Description (optional) ;
```

The operator [write_cam_par](#) writes the parameter group Camera:Parameter into the text file [CamParFile](#). This parameter group consists of the 8 interior camera parameters, but in a different sequence than in [CamParam](#) (see [read_cam_par](#) for an example).

<i>Parameter</i>
▷ CamParam (input_control) number-array $\leadsto$ <i>real</i> / integer Interior camera parameters. Number of elements : 8 ▷ CamParFile (input_control) string $\leadsto$ <i>string</i> File name of interior camera parameters. Default Value : 'campar.dat' List of values : CamParFile $\in$ {'campar.dat', 'campar.initial', 'campar.final'}

Example

```

* read calibration images
read_image(Image1, 'calib-01')
read_image(Image2, 'calib-02')
read_image(Image3, 'calib-03')
* find calibration pattern
find_caltab(Image1, Caltab1, 'caltab.descr', 3, 112, 5)
find_caltab(Image2, Caltab2, 'caltab.descr', 3, 112, 5)
find_caltab(Image3, Caltab3, 'caltab.descr', 3, 112, 5)
* find calibration marks and start poses
StartCamPar := [Focus, Kappa, Sx, Sy, Cx, Cy, ImageWidth, ImageHeight]
find_marks_and_pose(Image1, Caltab1, 'caltab.descr', StartCamPar,
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord1, CCoord1,
                    StartPose1)
find_marks_and_pose(Image2, Caltab2, 'caltab.descr', StartCamPar,
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord2, CCoord2,
                    StartPose2)
find_marks_and_pose(Image3, Caltab3, 'caltab.descr', StartCamPar,
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord3, CCoord3,
                    StartPose3)
* read 3D positions of calibration marks
caltab_points('caltab.descr', NX, NY, NZ)
* camera calibration
camera_calibration(NX, NY, NZ, [RCoord1, RCoord2, RCoord3],
                  [CCoord1, CCoord2, CCoord3], StartCamPar,
                  [StartPose1, StartPose2, StartPose3], 'all',
                  CamParam, NFinalPose, Errors)
* write interior camera parameters to file
write_cam_par(CamParam, 'campar.dat')

```

Result

[write_cam_par](#) returns 2 (H_MSG.TRUE) if all parameter values are correct and the file has been written successfully. If necessary an exception handling is raised.

Parallelization Information

[write_cam_par](#) is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[camera_calibration](#)

See also

[find_caltab](#), [find_marks_and_pose](#), [camera_calibration](#), [disp_caltab](#), [sim_caltab](#),
[read_cam_par](#), [write_pose](#), [read_pose](#), [project_3d_point](#), [get_line_of_sight](#)

Module

Camera calibration

write_pose (: : Pose, PoseFile :)

Write a 3D pose to a text file.

[write_pose](#) is used to write a 3D pose [Pose](#) into a text file with the name [PoseFile](#).

A pose describes a rigid 3D transformation, i.e., a transformation consisting of an arbitrary translation and rotation, with 6 parameters, three for the translation, three for the rotation. With a seventh parameter different pose types can be indicated (see [create_pose](#)).

A file generated by [write_pose](#) looks like the following:

```
# 3D POSE PARAMETERS: rotation and translation
```

```
# Used representation type:
f 0

# Rotation angles [deg] or Rodriguez vector:
r -17.8134 1.83816 0.288092

# Translation vector (x y z [m]):
t 0.280164 0.150644 1.7554
```

Parameter

- ▷ **Pose** (input_control) pose-array $\leadsto$ *real* / integer
3D pose.
Number of elements : 7
- ▷ **PoseFile** (input_control) filename $\leadsto$ *string*
File name of the exterior camera parameters.
Default Value : 'campose.dat'
List of values : PoseFile $\in$ {'campose.dat', 'campose.initial', 'campose.final'}

Example

```
* read calibration images
read_image(Image1, 'calib-01')
read_image(Image2, 'calib-02')
read_image(Image3, 'calib-03')
* find calibration pattern
find_caltab(Image1, Caltab1, 'caltab.descr', 3, 112, 5)
find_caltab(Image2, Caltab2, 'caltab.descr', 3, 112, 5)
find_caltab(Image3, Caltab3, 'caltab.descr', 3, 112, 5)
* find calibration marks and start poses
StartCamPar := [0.008, 0.0, 0.000011, 0.000011, 384, 288, 768, 576]
find_marks_and_pose(Image1, Caltab1, 'caltab.descr', StartCamPar,
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord1, CCoord1,
                    StartPose1)
find_marks_and_pose(Image2, Caltab2, 'caltab.descr', StartCamPar,
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord2, CCoord2,
                    StartPose2)
find_marks_and_pose(Image3, Caltab3, 'caltab.descr', StartCamPar,
                    128, 10, 18, 0.9, 15.0, 100.0, RCoord3, CCoord3,
                    StartPose3)
* read 3D positions of calibration marks
caltab_points('caltab.descr', NX, NY, NZ)
* camera calibration
camera_calibration(NX, NY, NZ, [RCoord1, RCoord2, RCoord3],
                  [CCoord1, CCoord2, CCoord3], StartCamPar,
                  [StartPose1, StartPose2, StartPose3], 'all',
                  CamParam, NFinalPose, Errors)
* write exterior camera parameters of first calibration image
write_pose(NFinalPose[0:6], 'campose.dat')
```

Result

[write_pose](#) returns 2 (H_MSG_TRUE) if all parameter values are correct and the file has been written successfully. If necessary an exception handling is raised.

Parallelization Information

[write_pose](#) is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[camera_calibration](#), [hom_mat3d_to_pose](#)

See also

[create_pose](#), [find_marks_and_pose](#), [camera_calibration](#), [disp_caltab](#), [sim_caltab](#),
[read_pose](#), [pose_to_hom_mat3d](#), [hom_mat3d_to_pose](#)

15.5 Fourier-Descriptor

abs_invar_fourier_coeff (: : RealInvar, ImaginaryInvar, CoefP, CoefQ, AZInvar : RealAbsInvar, ImaginaryAbsInvar)

Normalizing of the Fourier coefficients with respect to the displacement of the starting point.

The operator **abs_invar_fourier_coeff** normalizes the Fourier coefficients with regard to the displacements of the starting point. These occur when an object is rotated. The contour tracer **get_region_contour** starts with recording the contour in the upper lefthand corner of the region and follows the contour clockwise. If the object is rotated, the starting value for the contour point chain is different which leads to a phase shift in the frequency space. The following two kinds of normalizing are available:

abs_amount: The phase information will be eliminated; the normalizing does not retain the structure, i.e. if the AZ-invariants are backtransformed, no similarity with the pattern can be recognized anymore.

az_invar1: AZ-invariants of the 1st order execute the normalizing with respect to displacing the starting point so that the structure is retained; they are however more prone to local and global disturbances, in particular to projective distortions.

Parameter

- ▷ **RealInvar** (input_control) real-array $\leadsto$ *real*
Real parts of the normalized Fourier coefficients.
- ▷ **ImaginaryInvar** (input_control) real-array $\leadsto$ *real*
Imaginary parts of the normalized Fourier coefficients.
- ▷ **CoefP** (input_control) integer $\leadsto$ *integer*
Normalizing coefficients p.
Default Value : 1
Suggested values : CoefP $\in \{1, 2\}$
Restriction : CoefP ≥ 1
- ▷ **CoefQ** (input_control) integer $\leadsto$ *integer*
Normalizing coefficients q.
Default Value : 1
Suggested values : CoefQ $\in \{1, 2\}$
Restriction : (CoefQ ≥ 1) $\wedge$ (CoefQ $\neq$ CoefP)
- ▷ **AZInvar** (input_control) string $\leadsto$ *string*
Order of the AZ-invariants.
Default Value : 'abs_amount'
List of values : AZInvar $\in \{'abs\_amount', 'az\_invar1'\}$
- ▷ **RealAbsInvar** (output_control) real-array $\leadsto$ *real*
Real parts of the normalized Fourier coefficients.
- ▷ **ImaginaryAbsInvar** (output_control) real-array $\leadsto$ *real*
Imaginary parts of the normalized Fourier coefficients.

Example

```
get_region_contour(single,&row,&col);
length_of_contour = length_tuple(row);
move_contour_orig(row,col,&trow,&tcoll);
prep_contour_fourier(trow,tcoll,"unsigned_area",&param_scale);
fourier_1dim(trow,tcoll,param_scale,50,&frow,&fcol);
invar_fourier_coeff(frow,fcol,1,"affine_invar",&invrow,&invcol);
abs_invar_fourier_coeff(invrow,invcol,1,2,"az_invar1",&absrow,&abscol);
fourier_1dim_inv(absrow,abscol,length_of_contour,&fsynrow,&fsyncol);
```

Parallelization Information

`abs_invar_fourier_coeff` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`invar_fourier_coeff`

Possible Successors

`fourier_1dim_inv`, `match_fourier_coeff`

Module

Fourier descriptors

fourier_1dim (: : Rows, Columns, ParContour, MaxCoef : RealCoef, ImaginaryCoef)

Calculate the Fourier coefficients of a parameterized contour.

The operator `fourier_1dim` calculates the Fourier coefficients of a parameterized contour by using a valid parameter scale. This parameter scale may, for instance, be created with the help of the procedure `prep_contour_fourier`. This function serves to calculate the Fourier coefficients of closed contours which are treated like complex-valued curves. Therefore, in order to determine the Fourier coefficients, the Fourier transform for periodical functions is used. Hereby the parameter `MaxCoef` determines the *absolutevalue* + 1 of the maximal number of Fourier coefficients, i.e. if n coefficients are indicated, the procedure will calculate coefficients ranging from $-n$ to n . The contour will be approximated without loss, if $n = \text{number of the contour points}$, whereby $n = 100$ approximates the contour so well that an error can hardly be distinguished; $n \in [40, 50]$ however is sufficient for most applications. If the parameter `MaxCoef` is set to 0, all coefficients will be determined.

Parameter

- ▷ **Rows** (input_control) contour.y-array $\leadsto$ *integer*
Row coordinates of the contour.
- ▷ **Columns** (input_control) contour.x-array $\leadsto$ *integer*
Column coordinates of the contour.
- ▷ **ParContour** (input_control) real-array $\leadsto$ *real*
Parameter scale.
- ▷ **MaxCoef** (input_control) integer $\leadsto$ *integer*
Desired number of Fourier coefficients or all of them (0).
Default Value : 50
Suggested values : `MaxCoef` $\in$ {0, 5, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 400}
Restriction : `MaxCoef` $\geq$ 0
- ▷ **RealCoef** (output_control) real-array $\leadsto$ *real*
Real parts of the Fourier coefficients.
- ▷ **ImaginaryCoef** (output_control) real-array $\leadsto$ *real*
Imaginary parts of the Fourier coefficients.

Example

```
get_region_contour(single,&row,&col);
move_contour_orig(row,col,&trow,&tcoll);
prep_contour_fourier(trow,tcoll,"unsigned_area",&param_scale);
fourier_1dim(trow,tcoll,param_scale,&frow,&fcoll);
invar_fourier_coeff(frow,fcoll,1,"affine_invar",&invrow,&invcoll);
abs_invar_fourier_coeff(invrow,invcoll,1,2,"az_invar1",&absrow,&abscoll);
```

Parallelization Information

`fourier_1dim` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`prep_contour_fourier`

Possible Successors

`invar_fourier_coeff`, `disp_polygon`

Module

Fourier descriptors

fourier_1dim_inv (: : RealCoef, ImaginaryCoef, MaxCoef : Rows, Columns)

One dimensional Fourier synthesis (inverse Fourier transform).

Backtransformation of Fourier coefficients respectively of Fourier descriptors. The number of values to be backtransformed should not exceed the length of the transformed contour.

Parameter

- ▷ **RealCoef** (input_control) real-array $\leadsto$ *real*
Real parts.
- ▷ **ImaginaryCoef** (input_control) real-array $\leadsto$ *real*
Imaginary parts.
- ▷ **MaxCoef** (input_control) integer $\leadsto$ *integer*
Input of the steps for the backtransformation.
Default Value : 100
Suggested values : MaxCoef $\in \{5, 10, 15, 20, 30, 40, 50, 60, 70, 80, 90, 100, 150, 200, 400\}$
Restriction : MaxCoef ≥ 1
- ▷ **Rows** (output_control) contour.y-array $\leadsto$ *real*
Row coordinates.
- ▷ **Columns** (output_control) contour.x-array $\leadsto$ *real*
Column coordinates.

Example

```
get_region_contour(single,&row,&col);
length_of_contour = row.Num();
move_contour_orig(row,col,&trow,&tcoll);
prep_contour_fourier(trow,tcoll,"unsigned_area",&param_scale);
fourier_1dim(trow,tcoll,param_scale,50,&frow,&fcoll);
invar_fourier_coeff(frow,fcoll,1,"affine_invar",&invrow,&invcoll);
abs_invar_fourier_coeff(invrow,invcoll,1,2,"az_invar1",&absrow,&abscoll);
fourier_1dim_inv(absrow,abscoll,length_of_contour,&fsynrow,&fsyncoll);
```

Parallelization Information

fourier_1dim_inv is *reentrant* and processed *without* parallelization.

Possible Predecessors

invar_fourier_coeff, **fourier_1dim**

Possible Successors

disp_polygon

Module

Fourier descriptors

invar_fourier_coeff (: : RealCoef, ImaginaryCoef, NormPar, InvarType : RealInvar, ImaginaryInvar)

Normalize the Fourier coefficients.

Elimination of affine information from the Fourier coefficients, determination of affine invariants. The Fourier coefficients will be normalized suitably so that all affine correlated contours will be projected to one and the same contour. The following levels of affine mappings are available:

1. Translations (**InvarType** = 'transl_invar')

2. + Rotations (**InvarType** = 'congr_invar')
3. + Scalings (**InvarType** = 'simil_invar')
4. + Slanting (**InvarType** = 'affine_invar')

The control parameter **InvarType** indicates up to which level the affine representation shall be normalized. Please note that indicating a certain level implies that the normalizing is executed with regard to all levels below. For most applications a subsequent normalizing of the starting point is recommended!

Parameter

- ▷ **RealCoef** (input_control) real-array $\leadsto$ *real*
Real parts of the Fourier coefficients.
- ▷ **ImaginaryCoef** (input_control) real-array $\leadsto$ *real*
Imaginary parts of the Fourier coefficients.
- ▷ **NormPar** (input_control) integer $\leadsto$ *integer*
Input of the normalizing coefficients.
Default Value : 1
Suggested values : NormPar $\in \{1, 2\}$
Restriction : NormPar ≥ 1
- ▷ **InvarType** (input_control) string $\leadsto$ *string*
Indicates the level of the affine mappings.
Default Value : 'affine_invar'
List of values : InvarType $\in \{'affine\_invar', 'simil\_invar', 'congr\_invar', 'transl\_invar'\}$
- ▷ **RealInvar** (output_control) real-array $\leadsto$ *real*
Real parts of the normalized Fourier coefficients.
- ▷ **ImaginaryInvar** (output_control) real-array $\leadsto$ *real*
Imaginary parts of the normalized Fourier coefficients.

Example

```
prep_contour_fourier(trow,tcol,"unsigned_area",&param_scale);
fourier_1dim(trow,tcol,param_scale,&frow,&fcol);
invar_fourier_coeff(frow,fcol,1,"affine_invar",&invrow,&invcol);
abs_invar_fourier_coeff(invrow,invcol,1,2,"az_invar1",&absrow,&abscol);
```

Parallelization Information

invar_fourier_coeff is *reentrant* and processed *without* parallelization.

Possible Predecessors

fourier_1dim

Possible Successors

invar_fourier_coeff

Module

Fourier descriptors

```
match_fourier_coeff ( :: RealCoef1, ImaginaryCoef1, RealCoef2,
ImaginaryCoef2, MaxCoef, Damping : Distance )
```

Similarity of two contours.

The operator **match_fourier_coeff** calculates the Euclidean distance between two contours which are available as Fourier coefficients. In order to avoid that the higher frequencies are in some way too dominant, the following attenuation can be used:

none: No attenuation.

1/index: Absolute amounts of the Fourier coefficients will be divided by their index.

1/(index*index): Absolute amounts of the Fourier coefficients will be divided by their square index.

The higher the result value, the greater the differences between the pattern and the test contour. If the number of coefficients is not the same, only the first n coefficients will be compared. The parameter **MaxCoef** indicates the number of the coefficients to be compared. If **MaxCoef** is set to zero, all coefficients will be used.

Parameter

- ▷ **RealCoef1** (input_control) real-array $\leadsto$ *real*
Real parts of the pattern Fourier coefficients.
- ▷ **ImaginaryCoef1** (input_control) real-array $\leadsto$ *real*
Imaginary parts of the pattern Fourier coefficients.
- ▷ **RealCoef2** (input_control) real-array $\leadsto$ *real*
Real parts of the Fourier coefficients to be compared.
- ▷ **ImaginaryCoef2** (input_control) real-array $\leadsto$ *real*
Imaginary parts of the Fourier coefficients to be compared.
- ▷ **MaxCoef** (input_control) integer $\leadsto$ *integer*
Total number of Fourier coefficients.
Default Value : 50
Suggested values : $\text{MaxCoef} \in \{0, 5, 10, 15, 20, 30, 40, 50, 70, 100, 200, 400\}$
Restriction : $\text{MaxCoef} \geq 0$
- ▷ **Damping** (input_control) string $\leadsto$ *string*
Kind of attenuation.
Default Value : "1/index"
Suggested values : $\text{Damping} \in \{\text{'none'}, "1/\text{index}", "1/(\text{index}*\text{index})"\}$
- ▷ **Distance** (output_control) real $\leadsto$ *real*
Similarity of the contours.

Example

```
prep_contour_fourier(trow,tcol,"unsigned_area",&param_scale);
fourier_1dim(trow,tcol,param_scale,50,&frow,&fcol);
invar_fourier_coeff(frow,fcol,1,"affine_invar",&invrow,&invcol);
abs_invar_fourier_coeff(invrow,invcol,1,2,
    "az_invar1",&absrow,&abscol);
match_fourier_coeff(contur1_row, contur1_col,
    contur2_row, contur2_col, 50,
    "1/index", &Distance_wert);
```

Parallelization Information

match_fourier_coeff is *reentrant* and processed without parallelization.

Possible Predecessors

invar_fourier_coeff

Module

Fourier descriptors

move_contour_orig (: : Rows, Columns : RowsMoved, ColumnsMoved)

Transformation of the origin into the centre of gravity.

The operator **move_contour_orig** relocates the input contour so that the origin lies in the centre of gravity.

Parameter

- ▷ **Rows** (input_control) contour.y-array $\leadsto$ *integer*
Row coordinates of the contour.
- ▷ **Columns** (input_control) contour.x-array $\leadsto$ *integer*
Column coordinates of the contour.
- ▷ **RowsMoved** (output_control) contour.y-array $\leadsto$ *integer*
Row coordinates of the displaced contour.

- ▷ **ColumnsMoved** (output_control) contour.x-array $\leadsto$ *integer*
Column coordinates of the displaced contour.

Parallelization Information

`move_contour_orig` is processed *completely exclusively* without parallelization.

Possible Predecessors

`get_region_contour`

Possible Successors

`prep_contour_fourier`

Module

Fourier descriptors

prep_contour_fourier (: : Rows, Columns, TransMode : ParContour)

Parameterize the passed contour.

The operator `prep_contour_fourier` parameterizes the transmitted contour in order to prepare it for the one dimensional Fourier transformation. Hereby the contour must be available in closed form. Three parameter functions are available for the control parameter **TransMode**:

arc: Parameterization by the radian.

signed_area: Parameterization by the signed area.

unsigned_area: Parameterization by the absolute area.

Please note that in contrast to the signed or unsigned area the affine mapping of the radian will not be transformed linearly.

Parameter

- ▷ **Rows** (input_control) contour.y-array $\leadsto$ *integer*
Row indices of the contour.
- ▷ **Columns** (input_control) contour.x-array $\leadsto$ *integer*
Column indices of the contour.
- ▷ **TransMode** (input_control) string $\leadsto$ *string*
Kind of parameterization.
Default Value : 'signed_area'
Suggested values : TransMode $\in$ {'arc', 'unsigned_area', 'signed_area'}
- ▷ **ParContour** (output_control) real-array $\leadsto$ *real*
Parameterized contour.

Example

```
get_region_contour(single,&row,&col);
move_contour_orig(row,col,&trow,&tcol);
prep_contour_fourier(trow,tcol,"unsigned_area",&param_scale);
fourier_1dim(trow,tcol,param_scale,&frow,&fcol);
```

Parallelization Information

`prep_contour_fourier` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`move_contour_orig`

Possible Successors

`fourier_1dim`

Module

Fourier descriptors

15.6 Function

abs_funcnt_1d (: : Function : FunctionAbsolute)

Absolute value of the y values.

[abs_funcnt_1d](#) calculates the absolute values of all y values of [Function](#).

Parameter

- ▷ **Function** (input_control) function_1d-array $\leadsto$ real / integer
Input function.
- ▷ **FunctionAbsolute** (output_control) function_1d-array $\leadsto$ real / integer
Function with the absolute values of the y values.

Parallelization Information

[abs_funcnt_1d](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[create_funcnt_1d_pairs](#), [create_funcnt_1d_array](#)

Module

Tools

create_funcnt_1d_array (: : YValues : Function)

Create a function from a sequence of y-values.

[create_funcnt_1d_array](#) creates a one-dimensional function from a set of y-values [YValues](#). The resulting function can then be processed and analyzed with the operators for 1d functions. [YValues](#) is interpreted as follows: the first value of [YValues](#) is the function value at zero, the second value is the function value at one, etc. Thus, the values define a function at equidistant x values (with distance 1), starting at 0.

Alternatively, the operator [create_funcnt_1d_pairs](#) can be used to create a function. [create_funcnt_1d_pairs](#) also allows to define a function with non-equidistant x value by specifying them explicitly. Thus to get the same definition as with [create_funcnt_1d_array](#), one would pass a tuple of x values to [create_funcnt_1d_pairs](#) that has the same length as [YValues](#) and contains values starting at 0 and increasing by 1 in each position. Note, however, that [create_funcnt_1d_pairs](#) leads to a different internal representation of the function which needs more storage (because all (x,y) pairs are stored) and sometimes cannot be processed as efficiently as functions created by [create_funcnt_1d_array](#).

Parameter

- ▷ **YValues** (input_control) number(-array) $\leadsto$ real / integer
X value for function points.
- ▷ **Function** (output_control) function_1d-array $\leadsto$ real / integer
Created function.

Parallelization Information

[create_funcnt_1d_array](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[write_funcnt_1d](#), [gnuplot_plot_funcnt_1d](#), [y_range_funcnt_1d](#), [get_pair_funcnt_1d](#),
[transform_funcnt_1d](#)

Alternatives

[create_funcnt_1d_pairs](#), [read_funcnt_1d](#)

See also

[funcnt_1d.to_pairs](#)

Module

Tools

create_funcnt_1d_pairs (: : XValues, YValues : Function)

Create a function from a set of (x,y) pairs.

[create_funcnt_1d_pairs](#) creates a one-dimensional function from a set of pairs of (x,y) values. The [XValues](#) of the functions have to be passed in ascending order. The resulting function can then be processed and analyzed with the operators for 1d functions.

Alternatively, functions can be created with the operator [create_funcnt_1d_array](#). In contrast to this operator, x values with arbitrary positions can be specified with [create_funcnt_1d_pairs](#). Hence, it is the more general operator. It should be noted, however, that because of this generality the processing of a function created with [create_funcnt_1d_pairs](#) cannot be carried out as efficiently as for equidistant functions. In particular, not all operators accept such functions. If necessary, a function can be transformed into an equidistant function with the operator [sample_funcnt_1d](#).

Parameter

- ▷ **XValues** (input_control) number(-array) $\leadsto$ *real* / integer
X value for function points.
- ▷ **YValues** (input_control) number(-array) $\leadsto$ *real* / integer
Y-value for function points.
- ▷ **Function** (output_control) function_1d-array $\leadsto$ *real* / integer
Created function.

Parallelization Information

[create_funcnt_1d_pairs](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[write_funcnt_1d](#), [gnuplot_plot_funcnt_1d](#), [y_range_funcnt_1d](#), [get_pair_funcnt_1d](#)

Alternatives

[create_funcnt_1d_array](#), [read_funcnt_1d](#)

See also

[funcnt_1d_to_pairs](#)

Module

Tools

distance_funcnt_1d (: : Function1, Function2, Mode,
Sigma : Distance)

Compute the distance of two functions.

[distance_funcnt_1d](#) calculates the distance of two functions. The two functions may differ in length.

Parameter

- ▷ **Function1** (input_control) function_1d-array $\leadsto$ *real* / integer
Input function 1.
- ▷ **Function2** (input_control) function_1d-array $\leadsto$ *real* / integer
Input function 2.
- ▷ **Mode** (input_control) string(-array) $\leadsto$ *string*
Modes of invariants.
Default Value : 'length'
List of values : Mode $\in$ {'length', 'mean'}
- ▷ **Sigma** (input_control) number(-array) $\leadsto$ *real*
Variance of the optional smoothing with a Gaussian filter.
Default Value : 0.0
Suggested values : Sigma $\in$ {0.0, 0.5, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0, 15.0, 20.0, 25.0, 30.0, 40.0, 50.0}
- ▷ **Distance** (output_control) real-array $\leadsto$ *real* / integer
Distance of the functions.

Parallelization Information

`distance_funcnt_1d` is *reentrant* and processed *without* parallelization.

Module

Tools

funcnt_1d_to_pairs (: : Function : XValues, YValues)

Access to the x/y values of a function.

`funcnt_1d_to_pairs` splits the input function `Function` into tuples for the x and y values.

Parameter

- ▷ **Function** (input_control) function_1d-array $\leadsto$ *real* / integer
Input function.
- ▷ **XValues** (output_control) number-array $\leadsto$ *real* / integer
X values of the function.
- ▷ **YValues** (output_control) number-array $\leadsto$ *real* / integer
Y values of the function.

Parallelization Information

`funcnt_1d_to_pairs` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`create_funcnt_1d_pairs`, `create_funcnt_1d_array`

Module

Tools

get_pair_funcnt_1d (: : Function, Index : X, Y)

Access a function value using the index of the control points.

`get_pair_funcnt_1d` accesses a function value of `Function`. This is done by specifying the index of one or more control points of the function.

Parameter

- ▷ **Function** (input_control) function_1d-array $\leadsto$ *real* / integer
Input function.
- ▷ **Index** (input_control) integer(-array) $\leadsto$ *integer*
Index of the control points.
- ▷ **X** (output_control) number(-array) $\leadsto$ *real*
X value at the given control points.
- ▷ **Y** (output_control) number(-array) $\leadsto$ *real*
Y value at the given control points.

Parallelization Information

`get_pair_funcnt_1d` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`create_funcnt_1d_pairs`, `create_funcnt_1d_array`

Module

Tools

integrate_funcnt_1d (: : Function : Positive, Negative)

Compute the positive and negative areas of a function.

`integrate_func1d` integrates the function `Function` (see `create_func1d_array` and `create_func1d_pairs`) and returns the integral of the positive and negative parts of the function in `Positive` and `Negative`, respectively. Hence, the integral of the function is the difference `Positive - Negative`. The integration is done on the interval on which the function is defined. For the integration, the function is interpolated linearly.

Parameter

- ▷ **Function** (input_control) function_1d-array $\leadsto$ real
Input function.
- ▷ **Positive** (output_control) number $\leadsto$ real
Area under the positive part of the function.
- ▷ **Negative** (output_control) number-array $\leadsto$ real
Area under the negative part of the function.

Parallelization Information

`integrate_func1d` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`create_func1d_pairs`, `create_func1d_array`

See also

`create_func1d_array`, `create_func1d_pairs`

Module

Tools

match_func1d_trans (:: Function1, Function2, Border, ParamsConst, UseParams : Params, ChiSquare, Covar)

Calculate transformation parameters between two functions.

`match_func1d_trans` calculates the transformation parameters between two functions given as the tuples `Function1` and `Function2` (see `create_func1d_array` and `create_func1d_pairs`). The following model is used for the transformation between the two functions:

$$y_1(x) = a_1 y_2(a_3 x + a_4) + a_2 \quad .$$

The transformation parameters are determined by a least-squares minimization of the following function:

$$\sum_{i=0}^{n-1} (y_1(x_i) - a_1 y_2(a_3 x_i + a_4) + a_2)^2 \quad .$$

The values of the function y_2 are obtained by linear interpolation. The parameter `Border` determines the values of the function `Function2` outside of its domain. For `Border='zero'` these values are set to 0, for `Border='constant'` they are set to the corresponding value at the border, for `Border='mirror'` they are mirrored at the border, and for `Border='cyclic'` they are continued cyclically. The calculated transformation parameters are returned as a 4-tuple in `Params`. If some of the parameter values are known, the respective parameters can be excluded from the least-squares adjustment by setting the corresponding value in the tuple `UseParams` to the value `'false'`. In this case, the tuple `ParamsConst` must contain the known value of the respective parameter. If a parameter is used for the adjustment (`UseParams = 'true'`), the corresponding parameter in `ParamsConst` is ignored. On output, `match_func1d_trans` additionally returns the sum of the squared errors `ChiSquare` of the resulting function, i.e., the function obtained by transforming the input function with the transformation parameters, as well as the covariance matrix `Covar` of the transformation parameters `Params`. These parameters can be used to decide whether a successful matching of the functions was possible.

Parameter

- ▷ **Function1** (input_control) function_1d-array $\leadsto$ real / integer
Function 1.
- ▷ **Function2** (input_control) function_1d-array $\leadsto$ real / integer
Function 2.

- ▷ **Border** (input_control) string $\leadsto$ string
Border treatment for function 2.
Default Value : 'constant'
List of values : Border $\in \{ 'zero', 'constant', 'mirror', 'cyclic' \}$
- ▷ **ParamsConst** (input_control) number-array $\leadsto$ real
Values of the parameters to remain constant.
Default Value : '[1.0,0.0,1.0,0.0]'
Number of elements : 4
- ▷ **UseParams** (input_control) string-array $\leadsto$ string
Should a parameter be adapted for it?
Default Value : '['true','true','true','true']'
List of values : UseParams $\in \{ 'true', 'false' \}$
Number of elements : 4
- ▷ **Params** (output_control) number-array $\leadsto$ real
Transformation parameters between the functions.
Number of elements : 4
- ▷ **ChiSquare** (output_control) number $\leadsto$ real
Quadratic error of the output function.
- ▷ **Covar** (output_control) number-array $\leadsto$ real
Covariance Matrix of the transformation parameters.
Number of elements : 16

Parallelization Information

[match_func_1d_trans](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[create_func_1d_array](#), [create_func_1d_pairs](#)

See also

[gray_projections](#)

Module

Tools

negate_func_1d (: : Function : FunctionInverted)

Negation of the y values.

[negate_func_1d](#) negates all y values of [Function](#).

Parameter

- ▷ **Function** (input_control) function_1d-array $\leadsto$ real / integer
Input function.
- ▷ **FunctionInverted** (output_control) function_1d-array $\leadsto$ real / integer
Function with the negated y values.

Parallelization Information

[negate_func_1d](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[create_func_1d_pairs](#), [create_func_1d_array](#)

Module

Tools

num_points_func_1d (: : Function : Length)

Number of control points of the function.

[num_points_func_1d](#) calculates the number of control points of [Function](#).

Parameter
▷ Function (input_control) function_1d-array $\leadsto$ real / integer Input function. ▷ Length (output_control) integer $\leadsto$ integer Number of control points.
Parallelization Information
<code>num_points_funcnt_1d</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
Possible Predecessors
<code>create_funcnt_1d_pairs</code> , <code>create_funcnt_1d_array</code>
Module
Tools

read_funcnt_1d (: : FileName : Function)

Read a function from a file.

The operator `read_funcnt_1d` reads the contents of `FileName` and converts it into the function `Function`. The file has be generated by `write_funcnt_1d`.

Parameter
▷ FileName (input_control) filename $\leadsto$ string Name of the file to be read. ▷ Function (output_control) function_1d(-array) $\leadsto$ real / integer Function from the file.
Result
If the parameters are correct the operator <code>read_funcnt_1d</code> returns the value 2 (H_MSG.TRUE). If the file could not be opened <code>read_funcnt_1d</code> returns 5 (H_MSG.FAIL). Otherwise an exception handling is raised.
Parallelization Information
<code>read_funcnt_1d</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
<code>fread_string</code> , <code>read_tuple</code>
See also
<code>write_funcnt_1d</code> , <code>gnuplot_plot_ctrl</code> , <code>write_image</code> , <code>write_region</code> , <code>open_file</code>
Module
Basic operators

sample_funcnt_1d (: : Function, XMin, XMax, XDist, Border : SampledFunction)

Sample a function equidistantly in an interval.

`sample_funcnt_1d` samples the input function `Function` in the interval `[XMin,XMax]` at equidistant points with the distance `XDist`. The last point lies in the interval if `XMax-XMin` is not an integer multiple of `XDist`. To obtain the samples, the input function is interpolated linearly. The parameter `Border` determines the values of the function `Function` outside of its domain. For `Border='zero'` these values are set to 0, for `Border='constant'` they are set to the corresponding value at the border, for `Border='mirror'` they are mirrored at the border, and for `Border='cyclic'` they are continued cyclically.

Parameter
▷ Function (input_control) function_1d-array $\leadsto$ real / integer Input function. ▷ XMin (input_control) number $\leadsto$ real / integer Minimum x value of the output function.

- ▷ **XMax** (input_control) number $\leadsto$ real / integer
Maximum x value of the output function.
Restriction : XMax > XMin
- ▷ **XDist** (input_control) number $\leadsto$ real / integer
Distance of the samples.
Restriction : XDist > 0
- ▷ **Border** (input_control) string $\leadsto$ string
Border treatment for the input function.
Default Value : 'constant'
List of values : Border $\in$ {'zero', 'constant', 'mirror', 'cyclic'}
- ▷ **SampledFunction** (output_control) function_1d-array $\leadsto$ real
Sampled function.

Parallelization Information

`sample_func_1d` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`transform_func_1d`, `create_func_1d_array`, `create_func_1d_pairs`

Module

Tools

scale_y_func_1d (: : Function, Mult, Add : FunctionScaled)

Multiplication and addition of the y values.

`scale_y_func_1d` multiplies and adds the y values of `Function` with the parameters `Mult` and `Add`.

Parameter

- ▷ **Function** (input_control) function_1d-array $\leadsto$ real / integer
Input function.
- ▷ **Mult** (input_control) number $\leadsto$ real
Factor for scaling of the y values.
Default Value : 2
Suggested values : Mult $\in$ {0.1, 0.3, 0.5, 1, 2, 5, 10}
- ▷ **Add** (input_control) number $\leadsto$ real
Constant which is added to the y values.
Default Value : 0
Suggested values : Add $\in$ {-10, -5, 1, 0, 5, 10}
- ▷ **FunctionScaled** (output_control) function_1d-array $\leadsto$ real / integer
Transformed function.

Parallelization Information

`scale_y_func_1d` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`create_func_1d_pairs`, `create_func_1d_array`

Module

Tools

smooth_func_1d_gauss (: : Function, Sigma : SmoothedFunction)

Smooth an equidistant function with a Gaussian function.

The operator `smooth_func_1d_gauss` smooths a one-dimensional function with a Gaussian function.

Parameter
▷ Function (input_control) function_1d-array $\leadsto$ real / integer Function to be smoothed. ▷ Sigma (input_control) number $\leadsto$ real Sigma of the Gaussian function for the smoothing. Default Value : 2.0 Suggested values : $\text{Sigma} \in \{0.5, 1.0, 2.0, 3.0, 4.0, 5.0\}$ Typical range of values : $0.1 \leq \text{Sigma} \leq 50.0$ (lin) Minimum Increment : 0.01 Recommended Increment : 0.2 ▷ SmoothedFunction (output_control) function_1d-array $\leadsto$ real Smoothed function.
Parallelization Information
smooth_func_1d_gauss is <i>reentrant</i> and processed <i>without</i> parallelization.
Possible Predecessors
create_func_1d_pairs , create_func_1d_array
Possible Successors
match_func_1d_trans , distance_func_1d
Module
Tools

smooth_func_1d_mean (:: Function, SmoothSize, Iterations : SmoothedFunction)

Smooth a 1D function by averaging its values.

The operator [smooth_func_1d_mean](#) smooths a one dimensional function by applying an average (mean) filter multiple times.

Parameter
▷ Function (input_control) function_1d-array $\leadsto$ integer / real 1D function. ▷ SmoothSize (input_control) integer $\leadsto$ integer Size of the averaging mask. Default Value : 10 Suggested values : $\text{SmoothSize} \in \{1, 3, 5, 7, 9, 11, 13, 15, 21, 31, 51\}$ Typical range of values : $1 \leq \text{SmoothSize} \leq 1000$ (lin) Minimum Increment : 1 Recommended Increment : 1 Restriction : $\text{SmoothSize} > 0$ ▷ Iterations (input_control) integer $\leadsto$ integer Number of iterations for the smoothing. Default Value : 3 Suggested values : $\text{Iterations} \in \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ Typical range of values : $1 \leq \text{Iterations} \leq 100$ (lin) Minimum Increment : 1 Recommended Increment : 1 Restriction : $\text{Iterations} \geq 1$ ▷ SmoothedFunction (output_control) function_1d-array $\leadsto$ real Smoothed function.
Parallelization Information
smooth_func_1d_mean is <i>reentrant</i> and processed <i>without</i> parallelization.
Possible Predecessors
create_func_1d_array
Alternatives
smooth_func_1d_gauss

Module

Tools

transform_func1d (: : Function, Params : TransformedFunction)

Transform a function using given transformation parameters.

transform_func1d transforms the input function **Function** using the transformation parameters given in **Params**. The function **Function** is passed as a tuple (see **create_func1d_array** and **create_func1d_pairs**). The following model is used for the transformation between the two functions (see **match_func1d_trans**):

$$y_t(x) = a_1 y(a_3 x + a_4) + a_2 \text{ .}$$

The output function **TransformedFunction** is obtained by transforming the x and y values of the input function separately with the above formula, i.e., the output function is not sampled again. Therefore, the parameter a_3 is restricted to $a_3 \neq 0.0$. To resample a function, the operator **sample_func1d** can be used.

Parameter

- ▷ **Function** (input_control) function_1d-array $\leadsto$ real / integer
Input function.
- ▷ **Params** (input_control) number-array $\leadsto$ real
Transformation parameters between the functions.
Number of elements : 4
- ▷ **TransformedFunction** (output_control) function_1d-array $\leadsto$ real / integer
Transformed function.

Parallelization Information

transform_func1d is *reentrant* and processed *without* parallelization.

Possible Predecessors

create_func1d_pairs, **create_func1d_array**, **match_func1d_trans**

Module

Tools

write_func1d (: : Function, FileName :)

Write a function to a file.

The operator **write_func1d** writes the contents of **Function** to a file. The data is written in an ASCII format. Therefore, the file can be exchanged between different architectures. The data can be read by the operator **read_func1d**. There is no specific extension for this kind of file.

Parameter

- ▷ **Function** (input_control) function_1d-array $\leadsto$ real / integer
Function to be written.
- ▷ **FileName** (input_control) filename $\leadsto$ string
Name of the file to be written.

Result

If the parameters are correct the operator **write_func1d** returns the value 2 (H_MSG_TRUE). If the file could not be opened **write_func1d** returns 5 (H_MSG_FAIL). Otherwise an exception handling is raised.

Parallelization Information

write_func1d is *reentrant* and processed *without* parallelization.

Possible Predecessors

create_func1d_pairs, **create_func1d_array**

write_tuple , fwrite_string	Alternatives
read_funcnt_1d , write_image , write_region , open_file	See also
	Module
Basic operators	

x.range_funcnt_1d (: : Function : XMin, XMax)

Smallest and largest x value of the function.

[x.range_funcnt_1d](#) calculates the smallest and the largest x value of [Function](#).

	Parameter
▷ Function (input_control) function_1d-array $\leadsto$ real / integer	Input function.
▷ XMin (output_control) number $\leadsto$ real	Smallest x value.
▷ XMax (output_control) number $\leadsto$ real	Largest x value.

	Parallelization Information
x.range_funcnt_1d is <i>reentrant</i> and processed <i>without</i> parallelization.	

create_funcnt_1d_pairs , create_funcnt_1d_array	Possible Predecessors
	Module
Tools	

y.range_funcnt_1d (: : Function : YMin, YMax)

Smallest and largest y value of the function.

[y.range_funcnt_1d](#) calculates the smallest and the largest y value of [Function](#).

	Parameter
▷ Function (input_control) function_1d-array $\leadsto$ real / integer	Input function.
▷ YMin (output_control) number $\leadsto$ real	Smallest y value.
▷ YMax (output_control) number $\leadsto$ real	Largest y value.

	Parallelization Information
y.range_funcnt_1d is <i>reentrant</i> and processed <i>without</i> parallelization.	

create_funcnt_1d_pairs , create_funcnt_1d_array	Possible Predecessors
	Module
Tools	

15.7 Geometry

angle_ll (: : RowA1, ColumnA1, RowA2, ColumnA2, RowB1, ColumnB1, RowB2, ColumnB2 : Angle)

Calculate the angle between two lines.

The operator `angle_ll` calculates the angle between two lines. As input the rows and columns of the first line (`RowA1,ColumnA1, RowA2,ColumnA2`) and of the second line (`RowB1,ColumnB1, RowB2,ColumnB2`) are expected. The calculation is done as follows: We interpret the lines as vectors with starting points `RowA1,ColumnA1` and `RowB1,ColumnB1` and end points `RowA2,ColumnA2` and `RowB2,ColumnB2`, respectively. Turning the vector *A* counterclockwise onto the vector *B* (the center of rotation is the intersection point of the two lines) yields the angle. The result depends on the order of the points and on the order of the lines. The parameter `Angle` returns the angle in radians. The angles range from $-\pi \leq \text{Angle} \leq \pi$.

Parameter

- ▷ **RowA1** (input_control) point.y(-array) $\leadsto$ *real* / integer
Row of the first point of the first line.
- ▷ **ColumnA1** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the first point of the first line.
- ▷ **RowA2** (input_control) point.y(-array) $\leadsto$ *real* / integer
Row of the second point of the first line.
- ▷ **ColumnA2** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the second point of the first line.
- ▷ **RowB1** (input_control) point.y(-array) $\leadsto$ *real* / integer
Row of the first point of the second line.
- ▷ **ColumnB1** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the first point of the second line.
- ▷ **RowB2** (input_control) point.y(-array) $\leadsto$ *real* / integer
Row of the second point of the second line.
- ▷ **ColumnB2** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the second point of the second line.
- ▷ **Angle** (output_control) number(-array) $\leadsto$ *real*
Angle between the lines.

Example

```

RowA1 := 255
ColumnA1 := 10
RowA2 := 255
ColumnA2 := 501
disp_line (WindowHandle, RowA1, ColumnA1, RowA2, ColumnA2)
RowB1 := 255
ColumnB1 := 255
for i := 1 to 360 by 1
    RowB2 := 255 + sin(rad(i)) * 200
    ColumnB2 := 255 + cos(rad(i)) * 200
    disp_line (WindowHandle, RowB1, ColumnB1, RowB2, ColumnB2)
    angle_ll (RowA1, ColumnA1, RowA2, ColumnA2,
              RowB1, ColumnB1, RowB2, ColumnB2, Angle)
endfor

```

Result

`angle_ll` returns 2 (H_MSG_TRUE).

Parallelization Information

`angle_ll` is *reentrant* and processed *without* parallelization.

Alternatives

`angle_lx`

Module

Basic operators

angle_lx (: : Row1, Column1, Row2, Column2 : Angle)

Calculate the angle between one line and the vertical axis.

The operator `angle_lx` calculates the angle between one line and the abscissa. As input the row and column of the line (`Row1,Column1, Row2,Column2`) are expected. The calculation is done as follows: We interpret the line as a vector with starting point `Row1,Column1` and end point `Row2,Column2`. Turning the vector counter clockwise onto the abscissa (center of rotation is the intersection point of the abscissa) yields the angle. The result is dependant on the order of points of line. The parameters `Angle` returns the angle in radians. The angles range from $-\pi \leq \text{Angle} \leq \pi$.

Parameter

- ▷ **Row1** (input_control) point.y(-array) $\leadsto$ *real* / integer
Row of the first point of the line.
- ▷ **Column1** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the first point of the line.
- ▷ **Row2** (input_control) point.y(-array) $\leadsto$ *real* / integer
Row of the second point of the line.
- ▷ **Column2** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the second point of the line.
- ▷ **Angle** (output_control) number(-array) $\leadsto$ *real*
Angle between the line and the abscissa.

Example

```

RowX1 := 255
ColumnX1 := 10
RowX2 := 255
ColumnX2 := 501
disp_line (WindowHandle, RowX1, ColumnX1, RowX2, ColumnX2)
Row1 := 255
Column1 := 255
for i := 1 to 360 by 1
    Row2 := 255 + sin(rad(i)) * 200
    Column2 := 255 + cos(rad(i)) * 200
    disp_line (WindowHandle, Row1, Column1, Row2, Column2)
    angle_lx (Row1, Column1, Row2, Column2, Angle)
endfor

```

Result

`angle_lx` returns 2 (H_MSG_TRUE).

Parallelization Information

`angle_lx` is *reentrant* and processed *without* parallelization.

Alternatives

`angle_ll`

Module

Basic operators

distance_lr (Region : : Row1, Column1, Row2, Column2 : DistanceMin, DistanceMax)

Calculate the distance between one line and one region.

The operator `distance_lr` calculates the orthogonal distance between one line and one region. As input the coordinates of 2 points that the line represent (`Row1,Column1, Row2,Column2`) and one region are expected. The parameters `DistanceMin` and `DistanceMax` return the result of the calculation.

Attention

Due to efficiency of `distance_lr` holes are ignored. Furthermore, if the lines intersects the region a minimal distance larger than 0.5 can be returned.

Parameter	
▷ Region (input_object)	region $\leadsto$ <i>Hobject</i> Input region.
▷ Row1 (input_control)	point.y(-array) $\leadsto$ <i>real</i> / integer Row of the first point of the line.
▷ Column1 (input_control)	point.x(-array) $\leadsto$ <i>real</i> / integer Column of the first point of the line.
▷ Row2 (input_control)	point.y(-array) $\leadsto$ <i>real</i> / integer Row of the second point of the line.
▷ Column2 (input_control)	point.x(-array) $\leadsto$ <i>real</i> / integer Column of the second point of the line.
▷ DistanceMin (output_control)	number(-array) $\leadsto$ <i>real</i> Minimal distance between the line and the region
▷ DistanceMax (output_control)	number(-array) $\leadsto$ <i>real</i> Maximal distance between the line and the region
Example	

```

dev_close_window ()
read_image (Image, 'fabrik')
dev_open_window (0, 0, 512, 512, 'white', WindowHandle)
threshold (Image, Region, 180, 255)
connection (Region, ConnectedRegions)
select_shape (ConnectedRegions, SelectedRegions, 'area', 'and',
              5000, 100000000)
dev_clear_window ()
dev_set_color ('black')
dev_display (SelectedRegions)
dev_set_color ('red')
Row1 := 100
Row2 := 400
for Col := 50 to 400 by 4
    disp_line (WindowHandle, Row1, Col+100, Row2, Col)
    distance_lr (SelectedRegions, Row1, Col+100, Row2, Col,
                DistanceMin, DistanceMax)
endfor

```

Result

`distance_lr` returns 2 (H_MSG_TRUE).

Parallelization Information

`distance_lr` is *reentrant* and processed *without* parallelization.

Alternatives

`distance_pr`, `distance_sr`, `diameter_region`

See also

`hamming_distance`, `select_region_point`, `test_region_point`, `smallest_rectangle2`

Module

Basic operators

distance_pl (: : Row, Column, Row1, Column1, Row2,
Column2 : Distance)

Calculate the distance between one point and one line.

The operator `distance_pl` calculates the orthogonal distance between a point (**Row,Column**) and a line, given by two arbitrary points of the line. The result is passed in **Distance**.

<i>Parameter</i>	
▷ Row (input_control)	point.y(-array) $\leadsto$ <i>real</i> / integer Row of the point.
▷ Column (input_control)	point.x(-array) $\leadsto$ <i>real</i> / integer Column of the point.
▷ Row1 (input_control)	point.y(-array) $\leadsto$ <i>real</i> / integer Row of the first point of the line.
▷ Column1 (input_control)	point.x(-array) $\leadsto$ <i>real</i> / integer Column of the first point of the line.
▷ Row2 (input_control)	point.y(-array) $\leadsto$ <i>real</i> / integer Row of the second point of the line.
▷ Column2 (input_control)	point.x(-array) $\leadsto$ <i>real</i> / integer Column of the second point of the line.
▷ Distance (output_control)	number(-array) $\leadsto$ <i>real</i> Distance between the points
<i>Example</i>	

```

read_image (Image, 'mreut')
dev_open_window (0, 0, 512, 512, 'white', WindowHandle)
dev_display (Image)
dev_set_color ('black')
threshold (Image, Region, 180, 255)
dev_clear_window ()
dev_display (Region)
connection (Region, ConnectedRegions)
select_shape (ConnectedRegions, SelectedRegions, 'area', 'and',
              10000, 100000000)
get_region_contour (SelectedRegions, Rows, Columns)
RowLine1 := 5
ColLine1 := 300
RowLine2 := 300
ColLine2 := 400
NumberTuple := |Rows|
dev_set_color ('red')
disp_line (WindowHandle, RowLine1, ColLine1, RowLine2, ColLine2)
dev_set_color ('green')
for i := 1 to NumberTuple by 5
    disp_line (WindowHandle, Rows[i], Columns[i]-2, Rows[i], Columns[i]+2)
    disp_line (WindowHandle, Rows[i]-2, Columns[i], Rows[i]+2, Columns[i])
    distance_pl (Rows[i], Columns[i], RowLine1, ColLine1,
                RowLine2, ColLine2, Distance)
endfor

```

<i>Result</i>	
distance_pl	returns 2 (H_MSG_TRUE).
<i>Parallelization Information</i>	
distance_pl	is <i>reentrant</i> and processed <i>without</i> parallelization.
<i>Alternatives</i>	
distance_ps	
<i>See also</i>	
distance_pp , distance_pr	
<i>Module</i>	
Basic operators	

distance_pp (: : Row1, Column1, Row2, Column2 : Distance)

Calculate the distance between two points.

The operator **distance_pp** calculates the distance between pairs of points according to the following formula:

$$\text{Distance} = \sqrt{((\text{Row1} - \text{Row2})^2 + (\text{Column1} - \text{Column2})^2)}$$

The result is passed in **Distance**.

Parameter

- ▷ **Row1** (input_control) point.y(-array) $\leadsto$ real / integer
Row of the first point.
- ▷ **Column1** (input_control) point.x(-array) $\leadsto$ real / integer
Column of the first point.
- ▷ **Row2** (input_control) point.y(-array) $\leadsto$ real / integer
Row of the second point.
- ▷ **Column2** (input_control) point.x(-array) $\leadsto$ real / integer
Column of the second point.
- ▷ **Distance** (output_control) number(-array) $\leadsto$ real
Distance between the points

Example

```
dev_close_window ()
read_image (Image, 'mreut')
dev_open_window (0, 0, 512, 512, 'white', WindowHandle)
dev_display (Image)
dev_set_color ('black')
threshold (Image, Region, 180, 255)
dev_clear_window ()
dev_display (Region)
connection (Region, ConnectedRegions)
select_shape (ConnectedRegions, SelectedRegions, 'area', 'and', 10000, 100000000)
get_region_contour (SelectedRegions, Rows, Columns)
RowPoint := 80
ColPoint := 250
NumberTuple := |Rows|
dev_set_color ('red')
set_draw (WindowHandle, 'margin')
disp_circle (WindowHandle, RowPoint, ColPoint, 10)
dev_set_color ('green')
for i := 1 to NumberTuple by 10
    disp_line (WindowHandle, Rows[i], Columns[i]-2, Rows[i], Columns[i]+2)
    disp_line (WindowHandle, Rows[i]-2, Columns[i], Rows[i]+2, Columns[i])
    distance_pp (RowPoint, ColPoint, Rows[i], Columns[i], Distance)
endfor
```

Result

distance_pp returns 2 (H_MSG_TRUE).

Parallelization Information

distance_pp is *reentrant* and processed *without* parallelization.

Alternatives

distance_ps

See also

distance_pl, **distance_pr**

Module

Basic operators

distance_pr (Region : : Row, Column : DistanceMin, DistanceMax)

Calculate the distance between one point and one region.

The operator **distance_pr** calculates the distance between one point and one region. As input the column and Row of the point (**Row,Column**) and one region are expected. If the point is inside the region the minimal distance is zero. The parameters **DistanceMin** and **DistanceMax** return the result of the calculation.

Parameter

- ▷ **Region** (input_object) region $\leadsto$ *Hobject*
Input region.
- ▷ **Row** (input_control) point.y(-array) $\leadsto$ *real / integer*
Row of the point.
- ▷ **Column** (input_control) point.x(-array) $\leadsto$ *real / integer*
Column of the point.
- ▷ **DistanceMin** (output_control) number(-array) $\leadsto$ *real*
Minimal distance between the point and the region
- ▷ **DistanceMax** (output_control) number(-array) $\leadsto$ *real*
Maximal distance between the point and the region

Example

```
dev_close_window ()
read_image (Image, 'mreut')
dev_open_window (0, 0, 512, 512, 'white', WindowHandle)
dev_set_color ('black')
threshold (Image, Region, 180, 255)
connection (Region, ConnectedRegions)
select_shape (ConnectedRegions, SelectedRegions, 'area', 'and',
              10000, 100000000)

Row1 := 255
Column1 := 255
dev_clear_window ()
dev_display (SelectedRegions)
dev_set_color ('red')
for i := 1 to 360 by 1
    Row2 := 255 + sin(rad(i)) * 200
    Column2 := 255 + cos(rad(i)) * 200
    disp_line (WindowHandle, Row1, Column1, Row2, Column2)
    distance_pr (SelectedRegions, Row2, Column2,
                DistanceMin, DistanceMax)
endfor
```

Result

distance_pr returns 2 (H_MSG_TRUE).

Parallelization Information

distance_pr is *reentrant* and processed *without* parallelization.

Alternatives

distance_lr, **distance_sr**, **diameter_region**

See also

hamming_distance, **select_region_point**, **test_region_point**, **smallest_rectangle2**

Module

Basic operators

distance_ps (: : Row, Column, Row1, Column1, Row2, Column2 : DistanceMin, DistanceMax)

Calculate the distances between a point and a line segment.

The operator `distance_ps` calculates the minimal and maximal distance between a point (`Row,Column`) and a line segment which is represented by the start point (`Row1,Column1`) and the end point (`Row2,Column2`). `DistanceMax` is the maximal distance between the point and the end points of the line segment. `DistanceMin` is identical to `distance_pl` in the case that the point is “between” the two endpoints. Otherwise the minimal distance to one of the endpoints is used.

Parameter

- ▷ **Row** (input_control) point.y(-array) $\leadsto$ *real* / integer
Row of the first point.
- ▷ **Column** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the first point.
- ▷ **Row1** (input_control) point.y(-array) $\leadsto$ *real* / integer
Row of the first point of the line segment.
- ▷ **Column1** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the first point of the line segment.
- ▷ **Row2** (input_control) point.y(-array) $\leadsto$ *real* / integer
Row of the second point of the line segment.
- ▷ **Column2** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the second point of the line segment.
- ▷ **DistanceMin** (output_control) number(-array) $\leadsto$ *real*
Minimal distance between the point and the line segment
- ▷ **DistanceMax** (output_control) number(-array) $\leadsto$ *real*
Maximal distance between the point and the line segment

Example

```
read_image (Image, 'mreut')
dev_open_window (0, 0, 512, 512, 'white', WindowHandle)
dev_display (Image)
dev_set_color ('black')
threshold (Image, Region, 180, 255)
dev_clear_window ()
dev_display (Region)
connection (Region, ConnectedRegions)
select_shape (ConnectedRegions, SelectedRegions, 'area', 'and',
              10000, 100000000)
get_region_contour (SelectedRegions, Rows, Columns)
RowLine1 := 400
ColLine1 := 50
RowLine2 := 50
ColLine2 := 450
NumberTuple := |Rows|
dev_set_color ('red')
disp_line (WindowHandle, RowLine1, ColLine1, RowLine2, ColLine2)
dev_set_color ('green')
for i := 1 to NumberTuple by 10
    disp_line (WindowHandle, Rows[i], Columns[i]-2, Rows[i], Columns[i]+2)
    disp_line (WindowHandle, Rows[i]-2, Columns[i], Rows[i]+2, Columns[i])
    distance_ps (Rows[i], Columns[i], RowLine1, ColLine1, RowLine2, ColLine2,
                DistanceMin, DistanceMax)
endfor
```

Result

`distance_ps` returns 2 (H_MSG_TRUE).

Parallelization Information

`distance_ps` is *reentrant* and processed *without* parallelization.

Alternatives

`distance_pl`

See also

[distance_pp](#), [distance_pr](#)

Module

Basic operators

distance_rr_min (Regions1, Regions2 : : : MinDistance, Row1, Column1, Row2, Column2)

Minimum distance between the contour pixels of two regions each.

The operator [distance_rr_min](#) calculates the minimum distance of pairs of regions. If several regions are passed in [Regions1](#) and [Regions2](#) the distance between the contour pixels of each i-th element is calculated and then forms the i-th entry in the output parameter [MinDistance](#). The calculation is carried out by comparing all contour pixels. The Euclidean distance is used. The parameters ([Row1](#),[Column1](#)) and ([Row2](#), [Column2](#)) indicate the position on the contour of [Regions1](#) and [Regions2](#), respectively, the distance between which is the minimum distance.

Attention

Each region must consist of exactly one connection component. Both input parameters must contain the same number of regions. The regions must not be empty. If the regions overlap the distance is indicated as 0.0. In this case the positions are not reliable.

Parameter

- ▷ **Regions1** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be examined.
- ▷ **Regions2** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be examined.
- ▷ **MinDistance** (output_control) real(-array) $\leadsto$ *real*
Minimum distance between contours of the regions.
Assertion : $0 \leq \text{MinDistance}$
- ▷ **Row1** (output_control) point.y(-array) $\leadsto$ *integer*
Line index on contour in [Regions1](#).
- ▷ **Column1** (output_control) point.x(-array) $\leadsto$ *integer*
Column index on contour in [Regions1](#).
- ▷ **Row2** (output_control) point.y(-array) $\leadsto$ *integer*
Line index on contour in [Regions2](#).
- ▷ **Column2** (output_control) point.x(-array) $\leadsto$ *integer*
Column index on contour in [Regions2](#).

Complexity

If $N1, N2$ are the lengths of the contours the runtime complexity is $O(N1 * N2)$.

Result

The operator [distance_rr_min](#) returns the value 2 (H_MSG.TRUE) if the input is not empty. Otherwise an exception handling is raised.

Parallelization Information

[distance_rr_min](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[threshold](#), [regiongrowing](#), [connection](#)

Alternatives

[distance_rr_min_dil](#), [dilation1](#), [intersection](#)

Module

Region processing

distance_rr_min_dil (Regions1, Regions2 : : : MinDistance)

Minimum distance between two regions with the help of dilatation.

The operator `distance_rr_min_dil` calculates the minimum distance between pairs of regions. If several regions are passed in `Regions1` and `Regions2` the distance between the *i*-th elements in each case is calculated. It then forms the *i*-th entry in the output parameter `MinDistance`. The calculation is carried out with the help of dilatation with the Golay element 'h'. The result is:

$$Numberiterations * 2 - 1$$

The mask 'h' has the effect that precisely the maximum metrics are calculated.

Attention

Both parameters must contain the same number of regions. The regions must not be empty.

Parameter

- ▷ **Regions1** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be examined.
- ▷ **Regions2** (input_object) region(-array) $\leadsto$ *Hobject*
Regions to be examined.
- ▷ **MinDistance** (output_control) integer(-array) $\leadsto$ *integer*
Minimum distances of the regions.
Assertion : $-1 \leq \text{MinDistance}$

Result

The operator `distance_rr_min_dil` returns the value 2 (H_MSG_TRUE) if the input is not empty. Otherwise an exception handling is raised.

Parallelization Information

`distance_rr_min_dil` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`threshold`, `regiongrowing`, `connection`

Alternatives

`distance_rr_min`, `dilation1`, `intersection`

Module

Region processing

distance_sl (: : RowA1, ColumnA1, RowA2, ColumnA2, RowB1, ColumnB1, RowB2, ColumnB2 : DistanceMin, DistanceMax)

Calculate the distances between one line segment and one line.

The operator `distance_sl` calculates the minimal and maximal orthogonal distance between one line segment and one line. As input the columns and rows of the line segment (`RowA1,ColumnA1,RowA2,ColumnA2`) and of the line (`RowB1,ColumnB1,RowB2,ColumnB2`) are expected. The parameters `DistanceMin` and `DistanceMax` return the result of the calculation. If the line segments are intersecting `DistanceMin` returns zero.

Parameter

- ▷ **RowA1** (input_control) point.y(-array) $\leadsto$ *real* / *integer*
Row of the first point of the line segment.
- ▷ **ColumnA1** (input_control) point.x(-array) $\leadsto$ *real* / *integer*
Column of the first point of the line segment.
- ▷ **RowA2** (input_control) point.y(-array) $\leadsto$ *real* / *integer*
Row of the second point of the line segment.
- ▷ **ColumnA2** (input_control) point.x(-array) $\leadsto$ *real* / *integer*
Column of the second point of the line segment.
- ▷ **RowB1** (input_control) point.y(-array) $\leadsto$ *real* / *integer*
Row of the first point of the line.

- ▷ **ColumnB1** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the first point of the line.
- ▷ **RowB2** (input_control) point.y(-array) $\leadsto$ *real* / integer
Row of the second point of the line.
- ▷ **ColumnB2** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the second point of the line.
- ▷ **DistanceMin** (output_control) number(-array) $\leadsto$ *real*
Minimal distance between the line segment and the line
- ▷ **DistanceMax** (output_control) number(-array) $\leadsto$ *real*
Maximal distance between the line segment and the line

Example

```
dev_set_color ('black')
RowLine1 := 400
ColLine1 := 200
RowLine2 := 200
ColLine2 := 400
Rows := 300
Columns := 50
disp_line (WindowHandle, RowLine1, ColLine1, RowLine2, ColLine2)
dev_set_color ('green')
n := 0
for Rows := 40 to 200 by 4
    disp_line (WindowHandle, Rows+n, Columns+n, Rows, Columns+n)
    distance_sl (Rows+n, Columns+n, Rows, Columns+n, RowLine1, ColLine1,
                RowLine2, ColLine2, DistanceMin, DistanceMax)
    n := n+10
endfor
```

Result

`distance_sl` returns 2 (H_MSG_TRUE).

Parallelization Information

`distance_sl` is *reentrant* and processed *without* parallelization.

Alternatives

`distance_pl`

See also

`distance_ps`, `distance_pp`

Module

Basic operators

distance_sr (Region : : Row1, Column1, Row2, Column2 : DistanceMin, DistanceMax)

Calculate the distance between one line segment and one region.

The operator `distance_sr` calculates the distance between one line segment and one region. `Row1`, `Column1`, `Row2`, `Column2` are the the initial and end coordinates of the line segment. The parameters `DistanceMin` and `DistanceMax` contain the resulting distances.

Attention

Due to efficiency of `distance_sr` holes are ignored. Furthermore, if the lines intersects the region a minimal distance larger than 0.5 can be returned.

Parameter	
▷ Region (input_object)	region $\leadsto$ <i>Hobject</i> Input region.
▷ Row1 (input_control)	point.y(-array) $\leadsto$ <i>real</i> / integer Row of the first point of the line segment.
▷ Column1 (input_control)	point.x(-array) $\leadsto$ <i>real</i> / integer Column of the first point of the line segment.
▷ Row2 (input_control)	point.y(-array) $\leadsto$ <i>real</i> / integer Row of the second point of the line segment.
▷ Column2 (input_control)	point.x(-array) $\leadsto$ <i>real</i> / integer Column of the second point of the line segment.
▷ DistanceMin (output_control)	number(-array) $\leadsto$ <i>real</i> Minimal distance between the line segment and the region
▷ DistanceMax (output_control)	number(-array) $\leadsto$ <i>real</i> Maximal distance between the line segment and the region
Example	

```

read_image (Image, 'fabrik')
dev_open_window (0, 0, 512, 512, 'white', WindowHandle)
dev_display (Image)
threshold (Image, Region, 180, 255)
connection (Region, ConnectedRegions)
select_shape (ConnectedRegions, SelectedRegions, 'area', 'and',
              5000, 100000000)
dev_clear_window ()
dev_set_color ('black')
dev_display (SelectedRegions)
dev_set_color ('red')
Row1 := 100
Row2 := 400
n := 0
for Col := 50 to 400 by 5
    disp_line (WindowHandle, Row1+n, Col, Row2-n, Col+100)
    distance_sr (SelectedRegions, Row1+n, Col, Row2-n, Col+100,
                DistanceMin, DistanceMax)
    n := n+5
endfor

```

Result

[distance_sr](#) returns 2 (H_MSG_TRUE).

Parallelization Information

[distance_sr](#) is *reentrant* and processed *without* parallelization.

Alternatives

[distance_lr](#), [distance_pr](#), [diameter_region](#)

See also

[hamming_distance](#), [select_region_point](#), [test_region_point](#), [smallest_rectangle2](#)

Module

Basic operators

distance_ss (:: RowA1, ColumnA1, RowA2, ColumnA2, RowB1, ColumnB1,
RowB2, ColumnB2 : DistanceMin, DistanceMax)

Calculate the distances between two line segments.

The operator [distance_ss](#) calculates the minimal and maximal distance between two line segments. As input the rows and columns of the first line segments ([RowA1](#),[ColumnA1](#), [RowA2](#),[ColumnA2](#)) and of the

second line segment (**RowB1**,**ColumnB1**,**RowB2**,**ColumnB2**) are used. The parameters **DistanceMin** and **DistanceMax** return the result of the calculation. If the line segments are intersecting **DistanceMin** returns zero.

Parameter	
▷ RowA1 (input_control)	point.y(-array) $\leadsto$ real / integer Row of the first point of the line segment.
▷ ColumnA1 (input_control)	point.x(-array) $\leadsto$ real / integer Column of the first point of the line segment.
▷ RowA2 (input_control)	point.y(-array) $\leadsto$ real / integer Row of the second point of the line segment.
▷ ColumnA2 (input_control)	point.x(-array) $\leadsto$ real / integer Column of the second point of the line segment.
▷ RowB1 (input_control)	point.y(-array) $\leadsto$ real / integer Row of the first point of the line.
▷ ColumnB1 (input_control)	point.x(-array) $\leadsto$ real / integer Column of the first point of the line.
▷ RowB2 (input_control)	point.y(-array) $\leadsto$ real / integer Row of the second point of the line.
▷ ColumnB2 (input_control)	point.x(-array) $\leadsto$ real / integer Column of the second point of the line.
▷ DistanceMin (output_control)	number(-array) $\leadsto$ real Minimal distance between the line segments
▷ DistanceMax (output_control)	number(-array) $\leadsto$ real Maximal distance between the line segments
Example	

```
dev_set_color ('black')
RowLine1 := 400
ColLine1 := 200
RowLine2 := 240
ColLine2 := 400
Rows := 300
Columns := 50
disp_line (WindowHandle, RowLine1, ColLine1, RowLine2, ColLine2)
dev_set_color ('red')
n := 0
for Rows := 40 to 200 by 4
    disp_line (WindowHandle, Rows, Columns, Rows+n, Columns+n)
    distance_ss (Rows, Columns, Rows+n, Columns+n, RowLine1, ColLine1,
                RowLine2, ColLine2, DistanceMin, DistanceMax)
    n := n+8
endfor
```

Result
distance_ss returns 2 (H_MSG_TRUE).

Parallelization Information
distance_ss is <i>reentrant</i> and processed <i>without</i> parallelization.

Alternatives
distance_pp

See also
distance_pl , distance_ps

Module
Basic operators

get_points_ellipse (: : Angle, Row, Column, Phi, Radius1, Radius2 : RowPoint, ColPoint)

Points of an ellipse corresponding to specific angles.

get_points_ellipse returns the points (**RowPoint**, **ColPoint**) on the specified ellipse corresponding to the angles in **Angle**, which refer to the main axis of the ellipse. The ellipse itself is characterized by the center (**Row**, **Column**), the orientation of the main axis **Phi**, the length of the larger half axis **Radius1**, and the length of the smaller half axis **Radius2**.

Parameter

- ▷ **Angle** (input_control) real(-array) $\leadsto$ real
Angles corresponding to the resulting points [rad].
Default Value : 0
Restriction : $(\text{Angle} \geq 0) \wedge (\text{Angle} \leq 6.283185307)$
- ▷ **Row** (input_control) ellipse.center.y $\leadsto$ real
Row coordinate of the center of the ellipse.
- ▷ **Column** (input_control) ellipse.center.x $\leadsto$ real
Column coordinate of the center of the ellipse.
- ▷ **Phi** (input_control) ellipse.angle.rad $\leadsto$ real
Orientation of the main axis [rad].
Restriction : $(\text{Phi} \geq 0) \wedge (\text{Phi} \leq 6.283185307)$
- ▷ **Radius1** (input_control) ellipse.radius1 $\leadsto$ real
Length of the larger half axis.
Restriction : $\text{Radius1} > 0$
- ▷ **Radius2** (input_control) ellipse.radius2 $\leadsto$ real
Length of the smaller half axis.
Restriction : $\text{Radius2} \geq 0$
- ▷ **RowPoint** (output_control) point.row(-array) $\leadsto$ real
Row coordinates of the points on the ellipse.
- ▷ **ColPoint** (output_control) point.column(-array) $\leadsto$ real
Column coordinates of the points on the ellipse.

Example

```
draw_ellipse(WindowHandle, Row, Column, Phi, Radius1, Radius2)
```

```
get_points_ellipse([0, 3.14], Row, Column, Phi, Radius1, Radius2, RowPoint, ColPoint)
```

Result

get_points_ellipse returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Parallelization Information

get_points_ellipse is *reentrant* and processed *without* parallelization.

Possible Predecessors

fit_ellipse_contour_xld, **draw_ellipse**, **gen_ellipse_contour_xld**

See also

gen_ellipse_contour_xld

Module

Basic operators

intersection_ll (: : RowA1, ColumnA1, RowA2, ColumnA2, RowB1, ColumnB1, RowB2, ColumnB2 : Row, Column, IsParallel)

Calculate the intersection point of two lines.

The operator **intersection_ll** calculates the intersection point of two lines. As input the columns and rows of the lines (**RowA1**, **ColumnA1**, **RowA2**, **ColumnA2**) and (**RowB1**, **ColumnB1**, **RowB2**, **ColumnB2**) are expected.

The parameters **Row** and **Column** return the result of the calculation. If the lines are parallel **IsParallel** is 1 else 0. In addition the values of **Row** and **Column** are undefined.

Attention

If the lines are parallel the values of **Row** and **Column** are undefined.

Parameter

- ▷ **RowA1** (input_control) point.y(-array) $\leadsto$ *real* / integer
Row of the first point of the first line.
- ▷ **ColumnA1** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the first point of the first line.
- ▷ **RowA2** (input_control) point.y(-array) $\leadsto$ *real* / integer
Row of the second point of the first line.
- ▷ **ColumnA2** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the second point of the first line.
- ▷ **RowB1** (input_control) point.y(-array) $\leadsto$ *real* / integer
Row of the first point of the second line.
- ▷ **ColumnB1** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the first point of the second line.
- ▷ **RowB2** (input_control) point.y(-array) $\leadsto$ *real* / integer
Row of the second point of the second line.
- ▷ **ColumnB2** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the second point of the second line.
- ▷ **Row** (output_control) point.y(-array) $\leadsto$ *real*
Row of the intersection point
- ▷ **Column** (output_control) point.x(-array) $\leadsto$ *real*
Column of the intersection point
- ▷ **IsParallel** (output_control) number(-array) $\leadsto$ *integer*
Are the two lines parallel?

Example

```
dev_set_color ('black')
RowLine1 := 350
ColLine1 := 250
RowLine2 := 300
ColLine2 := 300
Rows := 300
Columns := 50
disp_line (WindowHandle, RowLine1, ColLine1, RowLine2, ColLine2)
n := 0
for Rows := 40 to 200 by 4
    dev_set_color ('red')
    disp_line (WindowHandle, Rows, Columns, Rows+n, Columns+n)
    intersection_ll (Rows, Columns, Rows+n, Columns+n, RowLine1, ColLine1,
                    RowLine2, ColLine2, Row, Column, IsParallel)
    dev_set_color ('blue')
    disp_line (WindowHandle, Row, Column-2, Row, Column+2)
    disp_line (WindowHandle, Row-2, Column, Row+2, Column)
    n := n+8
endfor
```

Result

intersection_ll returns 2 (H_MSG_TRUE).

Parallelization Information

intersection_ll is *reentrant* and processed *without* parallelization.

Module

Basic operators

projection_pl (: : Row, Column, Row1, Column1, Row2,
Column2 : RowProj, ColProj)

Calculate the projection of a point onto a line.

The operator **projection_pl** calculates the projection of a point (**Row,Column**) onto a line which is represent by the start point (**Row1,Column1**) and the end point (**Row2,Column2**). **RowProj** is the row of the projection point and **ColProj** is the column of the projection point.

Parameter

- ▷ **Row** (input_control) point.y(-array) $\leadsto$ *real* / integer
Row of the point.
- ▷ **Column** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the point.
- ▷ **Row1** (input_control) point.y(-array) $\leadsto$ *real* / integer
Row of the first point of the line.
- ▷ **Column1** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the first point of the line.
- ▷ **Row2** (input_control) point.y(-array) $\leadsto$ *real* / integer
Row of the second point of the line.
- ▷ **Column2** (input_control) point.x(-array) $\leadsto$ *real* / integer
Column of the second point of the line.
- ▷ **RowProj** (output_control) number(-array) $\leadsto$ *real*
Row of the projection
- ▷ **ColProj** (output_control) number(-array) $\leadsto$ *real*
Column of the projection

Example

```
dev_set_color ('black')
RowLine1 := 400
ColLine1 := 200
RowLine2 := 240
ColLine2 := 400
Rows := 300
Columns := 50
disp_line (WindowHandle, RowLine1, ColLine1, RowLine2, ColLine2)
n := 0
for Rows := 40 to 200 by 4
    dev_set_color ('red')
    disp_circle (WindowHandle, Rows+n, Columns, 2)
    projection_pl (Rows+n, Columns, RowLine1, ColLine1, RowLine2, ColLine2,
                  RowProj, ColProj)
    dev_set_color ('blue')
    disp_line (WindowHandle, RowProj-2, ColProj, RowProj+2, ColProj)
    disp_line (WindowHandle, RowProj, ColProj-2, RowProj, ColProj+2)
    n := n+8
endfor
```

Result

projection_pl returns 2 (H_MSG_TRUE).

Parallelization Information

projection_pl is *reentrant* and processed *without* parallelization.

Module

Basic operators

15.8 Hough

hough_circle_trans (Region : HoughImage : Radius :)

Return the Hough-Transform for circles with a given radius.

The operator **hough_circle_trans** calculates the Hough transform for circles with a certain **Radius** in the regions passed by **Region**. Hereby the centres of all possible circles in the parameter space (the Hough or accumulator space respectively) will be accumulated for each point in the image space. Circle hypotheses supported by many points in the input region thereby generate a maximum in the area showing the circle's centre in the output image (**HoughImage**). The circles' centres in the image space can be deduced from the coordinates of these maximums by subtracting the **Radius**. If more than one radius is transmitted, all Hough images will be shifted according to the maximal radius.

Parameter

- ▷ **Region** (input_object) region $\leadsto$ Hobject
Binary edge image in which the circles are to be detected.
- ▷ **HoughImage** (output_object) image(-array) $\leadsto$ Hobject : int2
Hough transform for circles with a given radius.
Number of elements : HoughImage = Radius
- ▷ **Radius** (input_control) integer(-array) $\leadsto$ integer
Radius of the circle to be searched in the image.
Default Value : 12
Typical range of values : $3 \leq \text{Radius} \text{ (lin)}$
Minimum Increment : 1
Recommended Increment : 1
Number of elements : $(1 \leq \text{Radius}) \leq 500$

Result

The operator **hough_circle_trans** returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator **set_system** ('no_object_result', <Result>), the behavior in case of empty region is set via **set_system** ('empty_region_result', <Result>). If necessary an exception handling is raised.

Parallelization Information

hough_circle_trans is *reentrant* and processed *without* parallelization.

Module

Region processing

hough_circles (RegionIn : RegionOut : Radius, Percent, Mode :)

Centres of circles for a specific radius.

hough_circle_trans detects the centres of circles in regions with the help of the Hough transform for circles with a specific radius.

Parameter

- ▷ **RegionIn** (input_object) region $\leadsto$ Hobject
Binary edge image in which the circles are to be detected.
- ▷ **RegionOut** (output_object) region(-array) $\leadsto$ Hobject
Centres of those circles which are included in the edge image by **Percent** percent.
Number of elements : RegionOut = ((Radius · Percent) · Mode)
- ▷ **Radius** (input_control) integer(-array) $\leadsto$ integer
Radius of the circle to be searched in the image.
Default Value : 12
Typical range of values : $2 \leq \text{Radius} \leq 500 \text{ (lin)}$
Minimum Increment : 1
Recommended Increment : 1
Number of elements : $(1 \leq \text{Radius}) \leq 500$

- ▷ **Percent** (input_control) integer(-array) $\leadsto$ integer
 Indicates the percentage (approximately) of the (ideal) circle which must be present in the edge image **RegionIn**.
Default Value : 60
Typical range of values : $10 \leq \text{Percent} \leq 100$ (lin)
Minimum Increment : 1
Recommended Increment : 5
Number of elements : $(1 \leq \text{Percent}) \leq 100$
- ▷ **Mode** (input_control) integer(-array) $\leadsto$ integer
 The modus defines the position of the circle in question:
 0 - the radius is equivalent to the outer border of the set pixels.
 1 - the radius is equivalent to the centres of the circle lines' pixels.
 2 - both 0 and 1 (a little more fuzzy, but more reliable in contrast to circles set slightly differently, necessitates 50 % more processing capacity compared to 0 or 1 alone).
List of values : $\text{Mode} \in \{0, 1, 2\}$
Number of elements : $(1 \leq \text{Mode}) \leq 3$

Result

The operator **hough_circles** returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator **set_system** ('no_object_result', <Result>), the behavior in case of empty region is set via **set_system** ('empty_region_result', <Result>). If necessary an exception handling is raised.

Parallelization Information

hough_circles is *reentrant* and processed *without* parallelization.

Module

Region processing

hough_line_trans (Region : HoughImage : AngleResolution :)

Produce the Hough transform for lines within regions.

The operator **hough_line_trans** calculates the Hough transform for lines in those regions transmitted by **Region**. Thereby the angles and the lengths of the lines' normal vectors are registered in the parameter space (the Hough- or accumulator space respectively). This means that the parameterization is executed according to the HNF.

The result is registered in a newly generated Int2-Image (**HoughImage**), whereby the x-axis is equivalent to the angle between the normal vector and the x-axis (in the original image), and the y-axis is equivalent to the distance of the line from the origin.

The angle ranges from -90 to 180 degrees and will be registered with a resolution of $1/\text{AngleResolution}$, which means that one pixel in x-direction is equivalent to $1/\text{AngleResolution}$ and that the **HoughImage** has a width of $270 * \text{AngleResolution} + 1$ pixel. The height of the **HoughImage** corresponds to the diagonal of the surrounding rectangle of the input region.

The maxima in the result image are equivalent to the parameter values of the lines in the original image.

Parameter

- ▷ **Region** (input_object) region $\leadsto$ Hobject
 Binary edge image in which lines are to be detected.
- ▷ **HoughImage** (output_object) image $\leadsto$ Hobject : int2
 Hough transform for lines.
- ▷ **AngleResolution** (input_control) integer $\leadsto$ integer
 Adjusting the resolution in the angle area.
Default Value : 4
List of values : $\text{AngleResolution} \in \{1, 2, 4, 8\}$

Result

The operator **hough_line_trans** returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator **set_system**

(`'no_object_result', <Result>`), the behavior in case of empty region is set via `set_system('empty_region_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`hough_line_trans` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`threshold`, `skeleton`

Possible Successors

`threshold`, `local_max`

See also

`hough_circle_trans`, `gen_region_hline`

Module

Region processing

`hough_line_trans_dir` (`ImageDir` : `HoughImage` : `DirectionUncertainty`,
`AngleResolution` :)

Compute the Hough transform for lines using local gradient direction.

The operator `hough_line_trans_dir` calculates the Hough transform for lines in those regions passed in the domain of `ImageDir`. To do so, the angles and the lengths of the lines' normal vectors are registered in the parameter space (the so-called Hough or accumulator space).

In contrast to `hough_line_trans`, additionally the edge direction in `ImageDir` (e.g., returned by `sobel_dir` or `edges_image`) is taken into account. This results in a more efficient computation and in a reduction of the noise in the Hough space.

The parameter `DirectionUncertainty` describes how much the edge direction of the individual points within a line is allowed to vary. For example, with `DirectionUncertainty` = 10 a horizontal line (i.e., edge direction = 0 degrees) may contain points with an edge direction between -10 and +10 degrees. The higher `DirectionUncertainty` is chosen, the higher the computation time will be. For `DirectionUncertainty` = 180 `hough_line_trans_dir` shows the same behavior as `hough_line_trans`, i.e., the edge direction is ignored. `DirectionUncertainty` should be chosen at least as high as the step width of the edge direction stored in `ImageDir`. The minimum step width is 2 degrees (defined by the image type 'direction').

The result is stored in a newly generated UINT2-Image (`HoughImage`), where the x-axis (i.e., columns) represents the angle between the normal vector and the x-axis of the original image, and the y-axis (i.e., rows) represents the distance of the line from the origin.

The angle ranges from -90 to 180 degrees and will be stored with a resolution of $1/\text{AngleResolution}$, which means that one pixel in x-direction is equivalent to $1/\text{AngleResolution}$ degrees and that the `HoughImage` has a width of $270 * \text{AngleResolution} + 1$ pixels. The height of the `HoughImage` corresponds to the length of the diagonal of the surrounding rectangle of the input region.

The local maxima in the result image are equivalent to the parameter values of the lines in the original image.

Parameter

- ▷ **`ImageDir`** (input_object) image $\rightsquigarrow$ *Hobject* : direction
Image containing the edge direction. The edges must be described by the image domain.
- ▷ **`HoughImage`** (output_object) image $\rightsquigarrow$ *Hobject* : uint2
Hough transform.
- ▷ **`DirectionUncertainty`** (input_control) angle.deg $\rightsquigarrow$ *integer*
Uncertainty of the edge direction (in degrees).
Default Value : 2
Typical range of values : $2 \leq \text{DirectionUncertainty} \leq 180$
Minimum Increment : 2
- ▷ **`AngleResolution`** (input_control) integer $\rightsquigarrow$ *integer*
Resolution in the angle area (in 1/degrees).
Default Value : 4
List of values : $\text{AngleResolution} \in \{1, 2, 4, 8\}$

Result

The operator `hough_line_trans_dir` returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input is set via the operator `set_system('no_object_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`hough_line_trans_dir` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`edges_image`, `sobel_dir`, `threshold`, `hysteresis_threshold`, `nonmax_suppression_dir`, `reduce_domain`

Possible Successors

`gauss_image`, `threshold`, `local_max`, `plateaus_center`

See also

`hough_line_trans`, `hough_lines`, `hough_lines_dir`

Module

Tools

hough_lines (`RegionIn` : : `AngleResolution`, `Threshold`, `AngleGap`,
`DistGap` : `Angle`, `Dist`)

Detect lines in edge images with the help of the Hough transform and returns it in HNF.

The operator `hough_lines` allows the selection of linelike structures in a region, whereby it is not necessary that the individual points of a line are connected. This process is based on the Hough transform. The lines are returned in HNF, that is by the direction and length of their normal vector.

The parameter `AngleResolution` defines the degree of exactness concerning the determination of the angles. It amounts to $1/\text{AngleResolution}$ degree. The parameter `Threshold` determines by how many points of the original region a line's hypothesis has to be supported at least in order to be taken over into the output. The parameters `AngleGap` and `DistGap` define a neighborhood of the points in the Hough image in order to determine the local maxima. The lines are returned in HNF.

Parameter

- ▷ **RegionIn** (input_object) region $\leadsto$ *Hobject*
Binary edge image in which the lines are to be detected.
- ▷ **AngleResolution** (input_control) integer $\leadsto$ *integer*
Adjusting the resolution in the angle area.
Default Value : 4
List of values : `AngleResolution` $\in \{1, 2, 4, 8\}$
- ▷ **Threshold** (input_control) integer $\leadsto$ *integer*
Threshold value in the Hough image.
Default Value : 100
Typical range of values : $1 \leq \text{Threshold}$
- ▷ **AngleGap** (input_control) integer $\leadsto$ *integer*
Minimal distance of two maxima in the Hough image (direction: angle).
Default Value : 5
Typical range of values : $0 \leq \text{AngleGap}$
- ▷ **DistGap** (input_control) integer $\leadsto$ *integer*
Minimal distance of two maxima in the Hough image (direction: distance).
Default Value : 5
Typical range of values : $0 \leq \text{DistGap}$
- ▷ **Angle** (output_control) `hesseline.angle.rad-array` $\leadsto$ *real*
Angles (in radians) of the detected lines' normal vectors.
Typical range of values : $-1.5707963 \leq \text{Angle} \leq 3.1415927$
- ▷ **Dist** (output_control) `hesseline.distance-array` $\leadsto$ *real*
Distance of the detected lines from the origin.
Typical range of values : $0 \leq \text{Dist}$
Number of elements : `Dist` = `Angle`

Result

The operator `hough_lines` returns the value 2 (`H_MSG.TRUE`) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result', <Result>)`, the behavior in case of empty region is set via `set_system('empty_region_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`hough_lines` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`threshold`, `skeleton`

Possible Successors

`select_matching_lines`

See also

`hough_line_trans`, `gen_region_hline`, `hough_circles`

Module

Region processing

hough_lines_dir (ImageDir : HoughImage, Lines : DirectionUncertainty, AngleResolution, Smoothing, FilterSize, Threshold, AngleGap, DistGap, GenLines : Angle, Dist)

Detect lines in edge images with the help of the Hough transform using local gradient direction and return them in normal form.

The operator `hough_lines_dir` selects line-like structures in a region based on the Hough transform. The individual points of a line can be unconnected. The region is given by the domain of `ImageDir`. The lines are returned in Hessian normal form (HNF), that is by the direction and length of their normal vector.

In contrast to `hough_lines`, additionally the edge direction in `ImageDir` (e.g., returned by `sobel_dir` or `edges_image`) is taken into account. This results in a more efficient computation and in a reduction of the noise in the Hough space.

The parameter `DirectionUncertainty` describes how much the edge direction of the individual points within a line is allowed to vary. For example, with `DirectionUncertainty = 10` a horizontal line (i.e., edge direction = 0 degrees) may contain points with an edge direction between -10 and +10 degrees. The higher `DirectionUncertainty` is chosen, the higher the computation time will be. For `DirectionUncertainty = 180` `hough_lines_dir` shows the same behavior as `hough_lines`, i.e., the edge direction is ignored. `DirectionUncertainty` should be chosen at least as high as the step width of the edge direction stored in `ImageDir`. The minimum step width is 2 degrees (defined by the image type 'direction').

The parameter `AngleResolution` defines how accurately the angles are determined. The accuracy amounts to $1/\text{AngleResolution}$ degrees. A subsequent smoothing of the Hough space results in an increased stability. The smoothing filter can be selected by `Smoothing`, the degree of smoothing by the parameter `FilterSize` (see `mean_image` or `gauss_image` for details). The parameter `Threshold` determines by how many points of the original region a line's hypothesis must at least be supported in order to be selected into the output. The parameters `AngleGap` and `DistGap` define a neighborhood of the points in the Hough image in order to determine the local maxima: `AngleGap` describes the minimum distance of two maxima in the Hough image in angle direction and `DistGap` in distance direction, respectively. Thus, maxima exceeding `Threshold` but lying close to an even higher maximum are eliminated. This can particularly be helpful when searching for short and long lines simultaneously. Besides the unsmoothed Hough image `HoughImage`, the lines are returned in HNF (`Angle`, `Dist`). If the parameter `GenLines` is set to 'true', additionally those regions in `ImageDir` are returned that contributed to the local maxima in Hough space. They are stored in the parameter `Lines`.

Parameter

- ▷ **ImageDir** (input_object) image $\leadsto$ Hobject : direction
Image containing the edge direction. The edges are described by the image domain.
- ▷ **HoughImage** (output_object) image $\leadsto$ Hobject : uint2
Hough transform.
- ▷ **Lines** (output_object) region-array $\leadsto$ Hobject
Regions of the input image that contributed to the local maxima.

- ▷ **DirectionUncertainty** (input_control) angle.deg $\leadsto$ integer
 Uncertainty of edge direction (in degrees).
Default Value : 2
Typical range of values : $2 \leq \text{DirectionUncertainty} \leq 180$
Minimum Increment : 2
- ▷ **AngleResolution** (input_control) integer $\leadsto$ integer
 Resolution in the angle area (in 1/degrees).
Default Value : 4
List of values : AngleResolution $\in \{1, 2, 4, 8\}$
- ▷ **Smoothing** (input_control) string $\leadsto$ string
 Smoothing filter for hough image.
Default Value : 'mean'
List of values : Smoothing $\in \{\text{'none'}, \text{'mean'}, \text{'gauss'}\}$
- ▷ **FilterSize** (input_control) integer $\leadsto$ integer
 Required smoothing filter size.
Default Value : 5
List of values : FilterSize $\in \{3, 5, 7, 9, 11\}$
- ▷ **Threshold** (input_control) integer $\leadsto$ integer
 Threshold value in the Hough image.
Default Value : 100
Typical range of values : $1 \leq \text{Threshold}$
- ▷ **AngleGap** (input_control) integer $\leadsto$ integer
 Minimum distance of two maxima in the Hough image (direction: angle).
Default Value : 5
Typical range of values : $0 \leq \text{AngleGap}$
- ▷ **DistGap** (input_control) integer $\leadsto$ integer
 Minimum distance of two maxima in the Hough image (direction: distance).
Default Value : 5
Typical range of values : $0 \leq \text{DistGap}$
- ▷ **GenLines** (input_control) string $\leadsto$ string
 Create line regions if 'true'.
Default Value : 'true'
List of values : GenLines $\in \{\text{'true'}, \text{'false'}\}$
- ▷ **Angle** (output_control) hesseline.angle.rad-array $\leadsto$ real
 Angles (in radians) of the detected lines' normal vectors.
Typical range of values : $-1.5707963 \leq \text{Angle} \leq 3.1415927$
- ▷ **Dist** (output_control) hesseline.distance-array $\leadsto$ real
 Distance of the detected lines from the origin.
Typical range of values : $0 \leq \text{Dist}$
Number of elements : Dist = Angle

Result

The operator `hough_lines` returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no-object.result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`hough_lines_dir` is *reentrant* and processed *without parallelization*.

Possible Predecessors

`edges_image`, `sobel_dir`, `threshold`, `nonmax_suppression_dir`, `reduce_domain`, `skeleton`

Possible Successors

`gen_region_hline`, `select_matching_lines`

See also

`hough_line_trans_dir`, `hough_line_trans`, `gen_region_hline`, `hough_circles`

Module

Region processing

select_matching_lines (RegionIn : RegionLines : AngleIn, DistIn, Linewidth, Thresh : AngleOut, DistOut)

Select those lines from a set of lines (in HNF) which fit best into a region.

Lines which fit best into a region can be selected from a set of lines which are available in HNF with the help of the operator `select_matching_lines`; the region itself is also transmitted as a parameter (`RegionIn`). The width of the lines can be indicated by the parameter `Linewidth`. The selected lines will be returned in HNF and as regions (`RegionLines`).

The lines are selected iteratively in a loop: At first, the line showing the greatest overlap with the input region is selected from the set of input lines. This line will then be taken over into the output set whereby all points belonging to that line will not be considered in the further steps determining overlaps. The loop will be left when the maximum overlap value of the region and the lines falls below a certain threshold value (`Thresh`). The selected lines will be returned as regions as well as in HNF.

Parameter

- ▷ **RegionIn** (input_object) region $\leadsto$ *Hobject*
Region in which the lines are to be matched.
- ▷ **RegionLines** (output_object) region(-array) $\leadsto$ *Hobject*
Region array containing the matched lines.
- ▷ **AngleIn** (input_control) hesseline.angle.rad(-array) $\leadsto$ *real*
Angles (in radians) of the normal vectors of the input lines.
Typical range of values : $-1.5707963 \leq \text{AngleIn} \leq 3.1415927$
- ▷ **DistIn** (input_control) hesseline.distance(-array) $\leadsto$ *real*
Distances of the input lines from the origin.
Number of elements : DistIn = AngleIn
- ▷ **Linewidth** (input_control) integer $\leadsto$ *integer*
Widths of the lines.
Default Value : 7
Typical range of values : $1 \leq \text{Linewidth}$
- ▷ **Thresh** (input_control) integer $\leadsto$ *integer*
Threshold value for the number of line points in the region.
Default Value : 100
Typical range of values : $1 \leq \text{Thresh}$
- ▷ **AngleOut** (output_control) hesseline.angle.rad(-array) $\leadsto$ *real*
Angles (in radians) of the normal vectors of the selected lines.
Typical range of values : $-1.5707963 \leq \text{AngleOut} \leq 3.1415927$
Number of elements : AngleOut $\leq$ AngleIn
- ▷ **DistOut** (output_control) hesseline.distance(-array) $\leadsto$ *real*
Distances of the selected lines from the origin.
Typical range of values : $0 \leq \text{DistOut}$
Number of elements : DistOut = AngleOut

Result

The operator `select_matching_lines` returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input regions available) is set via the operator `set_system('no_object_result', <Result>)`, the behavior in case of empty region is set via `set_system('empty_region_result', <Result>)`. If necessary an exception handling is raised.

Parallelization Information

`select_matching_lines` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`hough_lines`

Module

Region processing

15.9 Image-Comparison

clear_all_variation_models (: : :)

Free the memory of all variation models.

`clear_all_variation_models` frees the memory of all variation models that were created by calling `create_variation_model`. After calling `clear_all_variation_models`, no model can be used any longer.

Result

`clear_all_variation_models` always returns 2 (H_MSG_TRUE).

Parallelization Information

`clear_all_variation_models` is processed *completely exclusively* without parallelization.

Possible Predecessors

`create_variation_model`

Alternatives

`clear_variation_model`

Module

Image / region / XLD management

clear_variation_model (: : ModelID :)

Free the memory of a variation model.

`clear_variation_model` frees the memory of a variation model that was created by `create_variation_model`. After calling `clear_variation_model`, the model can no longer be used. The handle `ModelID` becomes invalid.

Parameter

▷ **ModelID** (input_control) variation_model $\leadsto$ integer
ID of the variation model.

Result

`clear_variation_model` returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

`clear_variation_model` is processed *completely exclusively* without parallelization.

Possible Predecessors

`create_variation_model`

Alternatives

`clear_all_variation_models`

Module

Image / region / XLD management

compare_variation_model (Image : Region : ModelID :)

Compare an image to a variation model.

`compare_variation_model` compares the input image `Image` to the variation model given by `ModelID`. Before `compare_variation_model` can be called, the two internal threshold images of the variation model must have been created with `prepare_variation_model`. Let $c(x, y)$ denote the input image `Image` and $t_{u,l}$ denote the two threshold images (see `prepare_variation_model`). Then the output region `Region` contains all points that differ substantially from the model, i.e., the points that fulfill the following condition:

$$c(x, y) > t_u(x, y) \vee c(x, y) < t_l(x, y) .$$

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Image of the object to be trained.
- ▷ **Region** (output_object) region(-array) $\leadsto$ *Hobject*
Region containing the points that differ substantially from the model.
- ▷ **ModelID** (input_control) variation_model $\leadsto$ *integer*
ID of the variation model.

Example

```

open_framegrabber ('File', 1, 1, 0, 0, 0, 0, 'default', -1,
                  'default', -1, 'default', 'model.seq', 'default',
                  -1, -1, FGHandle)
read_region (Region, 'model.reg')
area_center (Region, Area, RowRef, ColumnRef)
read_shape_model ('model.shape', TemplateID)
read_variation_model ('model.var', ModelID)
for K := 1 to 10000 by 1
    grab_image (Image, FGHandle)
    find_shape_model (Image, TemplateID, 0, rad(360), 0.5, 1, 0.5,
                    'true', 4, 0.9, Row, Column, Angle, Score)
    disp_obj (Image, WindowHandle)
    if (|Score| = 1)
        vector_angle_to_rigid (Row, Column, Angle, RowRef,
                              ColumnRef, 0, HomMat2D)
        affine_trans_image (Image, ImageTrans, HomMat2D, 'constant',
                            'false')
        compare_variation_model (ImageTrans, RegionDiff, ModelID)
        disp_obj (RegionDiff, WindowHandle)
    endif
endfor
clear_shape_model (TemplateID)
clear_variation_model (ModelID)
close_framegrabber (FGHandle)

```

Result

[compare_variation_model](#) returns 2 (H.MSG_TRUE) if all parameters are correct and if the internal threshold images have been generated with [prepare_variation_model](#).

Parallelization Information

[compare_variation_model](#) is *reentrant* and automatically *parallelized* (on *tuple level*, *domain level*).

Possible Predecessors

[prepare_variation_model](#)

Alternatives

[dyn_threshold](#)

Module

Image / region / XLD management

create_variation_model (: : Width, Height, Type, Mode : ModelID)

Create a variation model for image comparison.

[create_variation_model](#) creates a variation model that can be used for image comparison. The handle for the variation model is returned in [ModelID](#).

Typically, the variation model is used to discriminate correctly manufactured objects (“good objects”) from incorrectly manufactured objects (“bad objects”). It is assumed that the discrimination can be done solely based on the gray values of the object.

The variation model consists of an ideal image of the object to which the images of the objects to be tested are compared later on with `compare_variation_model` and an image that represents the amount of gray value variation at every point of the object. The size of the images with which the object model is trained and with which the model is compared later on is passed in `Width` and `Height`, respectively. The image type of the images used for training and comparison is passed in `Type`.

The variation model is trained using multiple images of good objects. Therefore, it is essential that the training images show the objects in the same position and rotation. If this cannot be guaranteed by external means, the pose of the object can, for example, be determined by using matching (see `find_shape_model`). The image can then be transformed to a reference pose with `affine_trans_image`.

The parameter `Mode` is used to determine how the image of the ideal object and the corresponding variation image are computed. For `Mode='standard'`, the ideal image of the object is computed as the mean of all training images at the respective image positions. The corresponding variation image is computed as the standard deviation of the training images at the respective image positions. This mode has the advantage that the variation model can be trained iteratively, i.e., as soon as an image of a good object becomes available, it can be trained with `train_variation_model`. The disadvantage of this mode is that great care must be taken to ensure that only images of good objects are trained, because the mean and standard deviation are not robust against outliers, i.e., if an image of a bad object is trained inadvertently, the accuracy of the ideal object image and that of the variation image might be degraded.

If it cannot be avoided that the variation model is trained with some images of objects that can contain errors, `Mode` can be set to `'robust'`. In this mode, the image of the ideal object is computed as the median of all training images at the respective image positions. The corresponding variation image is computed as a suitably scaled median absolute deviation of the training images and the median image at the respective image positions. This mode has the advantage that it is robust against outliers. It has the disadvantage that it cannot be trained iteratively, i.e., all training images must be accumulated using `concat_obj` and be trained with `train_variation_model` in a single call.

<i>Parameter</i>	
▷ Width (input_control)	extent.x $\leadsto$ integer
Width of the images to be compared.	
Default Value : 640	
Suggested values : Width $\in$ {160, 192, 320, 384, 640, 768}	
▷ Height (input_control)	extent.y $\leadsto$ integer
Height of the images to be compared.	
Default Value : 480	
Suggested values : Height $\in$ {120, 144, 240, 288, 480, 576}	
▷ Type (input_control)	string $\leadsto$ string
Type of the images to be compared.	
Default Value : 'byte'	
Suggested values : Type $\in$ {'byte', 'int2', 'uint2'}	
▷ Mode (input_control)	string $\leadsto$ string
Method used for computing the variation model.	
Default Value : 'standard'	
Suggested values : Mode $\in$ {'standard', 'robust'}	
▷ ModelID (output_control)	variation_model $\leadsto$ integer
ID of the variation model.	

Result

`create_variation_model` returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

`create_variation_model` is processed *completely exclusively* without parallelization.

Possible Successors

`train_variation_model`

See also

`clear_variation_model`, `find_shape_model`, `affine_trans_image`

Module

Image / region / XLD management

get_variation_model (: Image, VarImage : ModelID :)

Return the images used for image comparison by a variation model.

get_variation_model returns the image of the ideal object and the corresponding variation image of the variation model **ModelID** in **Image** and **VarImage**, respectively. The returned images can be used to check whether an image of a bad object has been trained with **train_variation_model**. This can be seen from the variation image. If an image of a bad object has been trained, the variation image typically has large variations in areas that should exhibit no variations.

Parameter

- ▷ **Image** (output_object) image $\leadsto$ *Hobject* : byte / int2 / uint2
Image of the trained object.
- ▷ **VarImage** (output_object) image $\leadsto$ *Hobject* : real
Variation image of the trained object.
- ▷ **ModelID** (input_control) variation_model $\leadsto$ integer
ID of the variation model.

Result

get_variation_model returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

get_variation_model is *reentrant* and processed *without* parallelization.

Possible Predecessors

train_variation_model

See also

prepare_variation_model, **compare_variation_model**

Module

Image / region / XLD management

prepare_variation_model (: : ModelID, AbsThreshold, VarThreshold :)

Prepare a variation model for comparison with an image.

prepare_variation_model prepares a variation model for the image comparison with **compare_variation_model**. This is done by converting the ideal image and the variation image that have been trained with **train_variation_model** into two threshold images and storing them in the variation model. These threshold images are used in **compare_variation_model** to speed up the comparison of the current image to the variation model.

Two thresholds are used to compute the threshold images. The parameter **AbsThreshold** determines the minimum amount of gray levels by which the image of the current object must differ from the image of the ideal object. The parameter **VarThreshold** determines a factor relative to the variation image for the minimum difference of the current image and the ideal image. Let $i(x, y)$ be the ideal image, $v(x, y)$ the variation image, $a = \mathbf{AbsThreshold}$, and $b = \mathbf{VarThreshold}$. Then the two threshold images $t_{u,l}$ are computed as follows:

$$t_u(x, y) = i(x, y) + \max\{a, bv(x, y)\} \quad t_l(x, y) = i(x, y) - \max\{a, bv(x, y)\} .$$

If the current image $c(x, y)$ is compared to the variation model using **compare_variation_model**, the output region contains all points that differ substantially from the model, i.e., that fulfill the following condition:

$$c(x, y) > t_u(x, y) \vee c(x, y) < t_l(x, y) .$$

Parameter

- ▷ **ModelID** (input_control) variation_model $\leadsto$ integer
ID of the variation model.
- ▷ **AbsThreshold** (input_control) number $\leadsto$ real / integer
Absolute minimum threshold for the differences between the image and the variation model.
Default Value : 10
Suggested values : $\mathbf{AbsThreshold} \in \{0, 5, 10, 15, 20, 30, 40, 50\}$
Restriction : $\mathbf{AbsThreshold} \geq 0$

- ▷ **VarThreshold** (input_control) number $\leadsto$ *real* / integer
 Threshold for the differences based on the variation of the variation model.
Default Value : 2
Suggested values : VarThreshold $\in \{1, 1.5, 2, 2.5, 3, 3.5, 4, 4.5, 5\}$
Restriction : VarThreshold ≥ 0

Result

`prepare_variation_model` returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

`prepare_variation_model` is processed under *mutual exclusion* against itself and without parallelization.

Possible Predecessors

`train_variation_model`

Possible Successors

`compare_variation_model`

See also

`create_variation_model`

Module

Image / region / XLD management

read_variation_model (: : FileName : ModelID)

Read a variation model from a file.

The operator `read_variation_model` reads a variation model, which has been written with `write_variation_model`, from the file `FileName`.

Parameter

- ▷ **FileName** (input_control) filename $\leadsto$ *string*
 File name.
 ▷ **ModelID** (output_control) variation_model $\leadsto$ *integer*
 ID of the variation model.

Result

If the file name is valid, the operator `read_variation_model` returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Parallelization Information

`read_variation_model` is *reentrant* and processed *without* parallelization.

Possible Successors

`compare_variation_model`

See also

`write_variation_model`

Module

Image / region / XLD management

train_variation_model (Images : : ModelID :)

Train a variation model.

`train_variation_model` trains the variation model that is passed in `ModelID` with one or more images, which are passed in `Images`.

As described for `create_variation_model`, a variation model that has been created using the mode '*standard*' can be trained iteratively, i.e., as soon as images of good objects become available, they can be trained with `train_variation_model`. The ideal image of the object is computed as the mean of all previous training images and the images that are passed in `Images`. The corresponding variation image is computed as the standard deviation of the training images and the images that are passed in `Images`.

If the variation model has been created using the mode *'robust'*, the model cannot be trained iteratively, i.e., all training images must be accumulated using `concat_obj` and be trained with `train_variation_model` in a single call. If any images have been trained previously, the training information of the previous call is discarded. The image of the ideal object is computed as the median of all training images passed in `Images`. The corresponding variation image is computed as a suitably scaled median absolute deviation of the training images and the median image.

Parameter

- ▷ **Images** (input_object) image(-array) $\leadsto$ *Hobject* : byte / int2 / uint2
Images of the object to be trained.
- ▷ **ModelID** (input_control) variation_model $\leadsto$ *integer*
ID of the variation model.

Example

```
open_framegrabber ('File', 1, 1, 0, 0, 0, 0, 'default', -1,
                  'default', -1, 'default', 'model.seq', 'default',
                  -1, -1, FGHandle)
grab_image (Image, FGHandle)
get_image_pointer1 (Image, Pointer, Type, Width, Height)
disp_obj (Image, WindowHandle)
draw_region (Region, WindowHandle)
reduce_domain (Image, Region, ImageReduced)
area_center (Region, Area, RowRef, ColumnRef)
create_shape_model (ImageReduced, 4, 0, rad(360), rad(1), 'none',
                  'use_polarity', 40, 10, TemplateID)
create_variation_model (Width, Height, Type, 'standard', ModelID)
for K := 1 to 100 by 1
    grab_image (Image, FGHandle)
    find_shape_model (Image, TemplateID, 0, rad(360), 0.5, 1, 0.5,
                    'true', 4, 0.9, Row, Column, Angle, Score)
    if (|Score| = 1)
        vector_angle_to_rigid (Row, Column, Angle, RowRef,
                              ColumnRef, 0, HomMat2D)
        affine_trans_image (Image, ImageTrans, HomMat2D, 'constant',
                            'false')
        train_variation_model (ImageTrans, ModelID)
    endif
endfor
prepare_variation_model (ModelID, 10, 4)
write_region (Region, 'model.reg')
write_shape_model (TemplateID, 'model.shape')
write_variation_model (ModelID, 'model.var')
clear_shape_model (TemplateID)
clear_variation_model (ModelID)
close_framegrabber (FGHandle)
```

Result

`train_variation_model` returns 2 (H_MSG_TRUE) if all parameters are correct.

Parallelization Information

`train_variation_model` is processed under *mutual exclusion* against itself and without parallelization.

Possible Predecessors

`create_variation_model`, `find_shape_model`, `affine_trans_image`, `concat_obj`

Possible Successors

`prepare_variation_model`

See also

`prepare_variation_model`, `compare_variation_model`, `clear_variation_model`

Module

Image / region / XLD management

write_variation_model (: : ModelID, FileName :)

Write a variation model to a file.

write_variation_model writes a variation model to the file **FileName**. The model can be read with **read_variation_model**.

Parameter

- ▷ **ModelID** (input_control) variation_model $\leadsto$ integer
ID of the variation model.
- ▷ **FileName** (input_control) filename $\leadsto$ string
File name.

Result

If the file name is valid (write permission), the operator **write_variation_model** returns 2 (H_MSG_TRUE). If necessary an exception handling is raised.

Parallelization Information

write_variation_model is *reentrant* and processed *without* parallelization.

Possible Predecessors

train_variation_model

See also

read_variation_model

Module

Image / region / XLD management

15.10 Kalman-Filter

filter_kalman (: : Dimension, Model, Measurement, PredictionIn : PredictionOut, Estimate)

Estimate the current state of a system with the help of the Kalman filtering.

The operator **filter_kalman** returns an estimate of the current state (or also a prediction of a future state) of a discrete, stochastically disturbed, linear system. In practice, Kalman filters are used successfully in image processing in the analysis of image sequences (background identification, lane tracking with the help of line tracing or region analysis, etc.). A short introduction concerning the theory of the Kalman filters will be followed by a detailed description of the routine **filter_kalman** itself.

KALMAN FILTER: A discrete, stochastically disturbed, linear system is characterized by the following markers:

- State $x(t)$: Describes the current state of the system (speeds, temperatures,...).
- Parameter $u(t)$: Inputs from outside into the system.
- Measurement $y(t)$: Measurements gained by observing the system. They indicate the state of the system (or at least parts of it).
- An output function describing the dependence of the measurements on the state.
- A transition function indicating how the state changes with regard to time, the current value and the parameters.

The output function and the transition function are linear. Their application can therefore be written as a multiplication with a matrix.

The transition function is described with the help of the transition matrix $A(t)$ and the parameter matrix, the initial function is described by the measurement matrix $C(t)$. Hereby $C(t)$ characterizes the dependency of the new state on the old, $G(t)$ indicates the dependency on the parameters. In practice it is rarely possible (or at least too time consuming) to describe a real system and its behaviour in a complete and exact way. Normally only a relatively small number of variables will be used to simulate the behaviour of the system. This leads to an error, the so called system error (also called system disturbance) $v(t)$.

The output function, too, is usually not exact. Each measurement is faulty. The measurement errors will be called $w(t)$. Therefore the following system equations arise:

$$\begin{aligned} x(t+1) &= A(t)x(t) + G(t)u(t) + v(t) \\ y(t) &= c(t)x(t) + w(t) \end{aligned}$$

The system error $v(t)$ and the measurement error $w(t)$ are not known. As far as systems are concerned which are interpreted with the help of the Kalman filter, these two errors are considered as Gaussian distributed random vectors (therefore the expression "stochastically disturbed systems"). Therefore the system can be calculated, if the corresponding expected values for $v(t)$ and $w(t)$ as well as the covariance matrices are known.

The estimation of the state of the system is carried out in the same way as in the Gaussian-Markov-estimation. However, the Kalman filter is a recursive algorithm which is based only on the current measurements $y(t)$ and the latest state $x(t)$. The latter implicitly also includes the knowledge about earlier measurements.

A suitable estimate value x_0 , which is interpreted as the expected value of a random variable for $x(0)$, must be indicated for the initial value $x(0)$. This variable should have an expected error value of 0 and the covariance matrix P_0 which also has to be indicated. At a certain time t the expected values of both disturbances $v(t)$ and $w(t)$ should be 0 and their covariances should be $Q(t)$ and $R(t)$. $x(t)$, $v(t)$ and $w(t)$ will usually be assumed to be not correlated (any kind of noise-process can be modelled - however the development of the necessary matrices by the user will be considerably more demanding). The following conditions must be met by the searched estimate values x^t :

- The estimate values x^t are linearly dependent on the actual value $x(t)$ and on the measurement sequence $y(0), y(1), \dots, y(t)$.
- x^t being hereby considered to meet its expectations, i.e. $Ex^t = Ex(t)$.
- The grade criterion for x^t is the criterion of minimal variance, i.e. the variance of the estimation error defined as $x(t) - x^t$, being as small as possible.

After the initialization

$$\hat{x}(0) = x_0, \hat{P}(0) = P_0$$

at each point in time t the Kalman filter executes the following calculation steps:

$$\begin{aligned} (K - III) \quad K(t) &= \frac{\hat{P}(t)C'(t)}{C(t)\hat{P}(t)C'(t) + R(t)} \\ (K - IV) \quad x^t &= \hat{x}(t) + K(t)(y(t) - C(t)\hat{x}(t)) \\ (K - V) \quad \tilde{P}(t) &= \hat{P}(t) - K(t)C(t)\hat{P}(t) \\ (K - I) \quad \hat{x}(t+1) &= A(t)x^t + G(t)u(t) \\ (K - II) \quad \hat{P}(t+1) &= A(t)\tilde{P}(t)A'(t) + Q(t) \end{aligned}$$

Hereby $\tilde{P}(t)$ is the covariance matrix of the estimation error, $\hat{x}(t)$ is the extrapolation value respective the prediction value of the state, $\hat{P}(t)$ are the covariances of the prediction error $\hat{x} - x$, K is the amplifier matrix (the so called Kalman gain), and X' is the transposed of a matrix X .

Please note that the prediction of the future state is also possible with the equation (K-I). Sometimes this is very useful in image processing in order to determine "regions of interest" in the next image.

As mentioned above, it is much more demanding to model any kind of noise processes. If for example the system noise and the measurement noise are correlated with the corresponding covariance matrix L , the equations for the Kalman gain and the error covariance matrix have to be modified:

$$\begin{aligned} (K - III) \quad K(t) &= \frac{\hat{P}(t)C'(t) + L(t)}{C(t)\hat{P}(t) + C(t)L(t) + L'(t)C'(t) + R(t)} \\ (K - V) \quad \tilde{P}(t) &= (\hat{P}(t) - K(t)C(t)\hat{P}(t))\hat{P}(t) - K(t)L(t) \end{aligned}$$

This means that the user himself has to establish the linear system equations from (K-I) up to (K-V) with respect to the actual problem. The user must therefore develop a mathematical model upon which the solution to the problem can be based. Statistical characteristics describing the inaccuracies of the system as well as the measurement errors, which are to be expected, thereby have to be estimated if they cannot be calculated exactly. Therefore the following individual steps are necessary:

1. Developing a mathematical model

2. Selecting characteristic state variables
3. Establishing the equations describing the changes of these state variables and their linearization (matrices A and G)
4. Establishing the equations describing the dependency of the measurement values of the system on the state variables and their linearization (matrix C)
5. Developing or estimating of statistical dependencies between the system disturbances (matrix Q)
6. Developing or estimating of statistical dependencies between the measurement errors (matrix R)
7. Initialization of the initial state

As mentioned above, the initialization of the system (point 7) hereby necessitates to indicate an estimate x_0 of the state of the system at the time 0 and the corresponding covariance matrix P_0 . If the exact initial state is not known, it is recommendable to set the components of the vector x_0 to the average values of the corresponding range, and to set high values for P_0 (about the size of the squares of the range). After a few iterations (when the number of the accumulated measurement values in total has exceeded the number of the system values), the values which have been determined in this way are also useable.

If on the other hand the initial state is known exactly, all entries for P_0 have to be set to 0, because P_0 describes the covariances of the error between the estimated value x_0 and the actual value $x(0)$.

THE FILTER ROUTINE:

A Kalman filter is dependent on a range of data which can be organized in four groups:

Model parameter: transition matrix A , control matrix G including the parameter u and the measurement matrix C

Model stochastic: system-error covariance matrix Q , system-error - measurement-error covariance matrix L , and measurement-error covariance matrix R

Measurement vector: y

History of the system: extrapolation vector $\hat{x}$ and extrapolation-error covariance matrix $\hat{P}$

Thereby many systems can work without input "from outside", i.e. without G and u . Further, system errors and measurement errors are normally not correlated (L is dropped).

Actually the data necessary for the routine will be set by the following parameters:

Dimension: This parameter includes the dimensions of the status vector, the measurement vector and the controller vector. **Dimension** thereby is a vector $[n,m,p]$, whereby n indicates the number of the state variables, m the number of the measurement values and p the number of the controller members. For a system without determining control (i.e. without influence "from outside") therefore $[n,m,0]$ has to be passed.

Model: This parameter includes the lined up matrices (vectors) A, C, Q, G, u and (if necessary) L having been stored in row-major order. **Model** therefore is a vector of the length $n \times n + n \times m + n \times n + n \times p + p[+n \times m]$. The last summand is dropped, in case the system errors and measurement errors are not correlated, i.e. there is no value for L .

Measurement: This parameter includes the matrix R which has been stored in row-major order, and the measurement vector y lined up. **Measurement** therefore is a vector of the dimension $m \times m + m$.

PredictionIn / PredictionOut: These two parameters include the matrix $\hat{P}$ (the extrapolation-error covariance matrix) which has been stored in row-major order and the extrapolation vector $\hat{x}$ lined up. This means, they are vectors of the length $n \times n + n$. **PredictionIn** therefore is an input parameter, which must contain $\hat{P}(t)$ and $\hat{x}(t)$ at the current time t . With **PredictionOut** the routine returns the corresponding predictions $\hat{P}(t+1)$ and $\hat{x}(t+1)$.

Estimate: With this parameter the routine returns the matrix $\tilde{P}$ (the estimation-error covariance matrix) which has been stored in row-major order and the estimated state $\tilde{x}$ lined up. **Estimate** therefore is a vector of the length $n \times n + n$.

Please note that the covariance matrices ($Q, R, \hat{P}, \tilde{P}$) must of course be symmetric.

<i>Parameter</i>	
▷ Dimension (input_control)	integer-array $\leadsto$ integer
The dimensions of the state vector, the measurement and the controller vector.	
Default Value : '[3,1,0]'	
Typical range of values : $0 \leq \text{Dimension} \leq 30$	
▷ Model (input_control)	real-array $\leadsto$ real
The lined up matrices A , C , Q , possibly G and u , and if necessary L which have been stored in row-major order.	
Default Value : '[1.0,1.0,0.5,0.0,1.0,1.0,0.0,0.0,1.0,1.0,0.0,0.0,54.3,37.9,48.0,37.9,34.3,42.5,48.0,42.5,43.7]'	
Typical range of values : $0.0 \leq \text{Model} \leq 10000.0$	
▷ Measurement (input_control)	real-array $\leadsto$ real
The matrix R stored in row-major order and the measurement vector y lined up.	
Default Value : '[1.2,1.0]'	
Typical range of values : $0.0 \leq \text{Measurement} \leq 10000.0$	
▷ PredictionIn (input_control)	real-array $\leadsto$ real
The matrix $\hat{P}$ (the extrapolation-error covariances) stored in row-major order and the extrapolation vector $\hat{x}$ lined up.	
Default Value : '[0.0,0.0,0.0,0.0,180.5,0.0,0.0,0.0,100.0,0.0,100.0,0.0]'	
Typical range of values : $0.0 \leq \text{PredictionIn} \leq 10000.0$	
▷ PredictionOut (output_control)	real-array $\leadsto$ real
The matrix P^* (the extrapolation-error covariances) stored in row-major order and the extrapolation vector $\hat{x}$ lined up.	
▷ Estimate (output_control)	real-array $\leadsto$ real
The matrix $\tilde{P}$ (the estimation-error covariances) stored in row-major order and the estimated state $\tilde{x}$ lined up.	
<i>Example</i>	

```
// Typical procedure:
// 1. To initialize the variables, which describe the model, e.g. with
read_kalman('kalman.init',Dim,Mod,Meas,Pred)
// Generation of the first measurements (typical of the first image of an
// image series) with an appropriate problem-specific routine (there is a
// fictitious routine extract_features in example):
extract_features(Image1,Meas,Meas1)
// first Kalman-Filtering:
filter_kalman(Dim,Mod,Meas1,Pred,Pred1,Est1)
// To use the estimate value (if need be the prediction too)
// with a problem-specific routine (here use_est):
use_est(Est1)
// To get the next measurements (e.g. from the next image):
extract_next_features(Image2,Meas1,Meas2)
// if need be Update of the model parameter (a constant model)
// second Kalman-Filtering:
filter_kalman(Dim,Mod,Meas2,Pred1,Pred2,Est2)
use_est(Est2)
extract_next_features(Image3,Meas2,Meas3).
// etc.
```

Result

If the parameter values are correct, the operator `filter_kalman` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling will be raised.

Parallelization Information

`filter_kalman` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`read_kalman`, `sensor_kalman`

Possible Successors

`update_kalman`

See also

[read_kalman](#), [update_kalman](#), [sensor_kalman](#)

References

W.Hartinger: "Entwurf eines anwendungsunabhängigen Kalman-Filters mit Untersuchungen im Bereich der Bildfolgenanalyse"; Diplomarbeit; Technische Universität München, Institut für Informatik, Lehrstuhl Prof. Radig; 1991.

R.E.Kalman: "A New Approach to Linear Filtering and Prediction Problems"; Transactions ASME, Ser.D: Journal of Basic Engineering; Vol. 82, S.34-45; 1960.

R.E.Kalman, P.I.Falb, M.A.Arbib: "Topics in Mathematical System Theory"; McGraw-Hill Book Company, New York; 1969.

K-P. Karmann, A.von Brandt: "Moving Object Recognition Using an Adaptive Background Memory"; Time-Varying Image Processing and Moving Object Recognition 2 (ed.: V. Cappellini), Proc. of the 3rd International Workshop, Florence, Italy, May, 29th - 31st, 1989; Elsevier, Amsterdam; 1990.

Module

Tools

read_kalman (: : FileName : Dimension, Model, Measurement, Prediction)

Read the description file of a Kalman filter.

The operator [read_kalman](#) reads the description file [FileName](#) of a Kalman filter. Kalman filters return an estimate of the current state (or even the prediction of a future state) of a discrete, stochastically disturbed, linear system. They are successfully used in image processing, especially in the analysis of image sequences. A Kalman filtering is based on a mathematical model of the system to be examined which at any point in time has the following characteristics:

Model parameter: transition matrix A , control matrix G including the controller output u and the measurement matrix C

Model stochastic: system-error covariance matrix Q , system-error - measurement-error covariance matrix L and measurement-error covariance matrix R

Estimate of the initial state of the system: state x_0 and corresponding covariance matrix P_0

Many systems do not need entries "from outside", and therefore G and u can be dropped. Further, system errors and measurement errors are normally not correlated (L is dropped). The characteristics mentioned above can be stored in an ASCII-file and then can be read with the help of the operator [read_kalman](#). This ASCII-file must have the following structure:

Dimension row

```
+ content row
+ matrix A
+ matrix C
+ matrix Q
[ + matrix G + vector u ]
[ + matrix L ]
+ matrix R
[ + matrix P0 ]
[ + vector x0 ]
```

The dimension row thereby is always of the following form:

$$n = \langle \text{integer} \rangle \quad m = \langle \text{integer} \rangle \quad p = \langle \text{integer} \rangle$$

whereby n indicates the number of the state variables, m the number of the measurement values and p the number of the controller members (see also [Dimension](#)). The maximal dimension will hereby be limited by a system constant (= 30 for the time being).

The content row has the following form:

$A * C * Q * G * u * L * R * P * x *$

and describes the following content of the file. Instead of '*', '+' (= parameter is available) respectively '-' (= parameter is missing) have to be set. Please note that only the parameters marked by [...] in the above list may be left out in the description file. If the initial state estimate a_0 is missing (i.e. 'x-'), the components of the vector will supposed to be 0.0. If the covariance matrix P_0 of the initial state estimate is missing (i.e. 'P-'), the error will be supposed to be tremendous. In this case the matrix elements will be set to 10000.0. This value seems to be very high, however, it is only sufficient if the range of components of the state vector x is smaller to the tenth power. $(r \times s)$ matrices will be stored per row in the following form:

$$\begin{array}{cccc} & \langle \text{Kommentar, d.h. string} \rangle & & \\ \langle a_{11} \rangle & \langle a_{12} \rangle & \cdots & \langle a_{1s} \rangle \\ \vdots & & \ddots & \vdots \\ \langle a_{r1} \rangle & \langle a_{r2} \rangle & \cdots & \langle a_{rs} \rangle \end{array}$$

(the spaces and line feed characters can be chosen at will),

vectors will be stored correspondingly in the following form:

$$\begin{array}{ccc} \langle \text{comment, i.e. string} \rangle & & \\ \langle a_1 \rangle & \cdots & \langle a_k \rangle \end{array}$$

The following parameter values are returned by the operator `read_kalman`:

Dimension: This parameter includes the dimensions of the status vector, the measurement vector and the controller vector. **Dimension** thereby is a vector $[n, m, p]$, whereby n indicates the number of the state variables, m the number of the measurement values and p the number of the controller members. For a system without determining control (i.e. without influence "from outside") therefore **Dimension** = $[n, m, 0]$.

Model: This parameter includes the lined up matrices (vectors) A , C , Q , G , u and (if necessary) L having been stored in row-major order. **Model** therefore is a vector of the length $n \times n + n \times m + n \times n + n \times p + p + [n \times m]$. The last summand is dropped, in case the system errors and measurement errors are not correlated, i.e. there is no value for L .

Measurement: This parameter includes the matrix R which has been stored in row-major order. **Measurement** therefore is vector of the dimension $m \times m$.

Prediction: This parameter includes the matrix P_0 (the error covariance matrix of the initial state estimate) and the initial state estimate x_0 lined up. This means, it is a vector of the length $n \times n + n$.

Parameter

- ▷ **FileName** (input_control) filename $\leadsto$ string
Description file for a Kalman filter.
Default Value : "kalman.init"
- ▷ **Dimension** (output_control) integer-array $\leadsto$ integer
The dimensions of the state vector, the measurement vector and the controller vector.
- ▷ **Model** (output_control) real-array $\leadsto$ real
The lined up matrices A , C , Q , possibly G and u , and if necessary L stored in row-major order.
- ▷ **Measurement** (output_control) real-array $\leadsto$ real
The matrix R stored in row-major order.
- ▷ **Prediction** (output_control) real-array $\leadsto$ real
The matrix P_0 (error covariance matrix of the initial state estimate) stored in row-major order and the initial state estimate x_0 lined up.

Example

%An example of the description-file:

%

%n=3 m=1 p=0

%A+C+Q+G-u-L-R+P+x+

%transition matrix A:

```

%1 1 0.5
%0 1 1
%0 0 1
%measurement matrix C:
%1 0 0
%system-error covariance matrix Q:
%54.3 37.9 48.0
%37.9 34.3 42.5
%48.0 42.5 43.7
%measurement-error covariance matrix R:
%1.2
%estimation-error covariance matrix (for the initial estimate) P0:
%0 0 0
%0 180.5 0
%0 0 100
%initial estimate x0:
%0 100 0
%
%the result of read_kalman with the upper descriptionfile
%as inputparameter:
%
%Dimension    = [3,1,0]
%Model        = [1.0,1.0,0.5,0.0,1.0,1.0,0.0,0.0,1.0,1.0,0.0,0.0,
%                54.3,37.9,48.0,37.9,34.3,42.5,48.0,42.5,43.7]
%Measurement  = [1.2]
%Prediction   = [0.0,0.0,0.0,0.0,180.5,0.0,0.0,0.0,100.0,0.0,100.0,
%                0.0].

```

Result

If the description file is readable and correct, the operator [read_kalman](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception handling will be raised.

Parallelization Information

[read_kalman](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[filter_kalman](#)

See also

[update_kalman](#), [filter_kalman](#), [sensor_kalman](#)

Module

Tools

sensor_kalman (: : Dimension, MeasurementIn : MeasurementOut)

Interactive input of measurement values for a Kalman filtering.

The operator [sensor_kalman](#) supports the interactive input of measurement values for a Kalman filtering. Kalman filters return an estimate of the current state (or even the prediction of a future state) of a discrete, stochastically disturbed, linear system. They are successfully used in image processing, especially in the analysis of image sequences.

Each filtering is hereby based on certain measurement values. How these values are extracted from images or sensor data depends strongly on the individual application and therefore must be entirely up to the user. However, the operator [sensor_kalman](#) allows an interactive input of (fictitious) measurement values y and the corresponding measurement-error covariance matrix R . Especially the testing of Kalman filters during the development can hereby be facilitated.

The parameters [MeasurementIn](#) and [MeasurementOut](#) include the matrix R which has been stored in row-major order and the measurement vector y lined up, i.e. they are vectors of the length $\text{Dimension} \times \text{Dimension} + \text{Dimension}$

Parameter	
▷ Dimension (input_control)	integer $\leadsto$ integer
Number of measurement values.	
Default Value : 1	
Typical range of values : $0 \leq \text{Dimension} \leq 30$	
▷ MeasurementIn (input_control)	real-array $\leadsto$ real
The matrix R stored in row-major order and the measurement vector y lined up.	
Default Value : '[1.2,1.0]'	
Typical range of values : $0.0 \leq \text{MeasurementIn} \leq 10000.0$	
▷ MeasurementOut (output_control)	real-array $\leadsto$ real
The matrix R stored in row-major order and the measurement vector y lined up.	
Result	
If the parameters are correct, the operator <code>sensor_kalman</code> returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.	
Parallelization Information	
<code>sensor_kalman</code> is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Successors	
<code>filter_kalman</code>	
See also	
<code>filter_kalman</code> , <code>read_kalman</code> , <code>update_kalman</code>	
Module	
Tools	

update_kalman (: : FileName, DimensionIn, ModelIn, MeasurementIn : DimensionOut, ModelOut, MeasurementOut)

Read an update file of a Kalman filter.

The operator `update_kalman` reads the update file `FileName` of a Kalman filter. Kalman filters return an estimate of the current state (or even the prediction of a future state) of a discrete, stochastically disturbed, linear system.

A Kalman filtering is based on a mathematical model of the system to be examined which at any point in time has the following characteristics:

Model parameter: transition matrix A , control matrix G including the controller output u and the measurement matrix C

Model stochastic: system-error covariance matrix Q , system-error - measurement-error covariance matrix L and measurement-error covariance matrix R

Measurement vector: y

History of the system: extrapolation vector $\hat{x}$ and extrapolation-error covariance matrix $\hat{P}$

Many systems do not need entries "from outside" and therefore G and u can be dropped. Further, system errors and measurement errors are normally not correlated (L is dropped). Some of the characteristics mentioned above may change dynamically (from one iteration to the next). The operator `update_kalman` serves to modify parts of the system according to an update file (ASCII) with the following structure (see also `read_kalman`):

```

Dimension row
+ content row
+ matrix A
+ matrix C
+ matrix Q
+ matrix G + vector u
+ matrix L
+ matrix R
```

The dimension row thereby has the following form:

$$n = \langle \text{integer} \rangle \quad m = \langle \text{integer} \rangle \quad p = \langle \text{integer} \rangle$$

whereby n indicates the number of the state variables, m the number of the measurement values and p the number of the controller members (see also [DimensionIn](#) / [DimensionOut](#)). The maximal dimension will hereby be limited by a system constant (= 30 for the time being). As in this case changes should take effect at a valid model, the dimensions n and m are invariant (and will only be indicated for purposes of control).

The content row has the following form:

$$A * C * Q * G * u * L * R *$$

and describes the further content of the file. Instead of '*', '+' (= parameter is available) respectively '-' (= parameter is missing) has to be set. In contrast to description files for [read_kalman](#), the system description needs not be complete in this case. Only those parts of the system which are changed must be indicated. The indication of estimated values is unnecessary, as these values must stem from the latest filtering according to the structure of the filter.

($r \times s$) matrices will be stored in row-major order in the following form:

$$\begin{array}{cccc} & \langle \text{comment, i.e. string} \rangle & & \\ \langle a_{11} \rangle & \langle a_{12} \rangle & \cdots & \langle a_{1s} \rangle \\ \vdots & & \ddots & \vdots \\ \langle a_{r1} \rangle & \langle a_{r2} \rangle & \cdots & \langle a_{rs} \rangle \end{array}$$

(the spaces/line feed characters can be chosen at will),

vectors will be stored correspondingly in the following form:

$$\begin{array}{ccc} \langle \text{commentar, i.e. string} \rangle & & \\ \langle a_1 \rangle & \cdots & \langle a_k \rangle \end{array}$$

The following parameter values of the operator [read_kalman](#) will be changed:

DimensionIn / DimensionOut: These parameters include the dimensions of the state vector, measurement vector and controller vector and therefore are vectors $[n, m, p]$, whereby n indicates the number of the state variables, m the number of the measurement values and p the number of the controller members. n and m are invariant for a given system, i.e. they must not differ from corresponding input values of the update file. For a system without influence "from outside" $p = 0$.

ModelIn / ModelOut: These parameters include the lined up matrices (vectors) A, C, Q, G, u and if necessary L which have been stored in row-major order. [ModelIn](#) / [ModelOut](#) therefore are vectors of the length $n \times n + n \times m + n \times n + n \times p + p[+n \times m]$. The last summand is dropped if system errors and measurement errors are not correlated, i.e. no value has been set for L .

MeasurementIn / MeasurementOut: These parameters include the matrix R stored in row-major order, and therefore are vectors of the dimension $m \times m$.

	Parameter
▷ FileName (input_control)	filename $\leadsto$ string
	Update file for a Kalman filter.
	Default Value : "kalman.updt"
▷ DimensionIn (input_control)	integer-array $\leadsto$ integer
	The dimensions of the state vector, measurement vector and controller vector.
	Default Value : '[3,1,0]'
	Typical range of values : $0 \leq \text{DimensionIn} \leq 30$
▷ ModelIn (input_control)	real-array $\leadsto$ real
	The lined up matrices A, C, Q , possibly G and u , and if necessary L which all have been stored in row-major order.
	Default Value : '[1.0,1.0,0.5,0.0,1.0,1.0,0.0,0.0,1.0,1.0,0.0,0.0,54.3,37.9,48.0,37.9,34.3,42.5,48.0,42.5,43.7]'
	Typical range of values : $0.0 \leq \text{ModelIn} \leq 10000.0$

- ▷ **MeasurementIn** (input_control) real-array $\leadsto$ *real*
The matrix R stored in row-major order.
Default Value : '[1,2]'
Typical range of values : $0.0 \leq \text{MeasurementIn} \leq 10000.0$
- ▷ **DimensionOut** (output_control) integer-array $\leadsto$ *integer*
The dimensions of the state vector, measurement vector and controller vector.
- ▷ **ModelOut** (output_control) real-array $\leadsto$ *real*
The lined up matrices A,C,Q, possibly G and u, and if necessary L which all have been stored in row-major order.
- ▷ **MeasurementOut** (output_control) real-array $\leadsto$ *real*
The matrix R stored in row-major order.

Example

```
%The following values are describing the system
%
%DimensionIn   = [3,1,0]
%ModelIn       = [1.0,1.0,0.5,0.0,1.0,1.0,0.0,0.0,1.0,1.0,0.0,0.0,
%                54.3,37.9,48.0,37.9,34.3,42.5,48.0,42.5,43.7]
%MeasurementIn = [1,2]
%
%An example of the Updatefile:
%
%n=3 m=1 p=0
%A+C-Q-G-u-L-R-
%transitions at time t=15:
%2 1 1
%0 2 2
%0 0 2
%
%the results of update_kalman:
%
%DimensionOut   = [3,1,0]
%ModelOut       = [2.0,1.0,1.0,0.0,2.0,2.0,0.0,0.0,2.0,1.0,0.0,0.0,
%                54.3,37.9,48.0,37.9,34.3,42.5,48.0,42.5,43.7]
%MeasurementOut = [1.2]
```

Result

If the update file is readable and correct, the operator [update_kalman](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

[update_kalman](#) is *reentrant* and processed *without* parallelization.

Possible Successors

[filter_kalman](#)

See also

[read_kalman](#), [filter_kalman](#), [sensor_kalman](#)

Module

Tools

15.11 Measure

close_all_measures (: : :)

Delete all measure objects.

[close_all_measures](#) deletes all measure objects that have been created using [gen_measure_rectangle2](#) or [gen_measure_arc](#). The memory used for the measure objects is freed.

Result

`close_all_measures` always returns 2 (H_MSG_TRUE).

Parallelization Information

`close_all_measures` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`gen_measure_rectangle2`, `gen_measure_arc`, `measure_pos`, `measure_pairs`

Alternatives

`close_measure`

Module

Sub-pixel operators

close_measure (: : MeasureHandle :)

Delete a measure object.

`close_measure` deletes the measure object given by `MeasureHandle`. The memory used for the measure object is freed.

Parameter

▷ **MeasureHandle** (input_control) measure_id $\leadsto$ integer
Measure object handle.

Result

If the parameter values are correct the operator `close_measure` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

`close_measure` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`gen_measure_rectangle2`, `gen_measure_arc`, `measure_pos`, `measure_pairs`

See also

`close_all_measures`

Module

Sub-pixel operators

fuzzy_measure_pairing (Image : : MeasureHandle, Sigma, AmpThresh, FuzzyThresh, Transition, Pairing, NumPairs : RowEdgeFirst, ColumnEdgeFirst, AmplitudeFirst, RowEdgeSecond, ColumnEdgeSecond, AmplitudeSecond, RowPairCenter, ColumnPairCenter, FuzzyScore, IntraDistance)

Extract straight edge pairs perpendicular to a rectangle or an annular arc.

`fuzzy_measure_pairing` serves to extract *straight* edge pairs that lie *perpendicular* to the major axis of a rectangle or an annular arc. In addition to `measure_pos` it uses fuzzy member functions to evaluate and select the edge pairs.

The extraction algorithm is identical to `fuzzy_measure_pos`. In addition, the edges are grouped to pairs: If `Transition` = 'positive', the edge points with a dark-to-light transition in the direction of the major axis of the rectangle or the annular arc are returned in `RowEdgeFirst` and `ColumnEdgeFirst`. In this case, the corresponding edges with a light-to-dark transition are returned in `RowEdgeSecond` and `ColumnEdgeSecond`. If `Transition` = 'negative', the behavior is exactly opposite. If `Transition` = 'all', the first detected edge defines the transition for `RowEdgeFirst` and `ColumnEdgeFirst`.

Having extracted subpixel edge locations, the edges are paired. The features of a possible edge pair are evaluated by a fuzzy function, set by `set_fuzzy_measure`. Which edge pairs are selected can be determined with the parameter `FuzzyThresh`, which constitutes a threshold on the weight over all fuzzy sets, i.e., the geometric

mean of the weights of the defined fuzzy membership functions. As an extension to `fuzzy_measure_pairs`, the pairing algorithm can be restricted by `Pairing`. Currently only `'no_restriction'` is available, which returns all possible edge pairs, allowing interleaving and inclusion of pairs. Finally, the best scored `NumPairs` edge pairs are returned, whereas 0 indicates to return all possible found edge combinations.

The selected edges are returned as single points, which lie on the major axis of the rectangle or annular arc. The corresponding edge amplitudes are returned in `AmplitudeFirst` and `AmplitudeSecond`, the fuzzy scores in `FuzzyScore`. In addition, the distance between each edge pair is returned in `IntraDistance`, corresponding to the distance between `EdgeFirst[i]` and `EdgeSecond[i]`.

Attention

`fuzzy_measure_pairing` only returns meaningful results if the assumptions that the edges are straight and perpendicular to the major axis of the rectangle or annular arc are fulfilled. Thus, it should not be used to extract edges from curved objects, for example. Furthermore, the user should ensure that the rectangle or annular arc is as close to perpendicular as possible to the edges in the image.

It should be kept in mind that `fuzzy_measure_pairing` ignores the domain of `Image` for efficiency reasons. If certain regions in the image should be excluded from the measurement a new measure object with appropriately modified parameters should be generated.

Parameter

- ▷ **Image** (input_object) singlechannel-image $\leadsto$ *Hobject* : byte / uint2
Input image.
- ▷ **MeasureHandle** (input_control) measure_id $\leadsto$ *integer*
Measure object handle.
- ▷ **Sigma** (input_control) number $\leadsto$ *real*
Sigma of Gaussian smoothing.
Default Value : 1.0
Suggested values : $\text{Sigma} \in \{0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0\}$
Typical range of values : $0.4 \leq \text{Sigma} \leq 100$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : $\text{Sigma} \geq 0.4$
- ▷ **AmpThresh** (input_control) number $\leadsto$ *real*
Minimum edge amplitude.
Default Value : 30.0
Suggested values : $\text{AmpThresh} \in \{5.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 90.0, 110.0\}$
Typical range of values : $1 \leq \text{AmpThresh} \leq 255$ (lin)
Minimum Increment : 0.5
Recommended Increment : 2
- ▷ **FuzzyThresh** (input_control) number $\leadsto$ *real*
Minimum fuzzy value.
Default Value : 0.5
Suggested values : $\text{FuzzyThresh} \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$
Typical range of values : $0.0 \leq \text{FuzzyThresh} \leq 1.0$ (lin)
Recommended Increment : 0.1
- ▷ **Transition** (input_control) string $\leadsto$ *string*
Select the first gray value transition of the edge pairs.
Default Value : 'all'
List of values : $\text{Transition} \in \{'all', 'positive', 'negative'\}$
- ▷ **Pairing** (input_control) string $\leadsto$ *string*
Constraint of pairing.
Default Value : 'no_restriction'
List of values : $\text{Pairing} \in \{'no\_restriction'\}$
- ▷ **NumPairs** (input_control) number $\leadsto$ *integer*
Number of edge pairs.
Default Value : 10
Suggested values : $\text{NumPairs} \in \{0, 1, 10, 20, 50\}$
Typical range of values : $0 \leq \text{NumPairs}$
Recommended Increment : 1

- ▷ **RowEdgeFirst** (output_control) point.y-array $\leadsto$ *real*
Row coordinate of the first edge.
- ▷ **ColumnEdgeFirst** (output_control) point.x-array $\leadsto$ *real*
Column coordinate of the first edge.
- ▷ **AmplitudeFirst** (output_control) real-array $\leadsto$ *real*
Edge amplitude of the first edge (with sign).
- ▷ **RowEdgeSecond** (output_control) point.y-array $\leadsto$ *real*
Row coordinate of the second edge.
- ▷ **ColumnEdgeSecond** (output_control) point.x-array $\leadsto$ *real*
Column coordinate of the second edge.
- ▷ **AmplitudeSecond** (output_control) real-array $\leadsto$ *real*
Edge amplitude of the second edge (with sign).
- ▷ **RowPairCenter** (output_control) point.y-array $\leadsto$ *real*
Row coordinate of the center of the edge pair.
- ▷ **ColumnPairCenter** (output_control) point.x-array $\leadsto$ *real*
Column coordinate of the center of the edge pair.
- ▷ **FuzzyScore** (output_control) real-array $\leadsto$ *real*
Fuzzy evaluation of the edge pair.
- ▷ **IntraDistance** (output_control) real-array $\leadsto$ *real*
Distance between the edges of the edge pair.

Result

If the parameter values are correct the operator [fuzzy_measure_pairing](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

[fuzzy_measure_pairing](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_measure_rectangle2](#), [gen_measure_arc](#), [set_fuzzy_measure](#)

Possible Successors

[close_measure](#)

Alternatives

[edges_sub_pix](#), [fuzzy_measure_pairs](#), [measure_pairs](#)

See also

[fuzzy_measure_pos](#), [measure_pos](#)

Module

Sub-pixel operators

fuzzy_measure_pairs (Image : : MeasureHandle, Sigma, AmpThresh, FuzzyThresh, Transition : RowEdgeFirst, ColumnEdgeFirst, AmplitudeFirst, RowEdgeSecond, ColumnEdgeSecond, AmplitudeSecond, RowEdgeCenter, ColumnEdgeCenter, FuzzyScore, IntraDistance, InterDistance)

Extract straight edge pairs perpendicular to a rectangle or an annular arc.

[fuzzy_measure_pairs](#) serves to extract *straight* edge pairs which lie *perpendicular* to the major axis of a rectangle or an annular arc. In addition to [measure_pairs](#) it uses fuzzy member functions to evaluate and select the edge pairs.

The extraction algorithm is identical to [fuzzy_measure_pos](#). In addition, the edges are grouped to pairs: If **Transition** = '*positive*', the edge points with a dark-to-light transition in the direction of the major axis of the rectangle or annular arc are returned in [RowEdgeFirst](#) and [ColumnEdgeFirst](#). In this case, the corresponding edges with a light-to-dark transition are returned in [RowEdgeSecond](#) and [ColumnEdgeSecond](#). If **Transition** = '*negative*', the behavior is exactly opposite. If **Transition** = '*all*', the first detected edge defines the transition for [RowEdgeFirst](#) and [ColumnEdgeFirst](#).

Having extracted subpixel edge locations, the edges are paired. The pairing algorithm groups the edges such that interleavings and inclusions of pairs are prohibited. The features of an edge pair are evaluated by a fuzzy function,

which can be set by `set_fuzzy_measure` or `set_fuzzy_measure_norm_pair`. Which edge pairs are selected can be determined with the parameter `FuzzyThresh`, which constitutes a threshold on the weight over all fuzzy sets, i.e., the geometric mean of the weights of the defined fuzzy member functions.

The selected edges are returned as single points, which lie on the major axis of the rectangle or annular arc. The corresponding edge amplitudes are returned in `AmplitudeFirst` and `AmplitudeSecond`, the fuzzy scores in `FuzzyScore`. In addition, the distance between each edge pair is returned in `IntraDistance` and the distance between consecutive edge pairs is returned in `InterDistance`. Here, `IntraDistance[i]` corresponds to the distance between `EdgeFirst[i]` and `EdgeSecond[i]`, while `InterDistance[i]` corresponds to the distance between `EdgeSecond[i]` and `EdgeFirst[i+1]`, i.e., the tuple `InterDistance` contains one element less than the tuples of the edge pairs.

Attention

`fuzzy_measure_pairs` only returns meaningful results if the assumptions that the edges are straight and perpendicular to the major axis of the rectangle or annular arc are fulfilled. Thus, it should not be used to extract edges from curved objects, for example. Furthermore, the user should ensure that the rectangle or a annular arc is as close to perpendicular as possible to the edges in the image.

It should be kept in mind that `fuzzy_measure_pairs` ignores the domain of `Image` for efficiency reasons. If certain regions in the image should be excluded from the measurement a new measure object with appropriately modified parameters should be generated.

Parameter

- ▷ **Image** (input_object) singlechannel-image $\leadsto$ *Hobject* : byte / uint2
Input image.
- ▷ **MeasureHandle** (input_control) measure_id $\leadsto$ *integer*
Measure object handle.
- ▷ **Sigma** (input_control) number $\leadsto$ *real*
Sigma of Gaussian smoothing.
Default Value : 1.0
Suggested values : $\text{Sigma} \in \{0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0\}$
Typical range of values : $0.4 \leq \text{Sigma} \leq 100$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : $\text{Sigma} \geq 0.4$
- ▷ **AmpThresh** (input_control) number $\leadsto$ *real*
Minimum edge amplitude.
Default Value : 30.0
Suggested values : $\text{AmpThresh} \in \{5.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 90.0, 110.0\}$
Typical range of values : $1 \leq \text{AmpThresh} \leq 255$ (lin)
Minimum Increment : 0.5
Recommended Increment : 2
- ▷ **FuzzyThresh** (input_control) number $\leadsto$ *real*
Minimum fuzzy value.
Default Value : 0.5
Suggested values : $\text{FuzzyThresh} \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$
Typical range of values : $0.0 \leq \text{FuzzyThresh} \leq 1.0$ (lin)
Recommended Increment : 0.1
- ▷ **Transition** (input_control) string $\leadsto$ *string*
Select the first gray value transition of the edge pairs.
Default Value : 'all'
List of values : $\text{Transition} \in \{'all', 'positive', 'negative'\}$
- ▷ **RowEdgeFirst** (output_control) point.y-array $\leadsto$ *real*
Row coordinate of the first edge point.
- ▷ **ColumnEdgeFirst** (output_control) point.x-array $\leadsto$ *real*
Column coordinate of the first edge point.
- ▷ **AmplitudeFirst** (output_control) real-array $\leadsto$ *real*
Edge amplitude of the first edge (with sign).
- ▷ **RowEdgeSecond** (output_control) point.y-array $\leadsto$ *real*
Row coordinate of the second edge point.

- ▷ **ColumnEdgeSecond** (output_control) point.x-array $\leadsto$ *real*
Column coordinate of the second edge point.
- ▷ **AmplitudeSecond** (output_control) real-array $\leadsto$ *real*
Edge amplitude of the second edge (with sign).
- ▷ **RowEdgeCenter** (output_control) point.y-array $\leadsto$ *real*
Row coordinate of the center of the edge pair.
- ▷ **ColumnEdgeCenter** (output_control) point.x-array $\leadsto$ *real*
Column coordinate of the center of the edge pair.
- ▷ **FuzzyScore** (output_control) real-array $\leadsto$ *real*
Fuzzy evaluation of the edge pair.
- ▷ **IntraDistance** (output_control) real-array $\leadsto$ *real*
Distance between edges of an edge pair.
- ▷ **InterDistance** (output_control) real-array $\leadsto$ *real*
Distance between consecutive edge pairs.

Result

If the parameter values are correct the operator [fuzzy_measure_pairs](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

[fuzzy_measure_pairs](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_measure_rectangle2](#), [gen_measure_arc](#), [set_fuzzy_measure](#)

Possible Successors

[close_measure](#)

Alternatives

[edges_sub_pix](#), [fuzzy_measure_pairing](#), [measure_pairs](#)

See also

[fuzzy_measure_pos](#), [measure_pos](#)

Module

Sub-pixel operators

fuzzy_measure_pos (Image : : MeasureHandle, Sigma, AmpThresh, FuzzyThresh, Transition : RowEdge, ColumnEdge, Amplitude, FuzzyScore, Distance)

Extract straight edges perpendicular to a rectangle or an annular arc.

[fuzzy_measure_pos](#) extracts *straight* edges which lie *perpendicular* to the major axis of a rectangle or an annular arc. In addition to [measure_pos](#) it uses fuzzy member functions to evaluate and select the edges.

The algorithm works by averaging the gray values in “slices” perpendicular to the major axis of the rectangle or annular arc in order to obtain a one-dimensional edge profile. The sampling is done at subpixel positions in the image [Image](#) at integer row and column distances (in the coordinate frame of the rectangle) from the center of the rectangle. Since this involves some calculations which can be used repeatedly in several measurements, the operator [gen_measure_rectangle2](#) is used to perform these calculations only once, thus increasing the speed of [fuzzy_measure_pos](#) significantly. Since there is a trade-off between accuracy and speed in the subpixel calculations of the gray values, and thus in the accuracy of the extracted edge positions, different interpolation schemes can be selected in [gen_measure_rectangle2](#). (The interpolation only influences rectangles not aligned with the image axes and annular arcs.) The measure object generated with [gen_measure_rectangle2](#) is passed in [MeasureHandle](#).

After the one-dimensional edge profile has been calculated, subpixel edge locations are computed by convolving the profile with the derivatives of a Gaussian smoothing kernel of standard deviation [Sigma](#). Salient edges can be selected with the parameter [AmpThresh](#), which constitutes a threshold on the amplitude, i.e., the absolute value of the first derivative of the edge. Additionally, it is possible to select only positive edges, i.e., edges which constitute a dark-to-light transition in the direction of the major axis of the rectangle ([Transition](#) = ‘positive’), only negative edges, i.e., light-to-dark transitions ([Transition](#) = ‘negative’), or both types of edges ([Transition](#) = ‘all’). Finally, it is possible to select which edge points are returned.

Having extracted subpixel edge locations, features of these edges are evaluated by a corresponding fuzzy function, which can be set by `set.fuzzy_measure`. Which edges are selected can be determined with the parameter `FuzzyThresh`, which constitutes a threshold on the weight over all fuzzy sets, i.e., the geometric mean of the weights of the defined sets.

The selected edges are returned as single points, which lie on the major axis of the rectangle or annular arc, in (`RowEdge`,`ColumnEdge`). The corresponding edge amplitudes are returned in `Amplitude`, the fuzzy scores in `FuzzyScore`. In addition, the distance between consecutive edge points is returned in `Distance`. Here, `Distance[i]` corresponds to the distance between `Edge[i]` and `Edge[i+1]`, i.e., the tuple `Distance` contains one element less than the tuples `RowEdge` and `ColumnEdge`.

Attention

`fuzzy_measure_pos` only returns meaningful results if the assumptions that the edges are straight and perpendicular to the major axis of the rectangle are fulfilled. Thus, it should not be used to extract edges from curved objects, for example. Furthermore, the user should ensure that the rectangle is as close to perpendicular as possible to the edges in the image.

It should be kept in mind that `fuzzy_measure_pos` ignores the domain of `Image` for efficiency reasons. If certain regions in the image should be excluded from the measurement a new measure object with appropriately modified parameters should be generated.

Parameter

- ▷ **Image** (input_object) singlechannel-image $\leadsto$ *Hobject* : byte / uint2
Input image.
- ▷ **MeasureHandle** (input_control) measure_id $\leadsto$ *integer*
Measure object handle.
- ▷ **Sigma** (input_control) number $\leadsto$ *real*
Sigma of Gaussian smoothing.
Default Value : 1.0
Suggested values : $\text{Sigma} \in \{0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0\}$
Typical range of values : $0.4 \leq \text{Sigma} \leq 100$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : $\text{Sigma} \geq 0.4$
- ▷ **AmpThresh** (input_control) number $\leadsto$ *real*
Minimum edge amplitude.
Default Value : 30.0
Suggested values : $\text{AmpThresh} \in \{5.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 90.0, 110.0\}$
Typical range of values : $1 \leq \text{AmpThresh} \leq 255$ (lin)
Minimum Increment : 2
Recommended Increment : 0.5
- ▷ **FuzzyThresh** (input_control) number $\leadsto$ *real*
Minimum fuzzy value.
Default Value : 0.5
Suggested values : $\text{FuzzyThresh} \in \{0.1, 0.3, 0.5, 0.6, 0.7, 0.9\}$
Typical range of values : $0.0 \leq \text{FuzzyThresh} \leq 1.0$ (lin)
Recommended Increment : 0.1
- ▷ **Transition** (input_control) string $\leadsto$ *string*
Select light/dark or dark/light edges.
Default Value : 'all'
List of values : $\text{Transition} \in \{'all', 'positive', 'negative'\}$
- ▷ **RowEdge** (output_control) point.y-array $\leadsto$ *real*
Row coordinate of the edge point.
- ▷ **ColumnEdge** (output_control) point.x-array $\leadsto$ *real*
Column coordinate of the edge point.
- ▷ **Amplitude** (output_control) real-array $\leadsto$ *real*
Edge amplitude of the edge (with sign).
- ▷ **FuzzyScore** (output_control) real-array $\leadsto$ *real*
Fuzzy evaluation of the edges.

- ▷ **Distance** (output_control) real-array $\leadsto$ real
Distance between consecutive edges.

Result

If the parameter values are correct the operator `fuzzy_measure_pos` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

`fuzzy_measure_pos` is *reentrant* and processed *without parallelization*.

Possible Predecessors

`gen_measure_rectangle2`, `gen_measure_arc`, `set_fuzzy_measure`

Possible Successors

`close_measure`

Alternatives

`edges_sub_pix`, `measure_pos`

See also

`fuzzy_measure_pairing`, `fuzzy_measure_pairs`, `measure_pairs`

Module

Sub-pixel operators

gen_measure_arc (: : CenterRow, CenterCol, Radius, AngleStart, AngleExtent, AnnulusRadius, Width, Height, Interpolation : MeasureHandle)

Prepare the extraction of straight edges perpendicular to an annular arc.

`gen_measure_arc` prepares the extraction of *straight* edges which lie *perpendicular* to an annular arc. Here, annular arc denotes a circular arc with an associated width. The center of the arc is passed in the parameters `CenterRow` and `CenterCol`, its radius in `Radius`, the starting angle in `AngleStart`, and its angular extent relative to the starting angle in `AngleExtent`. If `AngleExtent` > 0, an arc with counterclockwise orientation is generated, otherwise an arc with clockwise orientation. The radius of the annular arc, i.e., half its width, is determined by `AnnulusRadius`.

The edge extraction algorithm is described in the documentation of the operator `measure_pos`. As discussed there, different types of interpolation can be used for the calculation of the one-dimensional gray value profile. For `Interpolation` = 'nearest_neighbor', the gray values in the measurement are obtained from the gray values of the closest pixel, i.e., by constant interpolation. For `Interpolation` = 'bilinear', bilinear interpolation is used, while for `Interpolation` = 'bicubic', bicubic interpolation is used.

With `gen_measure_arc`, all computations which can be used for multiple measurements are removed from the actual measurement. To effect this, an optimized data structure, a so-called measure object, is constructed and returned in `MeasureHandle`. In order to perform the measurement at the optimum speed, the size of the images for which subsequent measurements will be made must already be specified in `gen_measure_arc` with the parameters `Width` and `Height`. This technique increases the speed of the actual measurement significantly.

The system parameter 'int_zooming' (see `set_system`) affects the accuracy and speed of the calculations used to construct the measure object. If 'int_zooming' is set to 'true', the internal calculations are performed using fixed point arithmetic, leading to much shorter execution times. However, the geometric accuracy is slightly lower in this mode. If 'int_zooming' is set to 'false', the internal calculations are performed using floating point arithmetic, leading to the maximum geometric accuracy, but also to significantly increased execution times.

Parameter

- ▷ **CenterRow** (input_control) point.y $\leadsto$ real / integer
Row coordinate of the center of the arc.

Default Value : 100.0

Suggested values : CenterRow $\in$ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}

Typical range of values : $0.0 \leq \text{CenterRow} \leq 511.0$ (lin)

Minimum Increment : 1.0

Recommended Increment : 10.0

- ▷ **CenterCol** (input_control) point.x $\leadsto$ *real* / integer
 Column coordinate of the center of the arc.
Default Value : 100.0
Suggested values : CenterCol $\in$ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
Typical range of values : $0.0 \leq \text{CenterCol} \leq 511.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
- ▷ **Radius** (input_control) number $\leadsto$ *real* / integer
 Radius of the arc.
Default Value : 50.0
Suggested values : Radius $\in$ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
Typical range of values : $0.0 \leq \text{Radius} \leq 511.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
- ▷ **AngleStart** (input_control) angle.rad $\leadsto$ *real* / integer
 Start angle of the arc in radians.
Default Value : 0.0
Suggested values : AngleStart $\in$ {-3.14159265359, -2.35619449019, -1.5707963268, -0.785398163398, 0.0, 0.785398163398, 1.5707963268, 2.35619449019, 3.14159265359}
Typical range of values : $-3.14159265359 \leq \text{AngleStart} \leq 3.14159265359$ (lin)
Minimum Increment : 0.0314159265359
Recommended Increment : 0.314159265359
- ▷ **AngleExtent** (input_control) angle.rad $\leadsto$ *real* / integer
 Angular extent of the arc in radians.
Default Value : 6.28318530718
Suggested values : AngleExtent $\in$ {-6.28318530718, -5.49778714378, -4.71238898038, -3.926990817, -3.14159265359, -2.35619449019, -1.5707963268, -0.785398163398, 0.785398163398, 1.5707963268, 2.35619449019, 3.14159265359, 3.926990817, 4.71238898038, 5.49778714378, 6.28318530718}
Typical range of values : $-6.28318530718 \leq \text{AngleExtent} \leq 6.28318530718$ (lin)
Minimum Increment : 0.0314159265359
Recommended Increment : 0.314159265359
Restriction : AngleExtent $\neq 0.0$
- ▷ **AnnulusRadius** (input_control) number $\leadsto$ *real* / integer
 Radius (half width) of the annulus.
Default Value : 10.0
Suggested values : AnnulusRadius $\in$ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
Typical range of values : $0.0 \leq \text{AnnulusRadius} \leq 511.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
Restriction : AnnulusRadius $\leq$ Radius
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer*
 Width of the image to be processed subsequently.
Default Value : 512
Suggested values : Width $\in$ {128, 160, 192, 256, 320, 384, 512, 640, 768}
Typical range of values : $0 \leq \text{Width} \leq 1024$ (lin)
Minimum Increment : 1
Recommended Increment : 16
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer*
 Height of the image to be processed subsequently.
Default Value : 512
Suggested values : Height $\in$ {120, 128, 144, 240, 256, 288, 480, 512, 576}
Typical range of values : $0 \leq \text{Height} \leq 1024$ (lin)
Minimum Increment : 1
Recommended Increment : 16
- ▷ **Interpolation** (input_control) string $\leadsto$ *string*
 Type of interpolation to be used.
Default Value : 'nearest_neighbor'
List of values : Interpolation $\in$ {'nearest_neighbor', 'bilinear', 'bicubic'}

▷ **MeasureHandle** (output_control) measure_id $\leadsto$ integer
Measure object handle.

Result

If the parameter values are correct, the operator **gen_measure_arc** returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

gen_measure_arc is *reentrant* and processed *without* parallelization.

Possible Predecessors

draw_circle

Possible Successors

measure_pos, **measure_pairs**, **fuzzy_measure_pos**, **fuzzy_measure_pairs**,
fuzzy_measure_pairing

Alternatives

edges_sub_pix

See also

gen_measure_rectangle2

Module

Sub-pixel operators

gen_measure_rectangle2 (: : Row, Column, Phi, Length1, Length2,
Width, Height, Interpolation : MeasureHandle)

Prepare the extraction of straight edges perpendicular to a rectangle.

gen_measure_rectangle2 prepares the extraction of *straight* edges which lie *perpendicular* to the major axis of a rectangle. The center of the rectangle is passed in the parameters **Row** and **Column**, the direction of the major axis of the rectangle in **Phi**, and the length of the two axes, i.e., half the diameter of the rectangle, in **Length1** and **Length2**.

The edge extraction algorithm is described in the documentation of the operator **measure_pos**. As discussed there, different types of interpolation can be used for the calculation of the one-dimensional gray value profile. For **Interpolation** = 'nearest_neighbor', the gray values in the measurement are obtained from the gray values of the closest pixel, i.e., by constant interpolation. For **Interpolation** = 'bilinear', bilinear interpolation is used, while for **Interpolation** = 'bicubic', bicubic interpolation is used.

With **gen_measure_rectangle2**, all computations which can be used for multiple measurements are removed from the actual measurement. To effect this, an optimized data structure, a so-called measure object, is constructed and returned in **MeasureHandle**. In order to perform the measurement at the optimum speed, the size of the images for which subsequent measurements will be made must already be specified in **gen_measure_rectangle2** with the parameters **Width** and **Height**. This technique increases the speed of the actual measurement significantly.

The system parameter 'int_zooming' (see **set_system**) affects the accuracy and speed of the calculations used to construct the measure object. If 'int_zooming' is set to 'true', the internal calculations are performed using fixed point arithmetic, leading to much shorter execution times. However, the geometric accuracy is slightly lower in this mode. If 'int_zooming' is set to 'false', the internal calculations are performed using floating point arithmetic, leading to the maximum geometric accuracy, but also to significantly increased execution times.

Parameter

▷ **Row** (input_control) rectangle2.center.y $\leadsto$ real / integer
Row coordinate of the center of the rectangle.
Default Value : 50.0
Suggested values : Row $\in$ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
Typical range of values : 0.0 $\leq$ Row $\leq$ 511.0 (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0

- ▷ **Column** (input_control) rectangle2.center.x $\leadsto$ *real* / *integer*
 Column coordinate of the center of the rectangle.
Default Value : 100.0
Suggested values : Column $\in \{10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0\}$
Typical range of values : $0.0 \leq \text{Column} \leq 511.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
- ▷ **Phi** (input_control) rectangle2.angle.rad $\leadsto$ *real* / *integer*
 Angle of longitudinal axis of the rectangle to horizontal (radians).
Default Value : 0.0
Suggested values : Phi $\in \{-1.178097, -0.785398, -0.392699, 0.0, 0.392699, 0.785398, 1.178097\}$
Typical range of values : $-1.178097 \leq \text{Phi} \leq 1.178097$ (lin)
Minimum Increment : 0.001
Recommended Increment : 0.1
Restriction : $(-\pi < \text{Phi}) \wedge (\text{Phi} \leq \pi)$
- ▷ **Length1** (input_control) rectangle2.hwidth $\leadsto$ *real* / *integer*
 Half width of the rectangle.
Default Value : 200.0
Suggested values : Length1 $\in \{3.0, 5.0, 10.0, 15.0, 20.0, 50.0, 100.0, 200.0, 300.0, 500.0\}$
Typical range of values : $0.0 \leq \text{Length1} \leq 511.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
- ▷ **Length2** (input_control) rectangle2.hheight $\leadsto$ *real* / *integer*
 Half height of the rectangle.
Default Value : 100.0
Suggested values : Length2 $\in \{1.0, 2.0, 3.0, 5.0, 10.0, 15.0, 20.0, 50.0, 100.0, 200.0\}$
Typical range of values : $0.0 \leq \text{Length2} \leq 511.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
Restriction : Length2 $\leq$ Length1
- ▷ **Width** (input_control) extent.x $\leadsto$ *integer*
 Width of the image to be processed subsequently.
Default Value : 512
Suggested values : Width $\in \{128, 160, 192, 256, 320, 384, 512, 640, 768\}$
Typical range of values : $0 \leq \text{Width} \leq 1024$ (lin)
Minimum Increment : 1
Recommended Increment : 16
- ▷ **Height** (input_control) extent.y $\leadsto$ *integer*
 Height of the image to be processed subsequently.
Default Value : 512
Suggested values : Height $\in \{120, 128, 144, 240, 256, 288, 480, 512, 576\}$
Typical range of values : $0 \leq \text{Height} \leq 1024$ (lin)
Minimum Increment : 1
Recommended Increment : 16
- ▷ **Interpolation** (input_control) string $\leadsto$ *string*
 Type of interpolation to be used.
Default Value : 'nearest_neighbor'
List of values : Interpolation $\in \{\text{'nearest\_neighbor'}, \text{'bilinear'}, \text{'bicubic'}\}$
- ▷ **MeasureHandle** (output_control) measure_id $\leadsto$ *integer*
 Measure object handle.

Result

If the parameter values are correct the operator `gen_measure_rectangle2` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

`gen_measure_rectangle2` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`draw_rectangle2`

Possible Successors

[measure_pos](#), [measure_pairs](#), [fuzzy_measure_pos](#), [fuzzy_measure_pairs](#),
[fuzzy_measure_pairing](#), [measure_thresh](#)

Alternatives

[edges_sub_pix](#)

See also

[gen_measure_arc](#)

Module

Sub-pixel operators

```
measure_pairs ( Image : : MeasureHandle, Sigma, Threshold, Transition,
Select : RowEdgeFirst, ColumnEdgeFirst, AmplitudeFirst, RowEdgeSecond,
ColumnEdgeSecond, AmplitudeSecond, IntraDistance, InterDistance )
```

Extract straight edge pairs perpendicular to a rectangle or annular arc.

[measure_pairs](#) serves to extract *straight* edge pairs which lie *perpendicular* to the major axis of a rectangle or annular arc.

The extraction algorithm is identical to [measure_pos](#). In addition the edges are grouped to pairs: If [Transition](#) = 'positive', the edge points with a dark-to-light transition in the direction of the major axis of the rectangle are returned in [RowEdgeFirst](#) and [ColumnEdgeFirst](#). In this case, the corresponding edges with a light-to-dark transition are returned in [RowEdgeSecond](#) and [ColumnEdgeSecond](#). If [Transition](#) = 'negative', the behavior is exactly opposite. If [Transition](#) = 'all', the first detected edge defines the transition for [RowEdgeFirst](#) and [ColumnEdgeFirst](#). If more than one consecutive edge with the same transition is found, the first one is used as a pair element. This behavior may cause problems in applications in which the threshold [Threshold](#) cannot be selected high enough to suppress consecutive edges of the same transition. For these applications, a second pairing mode exists that only selects the respective strongest edges of a sequence of consecutive rising and falling edges. This mode is selected by appending '*strongest*' to any of the above modes for [Transition](#), e.g., '*negative-strongest*'. Finally, it is possible to select which edge pairs are returned. If [Select](#) is set to 'all', all edge pairs are returned. If it is set to '*first*', only the first of the extracted edge pairs is returned, while it is set to '*last*', only the last one is returned.

The extracted edges are returned as single points which lie on the major axis of the rectangle. The corresponding edge amplitudes are returned in [AmplitudeFirst](#) and [AmplitudeSecond](#). In addition, the distance between each edge pair is returned in [IntraDistance](#) and the distance between consecutive edge pairs is returned in [InterDistance](#). Here, [IntraDistance](#)[i] corresponds to the distance between [EdgeFirst](#)[i] and [EdgeSecond](#)[i], while [InterDistance](#)[i] corresponds to the distance between [EdgeSecond](#)[i] and [EdgeFirst](#)[i+1], i.e., the tuple [InterDistance](#) contains one element less than the tuples of the edge pairs.

Attention

[measure_pairs](#) only returns meaningful results if the assumptions that the edges are straight and perpendicular to the major axis of the rectangle are fulfilled. Thus, it should not be used to extract edges from curved objects, for example. Furthermore, the user should ensure that the rectangle is as close to perpendicular as possible to the edges in the image.

It should be kept in mind that [measure_pairs](#) ignores the domain of [Image](#) for efficiency reasons. If certain regions in the image should be excluded from the measurement a new measure object with appropriately modified parameters should be generated.

Parameter

- ▷ **Image** (input_object) singlechannel-image $\leadsto$ *Hobject* : byte / uint2
Input image.
- ▷ **MeasureHandle** (input_control) measure_id $\leadsto$ *integer*
Measure object handle.

- ▷ **Sigma** (input_control) number $\leadsto$ *real*
Sigma of gaussian smoothing.
Default Value : 1.0
Suggested values : $\text{Sigma} \in \{0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0\}$
Typical range of values : $0.4 \leq \text{Sigma} \leq 100$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : $\text{Sigma} \geq 0.4$
- ▷ **Threshold** (input_control) number $\leadsto$ *real*
Minimum edge amplitude.
Default Value : 30.0
Suggested values : $\text{Threshold} \in \{5.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 90.0, 110.0\}$
Typical range of values : $1 \leq \text{Threshold} \leq 255$ (lin)
Minimum Increment : 0.5
Recommended Increment : 2
- ▷ **Transition** (input_control) string $\leadsto$ *string*
Type of gray value transition that determines how edges are grouped to edge pairs.
Default Value : 'all'
List of values : $\text{Transition} \in \{\text{'all'}, \text{'positive'}, \text{'negative'}, \text{'all\_strongest'}, \text{'positive\_strongest'}, \text{'negative\_strongest'}\}$
- ▷ **Select** (input_control) string $\leadsto$ *string*
Selection of edge pairs.
Default Value : 'all'
List of values : $\text{Select} \in \{\text{'all'}, \text{'first'}, \text{'last'}\}$
- ▷ **RowEdgeFirst** (output_control) point.y-array $\leadsto$ *real*
Row coordinate of the center of the first edge.
- ▷ **ColumnEdgeFirst** (output_control) point.x-array $\leadsto$ *real*
Column coordinate of the center of the first edge.
- ▷ **AmplitudeFirst** (output_control) real-array $\leadsto$ *real*
Edge amplitude of the first edge (with sign).
- ▷ **RowEdgeSecond** (output_control) point.y-array $\leadsto$ *real*
Row coordinate of the center of the second edge.
- ▷ **ColumnEdgeSecond** (output_control) point.x-array $\leadsto$ *real*
Column coordinate of the center of the second edge.
- ▷ **AmplitudeSecond** (output_control) real-array $\leadsto$ *real*
Edge amplitude of the second edge (with sign).
- ▷ **IntraDistance** (output_control) real-array $\leadsto$ *real*
Distance between edges of an edge pair.
- ▷ **InterDistance** (output_control) real-array $\leadsto$ *real*
Distance between consecutive edge pairs.

Result

If the parameter values are correct the operator `measure_pairs` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

`measure_pairs` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`gen_measure_rectangle2`

Possible Successors

`close_measure`

Alternatives

`edges_sub_pix`, `fuzzy_measure_pairs`, `fuzzy_measure_pairing`

See also

`measure_pos`, `fuzzy_measure_pos`

Module

Sub-pixel operators

```
measure_pos ( Image : : MeasureHandle, Sigma, Threshold, Transition,
Select : RowEdge, ColumnEdge, Amplitude, Distance )
```

Extract straight edges perpendicular to a rectangle or annular arc.

measure_pos extracts *straight* edges which lie *perpendicular* to the major axis of a rectangle or annular arc.

The algorithm works by averaging the gray values in “slices” perpendicular to the major axis of the rectangle or annular arc in order to obtain a one-dimensional edge profile. The sampling is done at subpixel positions in the image **Image** at integer row and column distances (in the coordinate frame of the rectangle) from the center of the rectangle. Since this involves some calculations which can be used repeatedly in several measurements, the operator **gen_measure_rectangle2** or **gen_measure_arc** is used to perform these calculations only once, thus increasing the speed of **measure_pos** significantly. Since there is a trade-off between accuracy and speed in the subpixel calculations of the gray values, and thus in the accuracy of the extracted edge positions, different interpolation schemes can be selected in **gen_measure_rectangle2**. (The interpolation only influences rectangles not aligned with the image axes.) The measure object generated with **gen_measure_rectangle2** is passed in **MeasureHandle**.

After the one-dimensional edge profile has been calculated, subpixel edge locations are computed by convolving the profile with the derivatives of a Gaussian smoothing kernel of standard deviation **Sigma**. Salient edges can be selected with the parameter **Threshold**, which constitutes a threshold on the amplitude, i.e., the absolute value of the first derivative of the edge. Additionally, it is possible to select only positive edges, i.e., edges which constitute a dark-to-light transition in the direction of the major axis of the rectangle or the arc (**Transition** = 'positive'), only negative edges, i.e., light-to-dark transitions (**Transition** = 'negative'), or both types of edges (**Transition** = 'all'). Finally, it is possible to select which edge points are returned. If **Select** is set to 'all', all edge points are returned. If it is set to 'first', only the first of the extracted edge points is returned, while it is set to 'last', only the last one is returned.

The extracted edges are returned as single points which lie on the major axis of the rectangle or arc in (**RowEdge**,**ColumnEdge**). The corresponding edge amplitudes are returned in **Amplitude**. In addition, the distance between consecutive edge points is returned in **Distance**. Here, Distance[i] corresponds to the distance between Edge[i] and Edge[i+1], i.e., the tuple **Distance** contains one element less than the tuples **RowEdge** and **ColumnEdge**.

Attention

measure_pos only returns meaningful results if the assumptions that the edges are straight and perpendicular to the major axis of the rectangle or arc are fulfilled. Thus, it should not be used to extract edges from curved objects, for example. Furthermore, the user should ensure that the rectangle or arc is as close to perpendicular as possible to the edges in the image.

It should be kept in mind that **measure_pos** ignores the domain of **Image** for efficiency reasons. If certain regions in the image should be excluded from the measurement a new measure object with appropriately modified parameters should be generated.

Parameter

- ▷ **Image** (input_object) singlechannel-image $\rightsquigarrow$ *Hobject* : byte / uint2
Input image.
- ▷ **MeasureHandle** (input_control) measure_id $\rightsquigarrow$ *integer*
Measure object handle.
- ▷ **Sigma** (input_control) number $\rightsquigarrow$ *real*
Sigma of gaussian smoothing.
Default Value : 1.0
Suggested values : $\text{Sigma} \in \{0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0\}$
Typical range of values : $0.4 \leq \text{Sigma} \leq 100$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : $\text{Sigma} \geq 0.4$

- ▷ **Threshold** (input_control) number $\leadsto$ *real*
Minimum edge amplitude.
Default Value : 30.0
Suggested values : Threshold $\in \{5.0, 10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 90.0, 110.0\}$
Typical range of values : $1 \leq \text{Threshold} \leq 255$ (lin)
Minimum Increment : 2
Recommended Increment : 0.5
- ▷ **Transition** (input_control) string $\leadsto$ *string*
Light/dark or dark/light edge.
Default Value : 'all'
List of values : Transition $\in \{\text{'all'}, \text{'positive'}, \text{'negative'}\}$
- ▷ **Select** (input_control) string $\leadsto$ *string*
Selection of end points.
Default Value : 'all'
List of values : Select $\in \{\text{'all'}, \text{'first'}, \text{'last'}\}$
- ▷ **RowEdge** (output_control) point.y-array $\leadsto$ *real*
Row coordinate of the center of the edge.
- ▷ **ColumnEdge** (output_control) point.x-array $\leadsto$ *real*
Column coordinate of the center of the edge.
- ▷ **Amplitude** (output_control) real-array $\leadsto$ *real*
Edge amplitude of the edge (with sign).
- ▷ **Distance** (output_control) real-array $\leadsto$ *real*
Distance between consecutive edges.

Result

If the parameter values are correct the operator `measure_pos` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

`measure_pos` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`gen_measure_rectangle2`

Possible Successors

`close_measure`

Alternatives

`edges_sub_pix`, `fuzzy_measure_pos`

See also

`measure_pairs`, `fuzzy_measure_pairs`, `fuzzy_measure_pairing`

Module

Sub-pixel operators

measure_projection (Image : : MeasureHandle : GrayValues)

Extract a gray value profile perpendicular to a rectangle or annular arc.

`measure_projection` extracts a one-dimensional gray value profile perpendicular to a rectangle or annular arc. This is done by averaging the gray values in “slices” perpendicular to the major axis of the rectangle or arc. The sampling is done at subpixel positions in the image `Image` at integer row and column distances (in the coordinate frame of the rectangle) from the center of the rectangle. Since this involves some calculations which can be used repeatedly in several projections, the operator `gen_measure_rectangle2` is used to perform these calculations only once, thus increasing the speed of `measure_projection` significantly. Since there is a trade-off between accuracy and speed in the subpixel calculations of the gray values, different interpolation schemes can be selected in `gen_measure_rectangle2` (the interpolation only influences rectangles not aligned with the image axes). The measure object generated with `gen_measure_rectangle2` is passed in `MeasureHandle`.

Attention

It should be kept in mind that `measure_projection` ignores the domain of `Image` for efficiency reasons. If

certain regions in the image should be excluded from the measurement a new measure object with appropriately modified parameters should be generated.

Parameter	
▷ Image (input_object)	singlechannel-image $\leadsto$ <i>Hobject</i> : byte / uint2 Input image.
▷ MeasureHandle (input_control)	measure_id $\leadsto$ <i>integer</i> Measure object handle.
▷ GrayValues (output_control)	number-array $\leadsto$ <i>real</i> Gray value profile.
Result	
If the parameter values are correct the operator <code>measure_projection</code> returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.	
Parallelization Information	
<code>measure_projection</code> is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Predecessors	
<code>gen_measure_rectangle2</code>	
Possible Successors	
<code>close_measure</code>	
Alternatives	
<code>gray_projections</code>	
Module	
Sub-pixel operators	

measure_thresh (Image : : MeasureHandle, Sigma, Threshold,
Select : RowThresh, ColumnThresh, Distance)

Extracting points with a particular grey value along a rectangle or an annular arc.

`measure_thresh` extracts points for which the gray value within an one-dimensional gray value profile is equal to the specified threshold `Threshold`. The gray value profile is projected onto the major axis of the measure rectangle which is passed with the parameter `MeasureHandle`, so the threshold points calculated within the gray value profile correspond to certain image coordinates on the rectangle's major axis. These coordinates are returned as the operator results in `RowThresh` and `ColumnThresh`.

If the gray value profile intersects the threshold line for several times, the parameter `Select` determines which values to return. Possible settings are 'first', 'last', 'first.last' (first and last) or 'all'. For the last two cases `Distance` returns the distances between the calculated points.

The gray value profile is created by averaging the gray values along all line segments, which are defined by the measure rectangle as follows:

1. The segments are perpendicular to the major axis of the rectangle,
2. they have an integer distance to the center of the rectangle,
3. the rectangle bounds the segments.

For every line segment, the average of the gray values of all points with an integer distance to the major axis is calculated. Due to translation and rotation of the measure rectangle with respect to the image coordinates the input image `Image` is in general sampled at subpixel positions.

Since this involves some calculations which can be used repeatedly in several projections, the operator `gen_measure_rectangle2` is used to perform these calculations only once in advance. Here, the measure object `MeasureHandle` is generated and different interpolation schemes can be selected.

Attention
`measure_thresh` only returns meaningful results if the assumptions that the edges are straight and perpendicular to the major axis of the rectangle are fulfilled. Thus, it should not be used to extract edges from curved objects, for example. Furthermore, the user should ensure that the rectangle is as close to perpendicular as possible to the edges in the image.

It should be kept in mind that `measure_thresh` ignores the domain of `Image` for efficiency reasons. If certain regions in the image should be excluded from the measurement a new measure object with appropriately modified parameters should be generated.

Parameter	
▷ Image (input_object)	singlechannel-image $\leadsto$ <i>Hobject</i> : byte / uint2 Input image.
▷ MeasureHandle (input_control)	measure_id $\leadsto$ <i>integer</i> Measure object handle.
▷ Sigma (input_control)	number $\leadsto$ <i>real</i> Sigma of gaussian smoothing. Default Value : 1.0 Suggested values : $\text{Sigma} \in \{0.0, 0.4, 0.6, 0.8, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 7.0, 10.0\}$ Typical range of values : $0.4 \leq \text{Sigma} \leq 100$ (lin) Minimum Increment : 0.01 Recommended Increment : 0.1 Restriction : $\text{Sigma} \geq 0.0$
▷ Threshold (input_control)	number $\leadsto$ <i>real</i> Threshold. Default Value : 128.0 Typical range of values : $0 \leq \text{Threshold} \leq 255$ (lin) Minimum Increment : 1 Recommended Increment : 0.5
▷ Select (input_control)	string $\leadsto$ <i>string</i> Selection of points. Default Value : 'all' List of values : $\text{Select} \in \{'all', 'first', 'last', 'first\_last'\}$
▷ RowThresh (output_control)	point.y-array $\leadsto$ <i>real</i> Row coordinates of points with threshold value.
▷ ColumnThresh (output_control)	point.x-array $\leadsto$ <i>real</i> Column coordinates of points with threshold value.
▷ Distance (output_control)	real-array $\leadsto$ <i>real</i> Distance between consecutive points.

Result
If the parameter values are correct the operator <code>measure_thresh</code> returns the value 2 (H_MSG_TRUE). Otherwise, an exception handling is raised.

Parallelization Information
<code>measure_thresh</code> is <i>reentrant</i> and processed <i>without</i> parallelization.

Possible Predecessors
<code>gen_measure_rectangle2</code>

Possible Successors
<code>close_measure</code>

Alternatives
<code>measure_pos</code> , <code>edges_sub_pix</code> , <code>measure_pairs</code>

Module
Sub-pixel operators

reset_fuzzy_measure (: : MeasureHandle, SetType :)

Reset a fuzzy member function.

`reset_fuzzy_measure` discards a fuzzy member function of the fuzzy set `SetType`. This member function should have been set by `set_fuzzy_measure` before.

Parameter
▷ MeasureHandle (input_control) <code>measure_id</code> $\leadsto$ <i>integer</i> Measure object handle. ▷ SetType (input_control) <code>string</code> $\leadsto$ <i>string</i> Selection of the fuzzy set. Default Value : 'contrast' List of values : <code>SetType</code> $\in$ {'position', 'position_pair', 'size', 'gray', 'contrast'}
Parallelization Information
<code>reset_fuzzy_measure</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
Possible Predecessors
<code>set_fuzzy_measure</code>
Possible Successors
<code>fuzzy_measure_pos</code> , <code>fuzzy_measure_pairs</code>
See also
<code>set_fuzzy_measure</code> , <code>set_fuzzy_measure_norm_pair</code>
Module
Sub-pixel operators

set_fuzzy_measure (: : MeasureHandle, SetType, Function :)

Specify a fuzzy member function.

`set_fuzzy_measure` specifies a fuzzy member function passed in `Function`. The specified fuzzy functions enable `fuzzy_measure_pos` and `fuzzy_measure_pairs` / `fuzzy_measure_pairing` to evaluate and select the detected edge candidates. For this purpose, weighting characteristics for different edge features can be defined by one function each. Such a specified feature is called fuzzy set. Specifying no function for a fuzzy set means not to use this feature for the final edge evaluation. Setting a second fuzzy function to a set means to discard the first defined function and replace it by the second one. A previously defined fuzzy member function can be discarded completely by `reset_fuzzy_measure`.

Functions for five different fuzzy set types selected by the `SetType` parameter can be defined, the sub types of a set being mutual exclusive:

- 'contrast' will use the fuzzy function to evaluate the amplitudes of the edge candidates. When extracting edge pairs, the fuzzy evaluation is obtained by the geometric average of the fuzzy contrast scores of both edges.
- The fuzzy function of 'position' evaluates the distance of each edge candidate to the reference point of the measure object, generated by `gen_measure_arc` or `gen_measure_rectangle2`. The reference point is located at the beginning whereas 'position_center' or 'position_end' sets the reference point to the middle or the end of the one-dimensional gray value profile instead. If the fuzzy position evaluation depends on the position of the object along the profile, 'position_first_edge' / 'position_last_edge' sets the reference point at the position of the first/last extracted edge. When extracting edge pairs the position of a pair is referenced by the geometric average of the fuzzy position scores of both edges.
- Similar to 'position', 'position_pair' evaluates the distance of each edge pair to the reference point of the measure object. The position of a pair is defined by the center point between both edges. The object's reference can be set by 'position_pair_center', 'position_pair_end' and 'position_first_pair', 'position_last_pair', respectively. Contrary to 'position', this set is only used by `fuzzy_measure_pairs` / `fuzzy_measure_pairing`.
- 'size' denotes a fuzzy set that evaluates the normed distance of the two edges of a pair in pixels. This set is only used by `fuzzy_measure_pairs` / `fuzzy_measure_pairing`. Specifying an upper bound for the size by terminating the member function with a corresponding fuzzy value of 0.0 will speed up `fuzzy_measure_pairs` / `fuzzy_measure_pairing` because not all possible pairs need to be considered.
- 'gray' sets a fuzzy function to weight the mean projected gray value between two edges of a pair. This set is only used by `fuzzy_measure_pairs` / `fuzzy_measure_pairing`.

A fuzzy member function is defined as a piecewise linear function by at least two pairs of values, sorted in an ascending order by their x value. The x values represent the edge feature and must lie within the parameter space of the set type, i.e., in case of 'contrast' and 'gray' feature and, e.g., byte images within the range $0.0 \leq x \leq 255.0$. In case of 'size' x has to satisfy $0.0 \leq x$ whereas in case of 'position' x can be any real number. The y values of the fuzzy function represent the weight of the corresponding feature value and have to satisfy the range of $0.0 \leq y \leq 1.0$. Outside of the function's interval, defined by the smallest and the greatest x value, the y values of the interval borders are continued constantly. Such Fuzzy member functions can be generated by [create_funcnt_1d_pairs](#).

If more than one set is defined, [fuzzy_measure_pos](#) / [fuzzy_measure_pairs](#) / [fuzzy_measure_pairing](#) yield the overall fuzzy weighting by the geometric middle of the weights of each set.

Parameter

- ▷ **MeasureHandle** (input_control) *measure_id* $\leadsto$ *integer*
Measure object handle.
- ▷ **SetType** (input_control) *string* $\leadsto$ *string*
Selection of the fuzzy set.
Default Value : 'contrast'
List of values : SetType $\in$ {'position', 'position_center', 'position_end', 'position_first_edge', 'position_last_edge', 'position_pair_center', 'position_pair_end', 'position_first_pair', 'position_last_pair', 'size', 'gray', 'contrast'}
- ▷ **Function** (input_control) *function_1d-array* $\leadsto$ *real* / *integer*
Fuzzy member function.

Example

```
/* how to use a fuzzy function */
...
gen_measure_rectangle2 (50, 100, 0, 200, 100, 512, 512, 'nearest_neighbor',
                      MeasureHandle)
/* create a generalized fuzzy function to evaluate edge pairs
 * (30% uncertainty). */
create_funcnt_1d_pairs ([0.7,1.0,1.3], [0.0,1.0,0.0], SizeFunction)
/* and transform it to expected size of 13.45 pixels */
transform_funcnt_1d (SizeFunction, [1.0,0.0,13.45,0.0], TransformedFunction)
set_fuzzy_measure (MeasureHandle, 'size', SizeFunction)

fuzzy_measure_pairs (Image, MeasureHandle, 1, 30, 0.5, 'all', RowEdgeFirst,
                    ColumnEdgeFirst, AmplitudeFirst, RowEdgeSecond,
                    ColumnEdgeSecond, AmplitudeSecond, RowEdgeCenter,
                    ColumnEdgeCenter, FuzzyScore, IntraDistance,
                    InterDistance)
```

Parallelization Information

[set_fuzzy_measure](#) is *reentrant* and processed *without parallelization*.

Possible Predecessors

[gen_measure_arc](#), [gen_measure_rectangle2](#), [create_funcnt_1d_pairs](#),
[transform_funcnt_1d](#)

Possible Successors

[fuzzy_measure_pos](#), [fuzzy_measure_pairs](#)

Alternatives

[set_fuzzy_measure_norm_pair](#)

See also

[reset_fuzzy_measure](#)

Module

Sub-pixel operators

set_fuzzy_measure_norm_pair (: : MeasureHandle, PairSize, SetType, Function :)

Specify a normalized fuzzy member function for edge pairs.

set_fuzzy_measure_norm_pair specifies a normalized fuzzy member function passed in **Function**. The specified fuzzy functions enables **fuzzy_measure_pos**, **fuzzy_measure_pairs** and **fuzzy_measure_pairing** to evaluate and select the detected candidates of edges and edge pairs. For this purpose, weighting characteristics for different edge features can be defined by one function each. Such a specified feature is called fuzzy set. Specifying no function for a fuzzy set means not to use this feature for the final edge evaluation. Setting a second fuzzy function to a fuzzy set means to discard the first defined function and replace it by the second one. In difference to **set_fuzzy_measure**, the abscissa x of these member functions must be defined relative to the desired size s of the edge pairs (passed in **PairSize**). This enables a generalized usage of the defined functions. A previously defined normalized fuzzy member function can be discarded completely by **reset_fuzzy_measure**.

Functions for three different fuzzy set types selected by the **SetType** parameter can be defined, the sub types of a set being mutual exclusive:

- 'size' denotes a fuzzy set that valuates the normalized distance of two edges of a pair in pixels:

$$x = \frac{d}{s}(x \geq 0) \text{ .}$$

Specifying an upper bound x_{max} for the size by terminating the member function with a corresponding fuzzy value of 0.0 will speed up **fuzzy_measure_pairs** / **fuzzy_measure_pairing** because not all possible pairs must be considered. Additionally, this fuzzy set can also be specified as a normalized size difference by 'size_diff'

$$x = \frac{s - d}{s}(x \leq 1)$$

and a absolute normalized size difference by 'size_abs_diff'

$$x = \frac{|s - d|}{s}(0 \leq x \leq 1) \text{ .}$$

- The fuzzy function of 'position' evaluates the signed distance p of each edge candidate to the reference point of the measure object, generated by **gen_measure_arc** or **gen_measure_rectangle2**:

$$x = \frac{p}{s} \text{ .}$$

The reference point is located at the beginning whereas 'position_center' or 'position_end' sets the reference point to the middle or the end of the one-dimensional gray value profile, instead. If the fuzzy position valuation depends on the position of the object along the profile 'position_first_edge' / 'position_last_edge' sets the reference point at the position of the first/last extracted edge. When extracting edge pairs, the position of a pair is referenced by the geometric average of the fuzzy position scores of both edges.

- Similar to 'position', 'position_pair' evaluates the signed distance of each edge pair to the reference point of the measure object. The position of a pair is defined by the center point between both edges. The object's reference can be set by 'position_pair_center', 'position_pair_end' and 'position_first_pair', 'position_last_pair', respectively. Contrary to 'position', this set is only used by **fuzzy_measure_pairs** / **fuzzy_measure_pairing**.

A normalized fuzzy member function is defined as a piecewise linear function by at least two pairs of values, sorted in an ascending order by their x value. The y values of the fuzzy function represent the weight of the corresponding feature value and must satisfy the range of $0.0 \leq y \leq 1.0$. Outside of the function's interval, defined by the smallest and the greatest x value, the y values of the interval borders are continued constantly. Such Fuzzy member functions can be generated by **create_funct_1d_pairs**.

If more than one set is defined, **fuzzy_measure_pos** / **fuzzy_measure_pairs** / **fuzzy_measure_pairing** yield the overall fuzzy weighting by the geometric mean of the weights of each set.

Parameter	
▷ MeasureHandle (input_control)	measure_id $\leadsto$ integer Measure object handle.
▷ PairSize (input_control)	number $\leadsto$ real / integer Favored width of edge pairs. Default Value : 10.0 List of values : PairSize $\in \{4.0, 6.0, 8.0, 10.0, 15.0, 20.0, 30.0\}$ Typical range of values : $0.0 \leq \text{PairSize}$ Minimum Increment : 0.1 Recommended Increment : 1.0
▷ SetType (input_control)	string $\leadsto$ string Selection of the fuzzy set. Default Value : 'size_abs_diff' List of values : SetType $\in \{\text{'size'}, \text{'size\_diff'}, \text{'size\_abs\_diff'}, \text{'position'}, \text{'position\_center'}, \text{'position\_end'}, \text{'position\_first\_edge'}, \text{'position\_last\_edge'}, \text{'position\_pair\_center'}, \text{'position\_pair\_end'}, \text{'position\_first\_pair'}, \text{'position\_last\_pair'}\}$
▷ Function (input_control)	function_1d-array $\leadsto$ real / integer Fuzzy member function.

Example

```

/* how to use a fuzzy function */
...
gen_measure_rectangle2 (50, 100, 0, 200, 100, 512, 512, 'nearest_neighbor',
                      MeasureHandle)
/* create a generalized fuzzy function to evaluate edge pairs
 * (30% uncertainty). */
create_func_1d_pairs ([0.7,1.0,1.3], [0.0,1.0,0.0], SizeFunction)
/* and set it for an expected pair size of 13.45 pixels */
set_fuzzy_measure_norm_pair (MeasureHandle, 13.45, 'size', SizeFunction)

fuzzy_measure_pairs (Image, MeasureHandle, 1, 30, 0.5, 'all', RowEdgeFirst,
                    ColumnEdgeFirst, AmplitudeFirst, RowEdgeSecond,
                    ColumnEdgeSecond, AmplitudeSecond, RowEdgeCenter,
                    ColumnEdgeCenter, FuzzyScore, IntraDistance,
                    InterDistance)

```

Parallelization Information

[set_fuzzy_measure_norm_pair](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_measure_arc](#), [gen_measure_rectangle2](#), [create_func_1d_pairs](#)

Possible Successors

[fuzzy_measure_pairs](#), [fuzzy_measure_pairing](#)

Alternatives

[transform_func_1d](#), [set_fuzzy_measure](#)

See also

[reset_fuzzy_measure](#)

Module

Sub-pixel operators

translate_measure (: : MeasureHandle, Row, Column :)

Translate a measure object.

[translate_measure](#) translates the reference point of the measure object given by [MeasureHandle](#) to the point ([Row](#),[Column](#)). If the measure object and the translated measure object lie completely within the image,

the measure object is shifted to the new reference point in an efficient manner. Otherwise, the measure object is generated anew with [gen_measure_rectangle2](#) or [gen_measure_arc](#) using the parameters that were specified when the measure object was created and the new reference point.

Parameter

- ▷ **MeasureHandle** (input_control) `measure_id` $\leadsto$ integer
Measure object handle.
- ▷ **Row** (input_control) `point.y` $\leadsto$ real / integer
Row coordinate of the new reference point.
Default Value : 50.0
Suggested values : `Row` $\in$ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
Typical range of values : $0.0 \leq \text{Row} \leq 511.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
- ▷ **Column** (input_control) `point.x` $\leadsto$ real / integer
Column coordinate of the new reference point.
Default Value : 100.0
Suggested values : `Column` $\in$ {10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0, 500.0}
Typical range of values : $0.0 \leq \text{Column} \leq 511.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0

Result

If the parameter values are correct the operator [translate_measure](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

[translate_measure](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_measure_rectangle2](#), [gen_measure_arc](#)

Possible Successors

[measure_pos](#), [measure_pairs](#), [fuzzy_measure_pos](#), [fuzzy_measure_pairs](#),
[fuzzy_measure_pairing](#), [measure_thresh](#)

Alternatives

[gen_measure_rectangle2](#), [gen_measure_arc](#)

See also

[close_measure](#)

Module

Sub-pixel operators

15.12 OCR

append_ocr_trainf (`Character`, `Image` : : `Class`, `FileName` :)

Add characters to a training file.

The operator [append_ocr_trainf](#) serves to prepare the training with the operator [trainf_ocr_class_box](#). Hereby regions, representing characters, including their gray values (region and pixel) and the corresponding class name will be written into a file. An arbitrary number of regions within one image is supported. For each character (region) in [Character](#) the corresponding class name must be specified in [Class](#). The gray values are passed via the parameter [Image](#). In contrast to the operator [write_ocr_trainf](#) the characters are appended to an existing file using the same training file format as this file. If the file does not exist, a new file is generated. In this case, the file format can be chosen by the parameter '[ocr_trainf_version](#)' of the operator [set_system](#).

Parameter

- ▷ **Character** (input_object) region(-array) $\leadsto$ *Hobject*
Characters to be trained.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte
Gray values of the characters.
- ▷ **Class** (input_control) string(-array) $\leadsto$ *string*
Class (name) of the characters.
- ▷ **FileName** (input_control) filename $\leadsto$ *string*
Name of the training file.
Default Value : "train_ocr"

Example

```

char      name[128];
char      class[128];

read_image(&Image, "character.tiff");
bin_threshold(Image, &Dark);
connection(Dark, &Character);
count_obj(Character, &num);
create_tuple(&Class, num);
open_window(0, 0, -1, -1, 0, "", "", &WindowHandle);
set_color(WindowHandle, "red");
for (i=0; i<num; i++) {
    select_obj(Character, &SingleCharacter, i);
    clear_window(WindowHandle);
    disp_region(SingleCharacter, WindowHandle);
    printf("class of character %d ?\n", i);
    scanf("%s\n", class);
    append_ocr_trainf(Character, Image, name, class);
}

```

Result

If the parameters are correct, the operator [append_ocr_trainf](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Parallelization Information

[append_ocr_trainf](#) is processed under *mutual exclusion* against itself and without parallelization.

Possible Predecessors

[threshold](#), [connection](#), [create_ocr_class_box](#), [read_ocr](#)

Possible Successors

[trainf_ocr_class_box](#), [info_ocr_class_box](#), [write_ocr](#), [do_ocr_multi](#), [do_ocr_single](#)

Alternatives

[write_ocr_trainf](#), [write_ocr_trainf_image](#)

Module

Optical character recognition

close_all_ocrs (: : :)

Destroy all OCR classifiers.

[close_all_ocrs](#) deletes all OCR classifiers and frees the used memory space. All the trained data will be lost.

Attention

Since all classifiers are closed by [close_all_ocrs](#) all handles become invalid.

Result

If it is possible to close the OCR classifiers the operator `close_all_ocrs` returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

`close_all_ocrs` is processed *completely exclusively* without parallelization.

Alternatives

`close_ocr`

Module

Optical character recognition

close_ocr (: : OcrHandle :)

Deallocation of the memory of an OCR classifier.

The operator `close_ocr` deallocates the memory of the classifier having the number `OcrHandle`. Hereby all corresponding data will be deleted. However, if necessary, they can be saved in advance using the operator `write_ocr`. The number `OcrHandle` will be invalid after the call; but later the system can use it again for new classifiers.

Attention

All data of the classifier will be deleted in main memory (not on the hard disk).

Parameter

- ▷ **OcrHandle** (input_control) ocr $\leadsto$ *integer*
ID of the OCR classifier to be deleted.

Result

If the parameter `OcrHandle` is valid, the operator `close_ocr` returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Parallelization Information

`close_ocr` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`write_ocr_trainf`

Possible Successors

`read_ocr`

Module

Optical character recognition

concat_ocr_trainf (: : SingleFiles, ComposedFile :)

Concat training files.

The operator `concat_ocr_trainf` stores all characters which are contained in the files `SingleFiles` into a new file with the name `ComposedFile`. The file format can be defined by the parameter 'ocr_trainf-version' of the operator `set_system`. The file name can be chosen at will.

Parameter

- ▷ **SingleFiles** (input_control) filename(-array) $\leadsto$ *string*
Name of the single training files.
Default Value : "
- ▷ **ComposedFile** (input_control) filename $\leadsto$ *string*
Name of the composed training file.
Default Value : 'all_characters'

Result

If the parameters are correct, the operator `concat_ocr_trainf` returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

<i>Parallelization Information</i>
<code>concat_ocr_trainf</code> is processed under <i>mutual exclusion</i> against itself and without parallelization.
<i>Possible Predecessors</i>
<code>write_ocr_trainf</code> , <code>append_ocr_trainf</code>
<i>Possible Successors</i>
<code>trainf_ocr_class_box</code> , <code>info_ocr_class_box</code> , <code>write_ocr</code> , <code>do_ocr_multi</code> , <code>do_ocr_single</code>
<i>Module</i>
Optical character recognition

create_ocr_class_box (: : WidthPattern, HeightPattern, Interpolation, Features, Character : OcrHandle)

Create a new OCR-classifier.

The operator `create_ocr_class_box` creates a new OCR classifier. For a description of this classifier see operator `learn_class_box`. This classifier must then be trained with the help of the operators `traind_ocr_class_box` or `trainf_ocr_class_box`.

The parameters `WidthPattern` and `HeightPattern` indicate the size of the input-layer of the network. This size is used for the features `'projection_horizontal'`, `'projection_vertical'`, `'pixel'`, `'pixel_invar'`, and `'pixel_binary'` to transform the character to a standard size. The bigger the standard size is, the more characters can be distinguished. Hereby the amount of time necessary for the training (as well as the number of training random samples) and the time necessary for the recognition, however, will increase as well. The parameter `Interpolation` indicates the interpolation mode concerning the adaptation of characters in the image to the network. For more detailed information on this parameter see also `affine_trans_image`. The value `0` results in the same interpolation as `'none'` in `affine_trans_image`, i.e., no interpolation is performed. For `1`, the same behavior as `'constant'` in `affine_trans_image` is obtained, i.e., equally weighted interpolation between adjacent pixels is used. Finally, `2` results in the same interpolation as `'weighted'`, i.e., Gaussian interpolation between adjacent pixels is used. The parameter `Interpolation` must be chosen such that no aliasing occurs when the character is scaled to the standard size. Typically, this means that `Interpolation` should be set to `1`, except in cases where the characters are scaled down by a large amount, in which case `Interpolation` = `2` should be chosen. `Interpolation` = `0` should only be chosen if the characters will not be scaled.

The parameter `Character` determines all the characters which have to be recognized. Normally the transmitted strings consist of one character (e.g. alphabet). But also strings of any length can be learned. The number of distinguishable characters (number of strings in `Character`) is limited to `2048`.

The parameter `Features` helps to chose additional features besides gray values in order to recognize characters. By using `'default'` the features `'ratio'` and `'pixel_invar'` will be set.

The following features are available:

`'ratio'` Ratio of the character.

`'width'` Width of the character (not invariant to scaling).

`'height'` Height of the character (not invariant to scaling).

`'zoom_factor'` Difference in size between the current character and the values of `WidthPattern` and `HeightPattern` (not invariant to scaling).

`'foreground'` Relative number of pixels in the foreground.

`'foreground_grid_9'` Relative number of foreground pixels in a 3×3 grid within the surrounding rectangle of the character.

`'foreground_grid_16'` Relative number of foreground pixels in a 4×4 grid within the surrounding rectangle of the character.

`'anisometry'` Form feature anisometry.

`'compactness'` Form feature compactness.

`'convexity'` Form feature convexity.

`'moments_region_2nd_invar'` Normed 2nd geometric moments of the region. See also `moments_region_2nd_invar`.

'*moments_region_2nd_rel_invar*' Normed 2nd relativ geometric moments of the region. See also [moments_region_2nd_rel_invar](#).

'*moments_region_3rd_invar*' Normed 3rd geometric moments of the region. See also [moments_region_3rd_invar](#).

'*moments_central*' Normed central geometric moments of the region. See also [moments_region_central](#).

'*phi*' Orientation (angle) of the character.

'*num_connect*' Number of connecting components.

'*num_holes*' Number of holes.

'*projection_horizontal*' Horizontal projection of the gray values.

'*projection_horizontal_invar*' Horizontal projection of the gray values with are automatically scaled to maximum range.

'*projection_vertical*' Vertical projection of the gray values.

'*projection_vertical_invar*' Vertical projection of the gray values with are automatically scaled to maximum range.

'*cooc*' Values of the binary cooccurrence matrix.

'*moments_gray_plane*' Normed gray value moments and the angles of the gray value level.

'*num_runs*' Number of chords in the region normed to the area.

'*chord_histo*' Frequency of the chords per row.

'*pixel*' Gray value of the character.

'*pixel_invar*' Gray values of the character with automatic maximal scaling of the gray values.

'*pixel_binary*' Region of the character as a binary image zoomed to a size of [WidthPattern](#) × [HeightPattern](#).

Parameter

- ▷ **WidthPattern** (input_control) integer $\leadsto$ integer
Width of the input layer of the network.
Default Value : 8
Suggested values : $\text{WidthPattern} \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 20\}$
Typical range of values : $1 \leq \text{WidthPattern} \leq 100$
- ▷ **HeightPattern** (input_control) integer $\leadsto$ integer
Height of the input layer of the network.
Default Value : 10
Suggested values : $\text{HeightPattern} \in \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 16, 20\}$
Typical range of values : $1 \leq \text{HeightPattern} \leq 100$
- ▷ **Interpolation** (input_control) integer $\leadsto$ integer
Interpolation mode concerning scaling of characters.
Default Value : 1
List of values : $\text{Interpolation} \in \{0, 1, 2\}$
- ▷ **Features** (input_control) string(-array) $\leadsto$ string
Additional features.
Default Value : 'default'
List of values : $\text{Features} \in \{\text{'default'}, \text{'zoom_factor'}, \text{'ratio'}, \text{'width'}, \text{'height'}, \text{'foreground'}, \text{'foreground\_grid\_9'}, \text{'foreground\_grid\_16'}, \text{'anisometry'}, \text{'compactness'}, \text{'convexity'}, \text{'moments\_region\_2nd\_invar'}, \text{'moments\_region\_2nd\_rel\_invar'}, \text{'moments\_region\_3rd\_invar'}, \text{'moments\_central'}, \text{'phi'}, \text{'num\_connect'}, \text{'num\_holes'}, \text{'projection\_horizontal'}, \text{'projection\_vertical'}, \text{'projection\_horizontal\_invar'}, \text{'projection\_vertical\_invar'}, \text{'chord\_histo'}, \text{'num\_runs'}, \text{'pixel'}, \text{'pixel\_invar'}, \text{'pixel\_binary'}, \text{'cooc'}, \text{'moments\_gray\_plane'}\}$
- ▷ **Character** (input_control) string-array $\leadsto$ string
All characters of a set.
Default Value : '['a', 'b', 'c']'
- ▷ **OcrHandle** (output_control) ocr $\leadsto$ integer
ID of the created OCR classifier.

Result

If the parameters are correct, the operator `create_ocr_class_box` returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Parallelization Information

`create_ocr_class_box` is processed *completely exclusively* without parallelization.

Possible Predecessors

`reset_obj_db`

Possible Successors

`traind_ocr_class_box`, `trainf_ocr_class_box`, `info_ocr_class_box`, `write_ocr`, `ocr_change_char`

See also

`affine_trans_image`, `ocr_change_char`, `moments_region_2nd_invar`, `moments_region_2nd_rel_invar`, `moments_region_3rd_invar`, `moments_region_central`

Module

Optical character recognition

do_ocr_multi (Character, Image : : OcrHandle : Class, Confidence)

Classify characters.

The operator `do_ocr_multi` assigns a class to every `Character` (character). For gray value features the gray values from the surrounding rectangles of the regions are used. The gray values will be taken from the parameter `Image`. For each character the corresponding class will be returned in `Class` and a confidence value will be returned in `Confidence`. The confidence value indicates the similarity between the input pattern and the assigned character.

Parameter

- ▷ **Character** (input_object) region(-array) $\leadsto$ *Hobject*
Characters to be recognized.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte
Gray values for the characters.
- ▷ **OcrHandle** (input_control) ocr $\leadsto$ *integer*
ID of the OCR classifier.
- ▷ **Class** (output_control) string(-array) $\leadsto$ *string*
Class (name) of the characters.
Number of elements : Class = Character
- ▷ **Confidence** (output_control) real(-array) $\leadsto$ *real*
Confidence values of the characters.
Number of elements : Confidence = Character

Result

If the input parameters are set correctly, the operator `do_ocr_multi` returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Parallelization Information

`do_ocr_multi` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`traind_ocr_class_box`, `trainf_ocr_class_box`, `read_ocr`, `connection`, `sort_region`

Alternatives

`do_ocr_single`

See also

`write_ocr`

Module

Optical character recognition

do_ocr_single (*Character*, *Image* : : *OcrHandle* : *Classes*,
Confidences)

Classify one character.

The operator **do_ocr_single** assigns classes to the **Character** (characters). For gray value features gray values of the surrounding rectangles of the regions will be used. The gray values will be taken from the parameter **Image**. For each character the two classes with the highest confidences will be returned in **Classes**. The corresponding confidences will be returned in **Confidences**. The confidence value indicates the similarity between the input pattern and the assigned character.

Parameter

- ▷ **Character** (input_object) region $\leadsto$ *Hobject*
Character to be recognized.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte
Gray values of the characters.
- ▷ **OcrHandle** (input_control) ocr $\leadsto$ *integer*
ID of the OCR classifier.
- ▷ **Classes** (output_control) string-array $\leadsto$ *string*
Classes (names) of the characters.
Number of elements : 2
- ▷ **Confidences** (output_control) real-array $\leadsto$ *real*
Confidence values of the characters.
Number of elements : 2

Result

If the input parameters are correct, the operator **do_ocr_single** returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Parallelization Information

do_ocr_single is *reentrant* and processed *without* parallelization.

Possible Predecessors

traind_ocr_class_box, **trainf_ocr_class_box**, **read_ocr**, **connection**, **sort_region**

Alternatives

do_ocr_multi

See also

write_ocr

Module

Optical character recognition

info_ocr_class_box (: : *OcrHandle* : *WidthPattern*, *HeightPattern*,
Interpolation, *WidthMaxChar*, *HeightMaxChar*, *Features*, *Characters*)

Get information about an OCR classifier.

The operator **info_ocr_class_box** returns some information about an OCR classifier. The parameters are equivalent to those of **create_ocr_class_box**. The parameters **WidthMaxChar** and **HeightMaxChar** indicate the extension of the largest trained character. These values can be used to control the segmentation.

Parameter

- ▷ **OcrHandle** (input_control) ocr $\leadsto$ *integer*
ID of the OCR classifier.
- ▷ **WidthPattern** (output_control) integer $\leadsto$ *integer*
Width of the scaled characters.
- ▷ **HeightPattern** (output_control) integer $\leadsto$ *integer*
Height of the scaled characters.
- ▷ **Interpolation** (output_control) integer $\leadsto$ *integer*
Interpolation mode for scaling the characters.

- ▷ **WidthMaxChar** (output_control) integer $\leadsto$ integer
Width of the largest trained character.
- ▷ **HeightMaxChar** (output_control) integer $\leadsto$ integer
Height of the largest trained character.
- ▷ **Features** (output_control) string-array $\leadsto$ string
Used features.
- ▷ **Characters** (output_control) string-array $\leadsto$ string
All characters of the set.

Result

The operator `info_ocr_class_box` always returns 2 (H_MSG_TRUE).

Parallelization Information

`info_ocr_class_box` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`read_ocr`, `create_ocr_class_box`

Possible Successors

`write_ocr`

Module

Optical character recognition

ocr_change_char (: : OcrHandle, Character :)

Define a new conversion table for the characters.

The operator `ocr_change_char` establishes a new look-up table for the characters. Hereby the number of strings of `Character` must be the same as of the classifier `OcrHandle`. In order to enlarge the font, the operator `ocr_change_char` may be used as follows: More characters than actually needed will be indicated when creating a network using (`create_ocr_class_box`). The last n characters will not be used so far. If more characters are needed at a later stage, these unused characters will be allocated and then trained with the help of the operator `ocr_change_char`.

Parameter

- ▷ **OcrHandle** (input_control) ocr $\leadsto$ integer
ID of the OCR-network to be changed.
- ▷ **Character** (input_control) string-array $\leadsto$ string
New assign of characters.
Default Value : '["a","b","c"]'

Result

If the number of characters in `Character` is identical with the number of the characters of the network, the operator `ocr_change_char` returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Parallelization Information

`ocr_change_char` is processed *completely exclusively* without parallelization.

Possible Predecessors

`read_ocr`

Possible Successors

`do_ocr_multi`, `do_ocr_single`, `write_ocr`

Module

Optical character recognition

ocr_get_features (Character : : OcrHandle : FeatureVector)

Access the features which correspond to a character.

The operator `ocr_get_features` calculates the features for the given `Character`. The type and number of features is determined by the classifier `OcrHandle`. `FeatureVector` contains the same values which are used inside operators like `traind_ocr_class_box` or `trainf_ocr_class_box`.

Parameter

- ▷ **Character** (input_object) image $\leadsto$ *Hobject*: byte
Characters to be trained.
- ▷ **OcrHandle** (input_control) ocr $\leadsto$ *integer*
ID of the desired OCR-classifier.
- ▷ **FeatureVector** (output_control) real-array $\leadsto$ *real*
Feature vector.

Result

If the parameters are correct, the operator `ocr_get_features` returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Parallelization Information

`ocr_get_features` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`create_ocr_class_box`, `read_ocr`, `reduce_domain`, `threshold`, `connection`

Possible Successors

`learn_class_box`

See also

`trainf_ocr_class_box`, `traind_ocr_class_box`

Module

Optical character recognition

read_ocr (: : FileName : OcrHandle)

Read an OCR classifier from a file.

The operator `read_ocr` reads an OCR classifier from a file `FileName`. This file will hereby be searched in the directory (\$HALCONROOT/ocr/) as well as in the currently used directory. If too many classifiers have been loaded, an error message will be displayed.

Parameter

- ▷ **FileName** (input_control) filename $\leadsto$ *string*
Name of the OCR classifier file.
Default Value : 'testnet'
- ▷ **OcrHandle** (output_control) ocr $\leadsto$ *integer*
ID of the read OCR classifier.

Result

If the indicated file is available and the format is correct, the operator `read_ocr` returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Parallelization Information

`read_ocr` is processed *completely exclusively* without parallelization.

Possible Predecessors

`reset_obj_db`

Possible Successors

`do_ocr_multi`, `do_ocr_single`, `traind_ocr_class_box`, `trainf_ocr_class_box`

See also

`write_ocr`, `do_ocr_multi`, `traind_ocr_class_box`, `trainf_ocr_class_box`

Module

Optical character recognition

read_ocr_trainf (: Characters : TrainFileNames : CharacterNames)

Read training characters from files and convert to images.

read_ocr_trainf reads all characters from the specified file names and converts them into images. The domain is defined according to the foreground of the characters (as specified in **write_ocr_trainf**). The names of the characters are returned in **CharacterNames**. If more than one file name is given the files are processed in the order the file names.

Parameter

- ▷ **Characters** (output_object) image-array $\leadsto$ *Hobject* : byte
Images read from file.
- ▷ **TrainFileNames** (input_control) filename.named(-array) $\leadsto$ *string*
Names of the training files.
Default Value : "
- ▷ **CharacterNames** (output_control) string-array $\leadsto$ *string*
Names of the read characters.

Result

If the parameter values are correct the operator **read_ocr_trainf** returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

read_ocr_trainf is *reentrant* and processed *without parallelization*.

Possible Predecessors

write_ocr_trainf

Possible Successors

disp_image, **select_obj**, **zoom_image_size**

Alternatives

read_ocr_trainf_select

See also

trainf_ocr_class_box

Module

Optical character recognition

read_ocr_trainf_names (: : TrainFileNames : CharacterNames, CharacterCount)

Query which characters are stored in a training file.

read_ocr_trainf_names extracts the names and frequency of all characters in the specified training files.

Parameter

- ▷ **TrainFileNames** (input_control) filename.named(-array) $\leadsto$ *string*
Names of the training files.
Default Value : "
- ▷ **CharacterNames** (output_control) string(-array) $\leadsto$ *string*
Names of the read characters.
- ▷ **CharacterCount** (output_control) integer(-array) $\leadsto$ *integer*
Number of the characters.

Result

If the parameter values are correct the operator **read_ocr_trainf_names** returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

read_ocr_trainf_names is *reentrant* and processed *without parallelization*.

Possible Predecessors

write_ocr_trainf

See also

[trainf_ocr_class_box](#)

Module

Optical character recognition

read_ocr_trainf_select (: Characters : TrainFileNames,
SearchNames : FoundNames)

Read training specific characters from files and convert to images.

[read_ocr_trainf_select](#) reads the characters given in [SearchNames](#) from the specified files and converts them into images. It works simimlär to [read_ocr_trainf](#) but here the characters which are extracted can be specified.

Parameter

- ▷ **Characters** (output_object) image-array $\leadsto$ Hobject : byte
Images read from file.
- ▷ **TrainFileNames** (input_control) filename.named(-array) $\leadsto$ string
Names of the training files.
Default Value : "
- ▷ **SearchNames** (input_control) string(-array) $\leadsto$ string
Names of the characters to be extracted.
Default Value : '0'
- ▷ **FoundNames** (output_control) string(-array) $\leadsto$ string
Names of the read characters.

Result

If the parameter values are correct the operator [read_ocr_trainf_select](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception handling is raised.

Parallelization Information

[read_ocr_trainf_select](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[write_ocr_trainf](#)

Possible Successors

[disp_image](#), [select_obj](#), [zoom_image_size](#)

Alternatives

[read_ocr_trainf](#)

See also

[trainf_ocr_class_box](#)

Module

Optical character recognition

testd_ocr_class_box (Character, Image : : OcrHandle,
Class : Confidence)

Train an OCR classifier by the input of regions.

The operator [testd_ocr_class_box](#) tests the confidence with which a character belongs to a given class. Any number of regions of an image can be passed. For each character (region) in [Character](#) the corresponding name (class) [Class](#) must be specified. The gray values are passed in [Image](#). When the operator has finished the parameter [Confidence](#) provides information about how sure a character belongs to the (arbitrary chosen) class.

Parameter	
▷ Character (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Characters to be trained.
▷ Image (input_object)	image $\leadsto$ <i>Hobject</i> : byte Gray values for the characters.
▷ OcrHandle (input_control)	ocr $\leadsto$ <i>integer</i> ID of the desired OCR-classifier.
▷ Class (input_control)	string(-array) $\leadsto$ <i>string</i> Class (name) of the characters. Default Value : "a"
▷ Confidence (output_control)	real(-array) $\leadsto$ <i>real</i> Confidence for the character to belong to the class.

Result
If the parameters are correct, the operator `testd_ocr_class_box` returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Parallelization Information
`testd_ocr_class_box` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors
`read_ocr`, `trainf_ocr_class_box`, `traind_ocr_class_box`

Module
Optical character recognition

traind_ocr_class_box (Character, Image : : OcrHandle,
Class : AvgConfidence)

Train an OCR classifier by the input of regions.

The operator `traind_ocr_class_box` trains the classifier directly via the input of regions in an image. Any number of regions of an image can be passed. For each character (region) in `Character` the corresponding name (class) `Class` must be specified. The gray values are passed in `Image`. When the procedure has finished the parameter `AvgConfidence` provides information about the success of the training: It contains the average confidence of the trained characters measured by a re-classification. The confidence of mismatched characters is set to 0 (thus, the average confidence will be decreased significantly).

Parameter	
▷ Character (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Characters to be trained.
▷ Image (input_object)	image $\leadsto$ <i>Hobject</i> : byte Gray values for the characters.
▷ OcrHandle (input_control)	ocr $\leadsto$ <i>integer</i> ID of the desired OCR-classifier.
▷ Class (input_control)	string(-array) $\leadsto$ <i>string</i> Class (name) of the characters. Default Value : "a"
▷ AvgConfidence (output_control)	real $\leadsto$ <i>real</i> Average confidence during a re-classification of the trained characters.

Result
If the parameters are correct, the operator `traind_ocr_class_box` returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Parallelization Information
`traind_ocr_class_box` is processed *completely exclusively* without parallelization.

Possible Predecessors
`create_ocr_class_box`, `read_ocr`

Possible Successors
`traind_ocr_class_box`, `write_ocr`, `do_ocr_multi`, `do_ocr_single`

Alternatives

[trainf_ocr_class_box](#)

Module

Optical character recognition

trainf_ocr_class_box (: : OcrHandle, FileName : AvgConfidence)

Train an OCR classifier with the help of a training file.

The operator [trainf_ocr_class_box](#) trains the classifier [OcrHandle](#) via the indicated training files. Any number of files can be indicated. The parameter [AvgConfidence](#) provides information about the success of the training: It contains the average confidence of the trained characters measured by a re-classification. The confidence of mismatched characters is set to 0 (thus, the average confidence will be decreased significantly).

Attention

The names of the characters in the file must fit the network.

Parameter

- ▷ **OcrHandle** (input_control) ocr $\leadsto$ integer
ID of the desired OCR-network.
- ▷ **FileName** (input_control) filename(-array) $\leadsto$ string
Name(s) of the training file(s).
Default Value : "train_ocr"
- ▷ **AvgConfidence** (output_control) real $\leadsto$ real
Average confidence during a re-classification of the trained characters.

Result

If the file name is correct and the data fit the network, the operator [trainf_ocr_class_box](#) returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Parallelization Information

[trainf_ocr_class_box](#) is processed *completely exclusively* without parallelization.

Possible Predecessors

[create_ocr_class_box](#), [read_ocr](#)

Possible Successors

[traind_ocr_class_box](#), [write_ocr](#), [do_ocr_multi](#), [do_ocr_single](#)

Alternatives

[traind_ocr_class_box](#)

Module

Optical character recognition

write_ocr (: : OcrHandle, FileName :)

Writing an OCR classifier into a file.

The operator [write_ocr](#) writes the OCR classifier [OcrHandle](#) into the file [FileName](#). Since the data of the classifier will be lost when the program is finished, they have to be stored after the training if the user wants to use them again at a later execution of the program. The data can then be read with the help of the operator [read_ocr](#). The extension will be added automatically to the parameter [FileName](#).

Attention

The output file [FileName](#) must be given without extension.

Parameter

- ▷ **OcrHandle** (input_control) ocr $\leadsto$ integer
ID of the OCR classifier.

- ▷ **FileName** (input_control) filename $\leadsto$ string
 Name of the file for the OCR classifier (without extension).
Default Value : 'my_ocr'

Result

If the parameter **OcrHandle** is valid and the indicated file can be written, the operator **write_ocr** returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Parallelization Information

write_ocr is *reentrant* and processed *without* parallelization.

Possible Predecessors

traind_ocr_class_box, **trainf_ocr_class_box**

Possible Successors

do_ocr_multi, **do_ocr_single**

See also

read_ocr, **do_ocr_multi**, **traind_ocr_class_box**, **trainf_ocr_class_box**

Module

Optical character recognition

write_ocr_trainf (Character, Image : : Class, FileName :)

Storing of trained characters into a file.

The operator **write_ocr_trainf** serves to prepare the training with the operator **trainf_ocr_class_box**. Hereby regions, representing characters, including their gray values (region and pixel) and the corresponding class name will be written into a file. An arbitrary number of regions within one image is supported. For each character (region) in **Character** the corresponding class name must be specified in **Class**. The gray values are passed via the parameter **Image**. The file name can be chosen at will. The version of the file format used for writing data can be defined by the parameter '*ocr_trainf_version*' of the operator **set_system**.

Parameter

- ▷ **Character** (input_object) region(-array) $\leadsto$ Hobject
 Characters to be trained.
- ▷ **Image** (input_object) image $\leadsto$ Hobject : byte
 Gray values of the characters.
- ▷ **Class** (input_control) string(-array) $\leadsto$ string
 Class (name) of the characters.
- ▷ **FileName** (input_control) filename $\leadsto$ string
 Name of the training file.
Default Value : "train_ocr"

Result

If the parameters are correct, the operator **write_ocr_trainf** returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Parallelization Information

write_ocr_trainf is *reentrant* and processed *without* parallelization.

Possible Predecessors

threshold, **connection**, **create_ocr_class_box**, **read_ocr**

Possible Successors

trainf_ocr_class_box, **info_ocr_class_box**, **write_ocr**, **do_ocr_multi**, **do_ocr_single**

Module

Optical character recognition

write_ocr_trainf_image (Character : : Class, FileName :)

Write characters into a training file.

The operator `write_ocr_trainf_image` is used to prepare the training with the operator `trainf_ocr_class_box`. Hereby regions, representing characters, including their gray values (region and pixel) and the corresponding class name will be written into a file. An arbitrary number of regions within one image is supported. For each character (region) in `Character` the corresponding class name must be specified in `Class`. The file name can be chosen at will. In contrast to `write_ocr_trainf` one image per character is passed. The domain of this image defines the pixels which belong to the character. The file format can be defined by the parameter `'ocr_trainf_version'` of the operator `set_system`.

Parameter

- ▷ **Character** (input_object) image(-array) $\leadsto$ *Hobject* : byte
Characters to be trained.
- ▷ **Class** (input_control) string(-array) $\leadsto$ *string*
Class (name) of the characters.
- ▷ **FileName** (input_control) filename $\leadsto$ *string*
Name of the training file.
Default Value : "train_ocr"

Result

If the parameters are correct, the operator `write_ocr_trainf_image` returns the value 2 (H_MSG_TRUE). Otherwise an exception will be raised.

Parallelization Information

`write_ocr_trainf_image` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`threshold`, `connection`, `create_ocr_class_box`, `read_ocr`

Possible Successors

`trainf_ocr_class_box`, `info_ocr_class_box`, `write_ocr`, `do_ocr_multi`, `do_ocr_single`

Alternatives

`write_ocr_trainf`, `append_ocr_trainf`

Module

Optical character recognition

15.13 OCV

close_all_ocvs (: : :)

Clear all OCV tools.

`close_all_ocvs` closes *all* OCV tools which have been opened using `create_ocv_proj` or `read_ocv`. All handles are invalid after this call.

Attention

This operator is only for internal use (like e.g. a reset program in HDevelop).

Result

`close_all_ocvs` returns always 2 (H_MSG_TRUE).

Parallelization Information

`close_all_ocvs` is processed *completely exclusively* without parallelization.

Possible Predecessors

`read_ocv`, `create_ocv_proj`

Alternatives

`close_ocv`

Module

Optical character verification

close_ocv (: : OCVHandle :)

Clear an OCV tool.

close_ocv closes an open OCV tool and frees the memory. The OCV tool has been created using **create_ocv_proj** or **read_ocv**. The handle is after this call no longer valid.

Parameter

- ▷ **OCVHandle** (input_control) ocv $\leadsto$ integer
 Handle of the OCV tool which has to be freed.

Example

```
read_ocv("ocv_file",&ocv_handle);
for (i=0; i<1000; i++)
{
    grab_image_async(&Image,fg_handle,-1);
    reduce_domain(Image,ROI,&Pattern);
    do_ocv_simple(Pattern,ocv_handle,"A",
                  "true","true","false","true",10,
                  &Quality);
}
close_ocv(ocv_handle);
```

Result

close_ocv returns 2 (H_MSG_TRUE), if the handle is valid. Otherwise, an exception handling is raised.

Parallelization Information

close_ocv is processed *completely exclusively* without parallelization.

Possible Predecessors

read_ocv, **create_ocv_proj**

See also

close_ocr

Module

Optical character verification

create_ocv_proj (: : PatternNames : OCVHandle)

Create a new OCV tool based on gray value projections.

create_ocv_proj creates a new OCV tool. This tool will be used to train good-patterns for the optical character verification. The training is done using the operator **traind_ocv_proj**. Thus **traind_ocv_proj** is normally called after **create_ocv_proj**.

The pattern comparison is based on the gray projections: For every training pattern the horizontal and vertical gray projections are calculated by summing up the gray values along the rows and columns inside the region of the pattern. This operation is applied to the training patterns and the test patterns. For the training patterns the result is stored inside the OCV tool to save runtime while comparing patterns. The OCV is done by comparing the corresponding projections. The Quality is the similarity of the projections.

Input for **create_ocv_proj** are the names of the patterns (**PatternNames**) which have to be trained. The number and the names can be chosen arbitrary. In most case only one pattern will be trained, thus only one name has to be specified. The names will be used when doing the OCV (**do_ocv_simple**). It is possible to specify more names than actually used. These might be trained later.

To close the OCV tool, i.e. to free the memory, the operator **close_ocv** is called.

Parameter

- ▷ **PatternNames** (input_control) string(-array) $\leadsto$ *string*
List of names for patterns to be trained.
Default Value : "a"
- ▷ **OCVHandle** (output_control) ocv $\leadsto$ *integer*
Handle of the created OCV tool.

Example

```
create_ocv_proj("A",&ocv_handle);
draw_region(&ROI>window_handle);
reduce_domain(Image,ROI,&Sample);
traind_ocv_proj(Sample,ocv_handle,"A","single");
```

Result

[create_ocv_proj](#) returns 2 (H_MSG_TRUE), if the parameters are correct. Otherwise, an exception handling is raised.

Parallelization Information

[create_ocv_proj](#) is processed *completely exclusively* without parallelization.

Possible Successors

[traind_ocv_proj](#), [write_ocv](#), [close_ocv](#)

Alternatives

[read_ocv](#)

See also

[create_ocr_class_box](#)

Module

Optical character verification

do_ocv_simple (Pattern : : OCVHandle, PatternName, AdaptPos,
AdaptSize, AdaptAngle, AdaptGray, Threshold : Quality)

Verification of a pattern using an OCV tool.

[do_ocv_simple](#) evaluates the pattern in ([Pattern](#)). Before the evaluation the good-pattern has be trained by using the operator [traind_ocv_proj](#). Both patterns should have roughly the same (relative) extent and shape. To specify which of the trained patterns is used as reference its name is specified in [PatternName](#). The next four parameters influence the automatic adaption: [AdaptPos](#) and [AdaptSize](#) refer to the geometry of the pattern. [AdaptPos](#) specifies whether a shift of the position will be adapted automatically. [AdaptSize](#) is used to adapt to changes in the size of the pattern. [AdaptAngle](#) is not yet implemented. The parameter [AdaptGray](#) controls the adaption to changes of the grayvalues. This comprises additive and multiplicative changes of the intensity.

The parameter [Threshold](#) specifies the minimum difference of the gray values to be treated as an error. In this case the percentage of wrong pixels is returned. If the value is below 0 the sum of all errors normalized with respect to the size is returned.

The result of the operator is the [Quality](#) of the pattern with a value between 0 and 1. The value 1 corresponds to a pattern with no faults. The value 0 corresponds to a very big fault.

Parameter

- ▷ **Pattern** (input_object) image(-array) $\leadsto$ *Hobject* : byte
Characters to be verified.
- ▷ **OCVHandle** (input_control) ocv $\leadsto$ *integer*
Handle of the OCV tool.
- ▷ **PatternName** (input_control) string(-array) $\leadsto$ *string*
Name of the character.
Default Value : "a"

- ▷ **AdaptPos** (input_control) string $\leadsto$ string
Adaption to vertical and horizontal translation.
Default Value : "true"
List of values : AdaptPos $\in$ {"true", "false"}
- ▷ **AdaptSize** (input_control) string $\leadsto$ string
Adaption to vertical and horizontal scaling of the size.
Default Value : "true"
List of values : AdaptSize $\in$ {"true", "false"}
- ▷ **AdaptAngle** (input_control) string $\leadsto$ string
Adaption to changes of the orientation (not yet implemented).
Default Value : "false"
List of values : AdaptAngle $\in$ {"false"}
- ▷ **AdaptGray** (input_control) string $\leadsto$ string
Adaption to additive and scaling gray value changes.
Default Value : "true"
List of values : AdaptGray $\in$ {"true", "false"}
- ▷ **Threshold** (input_control) number $\leadsto$ real
Minimum difference between objects.
Default Value : 10
Suggested values : Threshold $\in$ {-1, 0, 1, 5, 10, 15, 20, 30, 40, 50, 60, 80, 100, 150}
- ▷ **Quality** (output_control) real(-array) $\leadsto$ real
Evaluation of the character.
Typical range of values : $0.0 \leq \text{Quality} \leq 1.0$

Result

`do_ocv_simple` returns 2 (H_MSG_TRUE), if the handle and the characters are correct. Otherwise, an exception handling is raised.

Parallelization Information

`do_ocv_simple` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`traind_ocr_class_box`, `trainf_ocr_class_box`, `read_ocv`, `threshold`, `connection`, `select_shape`

Possible Successors

`close_ocv`

See also

`create_ocv_proj`

Module

Optical character verification

read_ocv (: : FileName : OCVHandle)

Reading an OCV tool from file.

`read_ocv` reads an OCV tool from file. The tool will contain the same information that it contained when saving it with `write_ocv`. After reading the tool the training can be completed for those patterns which have not been trained so far. Otherwise a pattern comparison can be applied directly by calling `do_ocv_simple`.

As extension '.ocv' is used. If this extension is not given with the file name it will be added automatically.

Parameter

- ▷ **FileName** (input_control) filename $\leadsto$ string
Name of the file which has to be read.
Default Value : 'test.ocv'
- ▷ **OCVHandle** (output_control) ocv $\leadsto$ integer
Handle of read OCV tool.

Example

```

read_ocv("ocv_file",&ocv_handle);
for (i=0; i<1000; i++)
{
    grab_image_async(&Image,fg_handle,-1);
    reduce_domain(Image,ROI,&Pattern);
    do_ocv_simple(Pattern,ocv_handle,"A",
                  "true","true","false","true",10,
                  &Quality);
}
close_ocv(ocv_handle);

```

Result

[read_ocv](#) returns 2 (H_MSG_TRUE), if the file is correct. If the file could not be opened [read_ocv](#) returns 5 (H_MSG_FAIL). Otherwise, an exception handling is raised.

Parallelization Information

[read_ocv](#) is processed *completely exclusively* without parallelization.

Possible Predecessors

[write_ocv](#)

Possible Successors

[do_ocv_simple](#), [close_ocv](#)

See also

[read_ocr](#)

Module

Optical character verification

traind_ocv_proj (Pattern : : OCVHandle, Name, Mode :)

Training of an OCV tool.

[traind_ocv_proj](#) trains patterns for an OCV tool that has been created using the operators [create_ocv_proj](#) or [read_ocv](#). For this training one or multiple patterns are offered the system. Such a pattern consists of an image with a reduced domain (ROI) for the area of the pattern. Note that the pattern should not only contain foreground pixels (e.g. dark pixels of a character) but also background pixels. This can be implemented e.g. by the smallest surrounding rectangle of the pattern. Without this context an evaluation of the pattern is not possible.

If more than one pattern has to be trained this can be achieved by multiple calls (one for each pattern) or by calling [traind_ocv_proj](#) once with all patterns and a tuple of the corresponding names. The result will be in both cases the same. However using multiple calls will normally result in a longer execution time than using one call with all patterns.

Parameter

- ▷ **Pattern** (input_object) image(-array) $\leadsto$ *Hobject* : byte
Pattern to be trained.
- ▷ **OCVHandle** (input_control) ocv $\leadsto$ *integer*
Handle of the OCV tool to be trained.
- ▷ **Name** (input_control) string(-array) $\leadsto$ *string*
Name(s) of the object(s) to analyse.
Default Value : "a"
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Mode for training (only one mode implemented).
Default Value : 'single'
List of values : Mode $\in$ {'single'}

Example

```
create_ocv_proj("A",&ocv_handle);
draw_region(&ROI,window_handle);
reduce_domain(Image,ROI,&Sample);
traind_ocv_proj(Sample,ocv_handle,"A","single");
```

Result

[traind_ocv_proj](#) returns 2 (H_MSG_TRUE), if the handle and the training pattern(s) are correct. Otherwise, an exception handling is raised.

Parallelization Information

[traind_ocv_proj](#) is processed *completely exclusively* without parallelization.

Possible Predecessors

[write_ocr_trainf](#), [create_ocv_proj](#), [read_ocv](#), [threshold](#), [connection](#), [select_shape](#)

Possible Successors

[close_ocv](#)

See also

[traind_ocr_class_box](#)

Module

Optical character verification

write_ocv (: : OCVHandle, FileName :)

Saving an OCV tool to file.

[write_ocv](#) writes an OCV tool to file. This can be used to save the result of a training ([traind_ocv_proj](#)). The whole information contained in the OCV tool is stored in the file. The file can be reloaded afterwards using the operator [read_ocv](#).

As file extension '.ocv' is used. If this extension is not given with the file name, it will be added automatically.

Parameter

- ▷ **OCVHandle** (input_control) ocv $\leadsto$ *integer*
Handle of the OCV tool to be written.
- ▷ **FileName** (input_control) filename $\leadsto$ *string*
Name of the file where the tool has to be saved.
Default Value : 'test_ocv'

Result

[write_ocv](#) returns 2 (H_MSG_TRUE), if the data is correct and the file can be written. Otherwise, an exception handling is raised.

Parallelization Information

[write_ocv](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[traind_ocv_proj](#)

Possible Successors

[close_ocv](#)

See also

[write_ocr](#)

Module

Optical character verification

15.14 Shape-from

depth_from_focus (MultiFocusImage : Depth, Confidence : Filter, Selection :)

Extract depth using mutiple focus levels.

The operator [depth_from_focus](#) extracts the depth using a focus sequence. The images of the focus sequence have to passed as a multi channel image ([MultiFocusImage](#)). The depth for each pixel will be returned in [Depth](#) as the channel number. The parameter [Confidence](#) returns a confidence value for each depth estimation: The larger this value, the higher the confidence of the depth estimation is.

[depth_from_focus](#) selects the pixels with the best focus of all focus levels. The method used to extract these pixels is specified by the parameters [Filter](#) and [Selection](#):

'highpass' The value of the focus is estimated by a highpass filter.

'bandpass' The value of the focus is estimated by a bandpass filter.

'next_maximum' To decide which focus level has be selected, the pixel in the neighborhood with the best confidence is used to determine this value.

'local' The decision for a focus level is based only on the locally calculated focus values.

Parameter

- ▷ **MultiFocusImage** (input_object) multichannel-image(-array) $\leadsto$ *Hobject* : byte
Multichannel gray image consisting of multiple focus levels.
- ▷ **Depth** (output_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte
Depth image.
- ▷ **Confidence** (output_object) singlechannel-image(-array) $\leadsto$ *Hobject* : byte
Confidence of depth estimation.
- ▷ **Filter** (input_control) string(-array) $\leadsto$ *string*
Filter used to find sharp pixels.
Default Value : 'highpass'
List of values : Filter $\in$ {'highpass', 'bandpass'}
- ▷ **Selection** (input_control) string(-array) $\leadsto$ *string*
Method used to find sharp pixels.
Default Value : 'next_maximum'
List of values : Selection $\in$ {'next_maximum', 'local'}

Example

```
compose3(Focus0,Focus1,Focus2,&MultiFocus);
depth_from_focus(MultiFocus,&Depth,&Confidence,'highpass','next_maximum');
mean_image(Depth,&Smooth,15,15);
select_grayvalues_from_channels(MultiChannel,Smooth,SharpImage);
threshold(Confidence,HighConfidence,10,255);
reduce_domain(SharpImage,HighConfidence,ConfidentSharp);
```

Parallelization Information

[depth_from_focus](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[compose2](#), [compose3](#), [compose4](#), [add_channels](#), [read_image](#), [read_sequence](#)

Possible Successors

[select_grayvalues_from_channels](#), [mean_image](#), [gauss_image](#), [threshold](#)

See also

[count_channels](#)

Module

Tools

estimate_alam (Image : : : Albedo, Ambient)

Estimate the albedo of a surface and the amount of ambient light.

estimate_alam estimates the **Albedo** of a surface, i.e. the percentage of light reflected by the surface, and the amount of ambient light **Ambient** by using the maximum and minimum gray values of the image.

Attention

It is assumed that the image contains at least one point for which the reflection function assumes its minimum, e.g., points in shadows. Furthermore, it is assumed that the image contains at least one point for which the reflection function assumes its maximum. If this is not the case, wrong values will be estimated.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte
Image for which albedo and ambient are to be estimated.
- ▷ **Albedo** (output_control) real(-array) $\leadsto$ *real*
Amount of light reflected by the surface.
- ▷ **Ambient** (output_control) real(-array) $\leadsto$ *real*
Amount of ambient light.

Result

estimate_alam always returns the value 2 (H_MSG_TRUE).

Parallelization Information

estimate_alam is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

sfs_mod_lr, **sfs_orig_lr**, **sfs_pentland**, **phot_stereo**, **shade_height_field**

Module

Tools

estimate_sl_al_lr (Image : : : Slant, Albedo)

Estimate the slant of a light source and the albedo of a surface.

estimate_sl_al_lr estimates the **Slant** of a light source, i.e., the angle between the light source and the positive z-axis, and the albedo of the surface in the input image **Image**, i.e. the percentage of light reflected by the surface, using the algorithm of Lee and Rosenfeld.

Attention

The **Albedo** is assumed constant for the entire surface depicted in the image.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte
Image for which slant and albedo are to be estimated.
- ▷ **Slant** (output_control) angle.deg(-array) $\leadsto$ *real*
Angle between the light sources and the positive z-axis (in degrees).
- ▷ **Albedo** (output_control) real(-array) $\leadsto$ *real*
Amount of light reflected by the surface.

Result

estimate_sl_al_lr always returns the value 2 (H_MSG_TRUE).

Parallelization Information

estimate_sl_al_lr is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

sfs_mod_lr, **sfs_orig_lr**, **sfs_pentland**, **phot_stereo**, **shade_height_field**

Module

Tools

estimate_sl_al_zc (Image : : : Slant, Albedo)

Estimate the slant of a light source and the albedo of a surface.

estimate_sl_al_zc estimates the **Slant** of a light source, i.e. the angle between the light source and the positive z-axis, and the albedo of the surface in the input image **Image**, i.e. the percentage of light reflected by the surface, using the algorithm of Zheng and Chellappa.

Attention

The **Albedo** is assumed constant for the entire surface depicted in the image.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte
Image for which slant and albedo are to be estimated.
- ▷ **Slant** (output_control) angle.deg(-array) $\leadsto$ *real*
Angle of the light sources and the positive z-axis (in degrees).
- ▷ **Albedo** (output_control) real(-array) $\leadsto$ *real*
Amount of light reflected by the surface.

Result

estimate_sl_al_zc always returns the value 2 (H_MSG_TRUE).

Parallelization Information

estimate_sl_al_zc is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

sfs_mod_lr, **sfs_orig_lr**, **sfs_pentland**, **phot_stereo**, **shade_height_field**

Module

Tools

estimate_tilt_lr (Image : : : Tilt)

Estimate the tilt of a light source.

estimate_tilt_lr estimates the tilt of a light source, i.e. the angle between the light source and the x-axis after projection into the xy-plane, from the image **Image** using the algorithm of Lee and Rosenfeld.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte
Image for which the tilt is to be estimated.
- ▷ **Tilt** (output_control) angle.deg(-array) $\leadsto$ *real*
Angle between the light source and the x-axis after projection into the xy-plane (in degrees).

Result

estimate_tilt_lr always returns the value 2 (H_MSG_TRUE).

Parallelization Information

estimate_tilt_lr is *reentrant* and automatically *parallelized* (on tuple level).

Possible Successors

sfs_mod_lr, **sfs_orig_lr**, **sfs_pentland**, **phot_stereo**, **shade_height_field**

Module

Tools

estimate_tilt_zc (Image : : : Tilt)

Estimate the tilt of a light source.

estimate_tilt_zc estimates the tilt of a light source, i.e. the angle between the light source and the x-axis after projection into the xy-plane, from the image **Image** using the algorithm of Zheng and Chellappa.

Parameter
▷ Image (input_object) image(-array) $\leadsto$ <i>Hobject</i> : byte Image for which the tilt is to be estimated. ▷ Tilt (output_control) angle.deg(-array) $\leadsto$ <i>real</i> Angle between the light source and the x-axis after projection into the xy-plane (in degrees).
Result
<code>estimate_tilt_zc</code> always returns the value 2 (H_MSG_TRUE).
Parallelization Information
<code>estimate_tilt_zc</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).
Possible Successors
<code>sfs_mod_lr</code> , <code>sfs_orig_lr</code> , <code>sfs_pentland</code> , <code>phot_stereo</code> , <code>shade_height_field</code>
Module
Tools

phot_stereo (Images : Height : Slants, Tilts :)

Reconstruct a surface from at least three gray value images.

`phot_stereo` reconstructs a surface (i.e., the relative height of each image point) using the algorithm of Woodham from at least three gray value images given by the multi-channel image `Images`. The light sources corresponding to the individual images are given by the parameters `Slants` and `Tilts` and are assumed to lie infinitely far away.

Attention
`phot_stereo` assumes that the heights are to be extracted on a lattice with step width 1. If this is not the case, the calculated heights must be multiplied by the step width after the call to `phot_stereo`. A Cartesian coordinate system with the origin in the lower left corner of the image is used internally. Since the operator is based on the Fast Fourier Transform, only square images with an edge length being a power of 2 are accepted. All given images must be byte-images. At least three images must be given in a multi-channel image. `Slants` and `Tilts` must contain exactly as many light sources as the number of channels in `Images`. At least three of the light source directions must be linearly independent.

Parameter
▷ Images (input_object) (multichannel-)image $\leadsto$ <i>Hobject</i> : byte Shaded input image with at least three channels. ▷ Height (output_object) image $\leadsto$ <i>Hobject</i> : real Reconstructed height field. ▷ Slants (input_control) angle.deg-array $\leadsto$ <i>real</i> / integer Angle between the light sources and the positive z-axis (in degrees). Default Value : 45.0 Suggested values : <code>Slants</code> $\in$ {1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0} Typical range of values : $0.0 \leq \text{Slants} \leq 180.0$ (lin) Minimum Increment : 0.01 Recommended Increment : 10.0 ▷ Tilts (input_control) angle.deg-array $\leadsto$ <i>real</i> / integer Angle between the light source and the x-axis after projection into the xy-plane (in degrees). Default Value : 45.0 Suggested values : <code>Tilts</code> $\in$ {1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0} Typical range of values : $0.0 \leq \text{Tilts} \leq 360.0$ (lin) Minimum Increment : 0.01 Recommended Increment : 10.0
Result

If all parameters are correct `phot_stereo` returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Parallelization Information
`phot_stereo` is *reentrant* and processed *without* parallelization.

Possible Predecessors

[estimate_slal_lr](#), [estimate_slal_zc](#), [estimate_tilt_lr](#), [estimate_tilt_zc](#)

Possible Successors

[shade_height_field](#)

Module

Tools

select_grayvalues_from_channels (MultichannelImage,
 IndexImage : Selected : :)
Selection of gray values of a multi channel image using an index image.

The operator [select_grayvalues_from_channels](#) selects gray values from a [MultichannelImage](#). The channel number for each pixel is determined by using [IndexImage](#): The gray value in [IndexImage](#) is used as the channel number in [MultichannelImage](#).

Parameter

- ▷ **MultichannelImage** (input_object) multichannel-image(-array) $\leadsto$ *Hobject*: byte Multichannel gray image.
- ▷ **IndexImage** (input_object) singlechannel-image(-array) $\leadsto$ *Hobject*: byte Image, where gray values are interpreted as indexes.
- ▷ **Selected** (output_object) singlechannel-image(-array) $\leadsto$ *Hobject*: byte Depth image.

Example

```
compose3(Focus0,Focus1,Focus2,&MultiFocus);
depth_from_focus(MultiFocus,&Depth,&Confidence,'highpass','next_maximum');
mean_image(Depth,&Smooth,15,15);
select_grayvalues_from_channels(MultiChannel,Smooth,SharpImage);
```

Parallelization Information

[select_grayvalues_from_channels](#) is *reentrant* and automatically *parallelized* (on tuple level, domain level).

Possible Predecessors

[depth_from_focus](#), [mean_image](#)

Possible Successors

[disp_image](#)

See also

[count_channels](#)

Module

Tools

sfs_mod_lr (Image : Height : Slant, Tilt, Albedo, Ambient :)
Reconstruct a surface from a gray value image.

[sfs_mod_lr](#) reconstructs a surface (i.e. the relative height of each image point) using the modified algorithm of Lee and Rosenfeld. The surface is reconstructed from the input image [Image](#), and the light source given by the parameters [Slant](#), [Tilt](#), [Albedo](#) and [Ambient](#), and is assumed to lie infinitely far away in the direction given by [Slant](#) and [Tilt](#). The parameter [Albedo](#) determines the albedo of the surface, i.e. the percentage of light reflected in all directions. [Ambient](#) determines the amount of ambient light falling onto the surface. It can be set to values greater than zero if, for example, the white balance of the camera was badly adjusted at the moment the image was taken.

Attention

[sfs_mod_lr](#) assumes that the heights are to be extracted on a lattice with step width 1. If this is not the case, the

calculated heights must be multiplied with the step width after the call to `sfs_mod_lr`. A Cartesian coordinate system with the origin in the lower left corner of the image is used internally. Since the operator is based on the Fast Fourier Transform, only square images with the edge length being a power of 2 are accepted. `sfs_mod_lr` can only handle byte-images.

Parameter	
▷ Image (input_object)	image(-array) $\leadsto$ Hobject : byte Shaded input image.
▷ Height (output_object)	image(-array) $\leadsto$ Hobject : real Reconstructed height field.
▷ Slant (input_control)	angle.deg $\leadsto$ real / integer Angle between the light source and the positive z-axis (in degrees). Default Value : 45.0 Suggested values : Slant $\in \{1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0\}$ Typical range of values : $0.0 \leq \text{Slant} \leq 180.0$ (lin) Minimum Increment : 0.01 Recommended Increment : 10.0
▷ Tilt (input_control)	angle.deg $\leadsto$ real / integer Angle between the light source and the x-axis after projection into the xy-plane (in degrees). Default Value : 45.0 Suggested values : Tilt $\in \{1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0\}$ Typical range of values : $0.0 \leq \text{Tilt} \leq 360.0$ (lin) Minimum Increment : 0.01 Recommended Increment : 10.0
▷ Albedo (input_control)	number $\leadsto$ real / integer Amount of light reflected by the surface. Default Value : 1.0 Suggested values : Albedo $\in \{0.1, 0.5, 1.0, 5.0\}$ Typical range of values : $0.0 \leq \text{Albedo} \leq 5.0$ (lin) Minimum Increment : 0.01 Recommended Increment : 0.1 Restriction : Albedo ≥ 0.0
▷ Ambient (input_control)	number $\leadsto$ real / integer Amount of ambient light. Default Value : 0.0 Suggested values : Ambient $\in \{0.1, 0.5, 1.0\}$ Typical range of values : $0.0 \leq \text{Ambient} \leq 1.0$ (lin) Minimum Increment : 0.01 Recommended Increment : 0.1 Restriction : Ambient ≥ 0.0

Result
If all parameters are correct <code>sfs_mod_lr</code> returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Parallelization Information
<code>sfs_mod_lr</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).

Possible Predecessors
<code>estimate_alam</code> , <code>estimate_slal_lr</code> , <code>estimate_slal_zc</code> , <code>estimate_tilt_lr</code> , <code>estimate_tilt_zc</code>

Possible Successors
<code>shade_height_field</code>

Module
Tools

sfs_orig_lr (Image : Height : Slant, Tilt, Albedo, Ambient :)

Reconstruct a surface from a gray value image.

`sfs_orig_lr` reconstructs a surface (i.e. the relative height of each image point) using the original algorithm of Lee and Rosenfeld. The surface is reconstructed from the input image `Image`. The light source is to be given by the parameters `Slant`, `Tilt`, `Albedo` and `Ambient`, and is assumed to lie infinitely far away in the direction given by `Slant` and `Tilt`. The parameter `Albedo` determines the albedo of the surface, i.e. the percentage of light reflected in all directions. `Ambient` determines the amount of ambient light falling onto the surface. It can be set to values greater than zero if, for example, the white balance of the camera was badly adjusted at the moment the image was taken.

Attention

`sfs_orig_lr` assumes that the heights are to be extracted on a lattice with step width 1. If this is not the case, the calculated heights must be multiplied with the step width after the call to `sfs_orig_lr`. A Cartesian coordinate system with the origin in the lower left corner of the image is used internally. Since the operator is based on the Fast Fourier Transform, only square images with the edge length being a power of 2 are accepted. `sfs_orig_lr` can only handle byte-images.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte
Shaded input image.
- ▷ **Height** (output_object) image(-array) $\leadsto$ *Hobject* : real
Reconstructed height field.
- ▷ **Slant** (input_control) angle.deg $\leadsto$ *real* / integer
Angle between the light source and the positive z-axis (in degrees).
Default Value : 45.0
Suggested values : `Slant` $\in$ {1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0}
Typical range of values : $0.0 \leq \text{Slant} \leq 180.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 10.0
- ▷ **Tilt** (input_control) angle.deg $\leadsto$ *real* / integer
Angle between the light source and the x-axis after projection into the xy-plane (in degrees).
Default Value : 45.0
Suggested values : `Tilt` $\in$ {1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0}
Typical range of values : $0.0 \leq \text{Tilt} \leq 360.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 10.0
- ▷ **Albedo** (input_control) number $\leadsto$ *real* / integer
Amount of light reflected by the surface.
Default Value : 1.0
Suggested values : `Albedo` $\in$ {0.1, 0.5, 1.0, 5.0}
Typical range of values : $0.0 \leq \text{Albedo} \leq 5.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : `Albedo` ≥ 0.0
- ▷ **Ambient** (input_control) number $\leadsto$ *real* / integer
Amount of ambient light.
Default Value : 0.0
Suggested values : `Ambient` $\in$ {0.1, 0.5, 1.0}
Typical range of values : $0.0 \leq \text{Ambient} \leq 1.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : `Ambient` ≥ 0.0

Result

If all parameters are correct `sfs_orig_lr` returns the value 2 (`H_MSG_TRUE`). Otherwise, an exception is raised.

Parallelization Information

`sfs_orig_lr` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`estimate_al_am`, `estimate_sl_al_lr`, `estimate_sl_al_zc`, `estimate_tilt_lr`,
`estimate_tilt_zc`

Possible Successors

[shade_height_field](#)

Module

Tools

sfs_pentland (Image : Height : Slant, Tilt, Albedo, Ambient :)

Reconstruct a surface from a gray value image.

[sfs_pentland](#) reconstructs a surface (i.e. the relative height of each image point) using the algorithm of Pentland. The surface is reconstructed from the input image [Image](#). The light source must be given by the parameters [Slant](#), [Tilt](#), [Albedo](#) and [Ambient](#), and is assumed to lie infinitely far away in the direction given by [Slant](#) and [Tilt](#). The parameter [Albedo](#) determines the albedo of the surface, i.e. the percentage of light reflected in all directions. [Ambient](#) determines the amount of ambient light falling onto the surface. It can be set to values greater than zero if, for example, the white balance of the camera was badly adjusted at the moment the image was taken.

Attention

[sfs_pentland](#) assumes that the heights are to be extracted on a lattice with step width 1. If this is not the case, the calculated heights must be multiplied with the step width after the call to [sfs_pentland](#). A Cartesian coordinate system with the origin in the lower left corner of the image is used internally. Since the operator is based on the Fast Fourier Transform, only square images with the edge length being a power of 2 are accepted. [sfs_pentland](#) can only handle byte-images.

Parameter

- ▷ **Image** (input_object) image(-array) $\leadsto$ *Hobject* : byte
Shaded input image.
- ▷ **Height** (output_object) image(-array) $\leadsto$ *Hobject* : real
Reconstructed height field.
- ▷ **Slant** (input_control) angle.deg $\leadsto$ *real* / integer
Angle between the light source and the positive z-axis (in degrees).
Default Value : 45.0
Suggested values : $\text{Slant} \in \{1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0\}$
Typical range of values : $0.0 \leq \text{Slant} \leq 180.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
- ▷ **Tilt** (input_control) angle.deg $\leadsto$ *real* / integer
Angle between the light source and the x-axis after projection into the xy-plane (in degrees).
Default Value : 45.0
Suggested values : $\text{Tilt} \in \{1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0\}$
Typical range of values : $0.0 \leq \text{Tilt} \leq 360.0$ (lin)
Minimum Increment : 1.0
Recommended Increment : 10.0
- ▷ **Albedo** (input_control) number $\leadsto$ *real* / integer
Amount of light reflected by the surface.
Default Value : 1.0
Suggested values : $\text{Albedo} \in \{0.1, 0.5, 1.0, 5.0\}$
Typical range of values : $0.0 \leq \text{Albedo} \leq 5.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : $\text{Albedo} \geq 0.0$

- ▷ **Ambient** (input_control) number $\leadsto$ real / integer
Amount of ambient light.
Default Value : 0.0
Suggested values : Ambient $\in \{0.1, 0.5, 1.0\}$
Typical range of values : $0.0 \leq \text{Ambient} \leq 1.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : Ambient ≥ 0.0

Result

If all parameters are correct `sfs_pentland` returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Parallelization Information

`sfs_pentland` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`estimate_alam`, `estimate_slal_lr`, `estimate_slal_zc`, `estimate_tilt_lr`,
`estimate_tilt_zc`

Possible Successors

`shade_height_field`

Module

Tools

shade_height_field (ImageHeight : ImageShade : Slant, Tilt, Albedo,
Ambient, Shadows :)

Shade a height field.

`shade_height_field` computes a shaded image from the height field `ImageHeight` as if the image were illuminated by an infinitely far away light source. It is assumed that the surface described by the height field has Lambertian reflection properties determined by `Albedo` and `Ambient`. The parameter `Shadows` determines whether shadows are to be calculated.

Attention

`shade_height_field` assumes that the heights are given on a lattice with step width 1. If this is not the case, the heights must be divided by the step width before the call to `shade_height_field`. Otherwise, the derivatives used internally to compute the orientation of the surface will be estimated to steep or too flat. Example: The height field is given on 100*100 points on the square $[0,1] \times [0,1]$. Then the heights must be divided by 1/100 first. A Cartesian coordinate system with the origin in the lower left corner of the image is used internally.

Parameter

- ▷ **ImageHeight** (input_object) image(-array) $\leadsto$ Hobject : byte / int4 / real
Height field to be shaded.
- ▷ **ImageShade** (output_object) image(-array) $\leadsto$ Hobject : byte
Shaded image.
- ▷ **Slant** (input_control) angle.deg $\leadsto$ real / integer
Angle between the light source and the positive z-axis (in degrees).
Default Value : 0.0
Suggested values : Slant $\in \{1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0\}$
Typical range of values : $0.0 \leq \text{Slant} \leq 180.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 10.0
- ▷ **Tilt** (input_control) angle.deg $\leadsto$ real / integer
Angle between the light source and the x-axis after projection into the xy-plane (in degrees).
Default Value : 0.0
Suggested values : Tilt $\in \{1.0, 5.0, 10.0, 20.0, 40.0, 60.0, 90.0\}$
Typical range of values : $0.0 \leq \text{Tilt} \leq 360.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 10.0

- ▷ **Albedo** (input_control) number $\leadsto$ real / integer
Amount of light reflected by the surface.
Default Value : 1.0
Suggested values : Albedo $\in \{0.1, 0.5, 1.0, 5.0\}$
Typical range of values : $0.0 \leq \text{Albedo} \leq 5.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : Albedo ≥ 0.0
- ▷ **Ambient** (input_control) number $\leadsto$ real / integer
Amount of ambient light.
Default Value : 0.0
Suggested values : Ambient $\in \{0.1, 0.5, 1.0\}$
Typical range of values : $0.0 \leq \text{Ambient} \leq 1.0$ (lin)
Minimum Increment : 0.01
Recommended Increment : 0.1
Restriction : Ambient ≥ 0.0
- ▷ **Shadows** (input_control) string $\leadsto$ string
Should shadows be calculated?
Default Value : 'false'
Suggested values : Shadows $\in \{\text{'true'}, \text{'false'}\}$

Result

If all parameters are correct `shade_height_field` returns the value 2 (H_MSG_TRUE). Otherwise, an exception is raised.

Parallelization Information

`shade_height_field` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`sfs_mod_lr`, `sfs_orig_lr`, `sfs_pentland`, `phot_stereo`

Module

Tools

Chapter 16

Tuple

16.1 Arithmetic

tuple_abs (: : T : Abs)

Compute the absolute value of a tuple.

tuple_abs computes the absolute value of the input tuple **T**. The absolute value of an integer number is again an integer number. The absolute value of a floating point number is a floating point number. The absolute value of a string is not allowed.

Parameter

- ▷ **T** (input_control) number(-array) $\leadsto$ real / integer
Input tuple.
- ▷ **Abs** (output_control) number(-array) $\leadsto$ real / integer
Absolute value of the input tuple.

Parallelization Information

tuple_abs is *reentrant* and processed *without* parallelization.

Alternatives

tuple_fabs

Module

Operators not requiring licensing

tuple_acos (: : T : ACos)

Compute the arccosine of a tuple.

tuple_acos computes the arccosine of the input tuple **T**. The arccosine is always returned as a floating point number. The arccosine of a string is not allowed.

Parameter

- ▷ **T** (input_control) number(-array) $\leadsto$ real / integer
Input tuple.
Restriction : $(-1 \leq T) \leq 1$
- ▷ **ACos** (output_control) number(-array) $\leadsto$ real
Arccosine of the input tuple.

Parallelization Information

tuple_acos is *reentrant* and processed *without* parallelization.

Alternatives

tuple_asin, **tuple_atan**, **tuple_atan2**

See also

[tuple_cos](#)

Module

Operators not requiring licensing

tuple_add (: : S1, S2 : Sum)

Add two tuples.

tuple_add computes the sum of the input tuples **S1** and **S2**. If both tuples have the same length the corresponding elements of both tuples are added. Otherwise, either **S1** or **S2** must have length 1. In this case, the addition is performed for each element of the longer tuple with the single element of the other tuple. If two integer numbers are added, the result is again an integer number. If a floating point number is added to another number, the result is a floating point number. If two strings are added, the addition corresponds to a string concatenation. If a number and a string are added, the number is converted to a string first. Thus, the addition also corresponds to a string concatenation in this case.

Parameter

- ▷ **S1** (input_control) number(-array) $\leadsto$ real / integer / string
Input tuple 1.
- ▷ **S2** (input_control) number(-array) $\leadsto$ real / integer / string
Input tuple 2.
- ▷ **Sum** (output_control) number(-array) $\leadsto$ real / integer / string
Sum of the input tuples.

Parallelization Information

tuple_add is *reentrant* and processed *without* parallelization.

Alternatives

[tuple_sub](#)

Module

Operators not requiring licensing

tuple_asin (: : T : ASin)

Compute the arcsine of a tuple.

tuple_asin computes the arcsine of the input tuple **T**. The arcsine is always returned as a floating point number. The arcsine of a string is not allowed.

Parameter

- ▷ **T** (input_control) number(-array) $\leadsto$ real / integer
Input tuple.
Restriction : $(-1 \leq T) \leq 1$
- ▷ **ASin** (output_control) number(-array) $\leadsto$ real
Arcsine of the input tuple.

Parallelization Information

tuple_asin is *reentrant* and processed *without* parallelization.

Alternatives

[tuple_acos](#), [tuple_atan](#), [tuple_atan2](#)

See also

[tuple_sin](#)

Module

Operators not requiring licensing

tuple_atan (: : T : ATan)

Compute the arctangent of a tuple.

tuple_atan computes the arctangent of the input tuple **T**. The arctangent is always returned as a floating point number. The arctangent of a string is not allowed.

Parameter

- ▷ **T** (input_control) number(-array) $\leadsto$ *real* / integer
Input tuple.
- ▷ **ATan** (output_control) number(-array) $\leadsto$ *real*
Arctangent of the input tuple.

Parallelization Information

tuple_atan is *reentrant* and processed *without* parallelization.

Alternatives

[tuple_atan2](#), [tuple_asin](#), [tuple_acos](#)

See also

[tuple_tan](#)

Module

Operators not requiring licensing

tuple_atan2 (: : Y, X : ATan)

Compute the arctangent of a tuple for all four quadrants.

tuple_atan2 computes the arctangent of the input tuples **Y/X** while treating all four quadrants correctly. The arctangent is always returned as a floating point number. The arctangent of a string is not allowed.

Parameter

- ▷ **Y** (input_control) number(-array) $\leadsto$ *real* / integer
Input tuple of the y-values.
- ▷ **X** (input_control) number(-array) $\leadsto$ *real* / integer
Input tuple of the x-values.
- ▷ **ATan** (output_control) number(-array) $\leadsto$ *real*
Arctangent of the input tuple.

Parallelization Information

tuple_atan2 is *reentrant* and processed *without* parallelization.

Alternatives

[tuple_atan](#), [tuple_asin](#), [tuple_acos](#)

See also

[tuple_tan](#)

Module

Operators not requiring licensing

tuple_ceil (: : T : Ceil)

Compute the ceiling function of a tuple.

tuple_ceil computes the ceiling function of the input tuple **T**, i.e., the smallest integer greater than or equal to **T**. The ceiling function is always returned as a floating point number. The ceiling function of a string is not allowed.

Parameter
▷ T (input_control) number(-array) $\leadsto$ <i>real</i> / integer Input tuple. ▷ Ceil (output_control) number(-array) $\leadsto$ <i>real</i> Ceiling function of the input tuple.
Parallelization Information
tuple_ceil is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
tuple_ceil
Module
Operators not requiring licensing

tuple_cos (: : T : Cos)

Compute the cosine of a tuple.

tuple_cos computes the cosine of the input tuple **T**. The cosine is always returned as a floating point number. The cosine of a string is not allowed.

Parameter
▷ T (input_control) number(-array) $\leadsto$ <i>real</i> / integer Input tuple. ▷ Cos (output_control) number(-array) $\leadsto$ <i>real</i> Cosine of the input tuple.
Parallelization Information
tuple_cos is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
tuple_sin , tuple_tan
See also
tuple_acos
Module
Operators not requiring licensing

tuple_cosh (: : T : Cosh)

Compute the hyperbolic cosine of a tuple.

tuple_cosh computes the hyperbolic cosine of the input tuple **T**. The hyperbolic cosine is always returned as a floating point number. The hyperbolic cosine of a string is not allowed.

Parameter
▷ T (input_control) number(-array) $\leadsto$ <i>real</i> / integer Input tuple. ▷ Cosh (output_control) number(-array) $\leadsto$ <i>real</i> Hyperbolic cosine of the input tuple.
Parallelization Information
tuple_cosh is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
tuple_sinh , tuple_tanh
Module
Operators not requiring licensing

tuple_div (: : Q1, Q2 : Quot)

Divide two tuples.

tuple_div computes the quotient of the input tuples **Q1** and **Q2**. If both tuples have the same length the corresponding elements of both tuples are divided. Otherwise, either **Q1** or **Q2** must have length 1. In this case, the division is performed for each element of the longer tuple with the single element of the other tuple. If two integer numbers are divided, the result is again an integer number. If one of the operands is a floating point number, the result is a floating point number. The division of strings is not allowed.

Parameter

- ▷ **Q1** (input_control) number(-array) $\leadsto$ real / integer
Input tuple 1.
- ▷ **Q2** (input_control) number(-array) $\leadsto$ real / integer
Input tuple 2.
- Restriction** : $Q2 \neq 0$
- ▷ **Quot** (output_control) number(-array) $\leadsto$ real / integer
Quotient of the input tuples.

Parallelization Information

tuple_div is *reentrant* and processed *without* parallelization.

Alternatives

tuple_mult

Module

Operators not requiring licensing

tuple_exp (: : T : Exp)

Compute the exponential of a tuple.

tuple_exp computes the exponential of the input tuple **T**. The exponential is always returned as a floating point number. The exponential of a string is not allowed.

Parameter

- ▷ **T** (input_control) number(-array) $\leadsto$ real / integer
Input tuple.
- ▷ **Exp** (output_control) number(-array) $\leadsto$ real
Exponential of the input tuple.

Parallelization Information

tuple_exp is *reentrant* and processed *without* parallelization.

Alternatives

tuple_pow

See also

tuple_log, **tuple_log10**

Module

Operators not requiring licensing

tuple_fabs (: : T : Abs)

Compute the absolute value of a tuple (as floating point numbers).

tuple_fabs computes the absolute value of the input tuple **T**. In contrast to **tuple_abs**, the absolute value is always returned as a floating point number by **tuple_fabs**. The absolute value of a string is not allowed.

Parameter
▷ T (input_control) number(-array) $\leadsto$ real / integer Input tuple. ▷ Abs (output_control) number(-array) $\leadsto$ real Absolute value of the input tuple.
Parallelization Information
tuple_fabs is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
tuple_abs
Module
Operators not requiring licensing

tuple_floor (: : T : Floor)

Compute the floor function of a tuple.

tuple_floor computes the floor function of the input tuple **T**, i.e., the largest integer less than or equal to **T**. The floor function is always returned as a floating point number. The floor function of a string is not allowed.

Parameter
▷ T (input_control) number(-array) $\leadsto$ real / integer Input tuple. ▷ Floor (output_control) number(-array) $\leadsto$ real Floor function of the input tuple.
Parallelization Information
tuple_floor is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
tuple_ceil
Module
Operators not requiring licensing

tuple_fmod (: : T1, T2 : Fmod)

Calculate the remainder of the floating point division of two tuples.

tuple_fmod computes the remainder of the floating point division of the input tuples **T1**/**T2**. If both tuples have the same length the division is performed for the corresponding elements of both tuples. Otherwise, either **T1** or **T2** must have length 1. In this case, the division is performed for each element of the longer tuple with the single element of the other tuple. The result is always a floating point number. The division of strings is not allowed.

Parameter
▷ T1 (input_control) number(-array) $\leadsto$ real / integer Input tuple 1. ▷ T2 (input_control) number(-array) $\leadsto$ real / integer Input tuple 2. Restriction : $T2 \neq 0.0$ ▷ Fmod (output_control) number(-array) $\leadsto$ real Remainder of the division of the input tuples.
Parallelization Information
tuple_fmod is <i>reentrant</i> and processed <i>without</i> parallelization.
See also
tuple_floor , tuple_ceil
Module
Operators not requiring licensing

tuple_ldexp (: : T1, T2 : Ldexp)

Calculate the ldexp function of two tuples.

tuple_ldexp computes the ldexp function of the input tuples, i.e., $T1 * 2^{T2}$. If both tuples have the same length the operation is performed for the corresponding elements of both tuples. Otherwise, either **T1** or **T2** must have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The result is always a floating point number. The ldexp function of strings is not allowed.

Parameter

- ▷ **T1** (input_control) number(-array) $\leadsto$ real / integer
Input tuple 1.
- ▷ **T2** (input_control) number(-array) $\leadsto$ real / integer
Input tuple 2.
- ▷ **Ldexp** (output_control) number(-array) $\leadsto$ real
Ldexp function of the input tuples.

Parallelization Information

tuple_ldexp is *reentrant* and processed *without* parallelization.

See also

[tuple_exp](#)

Module

Operators not requiring licensing

tuple_log (: : T : Log)

Compute the natural logarithm of a tuple.

tuple_log computes the natural logarithm of the input tuple **T**. The natural logarithm is always returned as a floating point number. The natural logarithm of a string is not allowed.

Parameter

- ▷ **T** (input_control) number(-array) $\leadsto$ real / integer
Input tuple.
Restriction : $T > 0$
- ▷ **Log** (output_control) number(-array) $\leadsto$ real
Natural logarithm of the input tuple.

Parallelization Information

tuple_log is *reentrant* and processed *without* parallelization.

Alternatives

[tuple_log10](#)

See also

[tuple_exp](#), [tuple_pow](#)

Module

Operators not requiring licensing

tuple_log10 (: : T : Log)

Compute the base 10 logarithm of a tuple.

tuple_log10 computes the base 10 logarithm of the input tuple **T**. The logarithm is always returned as a floating point number. The logarithm of a string is not allowed.

Parameter
▷ T (input_control) number(-array) $\leadsto$ real / integer Input tuple. Restriction : $T > 0$ ▷ Log (output_control) number(-array) $\leadsto$ real Base 10 logarithm of the input tuple.
Parallelization Information
<code>tuple_log10</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
<code>tuple_log</code>
See also
<code>tuple_exp</code> , <code>tuple_pow</code>
Module
Operators not requiring licensing

tuple_mult (: : P1, P2 : Prod)

Multiply two tuples.

`tuple_mult` computes the product of the input tuples **P1** and **P2**. If both tuples have the same length the corresponding elements of both tuples are subtracted. Otherwise, either **P1** or **P2** must have length 1. In this case, the multiplication is performed for each element of the longer tuple with the single element of the other tuple. If two integer numbers are multiplied, the result is again an integer number. If one of the operands is a floating point number, the result is a floating point number. The multiplication of strings is not allowed.

Parameter
▷ P1 (input_control) number(-array) $\leadsto$ real / integer Input tuple 1. ▷ P2 (input_control) number(-array) $\leadsto$ real / integer Input tuple 2. ▷ Prod (output_control) number(-array) $\leadsto$ real / integer Product of the input tuples.
Parallelization Information
<code>tuple_mult</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
<code>tuple_div</code>
Module
Operators not requiring licensing

tuple_neg (: : T : Neg)

Negate a tuple.

`tuple_neg` computes the negation of the input tuple **T**, i.e., **Neg** = $-T$. The negation of an integer number is again an integer number. The negation of a floating point number is a floating point number. The negation of a string is not allowed. The negation of an empty input tuple results in an empty output tuple.

Parameter
▷ T (input_control) number(-array) $\leadsto$ real / integer Input tuple. ▷ Neg (output_control) number(-array) $\leadsto$ real / integer Negation of the input tuple.

Parallelization Information

tuple_neg is *reentrant* and processed *without* parallelization.

Module

Operators not requiring licensing

tuple_pow (: : T1, T2 : Pow)

Calculate the power function two tuples.

tuple_pow computes the power function of the input tuples **T1**^{**T2**}. If both tuples have the same length the power function is applied to the corresponding elements of both tuples. Otherwise, either **T1** or **T2** must have length 1. In this case, the power function is performed for each element of the longer tuple with the single element of the other tuple. The result is always a floating point number. The power function of strings is not allowed.

Parameter

- ▷ **T1** (input_control) number(-array) $\leadsto$ *real* / integer
Input tuple 1.
- ▷ **T2** (input_control) number(-array) $\leadsto$ *real* / integer
Input tuple 2.
- ▷ **Pow** (output_control) number(-array) $\leadsto$ *real* / integer
Power function of the input tuples.

Parallelization Information

tuple_pow is *reentrant* and processed *without* parallelization.

Alternatives

tuple_exp

See also

tuple_log, **tuple_log10**

Module

Operators not requiring licensing

tuple_rad (: : Deg : Rad)

Convert a tuple from degrees to radians.

tuple_rad converts the input tuple **Deg** from degrees to radians. The result is always returned as a floating point number. The conversion of a string is not allowed.

Parameter

- ▷ **Deg** (input_control) number(-array) $\leadsto$ *real* / integer
Input tuple.
- ▷ **Rad** (output_control) number(-array) $\leadsto$ *real*
Input tuple in radians.

Parallelization Information

tuple_rad is *reentrant* and processed *without* parallelization.

See also

tuple_deg

Module

Operators not requiring licensing

tuple_sin (: : T : Sin)

Compute the sine of a tuple.

tuple_sin computes the sine of the input tuple **T**. The sine is always returned as a floating point number. The sine of a string is not allowed.

Parameter	
▷ T (input_control)	number(-array) $\leadsto$ <i>real</i> / integer Input tuple.
▷ Sin (output_control)	number(-array) $\leadsto$ <i>real</i> Sine of the input tuple.
Parallelization Information	
tuple_sin is <i>reentrant</i> and processed <i>without</i> parallelization.	
Alternatives	
tuple_cos , tuple_tan	
See also	
tuple_asin	
Module	
Operators not requiring licensing	

tuple_sinh (: : T : Sinh)

Compute the hyperbolic sine of a tuple.

tuple_sinh computes the hyperbolic sine of the input tuple **T**. The hyperbolic sine is always returned as a floating point number. The hyperbolic sine of a string is not allowed.

Parameter	
▷ T (input_control)	number(-array) $\leadsto$ <i>real</i> / integer Input tuple.
▷ Sinh (output_control)	number(-array) $\leadsto$ <i>real</i> Hyperbolic sine of the input tuple.
Parallelization Information	
tuple_sinh is <i>reentrant</i> and processed <i>without</i> parallelization.	
Alternatives	
tuple_cosh , tuple_tanh	
Module	
Operators not requiring licensing	

tuple_sqrt (: : T : Sqrt)

Compute the square root of a tuple.

tuple_sqrt computes the square root of the input tuple **T**. The square root is always returned as a floating point number. The square root of a string is not allowed.

Parameter	
▷ T (input_control)	number(-array) $\leadsto$ <i>real</i> / integer Input tuple. Restriction : $T \geq 0$
▷ Sqrt (output_control)	number(-array) $\leadsto$ <i>real</i> Square root of the input tuple.
Parallelization Information	
tuple_sqrt is <i>reentrant</i> and processed <i>without</i> parallelization.	
Module	
Operators not requiring licensing	

tuple_sub (: : D1, D2 : Diff)

Subtract two tuples.

tuple_sub computes the difference of the input tuples **D1** and **D2**. If both tuples have the same length the corresponding elements of both tuples are subtracted. Otherwise, either **D1** or **D2** must have length 1. In this case, the subtraction is performed for each element of the longer tuple with the single element of the other tuple. If two integer numbers are subtracted, the result is again an integer number. If one of the operands is a floating point number, the result is a floating point number. The subtraction of strings is not allowed.

Parameter

- ▷ **D1** (input_control) number(-array) $\leadsto$ real / integer
Input tuple 1.
- ▷ **D2** (input_control) number(-array) $\leadsto$ real / integer
Input tuple 2.
- ▷ **Diff** (output_control) number(-array) $\leadsto$ real / integer
Difference of the input tuples.

Parallelization Information

tuple_sub is *reentrant* and processed *without* parallelization.

Alternatives

tuple_add

Module

Operators not requiring licensing

tuple_tan (: : T : Tan)

Compute the tangent of a tuple.

tuple_tan computes the tangent of the input tuple **T**. The tangent is always returned as a floating point number. The tangent of a string is not allowed.

Parameter

- ▷ **T** (input_control) number(-array) $\leadsto$ real / integer
Input tuple.
- ▷ **Tan** (output_control) number(-array) $\leadsto$ real
Tangent of the input tuple.

Parallelization Information

tuple_tan is *reentrant* and processed *without* parallelization.

Alternatives

tuple_sin, **tuple_cos**

See also

tuple_atan, **tuple_atan2**

Module

Operators not requiring licensing

tuple_tanh (: : T : Tanh)

Compute the hyperbolic tangent of a tuple.

tuple_tanh computes the hyperbolic tangent of the input tuple **T**. The hyperbolic tangent is always returned as a floating point number. The hyperbolic tangent of a string is not allowed.

Parameter
▷ T (input_control) number(-array) $\leadsto$ <i>real</i> / <i>integer</i> Input tuple. ▷ Tanh (output_control) number(-array) $\leadsto$ <i>real</i> Hyperbolic tangent of the input tuple.
Parallelization Information
tuple_tanh is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.
Alternatives
tuple_sinh , tuple_cosh
Module
Operators not requiring licensing

16.2 Bit-Operations

tuple_band (: : T1, T2 : BAnd)

Compute the bitwise and of two tuples.

tuple_band computes the bitwise and of the input tuples **T1** and **T2**. If both tuples have the same length the operation is performed on the corresponding elements of both tuples. Otherwise, either **T1** or **T2** must have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The input tuples must contain only integer numbers.

Parameter
▷ T1 (input_control) integer(-array) $\leadsto$ <i>integer</i> Input tuple 1. ▷ T2 (input_control) integer(-array) $\leadsto$ <i>integer</i> Input tuple 2. ▷ BAnd (output_control) integer(-array) $\leadsto$ <i>integer</i> Binary and of the input tuples.
Parallelization Information
tuple_band is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
tuple_bor , tuple_bxor , tuple_bnot
See also
tuple_and , tuple_or , tuple_xor , tuple_not
Module
Operators not requiring licensing

tuple_bnot (: : T : BNot)

Compute the bitwise not of two tuples.

tuple_bnot computes the bitwise not of the input tuple **T**. The input tuple must contain only integer numbers.

Parameter
▷ T (input_control) integer(-array) $\leadsto$ <i>integer</i> Input tuple. ▷ BNot (output_control) integer(-array) $\leadsto$ <i>integer</i> Binary not of the input tuple.
Parallelization Information
tuple_bnot is <i>reentrant</i> and processed <i>without</i> parallelization.

	Alternatives
tuple_band , tuple_bor , tuple_bxor	
	See also
tuple_and , tuple_or , tuple_xor , tuple_not	
	Module
Operators not requiring licensing	

tuple_bor (: : T1, T2 : BOr)

Compute the bitwise or of two tuples.

tuple_bor computes the bitwise or of the input tuples **T1** and **T2**. If both tuples have the same length the operation is performed on the corresponding elements of both tuples. Otherwise, either **T1** or **T2** must have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The input tuples must contain only integer numbers.

	Parameter
▷ T1 (input_control) integer(-array) $\leadsto$ integer Input tuple 1.	
▷ T2 (input_control) integer(-array) $\leadsto$ integer Input tuple 2.	
▷ BOr (output_control) integer(-array) $\leadsto$ integer Binary or of the input tuples.	

	Parallelization Information
tuple_bor is <i>reentrant</i> and processed <i>without</i> parallelization.	

	Alternatives
tuple_band , tuple_bxor , tuple_bnot	
	See also
tuple_and , tuple_or , tuple_xor , tuple_not	
	Module
Operators not requiring licensing	

tuple_bxor (: : T1, T2 : BXor)

Compute the bitwise exclusive or of two tuples.

tuple_bxor computes the bitwise exclusive or of the input tuples **T1** and **T2**. If both tuples have the same length the operation is performed on the corresponding elements of both tuples. Otherwise, either **T1** or **T2** must have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The input tuples must contain only integer numbers.

	Parameter
▷ T1 (input_control) integer(-array) $\leadsto$ integer Input tuple 1.	
▷ T2 (input_control) integer(-array) $\leadsto$ integer Input tuple 2.	
▷ BXor (output_control) integer(-array) $\leadsto$ integer Binary exclusive or of the input tuples.	

	Parallelization Information
tuple_bxor is <i>reentrant</i> and processed <i>without</i> parallelization.	

	Alternatives
tuple_band , tuple_bor , tuple_bnot	
	See also
tuple_and , tuple_or , tuple_xor , tuple_not	

Module

Operators not requiring licensing

tuple_lsh (: : T, Shift : Lsh)

Shift a tuple bitwise to the left.

tuple_lsh shifts the tuple **T** bitwise to the left by **Shift** places. If no overflow occurs, this operation is equivalent to a multiplication by 2^{Shift} . If **T** is negative, the result depends on the hardware. If **Shift** is negative or larger than 32, the result is undefined. If both tuples have the same length the corresponding elements of both tuples are shifted. Otherwise, either **T** or **Shift** must have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The input tuples must contain only integer numbers.

Parameter

- ▷ **T** (input_control) integer(-array) $\leadsto$ integer
Input tuple.
- ▷ **Shift** (input_control) integer(-array) $\leadsto$ integer
Number of places to shift the input tuple.
- ▷ **Lsh** (output_control) integer(-array) $\leadsto$ integer
Shifted input tuple.

Parallelization Information

tuple_lsh is *reentrant* and processed *without* parallelization.

Alternatives

[tuple_mult](#)

See also

[tuple_rsh](#)

Module

Operators not requiring licensing

tuple_rsh (: : T, Shift : Rsh)

Shift a tuple bitwise to the right.

tuple_rsh shifts the tuple **T** bitwise to the right by **Shift** places. This operation is equivalent to a division by 2^{Shift} . If **T** is negative, the result depends on the hardware. If **Shift** is negative or larger than 32, the result is undefined. If both tuples have the same length the corresponding elements of both tuples are shifted. Otherwise, either **T** or **Shift** must have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The input tuples must contain only integer numbers.

Parameter

- ▷ **T** (input_control) integer(-array) $\leadsto$ integer
Input tuple.
- ▷ **Shift** (input_control) integer(-array) $\leadsto$ integer
Number of places to shift the input tuple.
- ▷ **Rsh** (output_control) integer(-array) $\leadsto$ integer
Shifted input tuple.

Parallelization Information

tuple_rsh is *reentrant* and processed *without* parallelization.

Alternatives

[tuple_div](#)

See also

[tuple_lsh](#)

Module

Operators not requiring licensing

16.3 Comparison

tuple_equal (: : T1, T2 : Equal)

Test, whether two tuples are equal.

tuple_equal tests whether the two input tuples **T1** and **T2** are equal by comparing the tuples elementwise. Two tuples are equal, if they have got the same number of elements and if their elements are equal. Two tuple elements are equal, if they are both (integer or floating point) numbers or both are strings and contain the same value.

Parameter

- ▷ **T1** (input_control) number(-array) $\leadsto$ integer / real / string
Input tuple 1.
- ▷ **T2** (input_control) number(-array) $\leadsto$ integer / real / string
Input tuple 2.
- ▷ **Equal** (output_control) integer $\leadsto$ integer
Result of the comparison of the input tuples.

Parallelization Information

tuple_equal is *reentrant* and processed *without* parallelization.

Alternatives

[tuple_not_equal](#), [tuple_less](#), [tuple_greater](#), [tuple_less_equal](#), [tuple_greater_equal](#)

Module

Operators not requiring licensing

tuple_greater (: : T1, T2 : Greater)

Test, whether a tuple is greater than another tuple.

tuple_greater tests whether the input tuple **T1** is greater than **T2**. A tuple **T1** is said to be greater than a tuple **T2**, if **T1** has been found to be greater when comparing it elementwise to **T2** or (for the case that the elementwise comparison did not show that **T1** is greater than **T2**) if **T1** has got more elements than **T2**. With the elementwise comparison the single elements of **T1** and **T2** are compared with each other one after another (i.e., the first element of **T1** is compared to the first element of **T2** and the second element of **T1** is compared to the second element of **T2** etc.). If an element of **T1** is greater than its counterpart of **T2**, **T1** is said to be greater than **T2**. As a precondition for comparing the tuples elementwise two corresponding elements must either both be (integer or floating point) numbers or both be strings. Otherwise **tuple_greater** returns an error.

Parameter

- ▷ **T1** (input_control) number(-array) $\leadsto$ integer / real / string
Input tuple 1.
- ▷ **T2** (input_control) number(-array) $\leadsto$ integer / real / string
Input tuple 2.
- ▷ **Greater** (output_control) integer $\leadsto$ integer
Result of the comparison of the input tuples.

Parallelization Information

tuple_greater is *reentrant* and processed *without* parallelization.

Alternatives

[tuple_greater_equal](#), [tuple_less](#), [tuple_less_equal](#), [tuple_equal](#), [tuple_not_equal](#)

Module

Operators not requiring licensing

tuple_greater_equal (: : T1, T2 : Greatereq)

Test, whether a tuple is greater or equal to another tuple.

`tuple_greater_equal` tests whether the input tuple **T1** is greater or equal to **T2**. A tuple **T1** is said to be greater or equal to a tuple **T2**, if **T1** is not less than **T2**. A tuple **T1** is said to be less than a tuple **T2**, if **T1** has been found to be less when comparing it elementwise to **T2** or (for the case that the elementwise comparison did not show that **T1** is less than **T2**) if **T1** has got less elements than **T2**. With the elementwise comparison the single elements of **T1** and **T2** are compared with each other one after another (i.e., the first element of **T1** is compared to the first element of **T2** and the second element of **T1** is compared to the second element of **T2** etc.). If an element of **T1** is less than its counterpart of **T2**, **T1** is said to be less than **T2**. As a precondition for comparing the tuples elementwise two corresponding elements must either both be (integer or floating point) numbers or both be strings. Otherwise `tuple_greater_equal` returns an error.

Parameter

- ▷ **T1** (input_control) number(-array) $\rightsquigarrow$ integer / real / string
Input tuple 1.
- ▷ **T2** (input_control) number(-array) $\rightsquigarrow$ integer / real / string
Input tuple 2.
- ▷ **Greaterreq** (output_control) integer $\rightsquigarrow$ integer
Result of the comparison of the input tuples.

Parallelization Information

`tuple_greater_equal` is *reentrant* and processed *without* parallelization.

Alternatives

`tuple_greater`, `tuple_less`, `tuple_less_equal`, `tuple_equal`, `tuple_not_equal`

Module

Operators not requiring licensing

tuple_less (: : T1, T2 : Less)

Test, whether a tuple is less than another tuple.

`tuple_less` tests whether the input tuple **T1** is less than **T2**. A tuple **T1** is said to be less than a tuple **T2**, if **T1** has been found to be less when comparing it elementwise to **T2** or (for the case that the elementwise comparison did not show that **T1** is less than **T2**) if **T1** has got less elements than **T2**. With the elementwise comparison the single elements of **T1** and **T2** are compared with each other one after another (i.e., the first element of **T1** is compared to the first element of **T2** and the second element of **T1** is compared to the second element of **T2** etc.). If an element of **T1** is less than its counterpart of **T2**, **T1** is said to be less than **T2**. As a precondition for comparing the tuples elementwise two corresponding elements must either both be (integer or floating point) numbers or both be strings. Otherwise `tuple_less` returns an error.

Parameter

- ▷ **T1** (input_control) number(-array) $\rightsquigarrow$ integer / real / string
Input tuple 1.
- ▷ **T2** (input_control) number(-array) $\rightsquigarrow$ integer / real / string
Input tuple 2.
- ▷ **Less** (output_control) integer $\rightsquigarrow$ integer
Result of the comparison of the input tuples.

Parallelization Information

`tuple_less` is *reentrant* and processed *without* parallelization.

Alternatives

`tuple_less_equal`, `tuple_greater`, `tuple_greater_equal`, `tuple_equal`, `tuple_not_equal`

Module

Operators not requiring licensing

tuple_less_equal (: : T1, T2 : Lesseq)

Test, whether a tuple is less or equal to another tuple.

tuple_less_equal tests whether the input tuple **T1** is less or equal to **T2**. A tuple **T1** is said to be less or equal to a tuple **T2**, if **T1** is not greater than **T2**. A tuple **T1** is said to be greater than a tuple **T2**, if **T1** has been found to be greater when comparing it elementwise to **T2** or (for the case that the elementwise comparison did not show that **T1** is greater than **T2**) if **T1** has got more elements than **T2**. With the elementwise comparison the single elements of **T1** and **T2** are compared with each other one after another (i.e., the first element of **T1** is compared to the first element of **T2** and the second element of **T1** is compared to the second element of **T2** etc.). If an element of **T1** is greater than its counterpart of **T2**, **T1** is said to be greater than **T2**. As a precondition for comparing the tuples elementwise two corresponding elements must either both be (integer or floating point) numbers or both be strings. Otherwise **tuple_less_equal** returns an error.

Parameter	
▷ T1 (input_control)	number(-array) $\rightsquigarrow$ integer / real / string Input tuple 1.
▷ T2 (input_control)	number(-array) $\rightsquigarrow$ integer / real / string Input tuple 2.
▷ Lesseq (output_control)	integer $\rightsquigarrow$ integer Result of the comparison of the input tuples.

Parallelization Information

tuple_less_equal is *reentrant* and processed *without* parallelization.

Alternatives

tuple_less, **tuple_greater**, **tuple_greater_equal**, **tuple_equal**, **tuple_not_equal**

Module

Operators not requiring licensing

tuple_not_equal (: : T1, T2 : Nequal)

Test, whether two tuples are not equal.

tuple_not_equal tests whether the two input tuples **T1** and **T2** are not equal. Two tuples are not equal, if they consist of a different number of elements or if they differ when compared elementwise. Two tuple elements differ, if they are of types that may not be compared (e.g., one string and one integer) or if they differ in their values.

Parameter	
▷ T1 (input_control)	number(-array) $\rightsquigarrow$ integer / real / string Input tuple 1.
▷ T2 (input_control)	number(-array) $\rightsquigarrow$ integer / real / string Input tuple 2.
▷ Nequal (output_control)	integer $\rightsquigarrow$ integer Result of the comparison of the input tuples.

Parallelization Information

tuple_not_equal is *reentrant* and processed *without* parallelization.

Alternatives

tuple_equal, **tuple_less**, **tuple_greater**, **tuple_less_equal**, **tuple_greater_equal**

Module

Operators not requiring licensing

16.4 Conversion

tuple_chr (: : T : Chr)

Convert a tuple of integers into strings with the corresponding ASCII codes.

tuple_chr converts the input tuple **T**, consisting of integer numbers, into a tuple of strings of length 1, the characters of which have the ASCII code of the corresponding input number.

Parameter
▷ T (input_control) integer(-array) $\leadsto$ integer Input tuple. Restriction : $(0 \leq T) \leq 255$ ▷ Chr (output_control) string(-array) $\leadsto$ string Strings corresponding to the ASCII code of the input tuple.
Parallelization Information
<code>tuple_chr</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
<code>tuple_chrt</code>
See also
<code>tuple_ord</code> , <code>tuple_ords</code>
Module
Operators not requiring licensing

tuple_chrt (: : T : Chrt)

Convert a tuple of integers into strings with the corresponding ASCII codes.

`tuple_chrt` converts the input tuple **T**, consisting of integer numbers, into a tuple of strings and integer numbers (where only the number 0 can occur in the output), the characters of which have the ASCII code of the corresponding input number. The operator tries to pack as many of the input numbers into one string as possible. Only if the value 0 occurs in **T** the current string is terminated at this point, the integer number 0 is inserted into the output, and a new string with the remaining input values is started. This operator can be used to convert data read with `read_serial` into strings. With this mechanism it is possible to read bytes with the value 0.

Parameter
▷ T (input_control) integer(-array) $\leadsto$ integer Input tuple. Restriction : $(0 \leq T) \leq 255$ ▷ Chrt (output_control) string(-array) $\leadsto$ string / integer Strings corresponding to the ASCII code of the input tuple.
Example
<pre>read_serial (SerialHandle, 100, Data) tuple_chrt (Data, Strings)</pre>
Parallelization Information
<code>tuple_chrt</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
<code>tuple_chr</code>
See also
<code>tuple_ord</code> , <code>tuple_ords</code> , <code>read_serial</code>
Module
Operators not requiring licensing

tuple_deg (: : Rad : Deg)

Convert a tuple from radians to degrees.

`tuple_deg` converts the input tuple **Rad** from radians to degrees. The result is always returned as a floating point number. The conversion of a string is not allowed.

Parameter

- ▷ **Rad** (input_control) number(-array) $\leadsto$ *real* / integer
Input tuple.
- ▷ **Deg** (output_control) number(-array) $\leadsto$ *real*
Input tuple in degrees.

Parallelization Information

`tuple_deg` is *reentrant* and processed *without* parallelization.

See also

`tuple_rad`

Module

Operators not requiring licensing

tuple_is_number (: : T : IsNumber)

Check a tuple (of strings) whether it represents numbers.

`tuple_is_number` checks each element of the input tuple **T** whether it represents a number. If the element is a number (real or integer), 1 is returned for that element. If the element is a string, it is checked whether the string represents a number. If so, 1 is returned, otherwise 0.

Parameter

- ▷ **T** (input_control) number(-array) $\leadsto$ *real* / integer / string
Input tuple.
- ▷ **IsNumber** (output_control) integer(-array) $\leadsto$ *integer*
Tuple with boolean numbers.

Parallelization Information

`tuple_is_number` is *reentrant* and processed *without* parallelization.

See also

`tuple_number`

Module

Operators not requiring licensing

tuple_number (: : T : Number)

Convert a tuple (of strings) into a tuple of numbers.

`tuple_number` converts the input tuple **T** into a tuple of numbers. If the input tuple contains numbers, they are simply copied into the output tuple. Strings are converted into the appropriate type of number (integer or floating point numbers) or are copied as strings if they do not represent a number. Note that strings starting with 0x are interpreted as hexadecimal numbers, and strings starting with 0 as octal numbers. For example, the string '20' is converted to the integer 20, '020' to 16, and '0x20' to 32.

Parameter

- ▷ **T** (input_control) number(-array) $\leadsto$ *real* / integer / string
Input tuple.
- ▷ **Number** (output_control) number(-array) $\leadsto$ *real* / integer / string
Input tuple as numbers.

Parallelization Information

`tuple_number` is *reentrant* and processed *without* parallelization.

See also

`tuple_is_number`, `tuple_string`

Module

Operators not requiring licensing

tuple_ord (: : T : Ord)

Convert a tuple of strings of length 1 into a tuple of their ASCII codes.

tuple_ord converts the input tuple **T**, which may only contain strings of length 1, into a tuple of integer numbers that represent the ASCII code of the characters of the strings.

Parameter

- ▷ **T** (input_control) string(-array) $\leadsto$ *string*
Input tuple.
- ▷ **ord** (output_control) integer(-array) $\leadsto$ *integer*
ASCII code of the input tuple.

Parallelization Information

tuple_ord is *reentrant* and processed *without* parallelization.

Alternatives

tuple_ords

See also

tuple_chr, **tuple_chrt**

Module

Operators not requiring licensing

tuple_ords (: : T : Ords)

Convert a tuple of strings into a tuple of their ASCII codes.

tuple_ords converts the input tuple **T**, which may only contain strings and integer numbers, into a tuple of integer numbers that represent the ASCII code of the characters of the strings. The characters of the individual strings are output according to their order within the string and within the tuple. Integer numbers are simply copied to an appropriate position in the output string. This operator can be used to prepare outputs with **write_serial**. In particular, a byte with value 0 can be written by inserting the integer number 0 into **T**.

Parameter

- ▷ **T** (input_control) string(-array) $\leadsto$ *string* / *integer*
Input tuple.
- ▷ **ords** (output_control) integer(-array) $\leadsto$ *integer*
ASCII code of the input tuple.

Example

```
tuple_ords (["String 1", 0, "String 2", 0], Data)
write_serial (SerialHandle, Data)
```

Parallelization Information

tuple_ords is *reentrant* and processed *without* parallelization.

Alternatives

tuple_ord

See also

tuple_chr, **tuple_chrt**, **write_serial**

Module

Operators not requiring licensing

tuple_real (: : T : Real)

Convert a tuple into a tuple of floating point numbers.

tuple_real converts the input tuple **T** into a tuple of floating point numbers. The conversion of a string is not allowed.

Parameter	
▷ T (input_control)	number(-array) $\leadsto$ <i>real</i> / integer Input tuple.
▷ Real (output_control)	number(-array) $\leadsto$ <i>real</i> Input tuple as floating point numbers.

tuple_real is *reentrant* and processed *without* parallelization.

Module

Operators not requiring licensing

tuple_round (: : T : Round)

Convert a tuple into a tuple of integer numbers.

tuple_round converts the input tuple **T** into a tuple of integer numbers by rounding **T** to the nearest integer. The conversion of a string is not allowed.

Parameter	
▷ T (input_control)	number(-array) $\leadsto$ <i>real</i> / integer Input tuple.
▷ Round (output_control)	number(-array) $\leadsto$ <i>integer</i> Input tuple as integer numbers.

tuple_round is *reentrant* and processed *without* parallelization.

Module

Operators not requiring licensing

tuple_string (: : T, Format : String)

Convert a tuple into a tuple of strings.

tuple_string converts numbers to strings or modifies strings. The operator has two parameters: **T** is the number or string that has to be converted. **Format** specifies the conversion. This format string consists of the following four parts

<flags><field width><precision><conversion characters>

So a conversion might look like

```
tuple\_string(1332.4554, '.6e', String)
```

flags Zero or more flags, in any order, which modify the meaning of the conversion specification. Flags may consist of the following characters:

- The result of the conversion is left justified within the field.
- + The result of a signed conversion always begins with a sign, + or -.
- <space> If the first character of a signed conversion is not a sign, a space character is prefixed to the result. This means that if the space flag and + flag both appear, the space flag is ignored.
- # The value is to be converted to an “alternate form”. For d and s conversions, this flag has no effect. For o conversion (see below), it increases the precision to force the first digit of the result to be a zero. For x or X conversion (see below), a non- zero result has 0x or 0X prefixed to it. For e, E, f, g, and G conversions, the result always contains a radix character, even if no digits follow the radix character. For g and G conversions, trailing zeros are not removed from the result, contrary to usual behavior.

field width An optional string of decimal digits to specify a minimum field width. For an output field, if the converted value has fewer characters than the field width, it is padded on the left (or right, if the left-adjustment flag, - has been given) to the field width.

precision The precision specifies the minimum number of digits to appear for the **d**, **o**, **x**, or **X** conversions (the field is padded with leading zeros), the number of digits to appear after the radix character for the **e** and **f** conversions, the maximum number of significant digits for the **g** conversion, or the maximum number of characters to be printed from a string in **S** conversion. The precision takes the form of a period . followed by a decimal digit string. A null digit string is treated as a zero.

conversion characters A conversion character indicates the type of conversion to be applied:

- d, o, x, X** The integer argument is printed in signed decimal (**d**), unsigned octal (**o**), or unsigned hexadecimal notation (**x** and **X**). The **x** conversion uses the numbers and letters **0123456789abcdef**, and the **X** conversion uses the numbers and letters **0123456789ABCDEF**. The precision component of the argument specifies the minimum number of digits to appear. If the value being converted can be represented in fewer digits than the specified minimum, it is expanded with leading zeroes. The default precision is 1. The result of converting a zero value with a precision of 0 is no characters.
- f** The floating-point number argument is printed in decimal notation in the style **[-]dddrrddd**, where the number of digits after the radix character, **r**, is equal to the precision specification. If the precision is omitted from the argument, six digits are output; if the precision is explicitly 0, no radix appears.
- e, E** The floating-point-number argument is printed in the style **[-]drddde±dd**, where there is one digit before the radix character, and the number of digits after it is equal to the precision. When the precision is missing, six digits are produced; if the precision is 0, no radix character appears. The **E** conversion character produces a number with **E** introducing the exponent instead of **e**. The exponent always contains at least two digits. However, if the value to be printed requires an exponent greater than two digits, additional exponent digits are printed as necessary.
- g, G** The floating-point-number argument is printed in style **f** or **e** (or in style **E** in the case of a **G** conversion character), with the precision specifying the number of significant digits. The style used depends on the value converted; style **e** is used only if the exponent resulting from the conversion is less than -4 or greater than or equal to the precision. Trailing zeros are removed from the result. A radix character appears only if it is followed by a digit.
- s** The argument is taken to be a string, and characters from the string are printed until the end of the string or the number of characters indicated by the precision specification of the argument is reached. If the precision is omitted from the argument, it is interpreted as infinite and all characters up to the end of the string are printed.
- b** Similar to the **S** conversion specifier, except that the string can contain backslash-escape sequences which are then converted to the characters they represent.

In no case does a nonexistent or insufficient field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is simply expanded to contain the conversion result.

Parameter

- ▷ **T** (input_control) number(-array) $\leadsto$ *real* / integer / string
Input tuple.
- ▷ **Format** (input_control) string $\leadsto$ *string*
Format string.
- ▷ **String** (output_control) string(-array) $\leadsto$ *string*
Input tuple converted to strings.

Parallelization Information

tuple_string is *reentrant* and processed *without* parallelization.

Alternatives

tuple_sub

Module

Operators not requiring licensing

16.5 Creation

tuple_concat (: : T1, T2 : Concat)

Concatenate two tuple to a new one.

tuple_concat concatenates the input tuples **T1** and **T2** to a new tuple **Concat**. The first elements of **Concat** conform to the elements of **T1** and the remaining elements of **Concat** conform to those of **T2**

Parameter

- ▷ **T1** (input_control) number(-array) $\leadsto$ integer / real / string
Input tuple 1.
- ▷ **T2** (input_control) number(-array) $\leadsto$ integer / real / string
Input tuple 2.
- ▷ **Concat** (output_control) number(-array) $\leadsto$ integer / real / string
Concatenation of input tuples.

Parallelization Information

tuple_concat is *reentrant* and processed *without* parallelization.

Alternatives

tuple_gen_const, **tuple_str_bit_select**, **tuple_select**, **tuple_str_first_n**,
tuple_str_last_n

Module

Operators not requiring licensing

tuple_gen_const (: : Length, Const : Newtuple)

Generate a tuple of a specific length and initialize its elements.

tuple_gen_const generates a new tuple in **Newtuple**. The input parameter **Length** determines the number of elements for the new tuple. Thus **Length** may only consist of a single number. If **Length** contains a floating point number, this may only represent an integer value (without fraction). The data type and value of each element of the new generated tuple is determined by the input parameter **Const** that may only consist of a single element. All elements in **Newtuple** have got the same data type and value as the single element in **Const**.

Parameter

- ▷ **Length** (input_control) number $\leadsto$ integer / real
Length of tuple to generate.
- ▷ **Const** (input_control) number $\leadsto$ integer / real / string
Constant for initializing the tuple elements.
- ▷ **Newtuple** (output_control) number(-array) $\leadsto$ integer / real / string
New Tuple.

Parallelization Information

tuple_gen_const is *reentrant* and processed *without* parallelization.

Alternatives

tuple_str_bit_select, **tuple_select**, **tuple_str_first_n**, **tuple_str_last_n**,
tuple_concat

Module

Operators not requiring licensing

16.6 Element-Order

tuple_inverse (: : Tuple : Inverted)

Invert a tuple.

`tuple_inverse` inverts the input tuple `Tuple`. Thus `Inverted` contains the same elements as `Tuple` but with the reverse order.

Parameter
▷ Tuple (input_control) number(-array) $\leadsto$ integer / real / string Input tuple. ▷ Inverted (output_control) number(-array) $\leadsto$ integer / real / string Inverted input tuple.
Parallelization Information
<code>tuple_inverse</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
<code>tuple_sort</code> , <code>tuple_sort_index</code>
Module
Operators not requiring licensing

tuple_sort (: : Tuple : Sorted)

Sorts the elements of a tuple in ascending order.

`tuple_sort` sorts all elements of `Tuple` in ascending order and returns the result with `Sorted`. As a precondition the single elements of `Tuple` must be comparable. Thus `Tuple` must either exclusively consist of strings or it must only contain (integer or floating point) numbers. In the latter case integers and floating point numbers may be mixed.

Parameter
▷ Tuple (input_control) number(-array) $\leadsto$ integer / real / string Input tuple. ▷ Sorted (output_control) number(-array) $\leadsto$ integer / real / string Sorted tuple.
Parallelization Information
<code>tuple_sort</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
<code>tuple_sort_index</code> , <code>tuple_inverse</code>
Module
Operators not requiring licensing

tuple_sort_index (: : Tuple : Indices)

Sorts the elements of a tuple and returns the indices of the sorted tuple.

`tuple_sort_index` sorts all elements of `Tuple` in ascending order and returns the indices of the elements of the sorted tuple (in relation to the input tuple `Tuple`) with `Indices`. As a precondition the single elements of `Tuple` must be comparable. Thus `Tuple` must either exclusively consist of strings or it must only contain (integer or floating point) numbers. In the latter case integers and floating point numbers may be mixed.

Parameter
▷ Tuple (input_control) number(-array) $\leadsto$ integer / real / string Input tuple. ▷ Indices (output_control) number(-array) $\leadsto$ integer Sorted tuple.
Parallelization Information
<code>tuple_sort_index</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
<code>tuple_sort</code> , <code>tuple_inverse</code>

Module

Operators not requiring licensing

16.7 Features

tuple_deviation (: : Tuple : Deviation)

Return the standard deviation of the elements of a tuple.

tuple_deviation calculates the standard derivation of all elements of the input tuple **Tuple**. It returns the derivation as a floating point number in the output parameter **Deviation**. The input tuple may only consist of numbers (integer or floating point numbers).

Parameter

- ▷ **Tuple** (input_control) number(-array) $\leadsto$ integer / real / string
Input tuple.
- ▷ **Deviation** (output_control) number(-array) $\leadsto$ real
Standard deviation of tuple elements.

Parallelization Information

tuple_deviation is *reentrant* and processed *without* parallelization.

Alternatives

tuple_mean, **tuple_sum**, **tuple_min**, **tuple_max**, **tuple_length**

Module

Operators not requiring licensing

tuple_length (: : Tuple : Length)

Returns the number of elements of a tuple.

tuple_length returns the number of elements of the input tuple **Tuple**.

Parameter

- ▷ **Tuple** (input_control) number(-array) $\leadsto$ integer / real / string
Input tuple.
- ▷ **Length** (output_control) number(-array) $\leadsto$ real
Number of elements of input tuple.

Parallelization Information

tuple_length is *reentrant* and processed *without* parallelization.

Alternatives

tuple_min, **tuple_max**, **tuple_mean**, **tuple_deviation**, **tuple_sum**

Module

Operators not requiring licensing

tuple_max (: : Tuple : Max)

Returns the maximal element of a tuple.

tuple_max returns the maximal element of all elements of the input tuple **Tuple**. All elements of **Tuple** either have to be strings or numbers (integer or floating point numbers). It is not allowed to mix strings with numerical values. The result parameter **Max** will contain a floating point number, if at least one element of **Tuple** is a floating point number. If all elements of **Tuple** are integer numbers the resulting sum will also be an integer number.

Parameter
▷ Tuple (input_control) number(-array) $\leadsto$ integer / real / string Input tuple. ▷ Max (output_control) number(-array) $\leadsto$ real Maximal element of the input tuple elements.
Parallelization Information
<code>tuple_max</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
<code>tuple_min</code> , <code>tuple_mean</code> , <code>tuple_deviation</code> , <code>tuple_sum</code> , <code>tuple_length</code>
Module
Operators not requiring licensing

tuple_mean (: : Tuple : Mean)

Return the mean value of a tuple of numbers.

`tuple_mean` returns the mean value of all elements of the input tuple `Tuple` as a floating point number in the output parameter `Mean`. The input tuple may only consist of numbers (integer or floating point numbers).

Parameter
▷ Tuple (input_control) number(-array) $\leadsto$ integer / real / string Input tuple. ▷ Mean (output_control) number(-array) $\leadsto$ real Mean value of tuple elements.
Parallelization Information
<code>tuple_mean</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
<code>tuple_deviation</code> , <code>tuple_sum</code> , <code>tuple_min</code> , <code>tuple_max</code> , <code>tuple_length</code>
Module
Operators not requiring licensing

tuple_min (: : Tuple : Min)

Returns the minimal element of a tuple.

`tuple_min` returns the minimal element of all elements of the input tuple `Tuple`. All elements of `Tuple` either have to be strings or numbers (integer or floating point numbers). It is not allowed to mix strings with numerical values. The result parameter `Min` will contain a floating point number, if at least one element of `Tuple` is a floating point number. If all elements of `Tuple` are integer numbers the resulting sum will also be an integer number.

Parameter
▷ Tuple (input_control) number(-array) $\leadsto$ integer / real / string Input tuple. ▷ Min (output_control) number(-array) $\leadsto$ real Minimal element of the input tuple elements.
Parallelization Information
<code>tuple_min</code> is <i>reentrant</i> and processed <i>without</i> parallelization.
Alternatives
<code>tuple_max</code> , <code>tuple_mean</code> , <code>tuple_deviation</code> , <code>tuple_sum</code> , <code>tuple_length</code>
Module
Operators not requiring licensing

tuple_sum (: : Tuple : Sum)

Return the sum of all elements of a tuple.

tuple_sum returns the sum of all elements of the input tuple **Tuple**. All elements of **Tuple** either have to be strings or numbers (integer or floating point numbers). It is not allowed to mix strings with numerical values. The result parameter **Sum** will contain a floating point number, if at least one element of **Tuple** is a floating point number. If all elements of **Tuple** are integer numbers the resulting sum will also be an integer number. If **Tuple** contains strings, the concatenation will be used for building the sum.

Parameter

- ▷ **Tuple** (input_control) number(-array) $\leadsto$ integer / real / string
Input tuple.
- ▷ **Sum** (output_control) number(-array) $\leadsto$ real
Sum of tuple elements.

Parallelization Information

tuple_sum is *reentrant* and processed *without* parallelization.

Alternatives

tuple_mean, **tuple_deviation**, **tuple_min**, **tuple_max**, **tuple_length**

Module

Operators not requiring licensing

16.8 Logical-Operations

tuple_and (: : T1, T2 : And)

Compute the logical and of two tuples.

tuple_and computes the logical and of the input tuples **T1** and **T2**. If both tuples have the same length the operation is performed on the corresponding elements of both tuples. Otherwise, either **T1** or **T2** must have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The input tuples must contain only integer numbers.

Parameter

- ▷ **T1** (input_control) integer(-array) $\leadsto$ integer
Input tuple 1.
- ▷ **T2** (input_control) integer(-array) $\leadsto$ integer
Input tuple 2.
- ▷ **And** (output_control) integer(-array) $\leadsto$ integer
Logical and of the input tuples.

Parallelization Information

tuple_and is *reentrant* and processed *without* parallelization.

Alternatives

tuple_or, **tuple_xor**, **tuple_not**

See also

tuple_band, **tuple_bor**, **tuple_bxor**, **tuple_bnot**

Module

Operators not requiring licensing

tuple_not (: : T : Not)

Compute the logical not of two tuples.

tuple_not computes the logical not of the input tuple **T**. The input tuple must contain only integer numbers.

Parameter	
▷ T (input_control)	integer(-array) $\leadsto$ integer Input tuple.
▷ Not (output_control)	integer(-array) $\leadsto$ integer Binary not of the input tuple.
Parallelization Information	
tuple_not is <i>reentrant</i> , <i>local</i> , and processed <i>without</i> parallelization.	
Alternatives	
tuple_and , tuple_or , tuple_xor	
See also	
tuple_band , tuple_bor , tuple_bxor , tuple_bnot	
Module	
Operators not requiring licensing	

tuple_or (: : T1, T2 : Or)

Compute the logical or of two tuples.

tuple_or computes the logical or of the input tuples **T1** and **T2**. If both tuples have the same length the operation is performed on the corresponding elements of both tuples. Otherwise, either **T1** or **T2** must have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The input tuples must contain only integer numbers.

Parameter	
▷ T1 (input_control)	integer(-array) $\leadsto$ integer Input tuple 1.
▷ T2 (input_control)	integer(-array) $\leadsto$ integer Input tuple 2.
▷ or (output_control)	integer(-array) $\leadsto$ integer Logical or of the input tuples.
Parallelization Information	
tuple_or is <i>reentrant</i> and processed <i>without</i> parallelization.	
Alternatives	
tuple_and , tuple_xor , tuple_not	
See also	
tuple_band , tuple_bor , tuple_bxor , tuple_bnot	
Module	
Operators not requiring licensing	

tuple_xor (: : T1, T2 : Xor)

Compute the logical exclusive or of two tuples.

tuple_xor computes the logical exclusive or of the input tuples **T1** and **T2**. If both tuples have the same length the operation is performed on the corresponding elements of both tuples. Otherwise, either **T1** or **T2** must have length 1. In this case, the operation is performed for each element of the longer tuple with the single element of the other tuple. The input tuples must contain only integer numbers.

Parameter	
▷ T1 (input_control)	integer(-array) $\leadsto$ integer Input tuple 1.
▷ T2 (input_control)	integer(-array) $\leadsto$ integer Input tuple 2.

- ▷ **Xor** (output_control) integer(-array) $\leadsto$ integer
Binary exclusive or of the input tuples.

Parallelization Information

tuple_xor is *reentrant* and processed *without* parallelization.

Alternatives

tuple_and, **tuple_or**, **tuple_not**

See also

tuple_band, **tuple_bor**, **tuple_bxor**, **tuple_bnot**

Module

Operators not requiring licensing

16.9 Selection

tuple_first_n (: : Tuple, Index : Selected)

Select the first elements of a tuple.

tuple_first_n selects the first elements of **Tuple** and returns them with **Selected**. Thus **Selected** contains all elements of **Tuple** from the first element up to the “n-th” element of **Tuple** (including the “n-th” element). The index “n” is determined by the input parameter **Index**. Thus **Index** must contain a single integer value (if **Index** consists of a floating point number, this must represent an integer value without fraction). Indices of tuple elements start at 0, that means, the first tuple element has got the index 0.

Parameter

- ▷ **Tuple** (input_control) number(-array) $\leadsto$ integer / real / string
Input tuple.
- ▷ **Index** (input_control) number $\leadsto$ integer / real
Index of the first element to select.
- ▷ **Selected** (output_control) number(-array) $\leadsto$ integer / real / string
Selected tuple elements.

Parallelization Information

tuple_first_n is *reentrant* and processed *without* parallelization.

Alternatives

tuple_last_n, **tuple_select**, **tuple_last_n**, **tuple_str_bit_select**, **tuple_concat**

Module

Operators not requiring licensing

tuple_last_n (: : Tuple, Index : Selected)

Select all elements from index “n” to the end of a tuple.

Starting with the “n-th” element of the tuple **Tuple**, **tuple_last_n** selects every element of **Tuple** and returns it with **Selected**. Thus **Selected** contains all elements of **Tuple** from index “n” up to the last element of **Tuple** (including the element at position “n”). The index “n” is determined by the input parameter **Index**. Thus **Index** must contain a single integer value (if **Index** consists of a floating point number, this must represent an integer value without fraction). Indices of tuple elements start at 0, that means, the first tuple element has got the index 0.

Parameter

- ▷ **Tuple** (input_control) number(-array) $\leadsto$ integer / real / string
Input tuple.
- ▷ **Index** (input_control) number $\leadsto$ integer / real
Index of the first element to select.

- ▷ **Selected** (output_control) number(-array) $\leadsto$ integer / real / string
Selected tuple elements.

Parallelization Information

`tuple_last.n` is *reentrant* and processed *without* parallelization.

Alternatives

`tuple_first.n`, `tuple_select`, `tuple_str_bit_select`, `tuple_concat`

Module

Operators not requiring licensing

tuple_select (: : Tuple, Index : Selected)

Select single elements of a tuple.

`tuple_select` selects one or more single elements of the tuple `Tuple` and returns them with `Selected`. At this, `Index` determines the indices of the elements to select. Thus `Index` may only contain integer values (any floating point number within `Index` must represent an integer value without fraction). Indices of tuple elements start at 0, that means, the first tuple element has got the index 0.

Parameter

- ▷ **Tuple** (input_control) number(-array) $\leadsto$ integer / real / string
Input tuple.
- ▷ **Index** (input_control) number(-array) $\leadsto$ integer / real
Indices of the elements to select.
- ▷ **Selected** (output_control) number(-array) $\leadsto$ integer / real / string
Selected tuple element.

Parallelization Information

`tuple_select` is *reentrant* and processed *without* parallelization.

Alternatives

`tuple_first.n`, `tuple_last.n`, `tuple_str_bit_select`, `tuple_concat`

Module

Operators not requiring licensing

tuple_select_range (: : Tuple, Leftindex, Rightindex : Selected)

Select several elements of a tuple.

`tuple_select_range` selects several consecutive elements of the input tuple `Tuple` and returns them with `Selected`. At this, `Leftindex` determines the index of the first element and `Rightindex` determines the index of the last element to select. Thus both parameters `Leftindex` and `Rightindex` must contain a single integer value (any floating point number must represent an integer value without fraction). Indices of tuple elements start at 0, that means, the first tuple element has got the index 0. The result tuple `Selected` contains every element from the tuple `Tuple` that has got an index between `Leftindex` and `Rightindex` (including the elements at position `Leftindex` and `Rightindex`). The index `Rightindex` must be greater or equal to `Leftindex`. If the parameters contain equal values, only one element is selected.

Parameter

- ▷ **Tuple** (input_control) number(-array) $\leadsto$ integer / real / string
Input tuple.
- ▷ **Leftindex** (input_control) number(-array) $\leadsto$ integer / real
Index of first element to select.
- ▷ **Rightindex** (input_control) number(-array) $\leadsto$ integer / real
Index of last element to select.
- ▷ **Selected** (output_control) number(-array) $\leadsto$ integer / real / string
Selected tuple elements.

Parallelization Information

`tuple_select_range` is *reentrant* and processed *without* parallelization.

Alternatives

`tuple_select`, `tuple_first_n`, `tuple_last_n`, `tuple_str_bit_select`, `tuple_concat`

Module

Operators not requiring licensing

<code>tuple_str_bit_select</code> (: : Tuple, Index : Selected)

Select single character or bit from a tuple.

`tuple_str_bit_select` selects a single character or bit from a tuple **Tuple** of integer numbers and/or strings. The input parameter **Index** determines the character or bit position to select. **Index** must contain a single number. If **Index** contains a floating point number, this may only represent an integer value (without fraction). The result tuple **Selected** contains a new element for each element of **Tuple**. Let **Index** contain the number “n” then each element of **Selected** consists of the “n-th” character (for strings) or “n-th” bit (for integers) of the corresponding element of **Tuple**.

Parameter

- ▷ **Tuple** (input_control) number(-array) $\leadsto$ *string* / integer
Input tuple.
- ▷ **Index** (input_control) number $\leadsto$ *integer* / real
Position of character or bit to select.
- ▷ **Selected** (output_control) number(-array) $\leadsto$ *string* / integer
Tuple containing the selected characters and bits.

Parallelization Information

`tuple_str_bit_select` is *reentrant* and processed *without* parallelization.

Alternatives

`tuple_str_bit_select`, `tuple_select`, `tuple_first_n`, `tuple_last_n`, `tuple_concat`,
`tuple_strchr`, `tuple_strrchr`, `tuple_str_first_n`, `tuple_str_last_n`, `tuple_and`,
`tuple_or`, `tuple_xor`, `tuple_not`

Module

Operators not requiring licensing

16.10 String-Operators

<code>tuple_environment</code> (: : Names : Values)

Read one or more environment variables.

`tuple_environment` reads the content of all environment variables that are referenced by their names in the input tuple **Names** and returns the content with the output tuple **Values**. The input tuple may only contain strings. An empty string is returned for every name within **Names** that does not denote a valid environment variable.

Parameter

- ▷ **Names** (input_control) string(-array) $\leadsto$ *string*
Tuple containing name(s) of the environment variable(s).
- ▷ **Values** (output_control) string(-array) $\leadsto$ *string*
Content of the environment variable(s).

Parallelization Information

`tuple_environment` is *reentrant* and processed *without* parallelization.

Alternatives

`tuple_strstr`, `tuple_strrstr`, `tuple_strchr`, `tuple_strrchr`, `tuple_strlen`,
`tuple_str_first_n`, `tuple_str_last_n`, `tuple_split`

Module

Operators not requiring licensing

tuple_split (: : T1, T2 : Splitted)

Split strings into substrings between predefined separator symbol(s).

tuple_split searches within the strings of the input tuple **T1** for one or more separators defined in the input tuple **T2**. **tuple_split** then splits the examined strings into the substrings between the separators. Both input tuples may only consist of strings. Otherwise **tuple_split** returns an error. If the elements of **T2** contain more than one character, each character defines a separator. If **T1** consists only of one string, this is split up several times according to the elements of **T2**. For example: If **T1** consists of the string “data1;data2:7;data3” and **T2** contains the strings “;” and “:;”, the output tuple **Splitted** will comprise the strings “data1”, “data2:7”, “data3” as the result of splitting the string of **T1** according to the first element of **T2** and “data1”, “data2”, “7” and “data3” as the result of splitting according to the second element of **T2**. If both input tuples show the same number of elements, the search is done elementwise. I.e., **tuple_split** will split the first string of **T1** according to the separators in the first element of **T2**, the second string of **T1** according to the separators in the second element of **T2** and so on. If **T2** only contains one string, the separators defined in this string will be used to split up all the strings of **T1**. If both input tuples contain more than one element and the number of elements differs for the input tuples, **tuple_split** returns an error.

Parameter

- ▷ **T1** (input_control) string(-array) $\leadsto$ *string*
Input tuple 1.
- ▷ **T2** (input_control) string(-array) $\leadsto$ *string*
Input tuple 2.
- ▷ **Splitted** (output_control) string(-array) $\leadsto$ *string*
Substrings after splitting the input strings.

Parallelization Information

tuple_split is *reentrant* and processed *without* parallelization.

Alternatives

tuple_strstr, **tuple_strrstr**, **tuple_strchr**, **tuple_strrchr**, **tuple_strlen**,
tuple_str_first_n, **tuple_str_last_n**, **tuple_environment**

Module

Operators not requiring licensing

tuple_str_first_n (: : T1, T2 : Substring)

Cut the first characters up to position “n” out of string tuple.

tuple_str_first_n cuts the first characters up to position “n” out of each string of the input tuple **T1** and returns them as new strings in the output tuple **Substring** (remark: the position within strings starts with 0 for the first character of a string). The number “n” is determined by the second input tuple **T2**. If **T2** only contains one element, this element contains the number “n”. If **T1** and **T2** have got the same number of elements, the first element of **T2** determines the number “n” of characters to cut out of the first string of **T1**, the second element of **T2** does so for the second string of **T1** and so on. If **T2** contains more than one element and **T1** contains only one string, **tuple_str_first_n** cuts more than one substrings out of this string. The elements of **T2** then determine the lengths of these substrings. If both input tuples contain more than one element but differ in the number of elements, **tuple_str_first_n** returns an error.

Parameter

- ▷ **T1** (input_control) string(-array) $\leadsto$ *string*
Input tuple 1.
- ▷ **T2** (input_control) number(-array) $\leadsto$ *integer / real*
Input tuple 2.

- ▷ **Substring** (output_control) string(-array) $\leadsto$ string
The first characters up to position “n” of each string.

Parallelization Information

`tuple_str_first_n` is *reentrant* and processed *without* parallelization.

Alternatives

`tuple_str_last_n`, `tuple_strstr`, `tuple_strstr`, `tuple_strlen`, `tuple_strchr`,
`tuple_strchr`, `tuple_split`, `tuple_environment`

Module

Operators not requiring licensing

tuple_str_last_n (: : T1, T2 : Substring)

Cut all characters starting at position “n” out of string tuple.

`tuple_str_last_n` cuts all characters from position “n” to the end of the string out of each string of the input tuple **T1** and returns them as new strings in the output tuple **Substring**. The position “n” is determined by the second input tuple **T2**. If **T2** only contains one element, this element contains “n”. If **T1** and **T2** have got the same number of elements, the first element of **T2** determines the start position for the first string of **T1**, the second element of **T2** does so for the second string of **T1** and so on. If **T2** contains more than one element and **T1** contains only one string, `tuple_str_last_n` cuts more than one substrings out of this string. The elements of **T2** then determine the start positions for these substrings. If both input tuples contain more than one element but differ in the number of elements, `tuple_str_last_n` returns an error.

Parameter

- ▷ **T1** (input_control) string(-array) $\leadsto$ string
Input tuple 1.
- ▷ **T2** (input_control) number(-array) $\leadsto$ integer / real
Input tuple 2.
- ▷ **Substring** (output_control) string(-array) $\leadsto$ string
The last characters starting at position “n”.

Parallelization Information

`tuple_str_last_n` is *reentrant* and processed *without* parallelization.

Alternatives

`tuple_str_last_n`, `tuple_strstr`, `tuple_strstr`, `tuple_strlen`, `tuple_strchr`,
`tuple_strchr`, `tuple_split`, `tuple_environment`

Module

Operators not requiring licensing

tuple_strchr (: : T1, T2 : Position)

Forward search for a character within a string tuple.

`tuple_strchr` searches within the strings of the input tuple **T1** for the characters of the input tuple **T2**. Both input tuples may only consist of strings. Otherwise `tuple_strchr` returns an error. If the elements of **T2** contain more than one character, only the first character of each element is considered for searching. If **T1** contains only one string, all the characters defined in **T2** are searched in this string. Thus the output tuple consists of as many elements as **T2**. Whenever a searched character has been found, the position of its first occurrence gets stored in the output tuple **Position** (remark: the position starts at 0 for the first character of a string). If a character can not be found, -1 will be returned instead of its position. If both input tuples show the same number of elements, the search is done elementwise. I.e., the first character of the first element of **T2** is searched within the first string of **T1**, the first character of the second element of **T2** is searched within the second string of **T1** and so on. The results of the elementwise searches are returned with **Position** that contains as many elements as **T1** and **T2**. If **T2** only contains one string, its first character is searched within all strings of **T1**. Thus in this case **Position** consists of as many elements as **T1**. If both input tuples contain more than one element and the number of elements differs for the input tuples, `tuple_strchr` returns an error.

Parameter	
▷ T1 (input_control)	string(-array) $\leadsto$ string Input tuple 1.
▷ T2 (input_control)	string(-array) $\leadsto$ string Input tuple 2.
▷ Position (output_control)	integer(-array) $\leadsto$ integer Position of searched character(s) within the string(s).
Parallelization Information	
<code>tuple_strchr</code> is <i>reentrant</i> and processed <i>without</i> parallelization.	
Alternatives	
<code>tuple_strchr</code> , <code>tuple_strstr</code> , <code>tuple_strrstr</code> , <code>tuple_strlen</code> , <code>tuple_str_first_n</code> , <code>tuple_str_last_n</code> , <code>tuple_split</code> , <code>tuple_environment</code>	
Module	
Operators not requiring licensing	

`tuple_strlen` (: : T1 : Length)

Length of every string within a tuple of strings.

`tuple_strlen` checks the length of every string within the input tuple **T1** and returns the length of each string with the output tuple **Length**. All elements of **T1** may only consist of strings. Otherwise `tuple_strlen` returns an error.

Parameter	
▷ T1 (input_control)	string(-array) $\leadsto$ string Input tuple.
▷ Length (output_control)	integer(-array) $\leadsto$ integer Length of the single strings of the input tuple.
Parallelization Information	
<code>tuple_strlen</code> is <i>reentrant</i> and processed <i>without</i> parallelization.	
Alternatives	
<code>tuple_strstr</code> , <code>tuple_strrstr</code> , <code>tuple_strchr</code> , <code>tuple_strchr</code> , <code>tuple_str_first_n</code> , <code>tuple_str_last_n</code> , <code>tuple_split</code> , <code>tuple_environment</code>	
Module	
Operators not requiring licensing	

`tuple_strchr` (: : T1, T2 : Position)

Backward search for a character within a string tuple.

`tuple_strchr` searches within the strings of the input tuple **T1** for the characters of the input tuple **T2**. Both input tuples may only consist of strings. Otherwise `tuple_strchr` returns an error. In any case backward search is used, i.e., every string is examined from its last to its first character. If the elements of **T2** contain more than one character, only the first character of each element is considered for searching. If **T1** contains only one string, all the characters defined in **T2** are searched in this string. Thus the output tuple consists of as many elements as **T2**. Whenever a searched character has been found, the position of its first occurrence gets stored in the output tuple **Position**. If a character can not be found, -1 will be returned instead of its position (remark: the position starts at 0 for the first character of a string). If both input tuples show the same number of elements, the search is done elementwise. I.e., the first character of the first element of **T2** is searched within the first string of **T1**, the first character of the second element of **T2** is searched within the second string of **T1** and so on. The results of the elementwise searches are returned with **Position** that contains as many elements as **T1** and **T2**. If **T2** only contains one string, its first character is searched within all strings of **T1**. Thus in this case **Position** consists of as many elements as **T1**. If both input tuples contain more than one element and the number of elements differs for the input tuples, `tuple_strchr` returns an error.

Parameter
▷ T1 (input_control) string(-array) $\leadsto$ string Input tuple 1.
▷ T2 (input_control) string(-array) $\leadsto$ string Input tuple 2.
▷ Position (output_control) integer(-array) $\leadsto$ integer Position of searched character within the string.
Parallelization Information
<code>tuple_strchr</code> is reentrant and processed without parallelization.
Alternatives
<code>tuple_strchr</code> , <code>tuple_strstr</code> , <code>tuple_strstr</code> , <code>tuple_strlen</code> , <code>tuple_str_first_n</code> , <code>tuple_str_last_n</code> , <code>tuple_split</code> , <code>tuple_environment</code>
Module
Operators not requiring licensing

tuple_strrstr (: : T1, T2 : Position)

Backward search for string(s) within a string tuple.

`tuple_strrstr` searches within the strings of the input tuple **T1** for the strings of the input tuple **T2**. Both input tuples may only consist of strings. Otherwise `tuple_strrstr` returns an error. In any case backward search is used, i.e., every string is examined from its last to its first character. If **T1** contains only one string, all strings of **T2** are searched in it. Thus the output tuple consists of as many elements as **T2**. Whenever a searched string has been found, the position of its first occurrence gets stored in the output tuple **Position** (positions in strings are counted starting with 0). If a string can not be found, -1 will be returned instead of its position. If both input tuples show the same number of elements, the strings are searched elementwise. I.e., the first string of **T2** is searched within the first string of **T1**, the second string of **T2** is searched within the second string of **T1** and so on. The results of the elementwise searches are returned with **Position** that contains as many elements as **T1** and **T2**. If **T2** only contains one string, this is searched within all strings of **T1**. Thus in this case **Position** consists of as many elements as **T1**. If both input tuples contain more than one element and the number of elements differs for the input tuples, `tuple_strrstr` returns an error.

Parameter
▷ T1 (input_control) string(-array) $\leadsto$ string Input tuple 1.
▷ T2 (input_control) string(-array) $\leadsto$ string Input tuple 2.
▷ Position (output_control) integer(-array) $\leadsto$ integer Position of searched string(s) within the examined string(s).
Parallelization Information
<code>tuple_strrstr</code> is reentrant and processed without parallelization.
Alternatives
<code>tuple_strstr</code> , <code>tuple_strlen</code> , <code>tuple_strchr</code> , <code>tuple_strrchr</code> , <code>tuple_str_first_n</code> , <code>tuple_str_last_n</code> , <code>tuple_split</code> , <code>tuple_environment</code>
Module
Operators not requiring licensing

tuple_strstr (: : T1, T2 : Position)

Forward search for string(s) within a string tuple.

`tuple_strstr` searches within the strings of the input tuple **T1** for the strings of the input tuple **T2**. Both input tuples may only consist of strings. Otherwise `tuple_strstr` returns an error. If **T1** contains only one string, all

strings of **T2** are searched in it. Thus the output tuple consists of as many elements as **T2**. Whenever a searched string has been found, the position of its first occurrence gets stored in the output tuple **Position** (positions in strings are counted starting with 0). If a string can not be found, -1 will be returned instead of its position. If both input tuples show the same number of elements, the strings are searched elementwise. I.e., the first string of **T2** is searched within the first string of **T1**, the second string of **T2** is searched within the second string of **T1** and so on. The results of the elementwise searches are returned with **Position** that contains as many elements as **T1** and **T2**. If **T2** only contains one string, this is searched within all strings of **T1**. Thus in this case **Position** consists of as many elements as **T1**. If both input tuples contain more than one element and the number of elements differs for the input tuples, **tuple_strstr** returns an error.

Parameter

- ▷ **T1** (input_control) string(-array) $\leadsto$ string
Input tuple 1.
- ▷ **T2** (input_control) string(-array) $\leadsto$ string
Input tuple 2.
- ▷ **Position** (output_control) integer(-array) $\leadsto$ integer
Position of searched string(s) within the examined string(s).

Parallelization Information

tuple_strstr is *reentrant* and processed *without* parallelization.

Alternatives

tuple_strrstr, **tuple_strlen**, **tuple_strchr**, **tuple_strrchr**, **tuple_str_first_n**,
tuple_str_last_n, **tuple_split**, **tuple_environment**

Module

Operators not requiring licensing

Chapter 17

XLD

17.1 Access

get_contour_xld (Contour : : : Row, Col)

Return the coordinates of an XLD contour.

[get_contour_xld](#) returns the following values of the XLD contour [Contour](#):

[Row](#) Row coordinate of the contour's points
[Col](#) Column coordinate of the contour's points

Parameter

- ▷ **Contour** (input_object) xld_cont $\leadsto$ *Hobject*
Input XLD contour.
- ▷ **Row** (output_control) point.y-array $\leadsto$ *real*
Row coordinate of the contour's points.
- ▷ **Col** (output_control) point.x-array $\leadsto$ *real*
Column coordinate of the contour's points.

Parallelization Information

[get_contour_xld](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_contours_skeleton_xld](#), [lines_gauss](#), [lines_facet](#), [edges_sub_pix](#)

See also

[get_contour_attrib_xld](#), [query_contour_attribs_xld](#), [get_contour_global_attrib_xld](#),
[query_contour_global_attribs_xld](#)

Module

Sub-pixel operators

get_lines_xld (Polygon : : : BeginRow, BeginCol, EndRow, EndCol,
Length, Phi)

Return an XLD polygon's data (as lines).

[get_lines_xld](#) returns the XLD polygon [Polygon](#) as a set of lines. The following values are returned:

[BeginRow](#): Rows coordinates of the lines' start points
[BeginCol](#): Columns coordinates of the lines' start points
[EndRow](#): Row coordinates of the lines' end points
[EndCol](#): Column coordinates of the lines' end points
[Length](#): Lengths of the line segments
[Phi](#): Angles of the line segments

Parameter	
▷ Polygon (input_object)	xld_poly(-array) $\leadsto$ <i>Hobject</i> Input XLD polygons.
▷ BeginRow (output_control)	line.begin.y-array $\leadsto$ <i>real</i> Row coordinates of the lines' start points.
▷ BeginCol (output_control)	line.begin.x-array $\leadsto$ <i>real</i> Column coordinates of the lines' start points.
▷ EndRow (output_control)	line.end.y-array $\leadsto$ <i>real</i> Column coordinates of the lines' end points.
▷ EndCol (output_control)	line.end.x-array $\leadsto$ <i>real</i> Column coordinates of the lines' end points.
▷ Length (output_control)	real-array $\leadsto$ <i>real</i> Lengths of the line segments.
▷ Phi (output_control)	angle.rad-array $\leadsto$ <i>real</i> Angles of the line segments.

Parallelization Information

[get_lines_xld](#) is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

[gen_polygons_xld](#)

Alternatives

[get_polygon_xld](#)

Module

Sub-pixel operators

get_parallels_xld (Parallels : : : Row1, Col1, Length1, Phi1, Row2, Col2, Length2, Phi2)

Return an XLD parallel's data (as lines).

[get_parallels_xld](#) returns the following values of the XLD parallels [Parallels](#):

Row1: Row coordinates of the points on polygon P1
Col1: Column coordinates of the points on polygon P1
Length1: Lengths of the line segments on polygon P1
Phi1: Angles of the line segments on polygon P1
Row2: Row coordinates of the points on polygon P2
Col2: Column coordinates of the points on polygon P2
Length2: Lengths of the line segments on polygon P2
Phi2: Angles of the line segments on polygon P2

Parameter	
▷ Parallels (input_object)	xld $\leadsto$ <i>Hobject</i> Input XLD parallels.
▷ Row1 (output_control)	polygon.y-array $\leadsto$ <i>real</i> Row coordinates of the points on polygon P1.
▷ Col1 (output_control)	polygon.x-array $\leadsto$ <i>real</i> Column coordinates of the points on polygon P1.
▷ Length1 (output_control)	real-array $\leadsto$ <i>real</i> Lengths of the line segments on polygon P1.
▷ Phi1 (output_control)	angle.rad-array $\leadsto$ <i>real</i> Angles of the line segments on polygon P1.
▷ Row2 (output_control)	polygon.y-array $\leadsto$ <i>integer</i> Row coordinates of the points on polygon P2.
▷ Col2 (output_control)	polygon.x-array $\leadsto$ <i>integer</i> Column coordinates of the points on polygon P2.

- ▷ **Length2** (output_control) real-array $\leadsto$ *real*
Lengths of the line segments on polygon P2.
- ▷ **Phi2** (output_control) angle.rad-array $\leadsto$ *real*
Angles of the line segments on polygon P2.

Parallelization Information

[get_parallels_xld](#) is *reentrant* and processed *without parallelization*.

Possible Predecessors

[gen_parallels_xld](#)

See also

[get_polygon_xld](#), [get_lines_xld](#)

Module

Sub-pixel operators

get_polygon_xld (Polygon : : : Row, Col, Length, Phi)

Return an XLD polygon's data.

[get_polygon_xld](#) returns the following values of the XLD polygon **Polygon**:

Row Row coordinates of the polygons' points
Col Column coordinates of the polygons' points
Length Lengths of the line segments between points i and $i + 1$, respectively
Phi Angles of the line segments between points i and $i + 1$, respectively

Parameter

- ▷ **Polygon** (input_object) xld_poly $\leadsto$ *Hobject*
Input XLD polygon.
- ▷ **Row** (output_control) point.y-array $\leadsto$ *real*
Row coordinates of the polygons' points.
- ▷ **Col** (output_control) point.x-array $\leadsto$ *real*
Column coordinates of the polygons' points.
- ▷ **Length** (output_control) real-array $\leadsto$ *real*
Lengths of the line segments.
- ▷ **Phi** (output_control) angle.rad-array $\leadsto$ *real*
Angles of the line segments.

Parallelization Information

[get_polygon_xld](#) is *reentrant* and processed *without parallelization*.

Possible Predecessors

[gen_polygons_xld](#)

Alternatives

[get_lines_xld](#)

Module

Sub-pixel operators

17.2 Creation

gen_contour_polygon_rounded_xld (: Contour : Row, Col, Radius, SamplingInterval :)

Generate a XLD contour with rounded corners from a polygon (given as tuples).

[gen_contour_polygon_rounded_xld](#) generates an XLD contour **Contour** with rounded corners from a polygon given in the tuples **Row** and **Col**. The rounded corners are created in form of arcs defined by **Radius**. For every specified point of the polygon there must be a corresponding rounding radius defined. In case of a

closed polygon the first and the last point have to be defined identical with the radii of these points being equal. In contrast, the radii of the first and last point of open polygons are ignored. Finally, the [SamplingInterval](#) parameter defines the distance of the control points of the formed contour [Contour](#).

<i>Parameter</i>	
▷ Contour (output_object)	xld_cont $\leadsto$ <i>Hobject</i> Resulting contour.
▷ Row (input_control)	number-array $\leadsto$ <i>real</i> / integer Row coordinates of the polygon. Default Value : '[20,80,80,20,20]' Suggested values : Row $\in \{0, 1, 2, 3, 4, 5, 10, 20, 50, 100, 200, 500\}$
▷ Col (input_control)	number-array $\leadsto$ <i>real</i> / integer Column coordinates of the polygon. Default Value : '[20,20,80,80,20]' Suggested values : Col $\in \{0, 1, 2, 3, 4, 5, 10, 20, 50, 100, 200, 500\}$
▷ Radius (input_control)	number-array $\leadsto$ <i>real</i> / integer Radii of the rounded corners. Default Value : '[20,20,20,20,20]' Suggested values : Radius $\in \{0, 1, 2, 5, 10, 20, 50\}$
▷ SamplingInterval (input_control)	number $\leadsto$ <i>real</i> / integer Distance of the samples. Default Value : 1.0 Suggested values : SamplingInterval $\in \{0.5, 1.0, 2.0, 5.0\}$

Parallelization Information
[gen_contour_polygon_rounded_xld](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors
[get_region_contour](#)

Possible Successors
[smooth_contours_xld](#), [gen_polygons_xld](#)

See also
[gen_contour_polygon_xld](#), [gen_contours_skeleton_xld](#)

Module
Sub-pixel operators

gen_contour_polygon_xld (: Contour : Row, Col :)

Generate an XLD contour from a polygon (given as tuples).

[gen_contour_polygon_xld](#) generates an XLD contour [Contour](#) from a polygon given in the tuples [Row](#) and [Col](#). This operator is useful if contours have been obtained from routines outside the core library, but higher level operators, e.g., polygon approximation and extraction of parallels, are to be performed on the contours.

<i>Parameter</i>	
▷ Contour (output_object)	xld_cont $\leadsto$ <i>Hobject</i> Resulting contour.
▷ Row (input_control)	number-array $\leadsto$ <i>real</i> / integer Row coordinates of the polygon. Default Value : '[0,1,2,2,2]' Suggested values : Row $\in \{0, 1, 2, 3, 4, 5, 10, 20, 50, 100, 200, 500\}$
▷ Col (input_control)	number-array $\leadsto$ <i>real</i> / integer Column coordinates of the polygon. Default Value : '[0,0,0,1,2]' Suggested values : Col $\in \{0, 1, 2, 3, 4, 5, 10, 20, 50, 100, 200, 500\}$

Parallelization Information
[gen_contour_polygon_xld](#) is *reentrant* and processed *without* parallelization.

	Possible Predecessors
get_region_contour	
	Possible Successors
smooth_contours_xld , gen_polygons_xld	
	See also
gen_contours_skeleton_xld	
	Module
Sub-pixel operators	

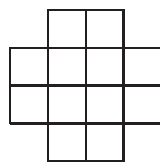
gen_contour_region_xld (Regions : Contours : Mode :)

Generate XLD contours from regions.

[gen_contour_region_xld](#) generates XLD contours [Contours](#) from the regions given in [Regions](#). This operator is useful if regions have been obtained from segmentation operations, but higher level operators, e.g., polygon approximation and extraction of parallels, are to be performed on their boundaries. For each connected component of the input regions a closed contour of the boundary is generated. The parameter [Mode](#) can take on the following values:

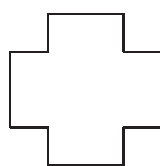
- *'center'*: The centers of the border pixels are used as contour points.
- *'border'*: The outer border of the border pixels is used as contour points.
- *'border_holes'*: In addition to the outer border of the input region you get the contours of all holes.

The difference between the two modes *'border'* and *'center'* can be seen by considering the following region:

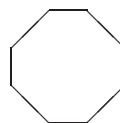


where $\square$ symbolizes a single pixel.

Then, computing the contour with *'border'* and *'center'* results in the following two contours, respectively:



'border'



'center'

This means, for example, that contours generated with *'border'* will in general have a much larger Euclidean length (see [length_xld](#)) than contours generated with *'center'*. This results from the fact that for diagonal border elements *'border'* uses two contour segments of length 1 each, whereas *'center'* uses a single segment of length $\sqrt{2}$. Other features, e.g., the area (see [area_center_xld](#)), will obviously also have different values.

	Parameter
▷ Regions (input_object)	region(-array) $\leadsto$ <i>Hobject</i> Input regions.
▷ Contours (output_object)	xld_cont(-array) $\leadsto$ <i>Hobject</i> Resulting contours.
▷ Mode (input_control)	string $\leadsto$ <i>string</i> Mode of contour generation. Default Value : <i>'border'</i> List of values : Mode $\in$ { <i>'border'</i> , <i>'border_holes'</i> , <i>'center'</i> }

Parallelization Information

`gen_contour_region_xld` is *reentrant* and processed *without* parallelization.

Possible Successors

`smooth_contours_xld`, `gen_polygons_xld`

Alternatives

`gen_contour_polygon_xld`, `get_region_contour`

See also

`gen_contours_skeleton_xld`

Module

Sub-pixel operators

gen_contours_skeleton_xld (Skeleton : Contours : Length, Mode :)

Convert a skeleton into XLD contours.

`gen_contours_skeleton_xld` converts the input skeleton (e.g., edges) `Skeleton`, which is assumed to contain mostly one pixel wide regions (see `skeleton`), into XLD contours. The regions are first transformed to contain only line segments in 8-neighborhood. In the process 12 special configurations are taken into account: Points for which there is a junction of three or more lines in 8-neighborhood are preserved (in all four rotations):

0 0 1	0 1 0	0 1 0
1 1 0	0 1 1	1 1 1
0 1 0	0 1 0	0 1 0

In a second step, all junction points are labelled, taking six different characteristic configurations of all four possible rotations into account:

1 0 1	1 0 1	1 0 0	1 0 0	0 1 0	0 1 0
0 2 0	0 2 0	0 2 1	0 2 1	0 2 1	1 2 1
0 0 1	1 0 1	0 1 0	1 0 0	0 1 0	0 1 0

where 0 = background, 1 = foreground, and 2 = junction point.

After this, all contours having at least `Length` points are returned. The contours generated by `gen_contours_skeleton_xld` always end in junction or end points. For closed contours the first point lies in the 8-neighborhood of the last point of the contour. Therefore, in order to determine the adjacency of contours it is sufficient to just take the end points into account.

Since contours are split at junction points, possibly long contours may be split into several short segments because of short adjacent lines, even if they are longer than `Length` points, if `Mode` = 'filter' was selected. This can be avoided by setting `Mode` = 'generalize1'. In this case, the contours are generated as if the segments shorter than `Length` were not contained in the input region. In order to preserve line segments, which are split into very short segments by the crossing of short lines, `Mode` = 'generalize2' can be selected. In this case, line segments are preserved if they end in two junction points, even if they are shorter than `Length`.

Parameter

- ▷ **Skeleton** (input_object) region $\leadsto$ *Hobject*
Skeleton of which the contours are to be determined.
- ▷ **Contours** (output_object) xld_cont-array $\leadsto$ *Hobject*
Resulting contours.
- ▷ **Length** (input_control) integer $\leadsto$ *integer*
Minimum number of points a contour has to have.
Default Value : 1
Suggested values : Length $\in$ {1, 2, 3, 5, 10, 20}
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Contour filter mode.
Default Value : 'filter'
List of values : Mode $\in$ {'filter', 'generalize1', 'generalize2'}

Parallelization Information

`gen_contours_skeleton_xld` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`skeleton`

Possible Successors

`smooth_contours_xld`, `get_contour_xld`, `gen_polygons_xld`

See also

`edges_image`, `threshold`, `get_region_contour`

Module

Sub-pixel operators

gen_ellipse_contour_xld (: ContEllipse : Row, Column, Phi, Radius1, Radius2, StartPhi, EndPhi, PointOrder, Resolution :)

Creation of an XLD contour corresponding to an elliptic arc.

`gen_ellipse_contour_xld` creates one or more elliptic arcs or closed ellipses. Ellipses are specified by their center (`Row`, `Column`), the orientation of the main axis `Phi`, the length of the larger half axis `Radius1`, and the length of the smaller half axis `Radius2`. In addition to that, elliptic arcs are characterized by the angle of the start point `StartPhi`, the angle of the end point `EndPhi`, and the point order `PointOrder` along the boundary. The angles in the interval $[0, 2\pi]$ are measured in the coordinate system of the ellipse relative to the main axis. Thus, the two main poles correspond to the angles 0 and π , the two other poles to the angles $\pi/2$ and $3\pi/2$. To create a closed ellipse the values 0 and 2π (with positive point order) have to be passed to the operator. The resolution of the resulting contours `ContEllipse` is controlled via `Resolution` containing the maximum Euclidean distance between neighboring contour points.

Parameter

- ▷ **ContEllipse** (output_object) xld_cont(-array) $\leadsto$ *Hobject*
Resulting contour.
- ▷ **Row** (input_control) ellipse.center.y(-array) $\leadsto$ *real*
Row coordinate of the center of the ellipse.
Default Value : 200.0
Suggested values : Row $\in$ {0.0, 10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0}
- ▷ **Column** (input_control) ellipse.center.x(-array) $\leadsto$ *real*
Column coordinate of the center of the ellipse.
Default Value : 200.0
Suggested values : Column $\in$ {0.0, 10.0, 20.0, 50.0, 100.0, 200.0, 300.0, 400.0}
- ▷ **Phi** (input_control) ellipse.angle.rad(-array) $\leadsto$ *real*
Orientation of the main axis [rad].
Default Value : 0.0
Suggested values : Phi $\in$ {-1.178097, -0.785398, -0.392699, 0.0, 0.392699, 0.785398, 1.178097}
Restriction : (Phi $\geq$ 0) $\wedge$ (Phi $\leq$ 6.283185307)
- ▷ **Radius1** (input_control) ellipse.radius1(-array) $\leadsto$ *real*
Length of the larger half axis.
Default Value : 100.0
Suggested values : Radius1 $\in$ {2.0, 5.0, 10.0, 20.0, 50.0, 100.0, 256.0, 300.0, 400.0}
Restriction : Radius1 $>$ 0
- ▷ **Radius2** (input_control) ellipse.radius2(-array) $\leadsto$ *real*
Length of the smaller half axis.
Default Value : 50.0
Suggested values : Radius2 $\in$ {1.0, 2.0, 4.0, 5.0, 10.0, 20.0, 50.0, 100.0, 256.0, 300.0, 400.0}
Restriction : (Radius2 $\geq$ 0) $\wedge$ (Radius2 $\leq$ Radius1)
- ▷ **StartPhi** (input_control) real(-array) $\leadsto$ *real*
Angle of the start point [rad].
Default Value : 0.0
Suggested values : StartPhi $\in$ {0.0, 0.78539, 1.57079, 2.35619, 3.14159, 3.92699, 4.71238, 5.49778,

- 6.28318}
Restriction : $(\text{StartPhi} \geq 0) \wedge (\text{StartPhi} \leq 6.283185307)$
- ▷ **EndPhi** (input_control) real(-array) $\leadsto$ *real*
 Angle of the end point [rad].
Default Value : 6.28318
Suggested values : $\text{EndPhi} \in \{0.0, 0.78539, 1.57079, 2.35619, 3.14159, 3.92699, 4.71238, 5.49778, 6.28318\}$
Restriction : $(\text{EndPhi} \geq 0) \wedge (\text{EndPhi} \leq 6.283185307)$
- ▷ **PointOrder** (input_control) string(-array) $\leadsto$ *string*
 point order along the boundary.
Default Value : 'positive'
List of values : $\text{PointOrder} \in \{'positive', 'negative'\}$
- ▷ **Resolution** (input_control) real $\leadsto$ *real*
 Resolution: Maximum distance between neighboring contour points.
Default Value : 1.5
Suggested values : $\text{Resolution} \in \{0.5, 1.0, 1.5, 2.0, 2.5, 3.0\}$
Restriction : $\text{Resolution} > 0$

Example

```
draw_ellipse(WindowHandle, Row, Column, Phi, Radius1, Radius2)
gen_ellipse_contour_xld(Ellipse, Row, Column, Phi, Radius1, Radius2, 0, 6.28319,
                        'positive', 1.5)
length_xld(Ellipse, Length).
```

Result

[gen_ellipse_contour_xld](#) returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Parallelization Information

[gen_ellipse_contour_xld](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[draw_ellipse](#)

Possible Successors

[disp_xld](#), [get_points_ellipse](#)

Module

Sub-pixel operators

gen_parallel_xld (Polygons : Parallels : Len, Dist, Alpha,
Merge :)

Extract parallel XLD polygons.

[gen_parallel_xld](#) examines the XLD polygons passed in **Polygons** for parallelism. The resulting parallel polygons are returned in **Parallels**. If the parameter **Merge** is set to 'true', adjacent parallel polygons are returned in a single parallel relation. Otherwise, one parallel relation is returned for each pair of parallel line segments. Whether two polygon segments are parallel depends on their distance (smaller than **Dist**), a maximum allowed angle difference (**Alpha**, in radians), and a minimum length of the two polygon segments. Furthermore, the two segments have to overlap. As a side effect, a quality factor is calculated for each pair of parallels. It is based on the normalized angular difference and the normalized length of the overlapping area:

$$quality = \frac{\pi - \delta\alpha}{\pi/2} \frac{2\overlap}{len_1 + len_2} \quad \text{with } 0 \leq q \leq 1$$

Here, $\delta\alpha$ is the angle difference of the polygon segments, *overlap* is the length of the overlap area, len_1 and len_2 the length of the polygon segments, and *quality* the resulting quality factor.

The quality factor is a measure of parallelism (the larger its value, the “more parallel” the polygons). Finally, the quality factors of all parallel polygon segments contained in a single polygon are added, weighted with their length of the overlapping area.

Parameter	
▷ Polygons (input_object)	xld_poly-array $\leadsto$ Hobject Input polygons.
▷ Parallels (output_object)	xld_para-array $\leadsto$ Hobject Parallel polygons.
▷ Len (input_control)	number $\leadsto$ real / integer Minimum length of the individual polygon segments. Default Value : 10.0 Suggested values : $\text{Len} \in \{5.0, 10.0, 15.0, 20.0\}$ Restriction : $\text{Len} > 0.0$
▷ Dist (input_control)	number $\leadsto$ real / integer Maximum distance between the polygon segments. Default Value : 30.0 Suggested values : $\text{Dist} \in \{20.0, 25.0, 30.0, 40.0, 50.0, 75.0\}$ Restriction : $\text{Dist} > 0.0$
▷ Alpha (input_control)	number $\leadsto$ real / integer Maximum angle difference of the polygon segments. Default Value : 0.15 Suggested values : $\text{Alpha} \in \{0.05, 0.10, 0.15, 0.20, 0.30\}$ Restriction : $(0 \leq \text{Alpha}) \wedge (\text{Alpha} \leq (\pi/2))$
▷ Merge (input_control)	string $\leadsto$ string Should adjacent parallel relations be merged? Default Value : 'true' List of values : $\text{Merge} \in \{\text{'true'}, \text{'false'}\}$
Parallelization Information	
gen_parallels_xld is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Predecessors	
gen_polygons_xld	
Possible Successors	
mod_parallels_xld , get_parallels_xld	
Module	
Sub-pixel operators	

gen_polygons_xld (Contours : Polygons : Type, Alpha :)

Approximate XLD contours by polygons.

[gen_polygons_xld](#) approximates XLD contours ([Contours](#)) by polygons. The type of the approximation can be set by [Type](#). The threshold for the approximation is set via [Alpha](#). The procedure is able to process open as well as closed contours. The resulting approximating XLD polygons are returned in [Polygons](#).

Contours can be approximated by the algorithms of Ramer, Ray, and Sato. The algorithm of Ramer approximates contours such that the Euclidian distance of the approximating polygon to the contour is at most [Alpha](#) pixel units. The algorithm of Ray does not need a threshold, and hence [Alpha](#) is ignored. Here, the contour is approximated by maximizing the line length, while minimizing the sum of the distances of the line segment from the contour. The algorithm of Sato produces a polygon point at the point of the contour in which the distance to the end points of the contour is maximal. The total approximation error for each iteration is then given by $(L - L')/L$, where L is the Euclidian contour length, and L' is the length of the approximating polygon.

Parameter	
▷ Contours (input_object)	xld_cont-array $\leadsto$ Hobject Contours to be approximated.
▷ Polygons (output_object)	xld_poly-array $\leadsto$ Hobject Approximating polygons.

- ▷ **Type** (input_control) string $\leadsto$ string
Type of approximation.
Default Value : 'ramer'
List of values : Type $\in$ {'ramer', 'ray', 'sato'}
- ▷ **Alpha** (input_control) number $\leadsto$ real / integer
Threshold for the approximation.
Default Value : 2.0
Suggested values : Alpha $\in$ {1.0, 1.5, 2.0, 3.0, 4.0}
Restriction : Alpha > 0.0

Parallelization Information

[gen_polygons_xld](#) is *reentrant* and processed without parallelization.

Possible Predecessors

[gen_contours_skeleton_xld](#), [lines_gauss](#), [lines_facet](#), [edges_sub_pix](#)

Possible Successors

[gen_parallel_xld](#)

See also

[get_region_polygon](#)

Module

Sub-pixel operators

mod_parallel_xld (Parallels, Image : ModParallels,
ExtParallels : Quality, MinGray, MaxGray, MaxStandard :)

Extract parallel XLD polygons enclosing a homogeneous area.

[mod_parallel_xld](#) selects XLD parallels enclosing homogeneous areas from the input parallels [Parallels](#). The parameter [Image](#) contains the corresponding gray value image.

Only parallels having a quality factor larger than [Quality](#) are examined. The algorithm performs parallel cross sections one pixel apart, and parallel to the line segments in the area of overlap between two parallel line segments.

In the first iteration, the mean gray value for each of the lines in the cross section is calculated. In the second iteration, the standard deviations of the gray values along each line are computed.

If the mean gray value of an area between parallels lies in the interval [[MinGray](#),[MaxGray](#)], and if the mean of all standard deviations is smaller than the upper threshold [MaxStandard](#), the corresponding parallels are returned as modified parallels in [ModParallels](#).

In a second step, all polygon segments adjacent to parallels bordering homogeneous areas are checked for homogeneity. To do so, a rectangle having the width of the last area enclosed by a modified parallel is constructed, and checked for homogeneity using the algorithm described above. This process is continued as long as there are adjacent polygon segments. The polygons thus found are returned as extended parallels in [ExtParallels](#).

Parameter

- ▷ **Parallels** (input_object) xld_para-array $\leadsto$ Hobject
Input XLD parallels.
- ▷ **Image** (input_object) image $\leadsto$ Hobject : byte
Corresponding gray value image.
- ▷ **ModParallels** (output_object) xld_mod_para-array $\leadsto$ Hobject
Modified XLD parallels.
- ▷ **ExtParallels** (output_object) xld_ext_para-array $\leadsto$ Hobject
Modified XLD parallels.
- ▷ **Quality** (input_control) number $\leadsto$ real / integer
Minimum quality factor (measure of parallelism).
Default Value : 0.4
Suggested values : Quality $\in$ {0.1, 0.2, 0.3, 0.4, 0.5, 0.6}
Restriction : (0.0 $\leq$ Quality) $\wedge$ (Quality $\leq$ 1.0)

- ▷ **MinGray** (input_control) integer $\leadsto$ integer
Minimum mean gray value.
Default Value : 160
Suggested values : $\text{MinGray} \in \{80, 100, 120, 140, 160, 180\}$
Restriction : $(0 \leq \text{MinGray}) \wedge (\text{MinGray} \leq 255)$
- ▷ **MaxGray** (input_control) integer $\leadsto$ integer
Maximum mean gray value.
Default Value : 220
Suggested values : $\text{MaxGray} \in \{140, 160, 180, 200, 220, 240\}$
Restriction : $((0 \leq \text{MaxGray}) \wedge (\text{MaxGray} \leq 255)) \wedge (\text{MaxGray} \geq \text{MinGray})$
- ▷ **MaxStandard** (input_control) number $\leadsto$ real / integer
Maximum allowed standard deviation.
Default Value : 10.0
Suggested values : $\text{MaxStandard} \in \{5.0, 10.0, 15.0, 20.0\}$
Restriction : $\text{MaxStandard} \geq 0.0$

Parallelization Information

[mod_parallel_xld](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_parallel_xld](#)

Possible Successors

[max_parallel_xld](#)

See also

[info_parallel_xld](#)

Module

Sub-pixel operators

17.3 Features

area_center_xld (XLD : : : Area, Row, Column, PointOrder)

Area and center of gravity (centroid) of contours and polygons.

[area_center_xld](#) calculates the area and center of gravity (centroid) of the region enclosed by the contours or polygons [XLD](#) as well as the order of the points along the boundary. The area and centroid are computed by applying Green's theorem using only the points on the contour or polygon, i.e., no region is generated explicitly for the purpose of calculating the features. If the points are arranged counterclockwise (i.e., in a positive mathematical sense) in the contour, [PointOrder](#) will be "positive". It is assumed that the contours or polygons are closed. If this is not the case [area_center_xld](#) will artificially close the contour. If more than one contour or polygon is passed the results are stored as tuples in which the index of a value corresponds to the index of the respective contour or polygon in [XLD](#).

Parameter

- ▷ **XLD** (input_object) xld(-array) $\leadsto$ *Hobject*
Contours or polygons to be examined.
- ▷ **Area** (output_control) real(-array) $\leadsto$ *real*
Area enclosed by the contour or polygon.
- ▷ **Row** (output_control) point.y(-array) $\leadsto$ *real*
Row coordinate of the centroid.
- ▷ **Column** (output_control) point.x(-array) $\leadsto$ *real*
Column coordinate of the centroid.
- ▷ **PointOrder** (output_control) string(-array) $\leadsto$ *string*
point order along the boundary ("positive"/"negative").

Complexity

Let n be the number of points of the contour or polygon. Then the run time is $O(n)$.

Result

[area_center_xld](#) returns 2 (H_MSG_TRUE) if the input is not empty. If the input is empty the behaviour can be set via [set_system\(':', 'no_object_result', <Result>:\)](#). If necessary, an exception is raised.

Parallelization Information

[area_center_xld](#) is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

[gen_contours_skeleton_xld](#), [smooth_contours_xld](#), [gen_polygons_xld](#)

See also

[moments_xld](#), [moments_any_xld](#), [area_center](#), [moments_region_2nd](#)

Module

Sub-pixel operators

circularity_xld (XLD : : : Circularity)

Shape factor for the circularity (similarity to a circle) of a contour.

The operator [circularity_xld](#) calculates the similarity of the input contour with a circle. The input contour must not intersect itself, otherwise the resulting parameter is not meaningful. If the input contour is not closed it will be closed automatically.

Calculation: If F is the enclosing area of the contour and max is the maximum distance from the center to all contour pixels, the shape factor [Circularity](#) is defined as:

$$\text{Circularity} = \frac{F}{(max^2 * \pi)}$$

The shape factor [Circularity](#) of a circle is 1. If the contour encloses an elongated area [Circularity](#) is smaller than 1. The operator [circularity_xld](#) especially responds to large bulges.

If more than one contour is passed the numerical values of the shape factor are stored in a tuple, the position of a value in the tuple corresponding to the position of the contour in the input tuple.

Parameter

- ▷ **XLD** (input_object) xld(-array) $\leadsto$ *Hobject*
Contour(s) to be examined.
 - ▷ **Circularity** (output_control) real(-array) $\leadsto$ *real*
Roundness of the input contour(s).
- Assertion :** $(0 \leq \text{Circularity}) \wedge (\text{Circularity} \leq 1.0)$

Result

The operator [circularity_xld](#) returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input contour available) is set via the operator [set_system\('no_object_result', <Result>:\)](#). If necessary an exception is raised.

Parallelization Information

[circularity_xld](#) is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

[gen_contours_skeleton_xld](#), [edges_sub_pix](#), [threshold_sub_pix](#),
[gen_contour_polygon_xld](#)

Alternatives

[compactness_xld](#), [convexity_xld](#), [eccentricity_xld](#)

See also

[area_center_xld](#), [select_shape_xld](#)

Module

Sub-pixel operators

compactness_xld (XLD : : : Compactness)

Shape factor for the compactness of a contour.

The operator `compactness_xld` calculates the compactness of the input contour `XLD`. The input contour must not intersect itself, otherwise the resulting parameter is not meaningful. If the input contour is not closed it will be closed automatically.

Calculation: If L is the length of the contour and F the enclosing area of the contour the shape factor **Compactness** is defined as:

$$\text{Compactness} = \frac{L^2}{4F\pi}$$

The shape factor **Compactness** of a circle is 1. If the contour encloses an elongated area **Compactness** is larger than 1. The operator `compactness_xld` responds to the course of the contour (roughness). If more than one contour is passed the numerical values of the shape factor are stored in a tuple, the position of a value in the tuple corresponding to the position of the contour in the input tuple.

Parameter

- ▷ **XLD** (input_object) xld(-array) $\leadsto$ *Hobject*
Contour(s) to be examined.
 - ▷ **Compactness** (output_control) real(-array) $\leadsto$ *real*
Compactness of the input contour(s).
- Assertion :** (Compactness $\geq$ 1.0) $\vee$ (Compactness = 0)

Result

The operator `compactness_xld` returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input contours available) is set via the operator `set_system('no_object_result', <Result>)`. If necessary an exception is raised.

Parallelization Information

`compactness_xld` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`gen_contours_skeleton_xld`, `edges_sub_pix`, `threshold_sub_pix`,
`gen_contour_polygon_xld`

Alternatives

`circularity_xld`, `convexity_xld`, `eccentricity_xld`

See also

`area_center_xld`, `select_shape_xld`

Module

Sub-pixel operators

contour_point_num_xld (Contour : : : Length)

Return the number of points in an XLD contour.

`contour_point_num_xld` returns the length of the input contour `Contour`, i.e., its number of points, in `Length`.

Parameter

- ▷ **Contour** (input_object) xld_cont $\leadsto$ *Hobject*
Input XLD contour.
- ▷ **Length** (output_control) integer $\leadsto$ *integer*
Number of contour points.

Parallelization Information

`contour_point_num_xld` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`gen_contours_skeleton_xld`, `lines_gauss`, `lines_facet`, `edges_sub_pix`

	Possible Successors
get_contour_xld , get_contour_attrib_xld	
	Alternatives
get_regress_params_xld	
	See also
query_contour_attribs_xld	
	Module
Sub-pixel operators	

convexity_xld (XLD : : : Convexity)

Shape factor for the convexity of a contour.

The operator [convexity_xld](#) calculates the convexity of the input contour. The input contour must not intersect itself, otherwise the resulting parameter is not meaningful. If the input contour is not closed it will be closed automatically.

Calculation: If F_c is the area of the convex hull and F_o the area enclosed by the original contour the shape factor **Convexity** is defined as:

$$\text{Convexity} = \frac{F_o}{F_c}$$

The shape factor **Convexity** is 1 if the contour is convex (e.g., a rectangle, circle, etc.). If there are indentations **Convexity** is smaller than 1.

If more than one contour is passed the numerical values of the shape factor are stored in a tuple, the position of a value in the tuple corresponding to the position of the contour in the input tuple.

	Parameter
▷ XLD (input_object)	xld(-array) $\leadsto$ <i>Hobject</i> Contour(s) to be examined.
▷ Convexity (output_control)	real(-array) $\leadsto$ <i>real</i> Convexity of the input contour(s).
Assertion : Convexity ≤ 1	

	Result
The operator convexity_xld returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input contour available) is set via the operator set_system ('no_object_result', <Result>). If necessary an exception is raised.	

	Parallelization Information
convexity_xld is <i>reentrant</i> and automatically <i>parallelized</i> (on <i>tuple level</i>).	

	Possible Predecessors
gen_contours_skeleton_xld , edges_sub_pix , threshold_sub_pix , gen_contour_polygon_xld	

	See also
area_center_xld , select_shape_xld , shape_trans_xld	

	Module
Sub-pixel operators	

diameter_xld (XLD : : : Row1, Column1, Row2, Column2, Diameter)

Maximum distance between two contour points.

The operator [diameter_xld](#) calculates the maximum distance between two contour points. The coordinates of these two extremes and the distance between them will be returned. The input contour must not intersect

itself, otherwise the resulting parameters are not meaningful. If the input contour is not closed it will be closed automatically.

Attention

If the contour is empty, the results of **Row1**, **Column1**, **Row2** and **Column2** (all of them = 0) may lead to confusion.

Parameter

- ▷ **XLD** (input_object) xld(-array) $\leadsto$ *Hobject*
Contour(s) to be examined.
- ▷ **Row1** (output_control) line.begin.y(-array) $\leadsto$ *real*
Row coordinate of the first extreme point of the contour(s).
- ▷ **Column1** (output_control) line.begin.x(-array) $\leadsto$ *real*
Column coordinate of the first extreme point of the contour(s).
- ▷ **Row2** (output_control) line.end.y(-array) $\leadsto$ *real*
Row coordinate of the second extreme point of the contour(s).
- ▷ **Column2** (output_control) line.end.x(-array) $\leadsto$ *real*
Column coordinate of the second extreme point of the contour(s).
- ▷ **Diameter** (output_control) number(-array) $\leadsto$ *real*
Distance of the two extreme points of the contour(s).

Result

The operator **diameter_xld** returns the value 2 (H_MSG_TRUE), if the input is not empty. The reaction to empty input (no input contours are available) may be determined with the help of the operator **set_system** ('no_object_result', <Result>). If necessary an exception is raised.

Parallelization Information

diameter_xld is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

gen_contours_skeleton_xld, **edges_sub_pix**, **threshold_sub_pix**,
gen_contour_polygon_xld

Alternatives

smallest_rectangle2_xld

See also

area_center_xld

Module

Sub-pixel operators

dist_ellipse_contour_xld (Contours : : Mode, MaxNumPoints,
ClippingEndPoints, Row, Column, Phi, Radius1, Radius2 : MinDist, MaxDist,
AvgDist, SigmaDist)

Distance of contours to an ellipse.

dist_ellipse_contour_xld determines the distance between the contours in **Contours** and an ellipse specified by the center (**Row**, **Column**), the orientation of the main axis **Phi**, the length of the larger half axis **Radius1**, and the length of the smaller half axis **Radius2**. Different measures for the distance of a contour point $X = (x_i, y_i)$ to the ellipse are available (**Mode**):

- 'algebraic' The distance is measured by the algebraic distance $ax_i^2 + bx_iy_i + cy_i^2 + dx_i + ey_i + f$, where the parameters $a - f$ describing the ellipse are normalized in order to obtain **Radius2** as distance of the center of the ellipse. This measure shows a *high curvature bias*: Near points of high curvature on the ellipse (like the poles on the main axis) the distance is smaller than near points with low curvature.
- 'geometric' The distance is measured by the deviation $XF_1 + XF_2 - 2a$, where F_1, F_2 are the focal points and a corresponds to **Radius1**. This measure shows a *low curvature bias*: Near points of high curvature on the ellipse (like the poles on the main axis) the distance is larger than near points with low curvature.
- 'bisec' The distance is measured by the distance between X and the intersection of the angular bisector of the two lines through X and the focal points with the ellipse. This is a very good approximation of the orthogonal distance from X to the ellipse.

The operator returns the minimum absolute distance **MinDist**, the maximum absolute distance **MaxDist**, the average absolute distance **AvgDist**, and the standard deviation of the absolute distances **SigmaDist** of all contour points.

To reduce the computational load, the computation of the distances can be restricted to a subset of the contour points: If a value other than -1 is assigned to **MaxNumPoints**, only up to **MaxNumPoints** points - uniformly distributed over the contour - are used. Due to artefacts in the pre-processing the start and end points of a contour might be faulty. Therefore, it is possible to exclude **ClippingEndPoints** points at the beginning and at the end of a contour from the computation.

Parameter

- ▷ **Contours** (input_object) xld_cont(-array) $\leadsto$ *Hobject*
Input contours.
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Method for the determination of the distances.
Default Value : 'bisec'
List of values : Mode $\in$ {'geometric', 'algebraic', 'bisec'}
- ▷ **MaxNumPoints** (input_control) integer $\leadsto$ *integer*
Maximum number of contour points used for the computation (-1 for all points).
Default Value : -1
Restriction : MaxNumPoints ≥ 3
- ▷ **ClippingEndPoints** (input_control) integer $\leadsto$ *integer*
Number of points at the beginning and the end of the contours to be ignored for the computation of distances.
Default Value : 0
Restriction : ClippingEndPoints ≥ 0
- ▷ **Row** (input_control) ellipse.center.y $\leadsto$ *real*
Row coordinate of the center of the ellipse of the ellipse.
- ▷ **Column** (input_control) ellipse.center.x $\leadsto$ *real*
Column coordinate of the center of the ellipse.
- ▷ **Phi** (input_control) ellipse.angle.rad $\leadsto$ *real*
Orientation of the main axis [rad].
Restriction : (Phi ≥ 0) $\wedge$ (Phi ≤ 6.283185307)
- ▷ **Radius1** (input_control) ellipse.radius1 $\leadsto$ *real*
Length of the larger half axis.
Restriction : Radius1 > 0
- ▷ **Radius2** (input_control) ellipse.radius2 $\leadsto$ *real*
Length of the smaller half axis.
Restriction : (Radius2 ≥ 0) $\wedge$ (Radius2 $\leq$ Radius1)
- ▷ **MinDist** (output_control) real(-array) $\leadsto$ *real*
Minimum distance.
- ▷ **MaxDist** (output_control) real(-array) $\leadsto$ *real*
Maximum distance.
- ▷ **AvgDist** (output_control) real(-array) $\leadsto$ *real*
Mean distance.
- ▷ **SigmaDist** (output_control) real(-array) $\leadsto$ *real*
Standard deviation of the distance.

Example

```
read_image (Image, 'caltab')
find_caltab (Image, Caltab, 'caltab_big.descr', 3, 112, 5)
reduce_domain (Image, Caltab, ImageReduced)
edges_sub_pix (ImageReduced, Edges, 'lanser2', 0.5, 20, 40)
select_contours_xld (Edges, EdgesClosed, 'closed', 0, 2.0, 0, 0)
select_contours_xld (EdgesClosed, EdgesMarks, 'contour_length', 20, 100,
                    0, 0)
fit_ellipse_contour_xld (EdgesMarks, 'fitzgibbon', -1, 2, 0, 200, 3, 2.0,
                        Row, Column, Phi, Radius1, Radius2, StartPhi,
                        EndPhi, PointOrder)
```

```

for i := 0 to |Row|-1 by 1
    Object_Selected := EdgesMarks[i+1]
    dist_ellipse_contour_xld (ObjectSelected, 'bisec', -1, 0, Row[i],
                             Column[i], Phi[i], Radius1[i], Radius2[i],
                             MinDist, MaxDist, AvgDist, SigmaDist)
endfor.

```

Result

dist_ellipse_contour_xld returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Parallelization Information

dist_ellipse_contour_xld is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

fit_ellipse_contour_xld

Module

Sub-pixel operators

eccentricity_xld (XLD : : : Anisometry, Bulkiness, StructureFactor)

Shape features derived from the ellipse parameters.

The operator **eccentricity_xld** calculates the three shape features **Anisometry**, **Bulkiness**, and **StructureFactor** derived from the geometric moments. The input contour must not intersect itself, otherwise the resulting parameters are not meaningful. If the input contour is not closed it will be closed automatically.

Definition: If the ellipse radii R_a , R_b (**elliptic_axis_xld**) and the enclosing area A of the contour are given, the following applies:

$$\begin{aligned}
 \text{Anisometry} &= \frac{R_a}{R_b} \\
 \text{Bulkiness} &= \frac{\pi \cdot R_a \cdot R_b}{A} \\
 \text{StructureFactor} &= \text{Anisometry} \cdot \text{Bulkiness} - 1
 \end{aligned}$$

The anisometry of a circle is 1.0.

If more than one contour is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a contour in the input.

Parameter

- ▷ **XLD** (input_object) xld(-array) $\leadsto$ *Hobject*
Contour(s) to be examined.
- ▷ **Anisometry** (output_control) real(-array) $\leadsto$ *real*
Anisometry of the contour(s).
Assertion: $\text{Anisometry} \geq 1.0$
- ▷ **Bulkiness** (output_control) real(-array) $\leadsto$ *real*
Bulkiness of the contour(s).
- ▷ **StructureFactor** (output_control) real(-array) $\leadsto$ *real*
Structure factor of the contour(s).

Result

The operator **eccentricity_xld** returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input contours available) is set via the operator **set_system** ('no_object_result', <Result>). If necessary an exception is raised.

Parallelization Information

eccentricity_xld is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

gen_contours_skeleton_xld, **edges_sub_pix**, **threshold_sub_pix**,
gen_contour_polygon_xld

See also

[elliptic_axis_xld](#), [moments_region_2nd](#), [select_shape_xld](#), [area_center_xld](#)

Module

Sub-pixel operators

elliptic_axis_xld (XLD : : : Ra, Rb, Phi)*Parameters of the equivalent ellipse.*

The operator [elliptic_axis_xld](#) calculates the radii and the orientation of the ellipse having the “same orientation” and the “same aspect ratio” as the input contour. The closed input contour must not intersect itself, otherwise the resulting parameters are not meaningful. Several input contours can be passed in [XLD](#) as tuples. The length of the major radius [Ra](#) and the minor radius [Rb](#) as well as the orientation of the main axis with regard to the horizontal ([Phi](#)) are determined. The angle is indicated in radians.

Calculation:

If the moments M_{20} , M_{02} and M_{11} are normalized and passed to the area (see [moments_xld](#)), the radii [Ra](#) and [Rb](#) are calculated as:

$$\text{Ra} = \frac{\sqrt{8(M_{20} + M_{02} + \sqrt{(M_{20} - M_{02})^2 + 4M_{11}^2})}}{2}$$

$$\text{Rb} = \frac{\sqrt{8(M_{20} + M_{02} - \sqrt{(M_{20} - M_{02})^2 + 4M_{11}^2})}}{2}$$

The orientation [Phi](#) is defined by:

$$\text{Phi} = -0.5 \text{atan2}(2M_{11}, M_{02} - M_{20})$$

If more than one contour is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a contour in the input.

Parameter

- ▷ **XLD** (input_object) xld(-array) $\leadsto$ *Hobject*
Contour(s) to be examined.
- ▷ **Ra** (output_control) real(-array) $\leadsto$ *real*
Major radius.
Assertion : $\text{Ra} \geq 0.0$
- ▷ **Rb** (output_control) real(-array) $\leadsto$ *real*
Minor radius.
Assertion : $(\text{Rb} \geq 0.0) \wedge (\text{Rb} \leq \text{Ra})$
- ▷ **Phi** (output_control) real(-array) $\leadsto$ *real*
Angle between the major axis and the x axis (radians).
Assertion : $((-\pi/2) < \text{Phi}) \wedge (\text{Phi} \leq (\pi/2))$

Complexity

If N is the number of contour points, the runtime complexity is $O(N)$.

Result

[elliptic_axis_xld](#) returns 2 (H_MSG_TRUE) if the input is not empty. If the input is empty the behavior can be set via [set_system\(::'no_object_result', <Result>:\)](#). If necessary, an exception is raised.

Parallelization Information

[elliptic_axis_xld](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[gen_contours_skeleton_xld](#), [edges_sub_pix](#), [threshold_sub_pix](#),
[gen_contour_polygon_xld](#)

Possible Successors

[area_center_xld](#), [gen_ellipse_contour_xld](#)

Alternatives

[smallest_rectangle2](#)

See also

[moments_xld](#), [smallest_circle_xld](#), [smallest_rectangle1_xld](#),
[smallest_rectangle2_xld](#), [shape_trans_xld](#)

References

R. Haralick, L. Shapiro “Computer and Robot Vision” Addison-Wesley, 1992, pp. 73-75

Module

Sub-pixel operators

fit_circle_contour_xld (Contours : : Algorithm, MaxNumPoints,
MaxClosureDist, ClippingEndPoints, Iterations, ClippingFactor : Row,
Column, Radius, StartPhi, EndPhi, PointOrder)
Approximation of XLD contours by circles.

[fit_circle_contour_xld](#) approximates the XLD contours [Contours](#) by circles. It does not perform a segmentation of the input contours. Thus, one has to make sure that each contour corresponds to one and only one circle. The operator returns for each contour the center ([Row](#), [Column](#)), and the [Radius](#).

The algorithm used for the fitting of circles can be selected via [Algorithm](#):

- 'algebraic' This approach minimizes the algebraic distance between the contour points and the resulting circle.
- 'ahuber' Similar to 'algebraic'. Here the contour points are weighted to decrease the impact of outliers based on the approach of Huber.
- 'atukey' Similar to 'algebraic'. Here the contour points are weighted to decrease the impact of outliers based on the approach of Tukey.

For 'ahuber' and 'atukey' a robust error statistics is used to estimate the standard deviation of the distances from the contour points *without* outliers from the approximating circle. The parameter [ClippingFactor](#) (a scaling factor for the standard deviation) controls the amount of damping outliers: The smaller the value chosen for [ClippingFactor](#) the more outliers are detected. The detection of outliers and the least squares fitting is repeated. The parameter [Iterations](#) specifies the number of iterations.

To reduce the computational load, the fitting of circles can be restricted to a subset of the contour points: If a value other than -1 is assigned to [MaxNumPoints](#), only up to [MaxNumPoints](#) points - uniformly distributed over the contour - are used.

For circular arcs, the points on the circle closest to the start points and end points of the original contours are chosen as start and end points. The corresponding angles referring to the X-axis are returned in [StartPhi](#) and [EndPhi](#), see also [gen_ellipse_contour_xld](#). Contours, for which the distance between their start points and their end points is less or equal [MaxClosureDist](#) are considered to be closed. Thus, they are approximated by circles instead of circular arcs. Due to artefacts in the pre-processing the start and end points of a contour might be faulty. Therefore, it is possible to exclude [ClippingEndPoints](#) points at the beginning and at the end of a contour from the fitting of circles. However, they are still used for the determination of [StartPhi](#) and [EndPhi](#).

Parameter

- ▷ **Contours** (input_object) xld_cont(-array) $\leadsto$ *Hobject*
Input contours.
- ▷ **Algorithm** (input_control) string $\leadsto$ *string*
Algorithm for the fitting of circles.
Default Value : 'algebraic'
List of values : Algorithm $\in$ {'algebraic', 'ahuber', 'atukey'}
- ▷ **MaxNumPoints** (input_control) integer $\leadsto$ *integer*
Maximum number of contour points used for the computation (-1 for all points).
Default Value : -1
Restriction : MaxNumPoints ≥ 3
- ▷ **MaxClosureDist** (input_control) real $\leadsto$ *real*
Maximum distance between the end points of a contour to be considered as 'closed'.
Default Value : 0.0
Restriction : MaxClosureDist ≥ 0.0

- ▷ **ClippingEndPoints** (input_control) integer $\leadsto$ integer
Number of points at the beginning and at the end of the contours to be ignored for the fitting.
Default Value : 0
Restriction : ClippingEndPoints ≥ 0
- ▷ **Iterations** (input_control) integer $\leadsto$ integer
Maximum number of iterations.
Default Value : 3
Restriction : Iterations ≥ 0
- ▷ **ClippingFactor** (input_control) real $\leadsto$ real
Clipping factor for the elimination of outliers (typical: 1.0 for Huber and 2.0 for Tukey).
Default Value : 2.0
List of values : ClippingFactor $\in \{1.0, 1.5, 2.0, 2.5, 3.0\}$
Restriction : ClippingFactor > 0
- ▷ **Row** (output_control) circle.center.y(-array) $\leadsto$ real
Row coordinate of the center of the circle.
- ▷ **Column** (output_control) circle.center.x(-array) $\leadsto$ real
Column coordinate of the center of the circle.
- ▷ **Radius** (output_control) circle.radius(-array) $\leadsto$ real
Radius of circle.
- ▷ **StartPhi** (output_control) real(-array) $\leadsto$ real
Angle of the start point [rad].
- ▷ **EndPhi** (output_control) real(-array) $\leadsto$ real
Angle of the end point [rad].
- ▷ **PointOrder** (output_control) string(-array) $\leadsto$ string
Point order along the boundary.
List of values : PointOrder $\in \{\text{'positive'}, \text{'negative'}\}$

Result

`fit_circle_contour_xld` returns 2 (H_MSG_TRUE) if all parameter values are correct, and circles could be fitted to the input contours. If the input is empty the behaviour can be set via `set_system('no_object_result', <Result>)`. If necessary, an exception is raised. If the parameter `ClippingFactor` is chosen too small, i.e., all points are classified as outliers, the error 3264 is raised.

Parallelization Information

`fit_circle_contour_xld` is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

`gen_contours_skeleton_xld`, `lines_gauss`, `lines_facet`, `edges_sub_pix`,
`smooth_contours_xld`

Possible Successors

`gen_ellipse_contour_xld`, `disp_circle`, `get_points_ellipse`

See also

`fit_ellipse_contour_xld`, `fit_line_contour_xld`

Module

Sub-pixel operators

fit_ellipse_contour_xld (Contours : : Algorithm, MaxNumPoints, MaxClosureDist, ClippingEndPoints, VossTabSize, Iterations, ClippingFactor : Row, Column, Phi, Radius1, Radius2, StartPhi, EndPhi, PointOrder)

Approximation of XLD contours by ellipses or elliptic arcs.

`fit_ellipse_contour_xld` approximates the XLD contours `Contours` by elliptic arcs or closed ellipses. It does not perform a segmentation of the input contours. Thus, one has to make sure that each contour corresponds to one and only one elliptic structure. The operator returns for each contour the center (**Row**, **Column**), the orientation of the main axis **Phi**, the length of the larger half axis **Radius1**, and the length of the smaller half axis **Radius2** of the underlying ellipse. In addition to that, the angle corresponding to the start point and the end

point [StartPhi](#), [EndPhi](#), and the point order along the boundary [PointOrder](#) is returned for elliptic arcs. These parameters are set to 0, 2π , and 'positive' for closed ellipses. The algorithm used for the fitting of ellipses can be selected via [Algorithm](#):

- 'fitzgibbon' This approach minimizes the algebraic distance $ax_i^2 + bx_iy_i + cy_i^2 + dx_i + ey_i + f$ between the contour points (x_i, y_i) and the resulting ellipse. The constraint $4ac - b^2 = 1$ guarantees that the resulting polynom characterizes an ellipse (instead of a hyperbola or a parabola).
- 'fhuber' Similar to 'fitzgibbon'. Here the contour points are weighted to decrease the impact of outliers based on the approach of Huber.
- 'ftukey' Similar to 'fitzgibbon'. Here the contour points are weighted to decrease the impact of outliers based on the approach of Tukey.
- 'voss' Each input contour is transformed in an affine standard position. Based on the moments of the transformed contour (that is of the enclosed image region) the standard circular segment is chosen whose standard position matches best with the standard position of the contour. The ellipse corresponding to the standard position of the selected circular segment is re-transformed based on the affine transformation which produced the standard position of the contour resulting in the ellipse matching the original contour. [VossTabSize](#) standard circular segments are used for this computation. To speed up the process the corresponding moments and other data is stored in a table which is created during the first call (with a specific value for [VossTabSize](#)) to [fit_ellipse_contour_xld](#).
- 'focpoints' Each point P on an ellipse satisfies the constraint that the sum of distances to the focal points F_1, F_2 equals twice the length of the larger half axis a . In this approach, the deviation $PF_1 + PF_2 - 2a$ is minimized for all contour points by a least squares optimization.
- 'fphuber' Similar to 'focpoints'. Here a *weighted* least squares optimization is done to decrease the impact of outliers based on the approach of Huber.
- 'fptukey' Similar to 'focpoints'. Here a *weighted* least squares optimization is done to decrease the impact of outliers based on the approach of Tukey.

For '*Huber' and '*Tukey' a robust error statistics is used to estimate the standard deviation of the distances from the contour points *without* outliers from the approximating ellipse. The parameter [ClippingFactor](#) (a scaling factor for the standard deviation) controls the amount of damping outliers: The smaller the value chosen for [ClippingFactor](#) the more outliers are detected. The detection of outliers and the least squares fitting in the mode 'focpoints' is repeated. The parameter [Iterations](#) specifies the number of iterations.

To reduce the computational load, the fitting of ellipses can be restricted to a subset of the contour points: If a value other than -1 is assigned to [MaxNumPoints](#), only up to [MaxNumPoints](#) points - uniformly distributed over the contour - are used.

For elliptic arcs, the points on the ellipse closest to the start points and end points of the original contours are chosen as start and end points. The corresponding angles referring to the main axis of the ellipse are returned in [StartPhi](#) and [EndPhi](#), see also [gen_ellipse_contour_xld](#). Contours, for which the distance between their start points and their end points is less or equal [MaxClosureDist](#) are considered to be closed. Thus, they are approximated by ellipses instead of elliptic arcs. Due to artefacts in the pre-processing the start and end points of a contour might be faulty. Therefore, it is possible to exclude [ClippingEndPoints](#) points at the beginning and at the end of a contour from the fitting of ellipses. However, they are still used for the determination of [StartPhi](#) and [EndPhi](#).

Parameter

- ▷ **Contours** (input_object) xld_cont(-array) $\leadsto$ *Hobject*
Input contours.
- ▷ **Algorithm** (input_control) string $\leadsto$ *string*
Algorithm for the fitting of ellipses.
Default Value : 'fitzgibbon'
List of values : $\text{Algorithm} \in \{\text{'fitzgibbon'}, \text{'fhuber'}, \text{'ftukey'}, \text{'voss'}, \text{'focpoints'}, \text{'fphuber'}, \text{'fptukey'}\}$
- ▷ **MaxNumPoints** (input_control) integer $\leadsto$ *integer*
Maximum number of contour points used for the computation (-1 for all points).
Default Value : -1
Restriction : $\text{MaxNumPoints} \geq 5$
- ▷ **MaxClosureDist** (input_control) real $\leadsto$ *real*
Maximum distance between the end points of a contour to be considered as 'closed'.
Default Value : 0.0
Restriction : $\text{MaxClosureDist} \geq 0.0$

- ▷ **ClippingEndPoints** (input_control) integer $\leadsto$ integer
Number of points at the beginning and at the end of the contours to be ignored for the fitting.
Default Value : 0
Restriction : ClippingEndPoints ≥ 0
- ▷ **VossTabSize** (input_control) integer $\leadsto$ integer
Number of circular segments used for the Voss approach.
Default Value : 200
Restriction : (VossTabSize ≥ 25) $\wedge$ (VossTabSize ≤ 5000)
- ▷ **Iterations** (input_control) integer $\leadsto$ integer
Maximum number of iterations (unused for 'fitzgibbon' and 'voss').
Default Value : 3
Restriction : Iterations ≥ 0
- ▷ **ClippingFactor** (input_control) real $\leadsto$ real
Clipping factor for the elimination of outliers (typical: 1.0 for '*Huber' and 2.0 for '*Tukey').
Default Value : 2.0
List of values : ClippingFactor $\in \{1.0, 1.5, 2.0, 2.5, 3.0\}$
Restriction : ClippingFactor > 0
- ▷ **Row** (output_control) ellipse.center.y(-array) $\leadsto$ real
Row coordinate of the center of the ellipse.
- ▷ **Column** (output_control) ellipse.center.x(-array) $\leadsto$ real
Column coordinate of the center of the ellipse.
- ▷ **Phi** (output_control) ellipse.angle.rad(-array) $\leadsto$ real
Orientation of the main axis [rad].
- ▷ **Radius1** (output_control) ellipse.radius1(-array) $\leadsto$ real
Length of the larger half axis.
- ▷ **Radius2** (output_control) ellipse.radius2(-array) $\leadsto$ real
Length of the smaller half axis.
- ▷ **StartPhi** (output_control) real(-array) $\leadsto$ real
Angle of the start point [rad].
- ▷ **EndPhi** (output_control) real(-array) $\leadsto$ real
Angle of the end point [rad].
- ▷ **PointOrder** (output_control) string(-array) $\leadsto$ string
point order along the boundary.
List of values : PointOrder $\in \{'positive', 'negative'\}$

Example

```

read_image (Image, 'caltab')
find_caltab (Image, Caltab, 'caltabBig.descr', 3, 112, 5)
reduce_domain (Image, Caltab, ImageReduced)
edges_sub_pix (ImageReduced, Edges, 'lanser2', 0.5, 20, 40)
select_contours_xld (Edges, EdgesClosed, 'closed', 0, 2.0, 0, 0)
select_contours_xld (EdgesClosed, EdgesMarks, 'length', 20, 80, 0, 0)
fit_ellipse_contour_xld (EdgesMarks, 'fitzgibbon', -1, 2, 0, 200, 3, 2.0,
                        Row, Column, Phi, Radius1, Radius2, StartPhi,
                        EndPhi, PointOrder)
gen_ellipse_contour_xld (EllMarks, Row, Column, Phi, Radius1, Radius2,
                        StartPhi, EndPhi, PointOrder, 1.5)
length_xld (EllMarks, Length).

```

Result

fit_ellipse_contour_xld returns 2 (H_MSG_TRUE) if all parameter values are correct, and ellipses could be fitted to the input contours. If the input is empty the behaviour can be set via **set_system('no.object.result', <Result>)**. If necessary, an exception is raised. If the parameter **ClippingFactor** is chosen too small, i.e., all points are classified as outliers, the error 3264 is raised.

Parallelization Information

fit_ellipse_contour_xld is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[gen_contours_skeleton_xld](#), [lines_gauss](#), [lines_facet](#), [edges_sub_pix](#),
[smooth_contours_xld](#)

Possible Successors

[gen_ellipse_contour_xld](#), [disp_ellipse](#), [get_points_ellipse](#)

See also

[fit_line_contour_xld](#)

Module

Sub-pixel operators

```
fit_line_contour_xld ( Contours : : Algorithm, MaxNumPoints,
ClippingEndPoints, Iterations, ClippingFactor : RowBegin, ColBegin,
RowEnd, ColEnd, Nr, Nc, Dist )
```

Approximation of XLD contours by line segments.

[fit_line_contour_xld](#) approximates the XLD contours [Contours](#) by line segments. It does not perform a segmentation of the input contours. Thus, one has to make sure that each contour corresponds to one and only one line segment. The operator returns for each contour the start point ([RowBegin](#), [ColBegin](#)), the end point ([RowEnd](#), [ColEnd](#)), and the regression line to the contour given by the normal vector ([Nr](#), [Nc](#)) of the line and its distance [Dist](#) from the origin, i.e., the line equation is given by $r \cdot Nr + c \cdot Nc - Dist = 0$.

The algorithm used for the fitting of lines can be selected via [Algorithm](#):

'regression' Standard 'least squares' line fitting.

'huber' Weighted 'least squares' line fitting, where the impact of outliers is decreased based on the approach of Huber.

'tukey' Weighted 'least squares' line fitting, where the impact of outliers is decreased based on the approach of Tukey.

'drop' Weighted 'least squares' line fitting, where outliers are eliminated.

'gauss' Weighted 'least squares' line fitting, where the impact of outliers is decreased based on the mean value and the standard deviation of the distances of all contour points from the approximating line.

For 'huber', 'tukey', and 'drop' a robust error statistics is used to estimate the standard deviation of the distances from the contour points *without* outliers from the approximating line. The parameter [ClippingFactor](#) (a scaling factor for the standard deviation) controls the amount of damping outliers: The smaller the value chosen for [ClippingFactor](#) the more outliers are detected. The detection of outliers is repeated. The parameter [Iterations](#) specifies the number of iterations. In the modus 'regression' this value is ignored.

To reduce the computational load, the fitting of lines can be restricted to a subset of the contour points: If a value other than -1 is assigned to [MaxNumPoints](#), only up to [MaxNumPoints](#) points - uniformly distributed over the contour - are used.

The start point and the end point of a line segment is determined by projecting the first and the last point of the corresponding contour to the approximating line. Due to artefacts in the pre-processing the start and end points of a contour might be faulty. Therefore, it is possible to exclude [ClippingEndPoints](#) points at the beginning and at the end of a contour from the line fitting. However, they are still used for the determination of the start point and the end point of the line segment.

Parameter

- ▷ **Contours** (input_object) xld_cont(-array) $\leadsto$ *Hobject*
Input contours.
- ▷ **Algorithm** (input_control) string $\leadsto$ *string*
Algorithm for the fitting of lines.
Default Value : 'tukey'
List of values : $Algorithm \in \{'regression', 'huber', 'tukey', 'gauss', 'drop'\}$
- ▷ **MaxNumPoints** (input_control) integer $\leadsto$ *integer*
Maximum number of contour points used for the computation (-1 for all points).
Default Value : -1
Restriction : $MaxNumPoints \geq 3$

- ▷ **ClippingEndPoints** (input_control) integer $\leadsto$ integer
Number of points at the beginning and at the end of the contours to be ignored for the fitting.
Default Value : 0
Restriction : ClippingEndPoints ≥ 0
- ▷ **Iterations** (input_control) integer $\leadsto$ integer
Maximum number of iterations (unused for 'regression').
Default Value : 5
Restriction : Iterations ≥ 0
- ▷ **ClippingFactor** (input_control) real $\leadsto$ real
Clipping factor for the elimination of outliers (typical: 1.0 for 'huber' and 'drop' and 2.0 for 'tukey').
Default Value : 2.0
List of values : ClippingFactor $\in \{1.0, 1.5, 2.0, 2.5, 3.0\}$
Restriction : ClippingFactor > 0
- ▷ **RowBegin** (output_control) line.begin.y(-array) $\leadsto$ real
Row coordinates of the starting points of the line segments.
- ▷ **ColBegin** (output_control) line.begin.x(-array) $\leadsto$ real
Column coordinates of the starting points of the line segments.
- ▷ **RowEnd** (output_control) line.end.y(-array) $\leadsto$ real
Row coordinates of the end points of the line segments.
- ▷ **ColEnd** (output_control) line.end.x(-array) $\leadsto$ real
Column coordinates of the end points of the line segments.
- ▷ **Nr** (output_control) number(-array) $\leadsto$ real
Line parameter: Row coordinate of the normal vector
- ▷ **Nc** (output_control) number(-array) $\leadsto$ real
Line parameter: Column coordinate of the normal vector
- ▷ **Dist** (output_control) number(-array) $\leadsto$ real
Line parameter: Distance of the line from the origin

Example

```
read_image (Image, 'mreut')
edges_sub_pix (Image, Edges, 'lanser2', 0.5, 20, 40)
gen_polygons_xld (Edges, Polygons, 'ramer', 2)
split_contours_xld (Polygons, Contours, 'polygon', 1, 5)
fit_line_contour_xld (Contours, 'regression', -1, 0, 5, 2, RowBegin,
                    ColBegin, RowEnd, ColEnd, Nr, Nc).
```

Result

fit_line_contour_xld returns 2 (H_MSG_TRUE) if all parameter values are correct, and line segments could be fitted to the input contours. If the input is empty the behaviour can be set via **set_system('no.object.result', <Result>)**. If necessary, an exception is raised. If the parameter **ClippingFactor** is chosen too small, i.e., all points are classified as outliers, the error 3264 is raised.

Parallelization Information

fit_line_contour_xld is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_contours_skeleton_xld](#), [lines_gauss](#), [lines_facet](#), [edges_sub_pix](#),
[smooth_contours_xld](#)

Possible Successors

[disp_line](#), [select_lines](#), [line_orientation](#)

See also

[regress_contours_xld](#), [get_regress_params_xld](#)

Module

Sub-pixel operators

get_contour_angle_xld (Contour : : AngleMode, CalcMode,
Lookaround : Angles)

Calculate the direction of an XLD contour for each contour point.

[get_contour_angle_xld](#) calculates for each point of the XLD contour [Contour](#) the direction of its tangent. Two modes for the output values can be chosen: By passing 'abs' for [AngleMode](#), the resulting angles are returned relative to the horizontal axis as angles between 0 and 2π (counter-clockwise). By passing 'rel', the absolute value of the difference of angles to the previous contour point is returned. In this case the range of values is between 0 and π .

There are three different ways of calculating the tangent direction ([CalcMode](#)) of the contour point i using the contour points in the interval from $i - \text{Lookaround}$ to $i + \text{Lookaround}$. By using 'range', the angle of the line segment between the first and last point of the interval is used. For 'mean', the average of all angles between consecutive points of the contour is used. Finally, for 'regress', the direction of the regression line (the least squares fit of a line to the contour points in the interval) is used. [Lookaround](#) is a measure of how strongly the contour is smoothed. The angles are returned in [Angles](#) in radians.

Parameter

- ▷ **Contour** (input_object) xld_cont $\leadsto$ *Hobject*
Input contour.
- ▷ **AngleMode** (input_control) string $\leadsto$ *string*
Return type of the angles.
Default Value : 'abs'
List of values : AngleMode $\in$ {'abs', 'rel'}
- ▷ **CalcMode** (input_control) string $\leadsto$ *string*
Method for computing the angles.
Default Value : 'range'
List of values : CalcMode $\in$ {'range', 'mean', 'regress'}
- ▷ **Lookaround** (input_control) integer $\leadsto$ *integer*
Number of points to take into account.
Default Value : 3
Restriction : Lookaround > 0
- ▷ **Angles** (output_control) real-array $\leadsto$ *real*
Direction of the tangent to the contour points.

Parallelization Information

[get_contour_angle_xld](#) is *reentrant* and processed *without parallelization*.

Possible Predecessors

[gen_contours_skeleton_xld](#), [lines_gauss](#), [lines_facet](#), [edges_sub_pix](#)

See also

[get_contour_xld](#), [get_contour_attrib_xld](#)

Module

Sub-pixel operators

get_contour_attrib_xld (Contour : : Name : Attrib)

Return point attribute values of an XLD contour.

[get_contour_attrib_xld](#) returns the values of the attribute [Name](#) of the XLD contour [Contour](#) in [Attrib](#). Attributes are additional values defined for each contour point, e.g., the direction perpendicular to the contour ('angle'). Operators which define this kind of attributes, e.g., [lines_gauss](#) and [edges_sub_pix](#), contain a description of the name and semantics of the defined values. [query_contour_attribs_xld](#) can be used to query which attributes are defined for a particular contour.

Parameter	
▷ Contour (input_object)	xld_cont $\leadsto$ <i>Hobject</i> Input XLD contour.
▷ Name (input_control)	string $\leadsto$ <i>string</i> Name of the attribute. Default Value : 'angle' Suggested values : Name $\in$ {'angle', 'edge_direction', 'width_right', 'width_left', 'response', 'contrast', 'asymmetry'}
▷ Attrib (output_control)	real-array $\leadsto$ <i>real</i> Attribute values.
Parallelization Information	
<code>get_contour_attrib_xld</code> is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Predecessors	
<code>lines_gauss</code> , <code>lines_facet</code> , <code>edges_sub_pix</code>	
See also	
<code>query_contour_attribs_xld</code> , <code>get_contour_global_attrib_xld</code> , <code>query_contour_global_attribs_xld</code>	
Module	
Sub-pixel operators	

get_contour_global_attrib_xld (Contour : : Name : Attrib)

Return global attributes values of an XLD contour.

`get_contour_global_attrib_xld` returns the values of the global attribute `Name` of the XLD contour `Contour` in `Attrib`. Global attributes are additional values defined for each contour, e.g., the normal vector of the regression line of a contour ('`regr_norm_row`' and '`regr_norm_col`'). Operators which define this kind of attributes contain a description of the name and semantics of the defined values. `query_contour_global_attribs_xld` can be used to query which attributes are defined for a particular contour.

Parameter	
▷ Contour (input_object)	xld_cont $\leadsto$ <i>Hobject</i> Input XLD contour.
▷ Name (input_control)	string(-array) $\leadsto$ <i>string</i> Name of the attribute. Default Value : 'regr_norm_row' Suggested values : Name $\in$ {'regr_norm_row', 'regr_norm_col', 'regr_mean_dist', 'regr_dev_dist', 'cont_approx'}
▷ Attrib (output_control)	real-array $\leadsto$ <i>real</i> Attribute values.
Parallelization Information	
<code>get_contour_global_attrib_xld</code> is <i>reentrant</i> and processed <i>without</i> parallelization.	
Possible Predecessors	
<code>lines_gauss</code> , <code>lines_facet</code> , <code>edges_sub_pix</code>	
See also	
<code>query_contour_global_attribs_xld</code> , <code>get_contour_attrib_xld</code> , <code>query_contour_attribs_xld</code>	
Module	
Sub-pixel operators	

get_regress_params_xld (*Contours* : : : *Length*, *Nx*, *Ny*, *Dist*, *Fpx*, *Fpy*, *Lpx*, *Lpy*, *Mean*, *Deviation*)

Return XLD contour parameters.

[get_regress_params_xld](#) returns the following parameters for all XLD contours given in [Contours](#):

- the number of contour points [Length](#),
- the coordinates [Nx](#) and [Ny](#) of the normal vector of the regression line (i.e., least-squares approximating line),
- the distance [Dist](#) of the regression line to the origin
- the sub-pixel precise coordinates [Fpx](#) and [Fpy](#) of the perpendicular projection of the start point of the contour onto the regression line,
- the sub-pixel precise coordinates [Lpx](#) and [Lpy](#) of the perpendicular projection of the end point of the contour onto the regression line,
- the mean of the Euclidian distance of the contour points from the regression line,
- the standard deviation of these distances.

Attention

Before the contour parameters can be returned by [get_regress_params_xld](#), the parameters of the regression line to the contour must be calculated by calling [regress_contours_xld](#).

Parameter

- ▷ **Contours** (input_object) xld_cont-array $\leadsto$ *Hobject*
Input XLD contours.
- ▷ **Length** (output_control) integer-array $\leadsto$ *integer*
Number of contour points.
- ▷ **Nx** (output_control) point.x-array $\leadsto$ *real*
X-coordinate of the normal vector of the regression line.
- ▷ **Ny** (output_control) point.y-array $\leadsto$ *real*
Y-coordinate of the normal vector of the regression line.
- ▷ **Dist** (output_control) number-array $\leadsto$ *real*
Distance of the regression line from the origin.
- ▷ **Fpx** (output_control) point.x-array $\leadsto$ *real*
X-coordinate of the projection of the start point of the contour onto the regression line.
- ▷ **Fpy** (output_control) point.y-array $\leadsto$ *real*
Y-coordinate of the projection of the start point of the contour onto the regression line.
- ▷ **Lpx** (output_control) point.x-array $\leadsto$ *real*
X-coordinate of the projection of the end point of the contour onto the regression line.
- ▷ **Lpy** (output_control) point.y-array $\leadsto$ *real*
Y-coordinate of the projection of the end point of the contour onto the regression line.
- ▷ **Mean** (output_control) real-array $\leadsto$ *real*
Mean distance of the contour points from the regression line.
- ▷ **Deviation** (output_control) real-array $\leadsto$ *real*
Standard deviation of the distances from the regression line.

Parallelization Information

[get_regress_params_xld](#) is *reentrant* and processed *without parallelization*.

Possible Predecessors

[regress_contours_xld](#)

Possible Successors

[disp_line](#), [select_lines](#), [line_orientation](#)

See also

[fit_line_contour_xld](#), [get_contour_global_attrib_xld](#),
[query_contour_global_attribs_xld](#), [get_contour_xld](#), [get_contour_attrib_xld](#),
[gen_contours_skeleton_xld](#), [lines_gauss](#), [lines_facet](#), [edges_sub_pix](#)

Module

Sub-pixel operators

info_parallels_xld (Parallels, Image : : : QualityMin, QualityMax, GrayMin, GrayMax, StandardMin, StandardMax)

Return information about the gray values of the area enclosed by XLD parallels.

[info_parallels_xld](#) calculates various gray value features of the area enclosed by the XLD parallels [Parallels](#). The input image [Image](#) is used to get the gray values needed for this. The algorithm used in this operator is very similar to the one used in [mod_parallels_xld](#). The operator returns ranges for the quality factor ([QualityMin](#) and [QualityMax](#)), the mean gray value ([GrayMin](#) and [GrayMax](#)), and the standard deviation with respect to the mean gray value ([StandardMin](#) and [StandardMax](#)).

This operator serves to determine appropriate thresholds for [mod_parallels_xld](#).

Parameter

- ▷ **Parallels** (input_object) xld_para-array $\leadsto$ *Hobject*
Input XLD Parallels.
- ▷ **Image** (input_object) image $\leadsto$ *Hobject* : byte
Corresponding gray value image.
- ▷ **QualityMin** (output_control) real $\leadsto$ *real*
Minimum quality factor.
- ▷ **QualityMax** (output_control) real $\leadsto$ *real*
Maximum quality factor.
- ▷ **GrayMin** (output_control) integer $\leadsto$ *integer*
Minimum mean gray value.
- ▷ **GrayMax** (output_control) integer $\leadsto$ *integer*
Maximum mean gray value.
- ▷ **StandardMin** (output_control) real $\leadsto$ *real*
Minimum standard deviation.
- ▷ **StandardMax** (output_control) real $\leadsto$ *real*
Maximum standard deviation.

Parallelization Information

[info_parallels_xld](#) is *reentrant* and processed *without parallelization*.

Possible Predecessors

[gen_parallels_xld](#)

Possible Successors

[mod_parallels_xld](#)

See also

[intensity](#), [min_max_gray](#)

Module

Sub-pixel operators

length_xld (XLD : : : Length)

Length of contours or polygons.

[length_xld](#) calculates the length of the contours or polygons [XLD](#). The length is calculated as the sum of the euclidian distances of successive points on the contour or polygon. If more than one contour or polygon is passed the results are stored as tuples in which the index of a value corresponds to the index of the respective contour or polygon in [XLD](#).

Parameter

- ▷ **XLD** (input_object) xld(-array) $\leadsto$ *Hobject*
Contours or polygons to be examined.
- ▷ **Length** (output_control) real(-array) $\leadsto$ *real*
Length of the contour or polygon.
Assertion : $\text{Length} \geq 0$

<i>Complexity</i>
Let n be the number of points of the contour or polygon. Then the run time is $O(n)$.
<i>Result</i>
<code>length_xld</code> returns 2 (H_MSG.TRUE) if the input is not empty. If the input is empty the behaviour can be set via <code>set_system(: 'no_object_result', <Result> :)</code> . If necessary, an exception is raised.
<i>Parallelization Information</i>
<code>length_xld</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).
<i>Possible Predecessors</i>
<code>gen_contours_skeleton_xld</code> , <code>smooth_contours_xld</code> , <code>gen_polygons_xld</code>
<i>See also</i>
<code>area_center_xld</code> , <code>moments_any_xld</code> , <code>moments_xld</code> , <code>contlength</code>
<i>Module</i>
Sub-pixel operators

local_max_contours_xld (Contours, Image : LocalMaxContours : MinPercent, MinDiff, Distance :)

Select XLD contours with a local maximum of gray values.

`local_max_contours_xld` selects XLD contours from the contours passed in `Contours`, which have a local maximum in gray values across the direction of the contour. In order to be selected, at least `MinPercent` of the contour points must have a local maximum perpendicular to the direction of the contour. The contours' direction is determined by fitting a regression line through five neighboring points of the contour. In order to decide whether there is a local maximum for a contour point, a gray value profile that is `Distance` points wide and perpendicular to the contour is computed on both sides of the contour. If the gray value at the contour point is at least `MinDiff` larger than the gray value at at least one point on each side of the profile, the contour point is labeled as a local maximum. The selected contours are returned in `LocalMaxContours`.

<i>Parameter</i>
▷ Contours (input_object) xld_cont-array $\leadsto$ <i>Hobject</i> XLD contours to be examined. ▷ Image (input_object) image $\leadsto$ <i>Hobject</i> : byte Corresponding gray value image. ▷ LocalMaxContours (output_object) xld_cont-array $\leadsto$ <i>Hobject</i> Selected contours. ▷ MinPercent (input_control) number $\leadsto$ <i>integer</i> / <i>real</i> Minimum percentage of maximum points. Default Value : 70 Suggested values : <code>MinPercent</code> $\in$ {60, 70, 75, 80, 85, 90, 95} Restriction : $(0.0 \leq \text{MinPercent}) \wedge (\text{MinPercent} \leq 100.0)$ ▷ MinDiff (input_control) integer $\leadsto$ <i>integer</i> Minimum amount by which the gray value at the maximum must be larger than in the profile. Default Value : 15 Suggested values : <code>MinDiff</code> $\in$ {5, 8, 10, 12, 15, 20} Restriction : $(0 \leq \text{MinDiff}) \wedge (\text{MinDiff} \leq 255)$ ▷ Distance (input_control) integer $\leadsto$ <i>integer</i> Maximum width of profile used to check for maxima. Default Value : 4 Suggested values : <code>Distance</code> $\in$ {2, 3, 4, 5, 6} Restriction : <code>Distance</code> ≥ 1

<i>Parallelization Information</i>
<code>local_max_contours_xld</code> is <i>reentrant</i> and processed <i>without parallelization</i> .
<i>Possible Predecessors</i>
<code>gen_contours_skeleton_xld</code> , <code>lines_gauss</code> , <code>lines_facet</code> , <code>edges_sub_pix</code>
<i>Possible Successors</i>
<code>gen_polygons_xld</code>

	See also	
smooth_contours_xld		
	Module	
Sub-pixel operators		

max_parallel_xld (ExtParallels : MaxPolygons : :)

Join modified XLD parallels lying on the same polygon.

[max_parallel_xld](#) joins all modified XLD parallels in [ExtParallels](#) into a polygon if they lie on the same original polygon segment. This means that polygons exhibiting parallelism and enclosing homogeneous areas in several places are joined into one long polygon (from the first parallel line to the last parallel line). The resulting polygons are returned in [MaxPolygons](#).

	Parameter	
▷ ExtParallels (input_object)	xld_ext_para-array	↪ Hobject
	Extended XLD parallels.	
▷ MaxPolygons (output_object)	xld_poly-array	↪ Hobject
	Maximally extended parallels.	

	Parallelization Information	
max_parallel_xld	is <i>reentrant</i> and processed <i>without</i> parallelization.	

	Possible Predecessors	
mod_parallel_xld		

	Possible Successors	
get_polygon_xld , get_lines_xld		

	Module	
Sub-pixel operators		

moments_any_xld (XLD : : Mode, PointOrder, Area, CenterRow, CenterCol, P, Q : M)

Arbitrary geometric moments of contours or polygons.

[moments_any_xld](#) calculates arbitrary moments **M** of the region enclosed by the contours or polygons **XLD**. The moments are computed by applying Green's theorem using only the points on the contour or polygon, i.e., no region is generated explicitly for the purpose of calculating the features. It is assumed that the contours or polygons are closed. If this is not the case [moments_any_xld](#) will artificially close the contours or polygons. The computed moments are normalized depending on the desired mode **Mode**:

'unnormalized': No normalization. Let R be the enclosed image region. Then the computed moment $M_{p,q}$ is equivalent to

$$M_{p,q} = \int_R r^p c^q dr dc$$

'unnormalized_central': Shift the region by its centroid $[\bar{r}, \bar{c}] = [\text{CenterRow}, \text{CenterCol}]$:

$$M_{p,q} = \int_R (r - \bar{r})^p (c - \bar{c})^q dr dc$$

'normalized': Normalization by the area $A = \text{Area}$ of the enclosed image region:

$$M_{p,q} = \frac{1}{A} \int_R r^p c^q dr dc$$

'central': Normalization by the area $A = \text{Area}$ of the enclosed image region and a shift of the region by it's centroid $[\bar{r}, \bar{c}] = [\text{CenterRow}, \text{CenterCol}]$:

$$M_{p,q} = \frac{1}{A} \int_R (r - \bar{r})^p (c - \bar{c})^q dr dc$$

For the normalization of the moments three specific moments are used: The area [Area](#) of the enclosed image region and the coordinates [CenterRow](#), [CenterCol](#) of it's centroid, see [area_center_xld](#). In addition to that [moments_any_xld](#) expects information about the point order of the input contours/polygons in [PointOrder](#), see [area_center_xld](#) again. If more than one contour or polygon is passed, **M** contains all desired moments of the first contour/polygon followed by all the moments of the second contour/polygon and so forth.

Parameter	
▷ XLD (input_object)	xld(-array) $\leadsto$ <i>Hobject</i> Contours or polygons to be examined.
▷ Mode (input_control)	string $\leadsto$ <i>string</i> computation mode. Default Value : 'unnormalized' Suggested values : Mode $\in \{ \text{'unnormalized'}, \text{'unnormalized\_central'}, \text{'normalized'}, \text{'central'} \}$
▷ PointOrder (input_control)	string(-array) $\leadsto$ <i>string</i> point order along the boundary. Default Value : 'positive' Suggested values : PointOrder $\in \{ \text{'positive'}, \text{'negative'} \}$
▷ Area (input_control)	real(-array) $\leadsto$ <i>real</i> Area enclosed by the contour or polygon.
▷ CenterRow (input_control)	point.y(-array) $\leadsto$ <i>real</i> Row coordinate of the centroid.
▷ CenterCol (input_control)	point.x(-array) $\leadsto$ <i>real</i> Column coordinate of the centroid.
▷ P (input_control)	point.x(-array) $\leadsto$ <i>integer</i> First index of the desired moments M[P,Q]. Default Value : 1
▷ Q (input_control)	point.x(-array) $\leadsto$ <i>integer</i> Second index of the desired moments M[P,Q]. Default Value : 1
▷ M (output_control)	real(-array) $\leadsto$ <i>real</i> The computed moments.

Complexity

Let n be the number of points of the contour or polygon. Then the run time is $O(n)$.

Result

[moments_any_xld](#) returns 2 (H_MSG_TRUE) if the input is not empty. If the input is empty the behaviour can be set via [set_system\(::'no_object_result', <Result>:\)](#). If necessary, an exception is raised.

Parallelization Information

[moments_any_xld](#) is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[area_center_xld](#), [gen_contours_skeleton_xld](#), [smooth_contours_xld](#), [gen_polygons_xld](#)

Alternatives

[moments_xld](#)

See also

[moments_xld](#), [area_center_xld](#), [moments_region_2nd](#), [area_center](#)

Module

Sub-pixel operators

moments_xld (XLD : : : M11, M20, M02)

Geometric moments $M20$, $M02$, and $M11$ of contours or polygons.

moments_xld calculates the moments (**M20**, **M02**, and **M11**) of the region enclosed by the contours or polygons **XLD**. See **moments_region_2nd** for the definition of these features. The moments are computed by applying Green's theorem using only the points on the contour or polygon, i.e., no region is generated explicitly for the purpose of calculating the features. It is assumed that the contours or polygons are closed. If this is not the case **moments_xld** will artificially close the contours or polygons. If more than one contour or polygon is passed the results are stored as tuples in which the index of a value corresponds to the index of the respective contour or polygon in **XLD**.

Parameter

- ▷ **XLD** (input_object) xld(-array) $\leadsto$ *Hobject*
Contours or polygons to be examined.
- ▷ **M11** (output_control) real(-array) $\leadsto$ *real*
Mixed second order moment.
- ▷ **M20** (output_control) real(-array) $\leadsto$ *real*
Second order moment along the row axis.
- ▷ **M02** (output_control) real(-array) $\leadsto$ *real*
Second order moment along the column axis.

Complexity

Let n be the number of points of the contour or polygon. Then the run time is $O(n)$.

Result

moments_xld returns 2 (H_MSG_TRUE) if the input is not empty. If the input is empty the behaviour can be set via **set_system**(::'no_object_result', <Result>:). If necessary, an exception is raised.

Parallelization Information

moments_xld is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

gen_contours_skeleton_xld, **smooth_contours_xld**, **gen_polygons_xld**

Alternatives

moments_any_xld

See also

moments_any_xld, **area_center_xld**, **moments_region_2nd**, **area_center**

Module

Sub-pixel operators

orientation_xld (XLD : : : Phi)

Orientation of a contour.

The operator **orientation_xld** calculates the orientation of the input contour. The operator is based on **elliptic_axis_xld**. In addition, the contour point with maximum distance to the center of gravity is calculated. If the column coordinate of this point is less than the column coordinate of the center of gravity the value of π is added to the angle. The input contour must not intersect itself, otherwise the resulting parameter is not meaningful. If the input contour is not closed it will be closed automatically.

If more than one contour is passed the results are stored in tuples, the index of a value in the tuple corresponding to the index of a contour in the input.

Parameter

- ▷ **XLD** (input_object) xld(-array) $\leadsto$ *Hobject*
Contour(s) to be examined.
- ▷ **Phi** (output_control) real(-array) $\leadsto$ *real*
Orientation of the contour(s) (radians).
Assertion : $((-\pi/2) < \text{Phi}) \wedge (\text{Phi} \leq ((3 \cdot \pi)/2))$

Result

The operator **orientation_xld** returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input contours available) is set via the operator **set_system**('no_object_result', <Result>). If necessary an exception is raised.

Parallelization Information

`orientation_xld` is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

`gen_contours_skeleton_xld`, `edges_sub_pix`, `threshold_sub_pix`,
`gen_contour_polygon_xld`

Alternatives

`elliptic_axis_xld`, `smallest_rectangle2_xld`

See also

`moments_region2nd`

Module

Sub-pixel operators

query_contour_attribs_xld (*Contour* : : : *Attribs*)

Return the names of the defined attributes of an XLD contour.

`query_contour_attribs_xld` returns the names of the defined attributes of an XLD contour *Contour* in *Attribs*. Attributes are additional values defined for each contour point, e.g., the direction perpendicular to the contour (*'angle'*). Operators which define this kind of attributes contain a description of the name and semantics of the defined values. `get_contour_attrib_xld` can be used to access the values of a particular attribute.

Parameter

- ▷ **Contour** (input_object) xld_cont $\leadsto$ *Hobject*
Input contour.
- ▷ **Attribs** (output_control) string-array $\leadsto$ *string*
List of the defined contour attributes.

Parallelization Information

`query_contour_attribs_xld` is *reentrant* and processed *without* parallelization.

Possible Predecessors

`lines_gauss`, `lines_facet`, `edges_sub_pix`

See also

`get_contour_attrib_xld`, `get_contour_global_attrib_xld`,
`query_contour_global_attribs_xld`

Module

Sub-pixel operators

query_contour_global_attribs_xld (*Contour* : : : *Attribs*)

Return the names of the defined global attributes of an XLD contour.

`query_contour_global_attribs_xld` returns the names of the globally defined attributes of an XLD contour *Contour* in *Attribs*. Global attributes are additional values defined for each contour, e.g., the normal vector of the regression line of a contour (*'regr_norm_row'* and *'regr_norm_col'*). Operators which define this kind of attributes contain a description of the name and semantics of the defined values. `get_contour_global_attrib_xld` can be used to access the value of a particular attribute.

Parameter

- ▷ **Contour** (input_object) xld_cont $\leadsto$ *Hobject*
Input contour.
- ▷ **Attribs** (output_control) string-array $\leadsto$ *string*
List of the defined global contour attributes.

Parallelization Information

`query_contour_global_attribs_xld` is *reentrant* and processed *without* parallelization.

<i>Possible Predecessors</i>
lines_gauss , lines_facet , edges_sub_pix
<i>See also</i>
get_contour_global_attrib_xld , get_contour_attrib_xld , query_contour_attribs_xld
<i>Module</i>
Sub-pixel operators

select_contours_xld (<i>Contours</i> : SelectedContours : <i>Feature</i> , <i>Min1</i> , <i>Max1</i> , <i>Min2</i> , <i>Max2</i> :)

Select XLD contours according to several features.

[select_contours_xld](#) selects XLD contours from the input contours [Contours](#) according to the following features (parameter [Feature](#)):

- '*contour_length*' all contours whose length is smaller than [Min1](#) or larger than [Max1](#) are not returned ([Min2](#) and [Max2](#) have no influence here).
- '*maximum_extent*' all contours whose maximum extent (as measured by their eight extremal points in row and column direction, according to Haralick und Shapiro: Computer and Robot Vision, Addison-Wesley 1992, chapter 3.2) is smaller than [Min1](#) or larger than [Max1](#) are not returned ([Min2](#) and [Max2](#) have no influence here).
- '*direction*' only contours for which the direction of the regression line is between [Min1](#) and [Max1](#) (in radians, counter-clockwise) are returned. [Min1](#) and [Max1](#) are mapped to the range of $[0, 2 \cdot \text{PI}]$. ([Min2](#) and [Max2](#) have no influence here).
- '*curvature*' only contours for which the mean distance from the regression line lies between [Min1](#) and [Max1](#), and for which the standard deviation of the distances is between [Min2](#) and [Max2](#) are returned.
- '*closed*' only contours for which the distance between their start point and their end point is less or equal [Max1](#) pixels are returned.
- '*open*' only contours for which the distance between their start point and their end point is greater than [Max1](#) pixels are returned.

If [Min1](#) = [Max1](#) = 0 or [Min2](#) = [Max2](#) = 0 is used for the selection according to curvature, the respective feature has no influence on the selection.

<i>Attention</i>
Before contour can be filtered by select_contours_xld according to 'direction' or 'curvature', the parameters of the regression lines to the contours must be calculated by calling regress_contours_xld .
<i>Parameter</i>

- ▷ **Contours** (input_object) xld_cont-array $\leadsto$ *Hobject*
Input XLD contours.
- ▷ **SelectedContours** (output_object) xld_cont-array $\leadsto$ *Hobject*
Output XLD contours.
- ▷ **Feature** (input_control) string $\leadsto$ *string*
Feature to select contours with.
Default Value : 'contour_length'
List of values : *Feature* $\in$ {'contour_length', 'maximum_extent', 'direction', 'curvature', 'closed', 'open'}
- ▷ **Min1** (input_control) real $\leadsto$ *real*
Lower threshold.
Default Value : 0.5
- ▷ **Max1** (input_control) real $\leadsto$ *real*
Upper threshold.
Default Value : 200.0
- ▷ **Min2** (input_control) real $\leadsto$ *real*
Lower threshold.
Default Value : -0.5
- ▷ **Max2** (input_control) real $\leadsto$ *real*
Upper threshold.
Default Value : 0.5

Parallelization Information

[select_contours_xld](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[regress_contours_xld](#)

See also

[get_contour_xld](#), [get_contour_attrib_xld](#), [gen_contours_skeleton_xld](#), [lines_gauss](#), [lines_facet](#), [edges_sub_pix](#), [get_regress_params_xld](#), [get_contour_global_attrib_xld](#), [query_contour_global_attribs_xld](#)

References

R. Haralick, L. Shapiro: "Computer and Robot Vision" Vol. 1; Kapitel 3.2, Addison-Wesley 1992

Module

Sub-pixel operators

select_shape_xld (XLD : SelectedXLD : Features, Operation, Min, Max :)

Select contours using shape features.

The operator [select_shape_xld](#) chooses regions according to their shape. For each input region in [XLD](#) the indicated features ([Features](#)) are calculated. If each ([Operation](#) = 'and') or at least one ([Operation](#) = 'or') of the calculated features is within the limits ([Min](#),[Max](#)) the contour is copied into the output.

Condition: $\text{Min}_i \leq \text{Features}_i(\text{Object}) \leq \text{Max}_i$

Possible values for [Features](#):

- 'area': Area of the contour
- 'row': Row index of the center of gravity
- 'column': Column index of the center of gravity
- 'width': Width of the contour
- 'height': Height of the contour
- 'row1': Row coordinate of upper left corner
- 'column1': Column coordinate of upper left corner
- 'row2': Row coordinate of lower right corner
- 'column2': Column coordinate of lower right corner
- 'circularity': Circularity (see [circularity_xld](#))
- 'compactness': Compactness (see [compactness_xld](#))
- 'contlength': Total length of contour (see [length_xld](#))
- 'convexity': Convexity (see [convexity_xld](#))
- 'ra': Major radius of the equivalent ellipse (see [elliptic_axis_xld](#))
- 'rb': Minor radius of the equivalent ellipse (see [elliptic_axis_xld](#))
- 'phi': Orientation of the equivalent ellipse (see [elliptic_axis_xld](#))
- 'anisometry': Anisometry (see [eccentricity_xld](#))
- 'bulkiness': Bulkiness (see [eccentricity_xld](#))
- 'struct_factor': Structure factor (see [eccentricity_xld](#))
- 'outer_radius': Radius of smallest enclosing circle (see [smallest_circle_xld](#))
- 'max_diameter': Maximum diameter of the contour (see [diameter_xld](#))
- 'orientation': Orientation of the contour (see [orientation_xld](#))
- 'rect2_phi': Orientation of the smallest surrounding rectangle (see [smallest_rectangle2_xld](#))
- 'rect2_len1': Half the length of the smallest surrounding rectangle (see [smallest_rectangle2_xld](#))
- 'rect2_len2': Half the width of the smallest surrounding rectangle (see [smallest_rectangle2_xld](#))

'moments_m11', 'moments_m20', 'moments_m02': Geometric moments of the contour (see [moments_region_2nd](#))

If only one feature ([Features](#)) is used the value of [Operation](#) is meaningless. Several features are processed in the sequence in which they are entered. The use of some features requires that the input contour must not intersect itself, otherwise the results are not meaningful.

<i>Parameter</i>	
▷ XLD (input_object)	xld-array $\leadsto$ <i>Hobject</i> Contours to be examined.
▷ SelectedXLD (output_object)	xld-array $\leadsto$ <i>Hobject</i> Contours fulfilling the condition(s).
▷ Features (input_control)	string(-array) $\leadsto$ <i>string</i> Shape features to be checked. Default Value : 'area' List of values : Features $\in$ {'area', 'row', 'column', 'width', 'height', 'row1', 'column1', 'row2', 'column2', 'circularity', 'compactness', 'contlength', 'convexity', 'ra', 'rb', 'phi', 'anisometry', 'bulkiness', 'struct_factor', 'outer_radius', 'max_diameter', 'orientation', 'rect2_phi', 'rect2_len1', 'rect2_len2', 'moments_m11', 'moments_m20', 'moments_m02'}
▷ Operation (input_control)	string $\leadsto$ <i>string</i> Operation type between the individual features. Default Value : 'and' List of values : Operation $\in$ {'and', 'or'}
▷ Min (input_control)	real(-array) $\leadsto$ <i>real / integer / string</i> Lower limits of the features or 'min'. Default Value : 150.0 Typical range of values : $0.0 \leq \text{Min} \leq 99999.0$ Minimum Increment : 0.001 Recommended Increment : 1.0
▷ Max (input_control)	real(-array) $\leadsto$ <i>real / integer / string</i> Upper limits of the features or 'max'. Default Value : 99999.0 Typical range of values : $0.0 \leq \text{Max} \leq 99999.0$ Minimum Increment : 0.001 Recommended Increment : 1.0 Restriction : Max $\geq$ Min

Result

The operator [select_shape_xld](#) returns the value 2 (H_MSG_TRUE) if the input is not empty. The behavior in case of empty input (no input objects available) is set via the operator [set_system](#) ('no_object_result', <Result>). If necessary an exception is raised.

Parallelization Information

[select_shape_xld](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_contours_skeleton_xld](#), [edges_sub_pix](#), [threshold_sub_pix](#),
[gen_contour_polygon_xld](#)

Possible Successors

[shape_trans_xld](#), [count_obj](#)

See also

[area_center_xld](#), [circularity_xld](#), [compactness_xld](#), [convexity_xld](#),
[elliptic_axis_xld](#), [eccentricity_xld](#), [smallest_circle_xld](#),
[smallest_rectangle1_xld](#), [smallest_rectangle2_xld](#), [diameter_xld](#), [orientation_xld](#),
[moments_xld](#), [select_obj](#)

Module

Sub-pixel operators

smallest_circle_xld (XLD : : : Row, Column, Radius)

Smallest enclosing circle of a contour.

The operator `smallest_circle_xld` determines the smallest enclosing circle of a contour, i.e., the circle with the smallest area of all circles containing the contour. For this circle the center (`Row`, `Column`) and the radius (`Radius`) are calculated.

If several contours are passed in `XLD` corresponding tuples are returned as output parameter. In case of empty contours all parameters have the value 0.0 if no other behavior was set (see `set_system`).

Parameter

- ▷ **XLD** (input_object) xld(-array) $\leadsto$ *Hobject*
Contour(s) to be examined.
 - ▷ **Row** (output_control) circle.center.y(-array) $\leadsto$ *real*
Row coordinate of the center of the enclosing circle.
 - ▷ **Column** (output_control) circle.center.x(-array) $\leadsto$ *real*
Column coordinate of the center of the enclosing circle.
 - ▷ **Radius** (output_control) circle.radius(-array) $\leadsto$ *real*
Radius of the enclosing circle.
- Assertion :** Radius ≥ 0

Complexity

If N is the number of contour points then the runtime complexity is $O(N * \ln(N))$.

Result

The operator `smallest_circle_xld` returns the value 2 (`H_MSG_TRUE`) if the input is not empty. If the input is empty the behavior can be set via `set_system(:, 'no_object_result', <Result>:)`. If necessary, an exception is raised.

Parallelization Information

`smallest_circle_xld` is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

`gen_contours_skeleton_xld`, `edges_sub_pix`, `threshold_sub_pix`,
`gen_contour_polygon_xld`

Possible Successors

`gen_ellipse_contour_xld`

Alternatives

`smallest_rectangle1_xld`, `smallest_rectangle2_xld`

See also

`elliptic_axis_xld`, `smallest_rectangle1_xld`, `smallest_rectangle2_xld`,
`shape_trans_xld`

Module

Sub-pixel operators

smallest_rectangle1_xld (XLD : : : Row1, Column1, Row2, Column2)

Enclosing rectangle parallel to the coordinate axes.

The operator `smallest_rectangle1_xld` calculates the enclosing rectangle for each input contour (parallel to the coordinate axes). The enclosing rectangle is described by the coordinates of the corner pixels (`Row1`, `Column1`, `Row2`, `Column2`)

If more than one contour is passed in `XLD`, the results are stored in tuples, the index of a value in the tuple corresponding to the index of a contour in the input. In case of an empty contour all parameters have the value 0 if no other behavior was set (see `set_system`).

Attention

In case of empty contours the result of `Row1`, `Column1`, `Row2` and `Column2` (all are 0) can lead to confusion.

Parameter
▷ XLD (input_object) xld(-array) $\leadsto$ <i>Hobject</i> Contour(s) to be examined. ▷ Row1 (output_control) rectangle.origin.y(-array) $\leadsto$ <i>real</i> Row coordinate of upper left corner point of the enclosing rectangle. ▷ Column1 (output_control) rectangle.origin.x(-array) $\leadsto$ <i>real</i> Column coordinate of upper left corner point of the enclosing rectangle. ▷ Row2 (output_control) rectangle.corner.y(-array) $\leadsto$ <i>real</i> Row coordinate of lower right corner point of the enclosing rectangle. ▷ Column2 (output_control) rectangle.corner.x(-array) $\leadsto$ <i>real</i> Column coordinate of lower right corner point of the enclosing rectangle.
Complexity
If N is the number of contour points, the runtime complexity is $O(N)$.
Result
<code>smallest_rectangle1_xld</code> returns 2 (H_MSG_TRUE) if the input is not empty. If the input is empty the behavior can be set via <code>set_system(':', 'no_object_result', <Result>:)</code> . If necessary, an exception is raised.
Parallelization Information
<code>smallest_rectangle1_xld</code> is <i>reentrant</i> and automatically <i>parallelized</i> (on tuple level).
Possible Predecessors
<code>gen_contours_skeleton_xld</code> , <code>edges_sub_pix</code> , <code>threshold_sub_pix</code> , <code>gen_contour_polygon_xld</code>
Possible Successors
<code>gen_polygons_xld</code>
Alternatives
<code>smallest_rectangle2_xld</code> , <code>shape_trans_xld</code>
See also
<code>shape_trans_xld</code> , <code>smallest_rectangle2_xld</code> , <code>smallest_circle_xld</code> , <code>elliptic_axis_xld</code> , <code>area_center_xld</code>
Module
Sub-pixel operators

smallest_rectangle2_xld (XLD : : : Row, Column, Phi, Length1, Length2)

Smallest enclosing rectangle with arbitrary orientation.

The operator `smallest_rectangle2_xld` determines the smallest enclosing rectangle of a contour, i.e., the rectangle with the smallest area of all rectangles containing the contour. For this rectangle the center, the inclination, and the two radii are calculated.

If more than one contour is passed in `XLD` the results are stored in tuples, the index of a value in the tuple corresponding to the index of a contour in the input. In case of an empty contour all parameters have the value 0.0 if no other behavior was set (see `set_system`).

Parameter
▷ XLD (input_object) xld(-array) $\leadsto$ <i>Hobject</i> Contour(s) to be examined. ▷ Row (output_control) rectangle2.center.y(-array) $\leadsto$ <i>real</i> Row coordinate of the center point of the enclosing rectangle. ▷ Column (output_control) rectangle2.center.x(-array) $\leadsto$ <i>real</i> Column coordinate of the center point of the enclosing rectangle. ▷ Phi (output_control) rectangle2.angle.rad(-array) $\leadsto$ <i>real</i> Orientation of the enclosing rectangle (arc measure) Assertion : $((-\pi/2) < \text{Phi}) \wedge (\text{Phi} \leq (\pi/2))$

- ▷ **Length1** (output_control) rectangle2.hwidth(-array) $\leadsto$ *real*
First radius (half length) of the enclosing rectangle.
Assertion : $\text{Length1} \geq 0.0$
- ▷ **Length2** (output_control) rectangle2.hheight(-array) $\leadsto$ *real*
Second radius (half width) of the enclosing rectangle.
Assertion : $(\text{Length2} \geq 0.0) \wedge (\text{Length2} \leq \text{Length1})$

Complexity

If N is the number of contour points and C is the number of points in the convex hull, the runtime complexity is $O(N * \ln(N) + C^2)$.

Result

[smallest_rectangle2_xld](#) returns 2 (H_MSG.TRUE) if the input is not empty. If the input is empty the behavior can be set via [set_system\(:, 'no_object_result', <Result>:\)](#). If necessary, an exception is raised.

Parallelization Information

[smallest_rectangle2_xld](#) is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

[gen_contours_skeleton_xld](#), [edges_sub_pix](#), [threshold_sub_pix](#),
[gen_contour_polygon_xld](#)

Possible Successors

[gen_polygons_xld](#)

Alternatives

[smallest_rectangle1](#), [shape_trans_xld](#)

See also

[smallest_rectangle1](#), [smallest_circle](#), [elliptic_axis_xld](#)

Module

Sub-pixel operators

17.4 Transformation

add_noise_white_contour_xld (
Contours : NoisyContours : NumRegrPoints, Amp :)

Add noise to XLD contours.

[add_noise_white_contour_xld](#) adds noise to the input XLD contours [Contours](#) and returns the noisy contours in [NoisyContours](#). For each point along the original contours the local regression line (i.e., a least-squares approximating line) based on [NumRegrPoints](#) neighboring points is determined. Then the point is shifted perpendicular to this line. The length of the shifts corresponds to white noise, equally distributed in the interval $[-\text{Amp}, \text{Amp}]$ generated by using the C function “drand48” with an initial time dependent seed.

Parameter

- ▷ **Contours** (input_object) xld_cont(-array) $\leadsto$ *Hobject*
Original contours.
- ▷ **NoisyContours** (output_object) xld_cont(-array) $\leadsto$ *Hobject*
Noisy contours.
- ▷ **NumRegrPoints** (input_control) integer $\leadsto$ *integer*
Number of points used to calculate the regression line.
Default Value : 5
Suggested values : $\text{NumRegrPoints} \in \{3, 5, 7, 9\}$
Restriction : $(\text{NumRegrPoints} \geq 3) \wedge \text{odd}(\text{NumRegrPoints})$
- ▷ **Amp** (input_control) real $\leadsto$ *real*
Maximum amplitude of the added noise (equally distributed in $[-\text{Amp}, \text{Amp}]$).
Default Value : 1.0
Suggested values : $\text{Amp} \in \{0.25, 0.5, 1.0, 1.5, 2.0, 3.0, 4.0, 5.0, 10.0\}$
Restriction : $\text{Amp} > 0$

Example

```
draw_ellipse(WindowHandle, Row, Column, Phi, Radius1, Radius2)
gen_ellipse_contour_xld(Ellipse, Row, Column, Phi, Radius1, Radius2, 0, 6.28319,
                        'positive')
add_noise_white_contour_xld(Ellipse, NoisyEllipse, 5, 0.5)
fit_ellipse_contour_xld (NoisyEllipse, 'fitzgibbon', -1, 2, 0, 200, 3, 2.0,
                        ERow, EColumn, EPhi, ERadius1, ERadius2, EStartPhi,
                        EEndPhi, 'EPointOrder').
```

Result

[add_noise_white_contour_xld](#) returns 2 (H_MSG_TRUE) if all parameter values are correct. If the input is empty the behaviour can be set via [set_system\('no_object_result', <Result>\)](#). If necessary, an exception is raised.

Parallelization Information

[add_noise_white_contour_xld](#) is *reentrant*, *local*, and processed *without* parallelization.

Possible Predecessors

[gen_contours_skeleton_xld](#), [lines_gauss](#), [lines_facet](#), [edges_sub_pix](#),
[gen_ellipse_contour_xld](#)

Possible Successors

[smooth_contours_xld](#)

See also

[smooth_contours_xld](#), [add_noise_white](#)

Module

Sub-pixel operators

affine_trans_contour_xld (Contours : ContoursAffinTrans : HomMat2D :)

Apply an arbitrary affine 2D transformation to XLD contours.

[affine_trans_contour_xld](#) applies an arbitrary affine 2D transformation, i.e., scaling, rotation, translation, and slant (skewing), to the XLD contours given in [Contours](#) and returns the transformed contours in [ContoursAffinTrans](#). The affine transformation is described by the homogeneous transformation matrix given in [HomMat2D](#), which can be created using the operators [hom_mat2d_identity](#), [hom_mat2d_scale](#), [hom_mat2d_rotate](#), [hom_mat2d_translate](#), etc., or be the result of operators like [vector_angle_to_rigid](#).

The components of the homogeneous transformation matrix are interpreted as follows: The *row* coordinate of the image corresponds to the *x* coordinate of the matrix, while the *column* coordinate of the image corresponds to the *y* coordinate of the matrix. This is necessary to obtain a right-handed coordinate system for the image. In particular, this assures that rotations are performed in the correct direction. Note that the *(x,y)* order of the matrices quite naturally corresponds to the usual (row,column) order for coordinates in the image.

Parameter

- ▷ **Contours** (input_object) xld_cont(-array) $\leadsto$ *Hobject*
Input XLD contours.
- ▷ **ContoursAffinTrans** (output_object) xld_cont(-array) $\leadsto$ *Hobject*
Transformed XLD contours.
- ▷ **HomMat2D** (input_control) affine2d-array $\leadsto$ *real*
Input transformation matrix.
Number of elements : 6

Result

[affine_trans_contour_xld](#) returns 2 (H_MSG_TRUE) if all parameter values are correct. If necessary, an exception is raised.

Parallelization Information

[affine_trans_contour_xld](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[hom_mat2d_identity](#), [hom_mat2d_translate](#), [hom_mat2d_rotate](#), [hom_mat2d_scale](#)

See also

[affine_trans_image](#), [affine_trans_region](#)

Module

Sub-pixel operators

affine_trans_polygon_xld (
 Polygons : PolygonsAffinTrans : HomMat2D :)
Apply an arbitrary affine transformation to XLD polygons.

[affine_trans_polygon_xld](#) applies an arbitrary affine transformation, i.e., scaling, rotation, translation, and slant (skewing), to the XLD polygons given in [Polygons](#) and returns the transformed polygons in [PolygonsAffinTrans](#). The affine transformation is described by the homogeneous transformation matrix given in [HomMat2D](#). This matrix can be created using the operators [hom_mat2d_identity](#), [hom_mat2d_scale](#), [hom_mat2d_rotate](#), [hom_mat2d_translate](#), etc., or be the result of operators like [vector_angle_to_rigid](#).

The components of the homogeneous transformation matrix are interpreted as follows: The *row* coordinate of the image corresponds to the *x* coordinate of the matrix, while the *column* coordinate of the image corresponds to the *y* coordinate of the matrix. This is necessary to obtain a right-handed coordinate system for the image. In particular, this assures that rotations are performed in the correct direction. Note that the (x,y) order of the matrices quite naturally corresponds to the usual (row,column) order for coordinates in the image.

Attention

The XLD contours that are possibly referenced by [Polygons](#) are neither transformed nor stored with the output polygons, since this is generally impossible without creating inconsistencies for the attributes of the XLD contours. Hence, operators that access the contours associated with a polygon, e.g., [split_contours_xld](#) will not work correctly.

Parameter

- ▷ **Polygons** (input_object) xld_poly(-array) $\leadsto$ *Hobject*
Input XLD polygons.
- ▷ **PolygonsAffinTrans** (output_object) xld_poly(-array) $\leadsto$ *Hobject*
Transformed XLD polygons.
- ▷ **HomMat2D** (input_control) affine2d-array $\leadsto$ *real*
Input transformation matrix.
Number of elements : 6

Result

[affine_trans_polygon_xld](#) returns 2 (H.MSG.TRUE) if all parameter values are correct. If necessary, an exception is raised.

Parallelization Information

[affine_trans_polygon_xld](#) is *reentrant* and automatically *parallelized* (on tuple level).

Possible Predecessors

[hom_mat2d_identity](#), [hom_mat2d_translate](#), [hom_mat2d_rotate](#), [hom_mat2d_scale](#)

See also

[affine_trans_image](#), [affine_trans_region](#), [affine_trans_contour_xld](#)

Module

Sub-pixel operators

clip_contours_xld (Contours : ClippedContours : Row1, Column1, Row2,
 Column2 :)
Clip an XLD contour.

clip_contours_xld clips all XLD contours given in **Contours**, i.e., only contour points contained in the rectangle given by **Row1**, **Column1**, **Row2**, and **Column2** are returned on output. If necessary, contours are split, and several new contours are produced. The resulting contours are returned in **ClippedContours**.

Parameter	
▷ Contours (input_object)	xld_cont-array $\leadsto$ Hobject Contours to be clipped.
▷ ClippedContours (output_object)	xld_cont(-array) $\leadsto$ Hobject Clipped contours.
▷ Row1 (input_control)	rectangle.origin.y $\leadsto$ integer Row coordinate of the upper left corner of the clip rectangle. Default Value : 0 Suggested values : Row1 $\in$ {0, 500, 1000, 1500, 2000}
▷ Column1 (input_control)	rectangle.origin.x $\leadsto$ integer Column coordinate of the upper left corner of the clip rectangle. Default Value : 0 Suggested values : Column1 $\in$ {0, 500, 1000, 1500, 2000}
▷ Row2 (input_control)	rectangle.corner.y $\leadsto$ integer Row coordinate of the lower right corner of the clip rectangle. Default Value : 512 Suggested values : Row2 $\in$ {512, 1024, 1536, 2048}
▷ Column2 (input_control)	rectangle.corner.x $\leadsto$ integer Column coordinate of the lower right corner of the clip rectangle. Default Value : 512 Suggested values : Column2 $\in$ {512, 1024, 1536, 2048}

Parallelization Information
clip_contours_xld is *reentrant* and processed *without* parallelization.

Possible Predecessors
gen_contours_skeleton_xld, **lines_gauss**, **lines_facet**, **edges_sub_pix**

Possible Successors
gen_polygons_xld

See also
clip_region, **crop_part**

Module
Sub-pixel operators

combine_roads_xld (EdgePolygons, ModParallels, ExtParallels,
CenterLines : RoadSides : MaxAngleParallel, MaxAngleColinear,
MaxDistanceParallel, MaxDistanceColinear :)

Combine road hypotheses from two resolution levels.

combine_roads_xld combines road hypotheses obtained from two different resolution levels. The algorithm selects only those hypotheses which mutually support each other in both resolution levels. The parameters **EdgePolygons**, **ModParallels** and **ExtParallels** correspond to the road hypotheses from the highest resolution level. The parameter **CenterLines** is the result of road extraction in a lower resolution level. Those of the polygons **EdgePolygons** are returned for which evidence of being roadsides is found in both resolution levels. The parameters **MaxAngleParallel** and **MaxAngleColinear** determine the angle, that may be formed by two parallel collinear line segments, respectively. The parameters **MaxDistanceParallel** and **MaxDistanceColinear** determine the maximum allowed distance of two parallel or collinear line segments, respectively. The combination is done using a number of rules.

Parameter	
▷ EdgePolygons (input_object)	xld_poly-array $\leadsto$ Hobject XLD polygons to be examined.
▷ ModParallels (input_object)	xld_mod_para-array $\leadsto$ Hobject Modified parallels obtained from EdgePolygons .

- ▷ **ExtParallels** (input_object) xld_ext_para-array $\leadsto$ *Hobject*
Extended parallels obtained from [EdgePolygons](#).
- ▷ **CenterLines** (input_object) xld_poly-array $\leadsto$ *Hobject*
Road-center-line polygons to be examined.
- ▷ **RoadSides** (output_object) xld_poly-array $\leadsto$ *Hobject*
Roadsides found.
- ▷ **MaxAngleParallel** (input_control) angle.rad $\leadsto$ *real / integer*
Maximum angle between two parallel line segments.
Default Value : 0.523598775598
Suggested values : $\text{MaxAngleParallel} \in \{0.349065850399, 0.523598775598, 0.6981317008\}$
Restriction : $(0 \leq \text{MaxAngleParallel}) \leq (\pi/2)$
- ▷ **MaxAngleColinear** (input_control) angle.rad $\leadsto$ *real / integer*
Maximum angle between two collinear line segments.
Default Value : 0.261799387799
Suggested values : $\text{MaxAngleColinear} \in \{0.174532925199, 0.261799387799, 0.349065850399\}$
Restriction : $(0 \leq \text{MaxAngleColinear}) \leq (\pi/2)$
- ▷ **MaxDistanceParallel** (input_control) real $\leadsto$ *real / integer*
Maximum distance between two parallel line segments.
Default Value : 40
Suggested values : $\text{MaxDistanceParallel} \in \{20, 30, 40, 50, 60\}$
Restriction : $\text{MaxDistanceParallel} > 0$
- ▷ **MaxDistanceColinear** (input_control) real $\leadsto$ *real / integer*
Maximum distance between two collinear line segments.
Default Value : 40
Suggested values : $\text{MaxDistanceColinear} \in \{20, 30, 40, 50, 60\}$
Restriction : $\text{MaxDistanceColinear} > 0$

Parallelization Information

[combine_roads_xld](#) is processed under *mutual exclusion* against itself and without parallelization.

Possible Predecessors

[mod_parallels_xld](#), [gen_polygons_xld](#), [affine_trans_contour_xld](#)

Possible Successors

[get_polygon_xld](#), [get_lines_xld](#)

See also

[lines_gauss](#), [lines_facet](#), [get_channel_info](#), [edges_sub_pix](#)

References

C. Steger, C. Glock, W. Eckstein, H. Mayer, B. Radig; "Model-based Road Extraction from Images"; in "Automatic Extraction of Man-Made Objects from Aerial and Space Images"; A. Gruen, O. Kuebler, P. Agouris (Editors); Birkhäuser Verlag (1995), pp. 275-284.

C. Heipke, C. Steger, R. Multhammer; "A Hierarchical Approach to Automatic Road Extraction from Aerial Imagery"; in "Integrating Photogrammetric Techniques with Scene Analysis and Machine Vision II"; D. M. McKeown, Jr., I. J. Dowman (Editors); Proc. SPIE 2486 (1995), pp. 222-231.

Module

Sub-pixel operators

gen_parallel_contour_xld (Contours : ParallelContours : Mode,
Distance :)

Compute the parallel contour of an XLD contour.

[gen_parallel_contour_xld](#) computes for each of the input contours [Contours](#) a parallel contour with distance [Distance](#). The resulting contours are returned in [ParallelContours](#). To calculate the parallel contour, the normal vector of the input contour is needed in every contour point. The parameter [Mode](#) determines how these normal vectors are computed. If [Mode](#) = 'gradient', it is assumed that the input contours are edges, and the normal information is obtained from the gradient direction of the edge (see [edges_sub_pix](#)). For this, the attribute 'edge_direction' must exist for the input contour. If [Mode](#) = 'contour_normal', a possibly existing normal

information is used to determine the normals. For this, the contour attribute *'angle'* must exist (see [lines_gauss](#) or [edges_sub_pix](#)). Finally, if **Mode** = *'regression_normal'*, the normal vectors are determined from a local line fit to each contour point. Here, the normal vectors are oriented such that they point to the right side of the contour when the contour is traversed from start to end. In contrast to the first two modes, this mode can be used for all XLD contours, no matter how they were generated.

Parameter

- ▷ **Contours** (input_object) xld_cont-array $\leadsto$ *Hobject*
Contours to be transformed.
- ▷ **ParallelContours** (output_object) xld_cont-array $\leadsto$ *Hobject*
Parallel contours.
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Mode, with which the direction information is computed.
Default Value : *'regression_normal'*
Suggested values : **Mode** $\in$ {*'gradient'*, *'contour_normal'*, *'regression_normal'*}
- ▷ **Distance** (input_control) number $\leadsto$ *real* / *integer*
Distance of the parallel contour.
Default Value : 1
List of values : **Distance** $\in$ {0.2, 0.4, 0.6, 0.8, 1, 2, 3, 4, 5, 7, 10, 15, 20, 30, 40, 50}

Parallelization Information

[gen_parallel_contour_xld](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_contours_skeleton_xld](#), [lines_gauss](#), [lines_facet](#), [edges_sub_pix](#),
[threshold_sub_pix](#)

Possible Successors

[gen_polygons_xld](#)

See also

[get_contour_xld](#)

Module

Sub-pixel operators

merge_cont_line_scan_xld (CurrConts, PrevConts : CurrMergedConts,
PrevMergedConts : ImageHeight, Margin, MergeBorder, MaxImagesCont :)

Merge XLD contours from successive line scan images.

The operator [merge_cont_line_scan_xld](#) connects adjacent XLD contours, which were extracted from adjacent images with the height [ImageHeight](#). This operator was especially designed to connect contours that were extracted from images grabbed by a line scan camera. [CurrConts](#) contains the contours from the current image and [PrevConts](#) the contours from the previous one.

With the help of the parameter [MergeBorder](#) two cases can be distinguished: If the top (first) line of the current image touches the bottom (last) line of the previous image, [MergeBorder](#) must be set to *'top'*, otherwise set [MergeBorder](#) to *'bottom'*. [MergeBorder](#) defines a margin to the border. Only those end points of the contours which are inside this margin are considered for the following merging process.

If the operator [merge_cont_line_scan_xld](#) is used recursively, the parameter [MaxImagesCont](#) determines the maximum number of images which are covered by a merged contour. All points of the merged contour from an older image are removed.

The operator [merge_cont_line_scan_xld](#) returns two contour arrays. [PrevMergedConts](#) contains all those contours from the previous input contours [PrevConts](#), which could not be merged with a current contour. [CurrMergedConts](#) collects all current contours together with the merged parts from the previous images. Merged contours will exceed the original image, because the previous contours are moved upward ([MergeBorder](#)=*'top'*) or downward ([MergeBorder](#)=*'bottom'*) according to the image height.

Parameter	
▷ CurrConts (input_object)	xld_cont(-array) $\leadsto$ Hobject Current input contours.
▷ PrevConts (input_object)	xld_cont(-array) $\leadsto$ Hobject Merged contours from the previous iteration.
▷ CurrMergedConts (output_object)	xld_cont(-array) $\leadsto$ Hobject Current contours, merged with old ones where applicable.
▷ PrevMergedConts (output_object)	xld_cont(-array) $\leadsto$ Hobject Contours from the previous iteration which could not be merged with the current ones.
▷ ImageHeight (input_control)	integer $\leadsto$ integer Height of the line scan images. Default Value : 512 List of values : ImageHeight $\in$ {240, 480, 512}
▷ Margin (input_control)	real $\leadsto$ real Maximum distance of contours from the image border. Default Value : 0 List of values : Margin $\in$ {0, 1, 2, 3, 4, 5}
▷ MergeBorder (input_control)	string $\leadsto$ string Image line of the current image, which touches the previous image. Default Value : "top" List of values : MergeBorder $\in$ {"top", "bottom"}
▷ MaxImagesCont (input_control)	integer $\leadsto$ integer Maximum number of images covered by one contour. Default Value : 3 Suggested values : MaxImagesCont $\in$ {1, 2, 3, 4, 5}

The operator `merge_cont_line_scan_xld` returns the value 2 (H_MSG_TRUE) if the given parameters are correct. Otherwise, an exception will be raised.

Parallelization Information
`merge_cont_line_scan_xld` is *reentrant* and processed *without* parallelization.

Module
Sub-pixel operators

regress_contours_xld (Contours : RegressContours : Mode, Iterations :)

Calculate the parameters of a regression line to an XLD contour.

`regress_contours_xld` calculates the following parameters for the input XLD contours `Contours`, and stores them with the resulting contours as global attributes:

- the coordinates of the normal vector of the regression line, i.e., the least-squares approximating line, of all contour points; the normal vector always points from the origin to the line (attributes: 'regr_norm_row', 'regr_norm_col'),
- the mean of the Euclidian distance of the contour points from the regression line (attribute: 'regr_mean_dist'),
- the standard deviation of these distances to the regression line (attribute: 'regr_dev_dist').

For `Mode` = 'no', the parameters of the regression line are calculated for all points of the contour. In addition, three different kinds of outlier treatment can be applied. Outliers are contour points which do not lie on the general contour direction in an "obvious" manner, and thus "distort" the resulting regression line.

`Mode` =

- 'drop': All contour points further away from the contour than the mean distance to the regression line are ignored for the calculation of the undistorted regression line.

- 'gauss': The distances of the contour points are weighted according to their probability of occurrence in a Gaussian distribution around the normal regression line.
- 'median': Here, also a normal distribution is assumed for the distances to the normal regression line, however with the outlier-independent standard deviation $\frac{\text{median}(\text{alldist.})}{0.6745}$. Again, the distances are weighted, and points further away than a certain distance are ignored for the undistorted regression line.

The calculation of the undistorted regression line can be iterated several times ([Iterations](#)).

Parameter	
▷ Contours (input_object)	xld_cont-array $\leadsto$ <i>Hobject</i> Input XLD contours.
▷ RegressContours (output_object)	xld_cont-array $\leadsto$ <i>Hobject</i> Resulting XLD contours.
▷ Mode (input_control)	string $\leadsto$ <i>string</i> Type of outlier treatment. Default Value : 'no' List of values : Mode $\in$ {'no', 'drop', 'gauss', 'median'}
▷ Iterations (input_control)	integer $\leadsto$ <i>integer</i> Number of iterations for the outlier treatment. Default Value : 1 Suggested values : Iterations $\in$ {1, 2, 3, 5, 10, 20}

[regress_contours_xld](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors
[gen_contours_skeleton_xld](#), [lines_gauss](#), [lines_facet](#), [edges_sub_pix](#)

Possible Successors
[get_regress_params_xld](#)

See also
[smooth_contours_xld](#), [get_contour_global_attrib_xld](#),
[query_contour_global_attribs_xld](#)

References
H. Suesse, K. Voss: "Adaptive Ausgleichsrechnung und Ausreißerproblematik für die digitale Bildverarbeitung";
Proc. 15. DAGM Symposium, Springer Verlag, Lübeck 1993
R. Haralick, L. Shapiro: "Computer and Robot Vision" Vol. 2; Kapitel 14.9, Addison-Wesley 1992

Module
Sub-pixel operators

segment_contours_xld (Contours : ContoursSplit : Mode, SmoothCont, MaxLineDist1, MaxLineDist2 :)

Segment XLD contours into line segments and circular or elliptic arcs.

[segment_contours_xld](#) segments the input contours [Contours](#) into lines if [Mode](#)='lines', into lines and circular arcs if [Mode](#)='lines_circles', or into lines and elliptic arcs if [Mode](#)='lines_ellipses'. The segmented contours are returned in [ContoursSplit](#). The information on whether an output contour represents a line or a circular or elliptic arc is done via the global contour attribute 'cont_approx' (see [get_contour_global_attrib_xld](#)). If 'cont_approx'=-1, the contour is best approximated by a line segment, if 'cont_approx'=0, by an elliptic arc, and if 'cont_approx'=1, by a circular arc.

[segment_contours_xld](#) first approximates the input contours by polygons. With this, the contours are over-segmented in curved areas. After this, neighboring line segments are substituted by circular or elliptic arcs, respectively, if the contour can be approximated better by an arc. If [SmoothCont](#) is set to a value > 0, the input contours are first smoothed (see [smooth_contours_xld](#)). This can be necessary to prevent very short segments in the polygonal approximation and to achieve a more robust fit with circular or elliptic arcs, because the smoothing suppresses outliers on the contours.

The initial polygonal approximation is done by using the Ramer algorithm (see [gen.polygons_xld](#)) with a maximum distance of **MaxLineDist1**. After this, circular or elliptic arcs are fit into neighboring line segments. If the maximum distance of the resulting arc to the contour is smaller than the maximum distance of the two line segments, the two line segments are replaced with the arc. This procedure is iterated until no more changes occur.

After this, the parts of the contour that are still approximated by line segments are again segmented with a polygonal approximation with maximum distance **MaxLineDist2**, and the newly created line segments are merged to circular or elliptical arcs where possible. Obviously, this changes the output only if **MaxLineDist2** < **MaxLineDist1**. This two-step approach is more efficient than a one-step approach with **MaxLineDist2**, since fewer line segments are generated in the first step, and hence the circle or ellipse fit has to be performed less often. Therefore, parts of the input contours that can be approximated by long arcs can be found more efficiently. In the second step, parts of the input contours that can be approximated by short arcs are found and the end parts of long arcs are refined.

Parameter

- ▷ **Contours** (input_object) xld_cont(-array) $\leadsto$ *Hobject*
Contours to be segmented.
- ▷ **ContoursSplit** (output_object) xld_cont-array $\leadsto$ *Hobject*
Segmented contours.
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Mode for the segmentation of the contours.
Default Value : 'lines_circles'
List of values : Mode $\in$ {'lines', 'lines_circles', 'lines_ellipses'}
- ▷ **SmoothCont** (input_control) integer $\leadsto$ *integer*
Number of points used for smoothing the contours.
Default Value : 5
Suggested values : SmoothCont $\in$ {0, 3, 5, 7, 9}
Restriction : (SmoothCont = 0) $\vee$ ((SmoothCont $\geq$ 3) $\wedge$ odd(SmoothCont))
- ▷ **MaxLineDist1** (input_control) real $\leadsto$ *real*
Maximum distance between a contour and the approximating line (first iteration).
Default Value : 4.0
Suggested values : MaxLineDist1 $\in$ {1.0, 1.5, 2.0, 2.5, 3.0, 3.5}
Restriction : MaxLineDist1 $\geq$ 0.0
- ▷ **MaxLineDist2** (input_control) real $\leadsto$ *real*
Maximum distance between a contour and the approximating line (second iteration).
Default Value : 2.0
Suggested values : MaxLineDist2 $\in$ {1.0, 1.5, 2.0, 2.5, 3.0, 3.5}
Restriction : MaxLineDist2 $\geq$ 0.0

Example

```
read_image (Image, 'pumpe')
edges_sub_pix (Image, Edges, 'canny', 1.5, 15, 40)
segment_contours_xld (Edges, ContoursSplit, 'lines_circles', 5, 4, 2)
count_obj (ContoursSplit, Number)
gen_empty_obj (Lines)
gen_empty_obj (Circles)
for I := 1 to Number by 1
    select_obj (ContoursSplit, Contour, I)
    get_contour_global_attrib_xld (Contour, 'cont_approx', Type)
    if (Type = -1)
        concat_obj (Lines, Contour, Lines)
    else
        concat_obj (Circles, Contour, Circles)
    endif
endfor
fit_line_contour_xld (Lines, 'tukey', -1, 0, 5, 2, RowBegin, ColBegin,
                    RowEnd, ColEnd, Nr, Nc)
fit_circle_contour_xld (Circles, 'atukey', -1, 2, 0, 3, 2, Row, Column,
                    Radius, StartPhi, EndPhi, PointOrder)
```

Parallelization Information

[segment_contours_xld](#) is *local* and processed *completely exclusively* without parallelization.

Possible Predecessors

[gen_contours_skeleton_xld](#), [lines_gauss](#), [edges_sub_pix](#)

Possible Successors

[fit_line_contour_xld](#), [fit_ellipse_contour_xld](#), [fit_circle_contour_xld](#)

See also

[split_contours_xld](#), [get_contour_global_attrib_xld](#), [smooth_contours_xld](#),
[gen_polygons_xld](#)

Module

Sub-pixel operators

shape_trans_xld (XLD : XLDTrans : Type :)

Transform the shape of a contour.

[shape_trans](#) transforms the input contour depending on the parameter [Type](#):

'convex' Convex hull.

'ellipse' Ellipse with the same moments and area as the input contour. The closed input contour must not intersect itself, otherwise the resulting ellipse is not meaningful.

'outer_circle' Smallest enclosing circle.

'rectangle1' Smallest enclosing rectangle parallel to the coordinate axes.

'rectangle2' Smallest enclosing rectangle.

Parameter

- ▷ **XLD** (input_object) xld(-array) $\leadsto$ *Hobject*
Contour(s) to be transformed.
- ▷ **XLDTrans** (output_object) xld(-array) $\leadsto$ *Hobject*
Transformed contour(s).
- ▷ **Type** (input_control) string $\leadsto$ *string*
Type of transformation.
Default Value : 'convex'
List of values : Type $\in$ { 'convex', 'ellipse', 'outer_circle', 'rectangle1', 'rectangle2' }

Result

[shape_trans_xld](#) returns 2 (H_MSG.TRUE) if all parameters are correct. The behavior in case of empty input (no contours given) can be set via [set_system\('no_object_result', <Result>\)](#). If necessary, an exception is raised.

Parallelization Information

[shape_trans_xld](#) is *reentrant* and automatically *parallelized* (on *tuple level*).

Possible Predecessors

[gen_contours_skeleton_xld](#), [edges_sub_pix](#), [threshold_sub_pix](#),
[gen_contour_polygon_xld](#)

Alternatives

[smallest_circle_xld](#), [smallest_rectangle1_xld](#), [smallest_rectangle2_xld](#),
[elliptic_axis_xld](#)

See also

[smallest_circle_xld](#), [smallest_rectangle1_xld](#), [smallest_rectangle2_xld](#),
[elliptic_axis_xld](#)

Module

Sub-pixel operators

smooth_contours_xld (Contours : SmoothedContours : NumRegrPoints :)

Smooth an XLD contour.

smooth_contours_xld smooths the input XLD contours **Contours** and returns the smoothed contours in **SmoothedContours**. The smoothing is done by projecting the contours' points onto a local regression line (i.e., a least-squares approximating line), which is computed from **NumRegrPoints** on each side of the current contour point. This operator should be called, for example, before contours are scaled.

Parameter

- ▷ **Contours** (input_object) xld_cont-array $\leadsto$ *Hobject*
Contour to be smoothed.
- ▷ **SmoothedContours** (output_object) xld_cont-array $\leadsto$ *Hobject*
Smoothed contour.
- ▷ **NumRegrPoints** (input_control) integer $\leadsto$ *integer*
Number of points used to calculate the regression line.
Default Value : 5
Suggested values : NumRegrPoints $\in \{3, 5, 7, 9\}$
Restriction : (NumRegrPoints ≥ 3) $\wedge$ odd(NumRegrPoints)

Parallelization Information

smooth_contours_xld is *reentrant* and processed *without* parallelization.

Possible Predecessors

gen_contours_skeleton_xld, **lines_gauss**, **lines_facet**, **edges_sub_pix**

Possible Successors

affine_trans_contour_xld, **gen_polygons_xld**, **local_max_contours_xld**

See also

get_contour_xld

Module

Sub-pixel operators

split_contours_xld (Polygons : Contours : Mode, Weight, Smooth :)

Split XLD contours at dominant points.

split_contours_xld splits the contours which were used to generate the polygons **Polygons** at prominent points. If the mode 'polygon' is selected, the contours are split at the polygons' control points. In mode 'dominant', they are split at dominant points, i.e., at points for which the calculated change of direction is larger than the (empirically determined) threshold $2\pi \text{Weight} / \text{contour length}$, and for which in the (empirically determined) neighborhood of $\sqrt{\text{Smooth} * \log(\text{contour length})}$ points no larger change of direction occurs. The contour direction is determined from the direction of the regression line (i.e., the least-squares approximating line) through all points in a neighborhood of **Smooth** points around a contour point. The directions thus determined are smoothed with a Gaussian of width **Smooth**. **Weight** is a weighting factor for the sensitiveness of the operator. The larger **Weight** is selected, the less dominant points are found.

Parameter

- ▷ **Polygons** (input_object) xld_poly(-array) $\leadsto$ *Hobject*
Polygons for which the corresponding contours are to be split.
- ▷ **Contours** (output_object) xld_cont(-array) $\leadsto$ *Hobject*
Split contours.
- ▷ **Mode** (input_control) string $\leadsto$ *string*
Mode for the splitting of the contours.
Default Value : 'polygon'
List of values : Mode $\in \{ \text{'polygon'}, \text{'dominant'} \}$
- ▷ **Weight** (input_control) integer $\leadsto$ *integer*
Weight for the sensitiveness.
Default Value : 1

- ▷ **Smooth** (input_control) integer $\leadsto$ integer
Width of the smoothing mask.
Default Value : 5

Parallelization Information

[split_contours_xld](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[gen_polygons_xld](#)

Possible Successors

[regress_contours_xld](#)

See also

[gen_contours_skeleton_xld](#), [lines_gauss](#), [lines_facet](#), [edges_sub_pix](#)

Module

Sub-pixel operators

union_straight_contours_histo_xld (Contours : UnionContours,
SelectedContours : RefLineStartRow, RefLineStartColumn, RefLineEndRow,
RefLineEndColumn, Width, MaxWidth, FilterSize : HistoValues)

Merge neighboring straight contours having a similar distance from a given line.

[union_straight_contours_histo_xld](#) merges neighboring XLD contours [Contours](#) if certain criteria are fulfilled.

The maximum and minium distance of the contours to an given reference line are calculated. From this distances a histogram is created. If the histogram should be smoothed, [FilterSize](#) must be greater then one. Afterwards the resulting histogram is divided into ranges (from minima to minima). Contours which lie in the same range are concatenated to a new contour. If the width of Range is greater than [MaxWidth](#), all contours in this range are ignored (removed). Lies an contour in two ranges or more it is also ignored. In case there are parallel contours, there is a danger of merging neighboring contours.

The parameters of the regression line are calculated new for merged contours.

The resulting contours cannot be displayed.

Attention

Before the contour parameters can be returned by [union_straight_contours_histo_xld](#), the parameters of the regression line to the contour must be calculated by calling [regress_contours_xld](#).

Parameter

- ▷ **Contours** (input_object) xld_cont-array $\leadsto$ Hobject
Input XLD contours.
- ▷ **UnionContours** (output_object) xld_cont-array $\leadsto$ Hobject
Output XLD contours.
- ▷ **SelectedContours** (output_object) xld_cont-array $\leadsto$ Hobject
Output XLD contours.
- ▷ **RefLineStartRow** (input_control) line.begin.y $\leadsto$ integer
y coordinate of the starting point of the reference line.
Default Value : 0
- ▷ **RefLineStartColumn** (input_control) line.begin.x $\leadsto$ integer
x coordinate of the starting point of the reference line.
Default Value : 0
- ▷ **RefLineEndRow** (input_control) line.end.y $\leadsto$ integer
y coordinate of the endpoint of the reference line.
Default Value : 0
- ▷ **RefLineEndColumn** (input_control) line.end.x $\leadsto$ integer
x coordinate of the endpoint of the reference line.
Default Value : 0

- ▷ **Width** (input_control) integer $\rightsquigarrow$ integer
Maximum distance.
Default Value : 1
- ▷ **MaxWidth** (input_control) integer $\rightsquigarrow$ integer
Maximum Width between two minimas.
Default Value : 1
- ▷ **FilterSize** (input_control) integer $\rightsquigarrow$ integer
Size of Smoothfilter
Default Value : 1
Typical range of values : $1 \leq \text{FilterSize} \leq 63$
- ▷ **HistoValues** (output_control) integer-array $\rightsquigarrow$ integer
Output Values of Histogram.

Parallelization Information

[union_straight_contours_histo_xld](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[regress_contours_xld](#)

See also

[fit_line_contour_xld](#), [get_contour_xld](#), [get_contour_attrib_xld](#),
[gen_contours_skeleton_xld](#), [lines_gauss](#), [lines_facet](#), [edges_sub_pix](#),
[get_regress_params_xld](#), [get_contour_global_attrib_xld](#),
[query_contour_global_attribs_xld](#)

Module

Sub-pixel operators

union_straight_contours_xld (Contours : UnionContours : MaxDist,
 MaxDiff, Percent, Mode, Iterations :)

Merge neighboring straight contours having a similar direction.

[union_straight_contours_xld](#) merges neighboring XLD contours [Contours](#) if certain criteria are fulfilled. The merging is not done recursively, and only between two contours. The parameter [Iterations](#) controls how often this union step is executed.

Two contours are merged if the shortest distance between their end points (end points are the projections of the contours' first and last points onto its regression line) is smaller than [MaxDist](#), and if the difference in direction (i.e., the regression lines' direction) is smaller than [MaxDiff](#) (radians).

If only one of the criteria is fulfilled, the decision on merging can be influenced by the weighting factor [Percent](#), which allows the exceeding of one limit to be balanced by the other value remaining below the limit by the same amount. The end point distance is weighted by [Percent](#), while the directional difference is weighted by $100 - \text{Percent}$.

If, for example, two contours have an end point distance of 5.0 and a directional difference of 0.5 (with the limits chosen being [MaxDist](#) = 4.0 und [MaxDiff](#) = 0.625), both values differ from the limits by 25%. By choosing [Percent](#) = 60%, the larger end point distance is weighed more than the smaller directional difference, and thus the contours are not merged. Contrary, if [Percent](#) = 40% is chosen, the contours are merged.

For [Percent](#) = 100%, only the end point distance is taken into account, while for [Percent](#) = 0% only the difference of direction is used. If [Percent](#) = 50% both criteria are equally valid.

In case there are parallel contours, there is a danger of merging neighboring contours. If this is to be avoided, [Mode](#) = 'noparallel' has to be chosen, while otherwise [Mode](#) = 'paralleltoo' suffices. For [Mode](#) = 'every', contours are merged unconditionally. All other parameters have no influence in this case.

The parameters of the regression line are calculated anew for merged contours.

Attention

Before the contour parameters can be returned by [get_regress_params_xld](#), the parameters of the regression line to the contour must be calculated by calling [regress_contours_xld](#).

<i>Parameter</i>	
▷ Contours (input_object)	xld_cont-array $\leadsto$ <i>Hobject</i> Input XLD contours.
▷ UnionContours (output_object)	xld_cont-array $\leadsto$ <i>Hobject</i> Output XLD contours.
▷ MaxDist (input_control)	real $\leadsto$ <i>real</i> Maximum distance of the contours' endpoints. Default Value : 5.0
▷ MaxDiff (input_control)	angle.rad $\leadsto$ <i>real</i> Maximum difference in direction. Default Value : 0.5
▷ Percent (input_control)	real $\leadsto$ <i>real</i> Weighting factor for the two selection criteria. Default Value : 50.0
▷ Mode (input_control)	string $\leadsto$ <i>string</i> Should parallel contours be taken into account? Default Value : 'noparallel' List of values : Mode $\in$ {'noparallel', 'paralleltoo', 'every'}
▷ Iterations (input_control)	string $\leadsto$ <i>string</i> / integer Number of iterations or 'maximum'. Default Value : 'maximum' Suggested values : Iterations $\in$ {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 'maximum'} Typical range of values : $1 \leq \text{Iterations} \leq 500$ (lin) Minimum Increment : 1 Recommended Increment : 1

Parallelization Information

[union_straight_contours_xld](#) is *reentrant* and processed *without* parallelization.

Possible Predecessors

[regress_contours_xld](#)

See also

[fit_line_contour_xld](#), [get_contour_xld](#), [get_contour_attrib_xld](#),
[gen_contours_skeleton_xld](#), [lines_gauss](#), [lines_facet](#), [edges_sub_pix](#),
[get_regress_params_xld](#), [get_contour_global_attrib_xld](#),
[query_contour_global_attribs_xld](#)

Module

Sub-pixel operators

Index

- 2D-Transformation, [46](#), [57](#), [59–63](#), [517–519](#), [521](#),
[693](#), [695–699](#), [701](#), [702](#), [709–712](#), [972](#),
[973](#)
- 3D-Projection, [769](#), [784](#)
- 3D-Transformation, [694](#), [703](#), [704](#), [706](#), [708](#), [754](#),
[758](#), [770](#), [778](#), [783](#), [786](#), [787](#), [791](#)
- `abs_funct_1d`, [799](#)
- `abs_image`, [64](#)
- `abs_invar_fourier_coeff`, [793](#)
- Absolute-Value, [64](#), [799](#), [897](#), [901](#)
- abstract, [659](#)
- Access, [801](#)
- `access_channel`, [309](#)
- `adapt_template`, [399](#)
- `add_channels`, [336](#)
- `add_image`, [65](#)
- `add_noise_distribution`, [157](#)
- `add_noise_white`, [158](#)
- `add_noise_white_contour_xld`, [971](#)
- Addition, [65](#), [805](#), [898](#)
- Affine-Transformation, [57](#), [59](#), [517](#), [695](#), [710](#), [972](#),
[973](#)
- `affine_trans_contour_xld`, [972](#)
- `affine_trans_image`, [57](#)
- `affine_trans_image_size`, [59](#)
- `affine_trans_point_2d`, [693](#)
- `affine_trans_point_3d`, [694](#)
- `affine_trans_polygon_xld`, [973](#)
- `affine_trans_region`, [517](#)
- Albedo, [888](#), [889](#)
- Ambient, [888](#)
- and, [72](#)
- `angle_ll`, [808](#)
- `angle_lx`, [809](#)
- Anisometry, [547](#), [567](#), [949](#), [967](#)
- `anisotrope_diff`, [161](#)
- Anisotropy, [343](#), [357](#)
- `append_channel`, [310](#)
- `append_ocr_trainf`, [867](#)
- `approx_chain`, [387](#)
- `approx_chain_simple`, [390](#)
- Approximation, [387](#), [390](#)
- Arc, [224](#), [387](#)
- Arccosine, [897](#)
- Arcs, [390](#)
- Arcsine, [898](#)
- Arctangent, [899](#)
- Area, [340](#), [541](#), [567](#), [574](#), [943](#), [967](#), [969](#)
- `area_center`, [541](#)
- `area_center_gray`, [340](#)
- `area_center_xld`, [943](#)
- Arithmetic, [64–71](#)
- Arrow, [225](#)
- assign, [11](#)
- assurance, [659](#), [661](#)
- Attributes, [957](#), [958](#), [965](#)
- Ausgabe, [244](#)
- `auto_threshold`, [603](#)
- Average, [335](#), [922](#), [942](#), [957](#), [959](#), [960](#), [966](#)
- Background, [581](#), [585](#)
- background, [713](#), [714](#), [716](#), [718](#), [720](#), [722](#)
- background-estimation, [713](#), [714](#), [716](#), [718](#), [720](#), [722](#)
- `background_seg`, [581](#)
- Band-Filter, [124](#)
- Bandpass-Filter, [125](#), [131](#), [136](#)
- `bandpass_image`, [136](#)
- Barcode, [723](#), [725](#), [726](#), [729](#), [734](#), [735](#), [738](#)
- Barcode-description, [734](#), [735](#)
- Best-Match, [399](#), [400](#), [402–404](#), [409](#), [410](#), [419](#), [422](#),
[426](#), [428](#)
- `best_match`, [399](#)
- `best_match_img`, [400](#)
- `best_match_pre_img`, [402](#)
- `best_match_rot`, [403](#)
- `best_match_rot_img`, [404](#)
- `bin_threshold`, [604](#)
- Binary-image, [46](#)
- Bit, [72–78](#)
- Bit-Operations, [908–910](#), [927](#)
- `bit_and`, [72](#)
- `bit_lshift`, [73](#)
- `bit_mask`, [74](#)
- `bit_not`, [75](#)
- `bit_or`, [75](#)
- `bit_rshift`, [76](#)
- `bit_slice`, [77](#)
- `bit_xor`, [78](#)
- Blurring, [183](#), [184](#), [186](#), [187](#)
- BMP, [41](#), [44](#), [46](#), [282](#)
- Border-Treatment, [149](#)
- Bottom-Hat, [439](#), [450](#), [485](#)
- `bottom_hat`, [450](#)
- Boundary, [451](#)
- `boundary`, [451](#)
- Branches, [493](#)
- Break, [12](#)
- `break`, [12](#)
- Buffer, [24](#), [292](#), [295](#), [670](#), [673](#)

- Bulkiness, 547, 949
- C-procedure, 663
- Calibration, 745, 746, 751–753, 755, 761, 763, 764, 766, 768, 769, 779, 781, 784, 785, 788, 789
- caltab_points, 745
- camera_calibration, 746
- Canny-Filter, 93, 96
- Ceiling-Function, 899
- Center, 340, 523, 541, 567, 574, 967, 969
- Center-of-Gravity, 393, 395
- Centroid, 943
- cfa_to_rgb, 79
- Chain-code, 513
- Chaincode, 387
- Chaincodes, 390
- change_domain, 337
- change_format, 361
- change_radial_distortion_cam_par, 751
- change_radial_distortion_contours_xld, 752
- change_radial_distortion_image, 753
- Channels, 651
- channels_to_image, 311
- Chapter, 658
- chapter, 659
- char_threshold, 605
- Character, 881–886
- Character-sequence, 50
- Characters, 876, 877
- check_difference, 607
- check_par_hw_potential, 668
- Checkerboard, 522
- Choice-of-regions, 567, 967
- Chord, 563, 564
- Chord-representation, 516
- Circle, 227, 453, 459, 467, 489, 523, 528, 554, 562, 573, 969, 978
- circle, 196, 268, 951
- circularity, 542
- circularity_xld, 944
- Class, 504
- class_2dim_sup, 608
- class_2dim_unsup, 610
- class_ndim_box, 612
- class_ndim_norm, 613
- Classification, 156, 608, 610, 612, 613, 628, 629
- Clear, 19
- clear_all_shape_models, 413
- clear_all_templates, 405
- clear_all_variation_models, 831
- clear_obj, 507
- clear_rectangle, 278
- clear_sampset, 1
- clear_serial, 678
- clear_shape_model, 413
- clear_template, 406
- clear_variation_model, 831
- clear_window, 279
- Clearing, 19, 329, 507
- clip_contours_xld, 973
- clip_region, 582
- clip_region_rel, 583
- Clipping, 362, 582, 583, 594, 595, 973
- Close, 881, 882
- close_all_bg_esti, 713
- close_all_class_box, 1
- close_all_files, 48
- close_all_framegrabbers, 368
- close_all_measures, 846
- close_all_ocrs, 868
- close_all_ocvs, 881
- close_all_serials, 679
- close_bg_esti, 713
- close_class_box, 2
- close_edges, 87
- close_edges_length, 88
- close_file, 48
- close_framegrabber, 368
- close_measure, 847
- close_ocr, 869
- close_ocv, 882
- close_serial, 679
- close_socket, 683
- close_window, 280
- Closing, 20, 280, 437, 439, 440, 450, 452, 453, 455, 471, 472, 485
- closing, 452
- closing_circle, 453
- closing_golay, 455
- closing_rectangle1, 456
- Clustering, 610, 612, 613, 628, 629
- Co-occurrence-Matrix, 341, 343, 349
- Coding, 564
- Coincidence, 528
- Color, 79, 214–217, 219
- color, 81, 84, 241, 246–249, 252, 253, 257, 267
- color-coding, 241, 257
- Color-Image, 228
- color-image, 80
- color-lookup-table, 246, 247
- Color-Space, 79
- color-space, 81, 84
- Color-Space-Transformation, 79
- color-space-transformation, 80, 81, 84
- Color-Table, 670, 673
- combine_roads_xld, 974
- Comment, 12
- comment, 12
- Compactness, 542, 543, 567, 944, 945, 967
- compactness, 543
- compactness_xld, 945
- compare_variation_model, 831
- Comparison, 505, 883
- Comparison-Operators, 911–913
- Complement, 578
- complement, 578

- complex_to_real, 383
- Components, 503
- compose2, 311
- compose3, 312
- compose4, 312
- compose5, 313
- compose6, 314
- compose7, 314
- concat_obj, 508
- concat_ocr_trainf, 869
- Concatenation, 508
- Condition, 15
- Configuration, 298
- configuration, 248–251, 272
- connect_and_holes, 544
- Connected-Components, 358, 360, 581, 584
- connection, 584
- Connection-component, 544, 549
- contlength, 545
- Contour, 513, 514, 516, 545, 816
- contour, 243, 244, 254, 260
- Contour-Length, 960
- Contour-length, 543, 545, 567, 945
- contour-output, 261
- contour_point_num_xld, 945
- contour_to_world_plane_xld, 753
- Contours, 87, 88, 451, 533, 538, 752, 933, 935–941, 945, 947, 951, 952, 955, 957–959, 961, 965, 966, 971–973, 975, 977, 978, 980–983
- Contrast, 120
- contrast, 115
- Contrast-enhancement, 116, 117
- Control, 653, 668–670, 673
- Control-Modes, 654
- control-parameter, 663
- Control-Variable, 23
- Conversion, 913–917
- convert_image_type, 384
- convert_pose_type, 754
- Convex, 514
- Convexity, 546, 567, 946, 967
- convexity, 546
- convexity_xld, 946
- convol_fft, 118
- convol_gabor, 119
- convol_image, 149
- Convolution, 118, 119, 126, 127, 149
- cooc_feature_image, 341
- cooc_feature_matrix, 341
- Coordinate-System, 24, 289, 292, 295
- copy_image, 321
- copy_obj, 508
- copy_rectangle, 280
- Copying, 321, 329, 379–381, 508, 511
- corner_response, 141
- Corners, 141
- Correlation, 142, 143, 399, 400, 402–404, 409, 410
- Correspondences, 764
- Cosine, 900
- count_channels, 315
- count_obj, 503
- count_relation, 649
- count_seconds, 666
- Create, 882
- create_bg_esti, 714
- create_caltab, 755
- create_class_box, 2
- create_func1d_array, 799
- create_func1d_pairs, 800
- create_ocr_class_box, 870
- create_ocv_proj, 882
- create_pose, 758
- create_scaled_shape_model, 414
- create_shape_model, 417
- create_template, 406
- create_template_rot, 408
- create_variation_model, 832
- Creation, 919
- crop_domain, 361
- crop_domain_rel, 362
- crop_part, 363
- crop_rectangle1, 363
- cross-reference, 659, 666
- Cursor, 222, 223
- cursor-position, 271
- Database, 19, 321, 507, 508, 510, 511, 649, 651
- Debugging, 653, 654
- Decode, 723, 725
- decode_1d_bar_code, 723
- decode_2d_bar_code, 724
- decompose2, 316
- decompose3, 316
- decompose4, 317
- decompose5, 318
- decompose6, 319
- decompose7, 319
- default-type, 661
- Degrees, 905, 914
- Delete, 506
- Deletion, 278, 279
- depth_from_focus, 887
- Deriche-Filter, 93, 96, 177
- derivate_gauss, 89
- descript_class_box, 2
- Description-file, 41
- description-file, 841
- detect_edge_segments, 615
- dev_clear_obj, 19
- dev_clear_window, 19
- dev_close_inspect_ctrl, 20
- dev_close_window, 20
- dev_display, 21
- dev_error_var, 22
- dev_inspect_ctrl, 23
- dev_map_par, 23
- dev_map_prog, 24

dev_map_var, 24
 dev_open_window, 24
 dev_set_check, 27
 dev_set_color, 27
 dev_set_colored, 28
 dev_set_draw, 29
 dev_set_line_width, 30
 dev_set_lut, 30
 dev_set_paint, 31
 dev_set_part, 32
 dev_set_shape, 33
 dev_set_window, 34
 dev_set_window_extents, 34
 dev_unmap_par, 36
 dev_unmap_prog, 36
 dev_unmap_var, 36
 dev_update_pc, 37
 dev_update_time, 37
 dev_update_var, 38
 dev_update_window, 38
 Deviation, 353, 356, 357
 deviation_image, 180
 Devicecontext, 299
 Diameter, 547, 946
 diameter_region, 547
 diameter_xld, 946
 diff_of_gauss, 92
 Difference, 578
 difference, 578
 Differential-Geometry, 89
 Diffusion-Filter, 161
 Dilatation, 816
 Dilation, 442, 450, 452, 453, 455, 457–460, 462, 463, 479, 481, 485, 488–492, 501
 dilation1, 457
 dilation2, 458
 dilation_circle, 459
 dilation_golay, 460
 dilation_rectangle1, 462
 dilation_seq, 463
 Direction, 933–935, 957, 959, 966, 971, 975, 981–983
 Direction-code, 513
 Discrete, 725
 discrete_1d_bar_code, 725
 disp_arc, 224
 disp_arrow, 225
 disp_caltab, 761
 disp_channel, 227
 disp_circle, 227
 disp_color, 228
 disp_distribution, 229
 disp_ellipse, 230
 disp_image, 232
 disp_info, 657
 disp_line, 233
 disp_lut, 213
 disp_obj, 234
 disp_polygon, 235
 disp_rectangle1, 236
 disp_rectangle2, 237
 disp_region, 238
 disp_xld, 239
 Displacement-Vector-Field, 145, 147, 384, 385
 DISPLAY, 292, 295
 Display, 239
 display, 242, 250, 251, 258
 dist_ellipse_contour_xld, 947
 Distance, 547, 585, 800, 816, 946, 947
 distance_func_1d, 800
 distance_lr, 810
 distance_pl, 811
 distance_pp, 813
 distance_pr, 814
 distance_ps, 814
 distance_rr_min, 816
 distance_rr_min_dil, 816
 distance_sl, 817
 distance_sr, 818
 distance_ss, 819
 distance_transform, 585
 Distribution, 343, 350–352, 354, 563
 div_image, 66
 Division, 66, 901, 902
 do_ocr_multi, 872
 do_ocr_single, 873
 do_ocv_simple, 883
 Documentation, 670, 673
 Domain, 336–339
 drag_region1, 193
 drag_region2, 194
 drag_region3, 195
 draw_circle, 196
 draw_circle_mod, 196
 draw_ellipse, 197
 draw_ellipse_mod, 198
 draw_line, 200
 draw_line_mod, 200
 draw_lut, 214
 draw_point, 202
 draw_point_mod, 202
 draw_polygon, 203
 draw_rectangle1, 204
 draw_rectangle1_mod, 205
 draw_rectangle2, 206
 draw_rectangle2_mod, 207
 draw_region, 208
 drawing, 196, 200, 202, 204–208
 dual_rank, 437
 dual_threshold, 616
 dump_window, 282
 dvf_to_hom_mat2d, 695
 dvf_to_int1, 384
 dyn_threshold, 618
 Dynamic-Threshold, 607, 618
 eccentricity, 547
 eccentricity_xld, 949

- Edge-Amplitude, [93](#), [96](#), [98](#), [99](#), [102](#), [103](#), [107](#), [108](#), [110–113](#)
- Edge-Closing, [87](#), [88](#)
- Edge-Detection, [87–89](#), [92](#), [93](#), [98–103](#), [105–113](#), [615](#), [631](#), [632](#)
- Edge-Direction, [93](#), [96](#), [99](#), [103](#), [108](#), [111](#), [113](#), [975](#)
- Edge-Preservation, [161](#), [176](#)
- Edge-Thinning, [93](#), [631](#), [632](#)
- Edges, [87](#), [88](#), [615](#), [616](#), [631](#), [632](#), [647](#), [846](#), [847](#), [849](#), [851](#), [853](#), [855](#), [857](#), [859–861](#), [866](#)
- edges_image, [93](#)
- edges_sub_pix, [96](#)
- Elements, [726](#)
- eliminate_min_max, [162](#)
- eliminate_runs, [587](#)
- eliminate_sp, [164](#)
- Ellipse, [230](#), [343](#), [524](#), [528](#), [547](#), [548](#), [821](#), [947](#), [949](#), [950](#), [978](#)
- ellipse, [197](#), [198](#), [268](#), [939](#), [952](#)
- elliptic_axis, [548](#)
- elliptic_axis.gray, [343](#)
- elliptic_axis.xld, [950](#)
- emphasize, [115](#)
- End-Points, [593](#)
- Energy, [120](#)
- energy_gabor, [120](#)
- enquire_class_box, [3](#)
- enquire_reject_class_box, [4](#)
- Entropy, [181](#), [343](#), [347](#), [357](#)
- entropy_gray, [343](#)
- entropy_image, [181](#)
- equ_histo_image, [116](#)
- Erosion, [444](#), [450](#), [452](#), [453](#), [455](#), [464](#), [465](#), [467–470](#), [476](#), [477](#), [479](#), [482](#), [483](#), [485](#), [488–492](#), [494–497](#), [499](#), [501](#)
- erosion1, [464](#)
- erosion2, [465](#)
- erosion_circle, [467](#)
- erosion_golay, [468](#)
- erosion_rectangle1, [469](#)
- erosion_seq, [470](#)
- Error-code, [22](#)
- Error-Control, [652](#)
- Error-Message, [652](#)
- Error-Text, [652](#)
- Errors, [654](#)
- estimate, [837](#), [841](#), [844](#)
- estimate_alam, [888](#)
- estimate_sl_al_lr, [888](#)
- estimate_sl_al_zc, [889](#)
- estimate_tilt_lr, [889](#)
- estimate_tilt_zc, [889](#)
- Estimation, [888](#), [889](#)
- estimation, [837](#), [841](#), [844](#)
- Euler-number, [549](#)
- euler_number, [549](#)
- Evaluation, [881–886](#)
- example, [659](#)
- Exception, [22](#), [652](#)
- exhaustive_match, [142](#)
- exhaustive_match_mg, [143](#)
- Exit, [13](#)
- exit, [13](#)
- expand_domain_gray, [150](#)
- expand_gray, [619](#)
- expand_gray_ref, [621](#)
- expand_line, [623](#)
- expand_region, [588](#)
- Expansion, [588](#), [619](#), [621](#), [623](#)
- Exponential, [901](#)
- Extension, [41](#)
- Extern, [289](#), [299](#)
- fast_match, [409](#)
- fast_match_mg, [410](#)
- fast_threshold, [624](#)
- Feature, [340](#), [341](#), [343–345](#), [347](#), [348](#), [354–356](#), [541–546](#), [548](#), [549](#), [552–563](#), [566](#), [572–574](#), [576](#), [921–923](#), [942–946](#), [950](#), [960](#), [962–964](#), [969](#), [970](#)
- Feature-Space, [608](#), [612](#), [613](#), [628](#), [629](#)
- Features, [921](#), [922](#)
- FFT, [118–125](#), [127](#), [129–135](#)
- fft_generic, [121](#)
- fft_image, [122](#)
- fft_image_inv, [123](#)
- File, [670](#), [673](#)
- file-access, [45](#)
- File-Format, [46](#)
- file_exists, [45](#)
- File_format, [44](#)
- Fileheader, [42](#)
- fill_dvf, [145](#)
- fill_interlace, [165](#)
- fill_up, [589](#)
- fill_up_shape, [589](#)
- Filter, [118](#), [119](#), [126](#), [127](#), [141](#), [150](#), [302](#)
- Filter-Mask, [149](#)
- Filter-Width, [101](#), [167](#)
- filter_kalman, [837](#)
- find_1d_bar_code, [726](#)
- find_1d_bar_code_region, [729](#)
- find_2d_bar_code, [731](#)
- find_caltab, [763](#)
- find_marks_and_pose, [764](#)
- find_neighbors, [550](#)
- find_scaled_shape_model, [419](#)
- find_scaled_shape_models, [422](#)
- find_shape_model, [426](#)
- find_shape_models, [428](#)
- fit_circle_contour_xld, [951](#)
- fit_ellipse_contour_xld, [952](#)
- fit_line_contour_xld, [955](#)
- fit_surface_first_order, [344](#)
- fit_surface_second_order, [345](#)
- Fitting, [471](#), [472](#)
- fitting, [471](#)
- Floor-Function, [902](#)

- fnew_line, 49
- Font, 670, 673
- font, 269, 270, 272, 275
- font-size, 270
- for, 13
- Foreground, 585
- foreground, 713, 714, 716, 718, 720, 722
- Form, 567, 967
- Form-feature, 556–561
- Fourier-Transformation, 121–125, 129–135
- fourier_1dim, 794
- fourier_1dim_inv, 795
- Framegrabber, 368–376, 378, 379
- fread_char, 49
- fread_string, 50
- Frei-Filter, 98, 99
- frei_amp, 98
- frei_dir, 99
- Frequency, 350–352, 354
- Frequency-Domain, 118, 119, 121, 122, 124, 125, 127, 129–131
- full_domain, 337
- funct_1d_to_pairs, 801
- Function, 800
- Functions, 799, 801–805, 807, 808
- Fuzzy-Sets, 347, 348
- fuzzy_entropy, 347
- fuzzy_measure_pairing, 847
- fuzzy_measure_pairs, 849
- fuzzy_measure_pos, 851
- fuzzy_perimeter, 348
- fwrite_string, 50

- Gabor-Filter, 119, 120, 127
- Gaps, 588, 619, 621
- Gauss-Filter, 166, 177
- Gauss-Pyramid, 146
- gauss_distribution, 158
- gauss_image, 166
- Gaussian, 89, 92, 105, 106, 112, 113, 131, 166, 177, 603, 604, 625
- Gaussian-Function, 805
- gen_1d_bar_code_descr, 734
- gen_1d_bar_code_descr_gen, 735
- gen_2d_bar_code_descr, 736
- gen_bandfilter, 124
- gen_bandpass, 125
- gen_checker_region, 522
- gen_circle, 523
- gen_contour_polygon_rounded_xld, 935
- gen_contour_polygon_xld, 936
- gen_contour_region_xld, 937
- gen_contours_skeleton_xld, 938
- gen_cooc_matrix, 349
- gen_disc_se, 438
- gen_ellipse, 524
- gen_ellipse_contour_xld, 939
- gen_empty_obj, 509
- gen_empty_region, 526
- gen_filter_mask, 126
- gen_gabor, 127
- gen_gauss_pyramid, 146
- gen_grid_region, 526
- gen_highpass, 129
- gen_image1, 321
- gen_image1_extern, 322
- gen_image1_rect, 324
- gen_image3, 325
- gen_image_const, 327
- gen_image_gray_ramp, 328
- gen_image_proto, 329
- gen_image_surface_first_order, 330
- gen_image_surface_second_order, 331
- gen_image_to_world_plane_map, 766
- gen_lowpass, 130
- gen_measure_arc, 853
- gen_measure_rectangle2, 855
- gen_parallel_contour_xld, 975
- gen_parallels_xld, 940
- gen_polygons_xld, 941
- gen_psf_defocus, 183
- gen_psf_motion, 184
- gen_radial_distortion_map, 768
- gen_random_region, 528
- gen_random_regions, 528
- gen_rectangle1, 530
- gen_rectangle2, 531
- gen_region_contour_xld, 533
- gen_region_histo, 533
- gen_region_hline, 534
- gen_region_line, 535
- gen_region_points, 536
- gen_region_polygon, 537
- gen_region_polygon_filled, 538
- gen_region_polygon_xld, 538
- gen_region_runs, 539
- gen_sin_bandpass, 131
- gen_std_bandpass, 131
- gen_struct_elements, 472
- generic, 735
- Geocoding, 46, 53–55
- Geometry, 808–811, 813, 814, 817–819, 821, 823
- get_1d_bar_code, 738
- get_2d_bar_code, 739
- get_2d_bar_code_pos, 743
- get_bg_esti_params, 716
- get_channel_info, 503
- get_chapter_info, 658
- get_check, 652
- get_class_box_param, 4
- get_comprise, 240
- get_contour_angle_xld, 957
- get_contour_attrib_xld, 957
- get_contour_global_attrib_xld, 958
- get_contour_xld, 933
- get_domain, 338
- get_draw, 240
- get_error_text, 652

get_fix, 241
 get_fixed_lut, 215
 get_font, 269
 get_framegrabber_lut, 369
 get_framegrabber_param, 369
 get_grayval, 305
 get_hsi, 241
 get_icon, 242
 get_image_pointer1, 306
 get_image_pointer1_rect, 307
 get_image_pointer3, 308
 get_image_time, 309
 get_insert, 242
 get_keywords, 659
 get_line_approx, 243
 get_line_of_sight, 769
 get_line_style, 243
 get_line_width, 244
 get_lines_xld, 933
 get_lut, 215
 get_lut_style, 215
 get_mbutton, 221
 get_modules, 650
 get_mposition, 222
 get_mshape, 222
 get_next_socket_data_type, 684
 get_obj_class, 504
 get_operator_info, 659
 get_operator_name, 660
 get_os_window_handle, 283
 get_paint, 244
 get_pair_func1d, 801
 get_parallel_xld, 934
 get_param_info, 661
 get_param_names, 663
 get_param_num, 663
 get_param_types, 664
 get_part, 245
 get_part_style, 245
 get_pixel, 246
 get_points_ellipse, 821
 get_polygon_xld, 935
 get_pose_type, 770
 get_region_chain, 513
 get_region_contour, 514
 get_region_convex, 514
 get_region_index, 551
 get_region_points, 515
 get_region_polygon, 516
 get_region_runs, 516
 get_region_thickness, 552
 get_regress_params_xld, 959
 get_rgb, 247
 get_serial_param, 680
 get_shape, 247
 get_shape_model_origin, 432
 get_shape_model_params, 432
 get_socket_timeout, 684
 get_spy, 653
 get_string_extents, 270
 get_system, 670
 get_tposition, 271
 get_tshape, 271
 get_variation_model, 834
 get_window_attr, 285
 get_window_extents, 285
 get_window_pointer3, 286
 get_window_type, 287
 give_bg_esti, 718
 Gnuplot, 209–212
 gnuplot_close, 209
 gnuplot_open_file, 209
 gnuplot_open_pipe, 210
 gnuplot_plot_ctrl, 211
 gnuplot_plot_func1d, 211
 gnuplot_plot_image, 212
 Golay-Alphabet, 455, 460, 463, 468, 470, 473, 477, 479, 486, 487, 490, 495, 496, 499
 golay_elements, 473
 grab_image, 370
 grab_image_async, 371
 grab_image_start, 372
 grab_region, 373
 grab_region_async, 374
 Gradient-Direction, 828
 Graphic, 670, 673
 graphic, 240, 242
 Graphics, 24, 34, 224, 225, 227, 230, 289, 292, 295, 299, 761
 graphics, 250, 251
 Graphics-Software, 287
 Grauwert, 244
 Gray-Channels, 309–320
 Gray-Projections, 726, 729, 738
 gray-scale-image, 80
 Gray-surface, 330, 331
 Gray-Value, 232, 234, 335
 Gray-Value-Feature, 340, 341, 343–345, 347, 348, 354–356, 942, 960
 Gray-Value-Morphology, 441–445, 447, 448
 Gray-value-scaling, 118
 gray-value-spreading, 118
 Gray-Value-Threshold, 603–605, 607, 618, 624, 625, 644
 Gray-Values, 152, 329, 341, 343, 349, 354, 356, 379–382, 438–449, 619, 621, 942, 960
 Gray-values, 305, 350–353, 357
 gray_bothat, 439
 gray_closing, 440
 gray_closing_shape, 441
 gray_dilation, 442
 gray_dilation_rect, 442
 gray_dilation_shape, 443
 gray_erosion, 444
 gray_erosion_rect, 444
 gray_erosion_shape, 445
 gray_histo, 350
 gray_histo_abs, 351

- gray_inside, 151
- gray_opening, 446
- gray_opening_shape, 447
- gray_projections, 351
- gray_range_rect, 448
- gray_skeleton, 152
- gray_tophat, 448
- grayvalue, 240, 244, 246, 247, 250, 251, 256, 261, 267
- grayvalue-interpolation, 245, 266
- Grayvalues, 141
- Half-axis, 343, 548, 950
- Half-Images, 165
- Hammin-distance, 552, 553
- Hamming-Distance, 590
- hamming_change_region, 590
- hamming_distance, 552
- hamming_distance_norm, 553
- Hand-Eye-Calibration, 771
- hand_eye_calibration, 771
- Height-Field, 890–892, 894, 895
- height-plot, 261
- help, 657, 659, 661, 665
- Hesse, 534
- High-Pass-Filter, 100
- Highpass-Filter, 129
- highpass_image, 100
- Hilbert-Transform, 119, 120, 127
- histo_2dim, 352
- histo_to_thresh, 625
- Histogram, 350–352, 354, 358, 360, 533, 603–605, 608, 610, 625
- histogram, 261
- Histogram-linearisation, 116
- Histogram-spreading, 116
- History, 19, 24
- Hit-Or-Miss-Transformation, 473, 476, 477, 479, 494–497, 499
- hit_or_miss, 476
- hit_or_miss_golay, 477
- hit_or_miss_seq, 479
- HNF, 825–828, 830
- hole, 544, 549
- Holes, 589
- hom_mat2d.compose, 695
- hom_mat2d.identity, 696
- hom_mat2d.invert, 697
- hom_mat2d.rotate, 697
- hom_mat2d.scale, 698
- hom_mat2d.slant, 699
- hom_mat2d.to_affine_par, 701
- hom_mat2d.translate, 702
- hom_mat3d.compose, 703
- hom_mat3d.identity, 703
- hom_mat3d.invert, 704
- hom_mat3d.rotate, 704
- hom_mat3d.scale, 706
- hom_mat3d.to_pose, 778
- hom_mat3d.translate, 708
- Homogeneous-Transformation-Matrix, 46, 57, 59, 517, 693–699, 701–704, 706, 708–712, 758, 778, 783, 787, 972, 973
- Hopfield, 552, 553
- Hough-Transform, 824–828, 830
- hough_circle_trans, 824
- hough_circles, 824
- hough_line_trans, 825
- hough_line_trans_dir, 826
- hough_lines, 827
- hough_lines_dir, 828
- Hull, 514, 554, 573, 574, 969, 970
- hull, 268
- Hyper-Cube, 613, 629
- Hyper-Cuboid, 612, 628
- Hyper-Sphere, 613, 629
- Hyperbolic-Cosine, 900
- Hyperbolic-Sine, 906
- Hyperbolic-Tangent, 907
- Hysteresis, 627
- Hysteresis-Thresholding, 93, 96
- hysteresis_threshold, 627
- Iconic-Object, 19, 321, 507, 508, 510, 511, 649
- if, 15
- ifelse, 15
- illuminate, 117
- Illumination, 117
- Image, 234, 321, 322, 324, 325, 327, 328, 330, 331, 670, 673, 686, 688
- image-clipping, 245, 265, 266
- Image-Comparison, 831, 832, 834, 835, 837
- Image-directory, 41
- Image-Generation, 333, 334
- Image-grabbing, 369–374, 379
- image-matrix, 253
- Image-Object, 232, 238, 670, 673
- Image-object, 41, 321, 322, 324, 325, 327, 328, 330, 331, 503–506
- image-object, 253
- Image-Part, 361, 363
- image-part, 245, 265, 266
- Image-Plane, 344, 345, 355, 356
- Image-plane, 328
- image-repeat-memory, 258
- Image-Restoration, 188, 190
- image-sequences, 713, 714, 716, 718, 720, 722
- Image-Size, 361, 363–366
- Image-Types, 384
- Image_data, 42
- image_points_to_world_plane, 779
- image_to_channels, 320
- image_to_world_plane, 781
- Images, 651
- info_edges, 101
- info_framegrabber, 375
- info_ocr_class_box, 873
- info_parallel_xld, 960

- info_smooth, 167
- Information, 343, 375, 503–505
- information, 659
- Initialization, 651, 668–670, 673
- Inner-circle, 554, 567
- inner_circle, 554
- input, 273, 274
- input-parameter, 663
- inquiry, 665
- insert, 16
- Inside, 151
- inspect_shape_model, 433
- Inspection, 23
- int1_to_dvf, 385
- integer_to_obj, 510
- integrate_funct_1d, 801
- intensity, 353
- Interaction, 221, 222, 302
- interaction, 193–198, 200, 202–208
- interjacent, 591
- Interlace, 165
- Interpolation, 57, 59, 165, 972, 973
- interpolation, 245
- Intersection, 362, 579, 582, 583
- intersection, 579
- intersection_ll, 821
- invar_fourier_coeff, 795
- Inversion, 67
- invert_image, 67
- Iteration, 161

- JPEG, 41, 44, 282
- Junctions, 593
- junctions_skeleton, 593

- Kalman-filter, 713, 716, 718, 720, 722, 837, 841, 843, 844
- keyword, 659, 666
- Kirsch-Filter, 102, 103
- kirsch_amp, 102
- kirsch_dir, 103
- kKalmanfilter, 714

- Label-Image, 334, 540
- label_to_region, 540
- Lanser-Filter, 93, 96
- laplace, 105
- Laplace-Filter, 89, 92, 105, 106
- Laplace-of-Gaussian, 89, 92, 106
- Laplace-Operator, 616
- laplace_of_gauss, 106
- Laws-Filter, 181
- Ldexp, 903
- learn_class_box, 5
- learn_ndim_box, 628
- learn_ndim_norm, 629
- learn_sampset_box, 6
- Least-Squares-Adjustment, 802
- Length, 933–935
- length_xld, 960

- Level-Crossings, 645
- Licence, 650
- Light-Source, 888–892, 894, 895
- Line, 233, 387, 955, 978
- line, 200
- Line-Extraction, 136, 137, 139
- Line-feed, 49
- Line-Length, 393, 395
- Line-length, 396
- Line-segments, 955, 978
- line-width, 261
- line_orientation, 392
- line_position, 392
- Lines, 390, 392, 393, 395, 396, 526, 528, 534, 535, 600, 601, 615, 933
- lines_facet, 137
- lines_gauss, 139
- load_par_knowledge, 669
- Local-Maximum, 631, 632, 961
- Local-Threshold, 607, 618
- local_max, 630
- local_max_contours_xld, 961
- LoG, 89, 92, 106
- Logarithm, 903
- Logical-Operations, 923, 924
- Look-Up-Table, 153, 213–217, 219, 220, 241, 255
- Loop, 13, 17
- Lowpass-Filter, 130
- LUT, 153, 213–217, 219, 220, 241, 255, 369, 378, 670
- lut, 267
- lut_trans, 153

- Main-axes-of-inertia, 555–561
- Main-radius, 567, 967
- Manipulation, 919, 920
- Manual, 658
- manual, 657, 665
- map_image, 153
- Mapping, 153
- Mask, 74
- match_fourier_coeff, 796
- match_funct_1d_trans, 802
- Matching, 142, 143, 147, 399, 400, 402–406, 408–414, 417, 419, 422, 426, 428, 432–435, 830
- Matrices, 651
- max_image, 68
- max_parallel_xld, 962
- Maxima, 630, 633–635
- Maximum, 68, 147, 152, 354, 357, 442, 443, 808, 921
- Mean, 353, 357, 806
- Mean-Filter, 162, 164, 167, 169
- mean_image, 167
- mean_n, 168
- mean_sp, 169
- measure_pairs, 857
- measure_pos, 859

- measure_projection, 860
- measure_thresh, 861
- Measurement, 846, 847, 849, 851, 853, 855, 857, 859–861, 866
- measurements, 837, 841, 843, 844
- Medial-Axis, 599
- Median, 596
- Median-Filter, 162, 170, 172–175, 179, 437
- median_image, 170
- median_separate, 172
- median_weighted, 173
- Memory, 670
- merge_cont_line_scan_xld, 976
- merge_regions_line_scan, 593
- Metric, 585
- Mexican-Hat-Operator, 89, 92, 106
- Midrange-Filter, 174
- midrange_image, 174
- min_image, 68
- min_max_gray, 354
- Minima, 603–605, 625
- Minimum, 68, 147, 354, 357, 441, 444, 445, 447, 808, 816, 922
- Minkowski-Addition, 479, 481, 488–492
- Minkowski-Subtraction, 452, 453, 455, 482, 483
- minkowski_add1, 479
- minkowski_add2, 481
- minkowski_sub1, 482
- minkowski_sub2, 483
- mirror_image, 60
- mirror_region, 518
- mod_parallel_xld, 942
- Modules, 650, 725
- Moments, 343–345, 355, 547, 548, 555–561, 949, 950, 962, 963
- moments_any_xld, 962
- moments_gray_plane, 355
- moments_region2nd, 555
- moments_region2nd_invar, 556
- moments_region2nd_rel_invar, 557
- moments_region3rd, 558
- moments_region3rd_invar, 559
- moments_region_central, 560
- moments_region_central_invar, 561
- moments_xld, 963
- Monotony, 147
- monotony, 147
- morph_hat, 485
- morph_skeleton, 486
- morph_skiz, 487
- Morphology, 438–453, 455, 457–460, 462–465, 467–473, 476, 477, 479, 481–483, 485–497, 499, 501, 520
- Motion-Blur, 184, 187, 188, 190
- Mouse, 221–223
- move_contour_orig, 797
- move_rectangle, 288
- move_region, 519
- movement, 193–195
- mult_image, 69
- multichannel, 227
- Multiplication, 69, 805, 904
- negate_funct_1d, 803
- Negation, 803, 904
- Neighborhood, 145, 550
- neighborhood, 566, 576
- new_extern_window, 289
- new_line, 272
- Noise, 157–160, 188, 190, 971
- noise, 162
- Noise-Distribution, 157–160, 229, 971
- Noise-Removal, 597
- noise_distribution_mean, 159
- Non-linear-Filter, 161
- Non-Maximum-Suppression, 93, 631, 632
- nonmax_suppression_amp, 631
- nonmax_suppression_dir, 632
- not, 75
- num_points_funct_1d, 803
- obj_to_integer, 510
- Object, 234, 506, 509
- object, 42
- Object-choice, 551, 565
- Object-comparison, 505
- Object-Key, 510
- Object-Selection, 357
- OCR, 876, 877
- ocr_change_char, 874
- ocr_get_features, 874
- online-text, 659, 665, 666
- open_file, 51
- open_framegrabber, 376
- open_serial, 680
- open_socket_accept, 685
- open_socket_connect, 686
- open_textwindow, 292
- open_window, 295
- Opening, 437, 446, 448, 471, 472, 485, 488–492, 501
- opening, 488
- opening_circle, 489
- opening_golay, 490
- opening_rectangle1, 491
- opening_seg, 492
- Operating-System-Window-Handle, 283
- Optical-Flow, 147, 384, 385
- optical_flow_match, 147
- or, 75
- Orientatio, 567
- Orientation, 343, 392, 393, 395, 524, 531, 548, 562, 574, 726, 950, 964, 967, 970
- orientation_region, 562
- orientation_xld, 964
- Out-Of-Focus-Blur, 183, 186, 188, 190
- Outer-circle, 567
- Outliers, 977

- Output, [224](#), [225](#), [227](#), [228](#), [230](#), [232–234](#), [236](#), [237](#), [761](#)
- output, [240](#), [243](#), [244](#), [258](#), [277](#)
- output-parameter, [663](#)
- Overlap, [570](#)
- paint, [203](#)
- paint-gray, [379](#)
- paint-region, [380](#)
- paint_xld, [381](#)
- Parallelization, [668–670](#)
- Parallels, [934](#), [940](#), [942](#), [960](#), [962](#), [974](#)
- parameter, [244](#), [659](#), [661](#), [663–665](#)
- parameter-number, [663](#)
- Partition, [594](#), [595](#)
- partition-dynamic, [594](#)
- partition-lines, [393](#)
- partition-rectangle, [595](#)
- Pattern, [522](#), [526](#), [881–886](#)
- Perimeter, [348](#)
- Phase, [132](#), [133](#)
- phase_deg, [132](#)
- phase_rad, [133](#)
- phot_stereo, [890](#)
- Photometric-Stereo, [890](#)
- Pixel, [515](#), [536](#)
- Pixel-Relations, [670](#), [673](#)
- Pixel-Types, [384](#)
- Pixel_types, [42](#)
- Pixels, [526](#), [528](#)
- plane_deviation, [356](#)
- Plateaus, [633](#), [634](#)
- plateaus, [633](#)
- plateaus_center, [634](#)
- plot, [261](#)
- PNG, [41](#)
- point, [202](#)
- Polar-Transformation, [61](#)
- polar_trans_image, [61](#)
- Polygon, [235](#), [537](#), [538](#), [562](#)
- polygon, [203](#)
- Polygon-Approximation, [941](#)
- Polygon-approximation, [516](#)
- polygon-approximation, [259](#)
- Polygon-Length, [960](#)
- Polygons, [933–937](#), [941](#), [962](#), [973](#), [974](#), [981](#)
- Polyline, [235](#)
- Pose, [754](#), [758](#), [770](#), [778](#), [783](#), [786](#), [787](#), [791](#)
- Pose-relation, [566](#), [576](#)
- pose_to_hom_mat3d, [783](#)
- PostScript, [282](#)
- Pouring, [635](#)
- pouring, [635](#)
- Power-Function, [905](#)
- Power-Spectrum, [132–135](#)
- power_byte, [134](#)
- power_ln, [134](#)
- power_real, [135](#)
- prediction, [837](#), [841](#), [844](#)
- prep_contour_fourier, [798](#)
- prepare_variation_model, [834](#)
- Prewitt-Filter, [107](#), [108](#)
- prewitt_amp, [107](#)
- prewitt_dir, [108](#)
- Principal-Components, [154](#)
- principal_comp, [154](#)
- procedure, [663](#), [665](#)
- Procedure-Chapters, [658](#)
- procedure-name, [660](#)
- Product-of-inertia, [555–561](#)
- project_3d_point, [784](#)
- projection, [261](#)
- projection_pl, [823](#)
- Pruning, [493](#)
- pruning, [493](#)
- Pyramid, [143](#)
- query_all_colors, [248](#)
- query_color, [248](#)
- query_colored, [249](#)
- query_contour_attribs_xld, [965](#)
- query_contour_global_attribs_xld, [965](#)
- query_font, [272](#)
- query_gray, [250](#)
- query_insert, [250](#)
- query_line_width, [251](#)
- query_lut, [216](#)
- query_mshape, [223](#)
- query_operator_info, [665](#)
- query_paint, [251](#)
- query_param_info, [665](#)
- query_shape, [251](#)
- query_spy, [654](#)
- query_tshape, [273](#)
- query_window_type, [298](#)
- Radians, [905](#), [914](#)
- Radius, [343](#), [523](#), [524](#), [548](#), [950](#)
- Random-pattern, [528](#)
- Range, [354](#), [448](#)
- Range-Filter, [174](#)
- Rank-Filter, [170](#), [173–175](#), [179](#), [437](#)
- Rank-Gray-Value, [170](#), [173–175](#), [179](#), [437](#)
- Rank-Operator, [596](#)
- rank_image, [175](#)
- rank_region, [596](#)
- read_cam_par, [785](#)
- read_char, [273](#)
- read_class_box, [7](#)
- read_contour_xld_arc_info, [53](#)
- read_funct_id, [804](#)
- read_gray_se, [449](#)
- read_image, [41](#)
- read_kalman, [841](#)
- read_ocr, [875](#)
- read_ocr_trainf, [876](#)
- read_ocr_trainf_names, [876](#)
- read_ocr_trainf_select, [877](#)

- read_ocv, 884
- read_polygon_xld_arc_info, 54
- read_pose, 786
- read_region, 46
- read_sampset, 7
- read_sequence, 42
- read_serial, 681
- read_shape_model, 434
- read_string, 274
- read_template, 411
- read_tuple, 52
- read_variation_model, 835
- read_world_file, 46
- real_to_complex, 385
- receive_image, 686
- receive_region, 687
- receive_tuple, 687
- receive_xld, 688
- Rectangle, 236, 237, 362, 462, 469, 491, 528, 530, 531, 574, 582, 583, 594, 595, 970
- rectangle, 200, 204–207, 268
- rectangle1_domain, 338
- Rectification, 751–753
- Recursive-Filter, 177
- reduce_domain, 339
- reference, 659
- Reflection, 60, 518, 520
- Region, 46, 47, 234, 238, 503, 505, 513–516, 522, 526, 528, 534, 535, 585, 670, 673, 937
- region, 193–195, 208, 240, 247, 251
- Region-choice, 570
- region_to_bin, 333
- region_to_label, 334
- region_to_mean, 335
- Regiongrowing, 637–639
- regiongrowing, 637
- regiongrowing_mean, 638
- regiongrowing_n, 639
- Regions, 333, 334, 450–453, 455, 457–460, 462–465, 467–473, 476, 477, 479, 481–483, 485–497, 499, 501, 520, 533, 538, 578–581, 589, 598, 651, 687, 689, 938
- regress_contours_xld, 977
- Regression, 955, 957, 959, 961, 966, 971, 975, 977, 981–983
- Rekursive-Filters, 93, 96
- Relation, 649
- Relations, 651
- Remainder, 902
- remove_noise_region, 597
- reset_fuzzy_measure, 862
- reset_obj_db, 651
- Resolution-Pyramid, 143
- Restoration, 183, 184, 186, 187
- return_value, 659
- RGB, 79–81, 84
- rgb1_to_gray, 80
- rgb3_to_gray, 80
- Ridge, 147
- Rigid-Transformation, 709, 711, 754, 758, 770, 778, 783, 786, 787, 791
- roberts, 109
- Roberts-Filter, 109
- Robinson-Filter, 110, 111
- robinson_amp, 110
- robinson_dir, 111
- Robot-Vision, 771
- rotate_image, 62
- Rotation, 62, 697, 701, 704
- Roundness, 542, 543, 944, 945
- roundness, 562
- Run, 563, 564, 587
- run_bg_esti, 718
- Runlength-Code, 673
- Runlength-Coding, 670
- Runlength-coding, 516, 539, 564
- runlength_encoding, 587
- runlength_distribution, 563
- runlength_features, 564
- Runtime, 666
- sample_func1d, 804
- Save, 884, 886
- scale_image, 70
- scale_image_max, 118
- scale_y_func1d, 805
- Scaling, 62, 63, 70, 521, 698, 701, 706
- scaling, 118
- search_operator, 666
- Secondary-radius, 567, 967
- segment_contours_xld, 978
- Segmentation, 335, 373, 374, 387, 390, 540, 581, 608, 610, 612, 613, 635, 646, 763, 978
- select_contours_xld, 966
- select_gray, 357
- select_grayvalues_from_channels, 891
- select_lines, 395
- select_lines_longest, 396
- select_matching_lines, 830
- select_obj, 511
- select_region_point, 565
- select_region_spatial, 566
- select_shape, 567
- select_shape_proto, 570
- select_shape_std, 572
- select_shape_xld, 967
- Selection, 508, 510, 511, 925–927
- send_image, 688
- send_region, 689
- send_tuple, 689
- send_xld, 689
- sensor_kalman, 843
- Separating-Lines, 591
- Serial-Interface, 678–681, 683
- set_bg_esti_params, 720
- set_check, 654
- set_class_box_param, 8
- set_color, 252

- set_colored, 253
- set_comprise, 253
- set_draw, 254
- set_fix, 255
- set_fixed_lut, 216
- set_font, 275
- set_framegrabber_lut, 378
- set_framegrabber_param, 379
- set_fuzzy_measure, 863
- set_fuzzy_measure_norm_pair, 865
- set_gray, 256
- set_grayval, 382
- set_hsi, 257
- set_icon, 258
- set_insert, 258
- set_line_approx, 259
- set_line_style, 260
- set_line_width, 261
- set_lut, 217
- set_lut_style, 219
- set_mshape, 223
- set_offset_template, 411
- set_origin_pose, 787
- set_paint, 261
- set_part, 265
- set_part_style, 266
- set_pixel, 267
- set_reference_template, 412
- set_rgb, 267
- set_serial_param, 681
- set_shape, 268
- set_shape_model_origin, 435
- set_socket_timeout, 690
- set_spy, 655
- set_system, 673
- set_tposition, 276
- set_tshape, 277
- set_window_attr, 299
- set_window_dc, 299
- set_window_extents, 300
- set_window_type, 301
- sfs_mod_lr, 891
- sfs_orig_lr, 892
- sfs_pentland, 894
- shade_height_field, 895
- Shading, 890–892, 894, 895
- Shadows, 895
- Shape, 570
- Shape-Feature, 960, 962, 963
- Shape-feature, 542, 543, 546–548, 552, 553, 555, 562, 572, 944–946, 949, 950, 964
- Shape-Features, 589
- Shape-from-Shading, 890–892, 894
- Shape-Transformation, 598, 980
- shape_histo_all, 358
- shape_histo_point, 360
- shape_trans, 598
- shape_trans_xld, 980
- Sharpness, 115
- Shen-Filter, 93, 96, 177
- Shift, 73, 76
- Sigma-Filter, 176
- sigma_image, 176
- sim_caltab, 788
- Similarity, 572, 800
- Similarity-Transformation, 712
- simulate_defocus, 186
- simulate_motion, 187
- Sine, 131, 905
- Skeleton, 486, 487, 493, 497, 499, 599, 938
- skeleton, 599
- Slant, 699, 701, 888, 889
- slide_image, 302
- Slope, 147
- smallest_circle, 573
- smallest_circle_xld, 969
- smallest_rectangle1, 574
- smallest_rectangle1_xld, 969
- smallest_rectangle2, 574
- smallest_rectangle2_xld, 970
- smooth_contours_xld, 981
- smooth_funct_1d_gauss, 805
- smooth_funct_1d_mean, 806
- smooth_image, 177
- Smoothing, 89, 161, 162, 164–167, 169, 170, 172–177, 179, 437, 805, 806, 981
- Sobel-Filter, 112, 113
- sobel_amp, 112
- sobel_dir, 113
- socket_accept_connect, 691
- Sockets, 683–691
- sort_region, 599
- sp_distribution, 160
- Spatial-Domain, 121, 123, 126
- spatial_relation, 576
- Split-And-Merge, 600, 601
- split_contours_xld, 981
- split_skeleton_lines, 600
- split_skeleton_region, 601
- spreading, 118
- Square-Root, 906
- Standard-Deviation, 180, 921, 942, 959, 960, 966
- Stop, 17
- stop, 17
- Store, 884, 886
- store_par_knowledge, 670
- Storing-Capacity, 673
- String, 50
- String-Operators, 927–931
- Structure-factor, 547, 949
- Structuring-Elements, 438, 449
- sub_image, 71
- Subtraction, 71, 907
- Sum, 923
- Surrogate, 506, 510
- Surrounding-circle, 567, 573, 967, 969
- surrounding-circle, 268
- Surrounding-rectangle, 567, 574, 967, 969, 970

- Symmetry, 155
- symmetry, 155
- System-Call., 667
- System-Parameter, 670, 673
- system_call, 667
- systems, 837, 841, 844

- Tangent, 907
- Termination, 13
- Test-pixel, 551, 565, 577
- test_equal_obj, 505
- test_equal_region, 505
- test_obj_def, 506
- test_region_point, 577
- test_sampset_box, 9
- testdocr_class_box, 877
- Text, 292
- text, 269–275, 277
- text-cursor, 271, 273, 277
- Text-file, 48–51
- text-position, 276
- text-size, 270
- Texture, 172, 180, 181, 341, 343, 349
- Texture-Transformation, 181
- texture_laws, 181
- Thickening, 494–496
- thickening, 494
- thickening_golay, 495
- thickening_seq, 496
- Thinning, 152, 486, 487, 497, 499, 599
- thinning, 497
- thinning_golay, 499
- thinning_seq, 499
- Threshold, 358, 360, 603–605, 607, 616, 618, 624, 625, 627, 644, 861
- threshold, 644
- threshold_sub_pix, 645
- Thresholding, 603–605, 625
- TIFF, 41, 282
- Tiff, 44, 46
- tile_channels, 364
- tile_images, 365
- tile_images_offset, 366
- Tiling, 364–366
- Tilt, 889
- Time-Measure, 666
- Top-Hat, 448, 485, 501
- top_hat, 501
- Topographic-Primal-Sketch, 156
- topographic_sketch, 156
- Trace, 653
- train_variation_model, 835
- traindocr_class_box, 878
- traindocr_proj, 885
- trainfocr_class_box, 879
- Training, 628, 629, 876, 877, 885
- trans_from_rgb, 81
- trans_to_rgb, 84
- transform_func_t1d, 807
- Transformation, 153, 585, 766, 768, 779, 781, 802, 804, 807
- translate_measure, 866
- Translation, 519, 701, 702, 708, 787
- transpose_region, 520
- Transposition, 520
- trimmed_mean, 179
- truecolor, 261
- Tuple, 509, 526, 897–917, 919–931
- Tuple-Length, 921
- Tuple-length, 503
- tuple_abs, 897
- tuple_acos, 897
- tuple_add, 898
- tuple_and, 923
- tuple_asin, 898
- tuple_atan, 899
- tuple_atan2, 899
- tuple_band, 908
- tuple_bnot, 908
- tuple_bor, 909
- tuple_bxor, 909
- tuple_ceil, 899
- tuple_chr, 913
- tuple_chrt, 914
- tuple_concat, 919
- tuple_cos, 900
- tuple_cosh, 900
- tuple_deg, 914
- tuple_deviation, 921
- tuple_div, 901
- tuple_environment, 927
- tuple_equal, 911
- tuple_exp, 901
- tuple_fabs, 901
- tuple_first_n, 925
- tuple_floor, 902
- tuple_fmod, 902
- tuple_gen_const, 919
- tuple_greater, 911
- tuple_greater_equal, 911
- tuple_inverse, 919
- tuple_is_number, 915
- tuple_last_n, 925
- tuple_ldexp, 903
- tuple_length, 921
- tuple_less, 912
- tuple_less_equal, 912
- tuple_log, 903
- tuple_log10, 903
- tuple_lsh, 910
- tuple_max, 921
- tuple_mean, 922
- tuple_min, 922
- tuple_mult, 904
- tuple_neg, 904
- tuple_not, 923
- tuple_not_equal, 913
- tuple_number, 915

- tuple_or, [924](#)
- tuple_ord, [916](#)
- tuple_ords, [916](#)
- tuple_pow, [905](#)
- tuple_rad, [905](#)
- tuple_real, [916](#)
- tuple_round, [917](#)
- tuple_rsh, [910](#)
- tuple_select, [926](#)
- tuple_select_range, [926](#)
- tuple_sin, [905](#)
- tuple_sinh, [906](#)
- tuple_sort, [920](#)
- tuple_sort_index, [920](#)
- tuple_split, [928](#)
- tuple_sqrt, [906](#)
- tuple_str_bit_select, [927](#)
- tuple_str_first_n, [928](#)
- tuple_str_last_n, [929](#)
- tuple_strchr, [929](#)
- tuple_string, [917](#)
- tuple_strlen, [930](#)
- tuple_strrchr, [930](#)
- tuple_strrstr, [931](#)
- tuple_strstr, [931](#)
- tuple_sub, [907](#)
- tuple_sum, [923](#)
- tuple_tan, [907](#)
- tuple_tanh, [907](#)
- tuple_xor, [924](#)
- type, [661](#)
- Type-Conversion, [383–385](#)
- types, [664](#)
- Union, [580, 581](#)
- union1, [580](#)
- union2, [581](#)
- union_straight_contours_histo_xld, [982](#)
- union_straight_contours_xld, [983](#)
- update-file, [844](#)
- update_bg_esti, [722](#)
- update_kalman, [844](#)
- Variation-Model, [831, 832, 834, 835, 837](#)
- vector_angle_to_rigid, [709](#)
- vector_to_hom_mat2d, [710](#)
- vector_to_rigid, [711](#)
- vector_to_similarity, [712](#)
- Verification, [881–886](#)
- wait_seconds, [667](#)
- Watersheds, [151, 646](#)
- watersheds, [646](#)
- Weighting, [173](#)
- while, [17](#)
- Wiener-Filter, [183, 184, 188, 190](#)
- wiener_filter, [188](#)
- wiener_filter_ni, [190](#)
- Window, [19, 20, 24, 34, 280, 289, 292, 295, 298, 299, 670, 673](#)
- Window-Dump, [282](#)
- Window-Position, [300](#)
- Window-Size, [285, 300](#)
- Window-size, [34](#)
- Window-Types, [298](#)
- World-File, [46, 53–55](#)
- write_cam_par, [789](#)
- write_class_box, [9](#)
- write_contour_xld_arc_info, [54](#)
- write_funct_1d, [807](#)
- write_image, [44](#)
- write_lut, [220](#)
- write_ocr, [879](#)
- write_ocr_trainf, [880](#)
- write_ocr_trainf_image, [880](#)
- write_ocv, [886](#)
- write_polygon_xld_arc_info, [55](#)
- write_pose, [791](#)
- write_region, [47](#)
- write_serial, [683](#)
- write_shape_model, [435](#)
- write_string, [277](#)
- write_template, [412](#)
- write_tuple, [52](#)
- write_variation_model, [837](#)
- X-Window, [670, 673](#)
- x_range_funct_1d, [808](#)
- XLD, [234, 239, 381, 533, 538, 688, 689, 752, 933–942, 945, 947, 951, 952, 955, 957–962, 965, 966, 971–975, 977, 978, 981–983](#)
- XLD-Contours, [53, 54, 239, 533, 538, 688, 689, 752, 933, 935–941, 945, 947, 951, 952, 955, 957–959, 961, 965, 966, 971–973, 975, 977, 978, 981–983](#)
- XLD-Parallels, [239, 688, 689, 934, 940, 942, 960, 962, 974](#)
- XLD-Polygons, [54, 55, 239, 688, 689, 933–935, 941, 962, 973, 974, 981](#)
- xor, [78](#)
- y_range_funct_1d, [808](#)
- Zero-Crossings, [616, 647](#)
- zero_crossing, [647](#)
- zero_crossing_sub_pix, [647](#)
- zoom_image_factor, [62](#)
- zoom_image_size, [63](#)
- zoom_region, [521](#)
- Zooming, [521](#)