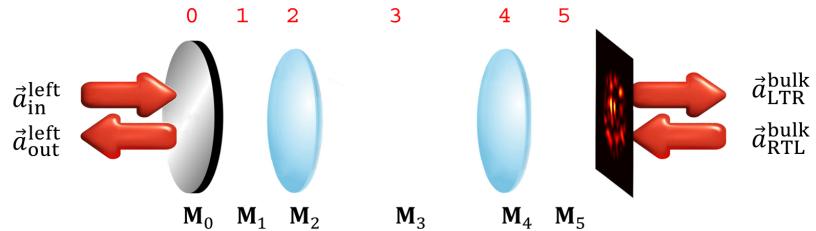


**fourier-cavity-sim**  
A Python Library for Fourier-Optics  
Cavity Simulations



Helmut Hörner  
TU Wien  
[helmut.hoerner@tuwien.ac.at](mailto:helmut.hoerner@tuwien.ac.at)

August 24, 2025



# License and Citation

**License (Documentation).** This user manual is licensed under the [Creative Commons Attribution 4.0 International \(CC BY 4.0\)](#) license. You are free to share and adapt the material for any purpose, provided you give appropriate credit.

This work is licensed under a [Creative Commons “Attribution 4.0 International”](#) license.



**License (Software).** The software `fourier-cavity-sim` is licensed under the MIT License. See the `LICENSE` file in the repository. The CC BY 4.0 above applies to the documentation only.

**How to Cite.** If you use this software or manual in a scientific publication, please cite the user manual:

Hoerner, H. (2025). *fourier-cavity-sim: A Python Library for Fourier-Optics Cavity Simulations* (User Manual). TU Wien.

Available at: [https://github.com/HelmutHoerner/fourier-cavity-sim/blob/main/docs/fo\\_cavity\\_sim\\_user\\_manual.pdf](https://github.com/HelmutHoerner/fourier-cavity-sim/blob/main/docs/fo_cavity_sim_user_manual.pdf)

BibTeX:

```
@report{Hoerner2025_fo_cavity_sim_manual,
  author      = {Helmut Hörner},
  title       = {fo_cavity_sim: A Python Library for Fourier-Optics
                 Cavity Simulations},
  institution = {TU Wien},
  year        = {2025},
  url         = {https://github.com/HelmutHoerner/fourier-cavity-sim/
                 blob/main/docs/fo_cavity_sim_user_manual.pdf}
}
```

A machine-readable citation is also provided in `CITATION.cff` at the repository root.

# Contents

<b>License and Citation</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Installation</b>	<b>3</b>
2.1 Installation on a Local Computer . . . . .	3
2.1.1 Prerequisites . . . . .	3
2.1.2 Get and install Conda . . . . .	3
2.1.3 Using Anaconda (Full Distribution) . . . . .	4
2.1.4 Create Virtual Environment and Install Library . . . . .	4
2.2 Installation on a Cluster . . . . .	5
<b>3 Light Propagation in One Direction</b>	<b>7</b>
3.1 Fourier Optics Theory . . . . .	7
3.1.1 Fourier-Optics Example 1: Free-Space Propagation . . . . .	8
3.1.2 Fourier-Optics Example 2: Thin-Lens Phase Mask . . . . .	9
3.2 Code Example 1: Light Propagation Through Two Lenses . . . . .	10
3.2.1 First Attempt . . . . .	11
3.2.2 Critical Sampling Condition . . . . .	14
3.2.3 Second Attempt: Grid Size Chosen by the Critical-Sampling Rule . . . . .	16
3.3 Code Example 2: Talbot Effect . . . . .	18
<b>4 Linear Cavities</b>	<b>23</b>
4.1 Multiple Round-Trips Method . . . . .	23
4.1.1 Theory . . . . .	23
4.1.2 Code Example: MAD-CPA, Multiple Roundtrips Method . . . . .	24
4.1.3 Simulation Results . . . . .	30
4.2 Geometric Series Method . . . . .	31
4.2.1 Theory: Transmission and Reflection Matrices . . . . .	31
4.2.2 Geometric-series method . . . . .	33
4.2.3 Code Example: MAD-CPA, Geometric Series Method . . . . .	34
4.2.4 Simulation Results . . . . .	37
4.3 Scattering and Transfer Matrix Approach . . . . .	41
4.3.1 Theory: Scattering- and Transfer-Matrices . . . . .	42
4.3.2 Algorithm for calculating left and right output fields . . . . .	43
4.3.3 Intracavity field from the scattering/transfer matrices . . . . .	44
4.3.4 Code Example: EP-MAD-CPA, Scattering-Matrix Method . . . . .	45

4.3.5	Simulation Results . . . . .	50
<b>5</b>	<b>Ring Cavities</b>	<b>57</b>
5.1	Theory . . . . .	58
5.1.1	Four-Port Scattering Matrices . . . . .	58
5.1.2	Four-Port Transfer Matrices . . . . .	60
5.1.3	Assembling a ring cavity from left to right . . . . .	60
5.1.4	Algorithm for calculating left and right output fields . . . . .	61
5.1.5	Algorithm for calculating intracavity fields . . . . .	62
5.1.6	Code Example: Ring Cavity, Scattering-Matrix Method . . . . .	63
5.2	Simulation Results . . . . .	68
<b>6</b>	<b>Running Parallel Tasks on a Cluster</b>	<b>75</b>
6.1	Single-Step Mode . . . . .	75
6.1.1	Code Example: EP-MAD-CPA, Single-Step Cluster Simulation . . . . .	75
6.1.2	Slurm File Configuration . . . . .	81
6.1.3	Simulation Results . . . . .	82
6.2	Generated Images . . . . .	92
<b>7</b>	<b>Component Overview</b>	<b>93</b>
7.1	Light Field Classes . . . . .	93
7.2	Optical Components . . . . .	93
7.2.1	2-Port Component Classes . . . . .	94
7.2.2	4-Port Component Classes . . . . .	98
<b>A</b>	<b>Helper Functions and Classes</b>	<b>101</b>
A.1	Enumeration Types . . . . .	101
Dir	. . . . .	101
Dir2	. . . . .	101
Side	. . . . .	102
Res	. . . . .	102
Path	. . . . .	102
A.2	Helper Functions . . . . .	102
get_sample_vec(NoOfPoints, LR_range, center, stacked, p=1)	. . . . .	102
delta_f_to_delta_lambda(delta_f_hz, lambda_center_m, n=1.0)	. . . . .	103
polar_interpolation(z1, z2, w1=0.5)	. . . . .	103
find_best_match(ax, x)	. . . . .	104
A.3	Matrix Representation and Operations . . . . .	104
is_matrix(X)	. . . . .	105
mat_is_zero(X)	. . . . .	105
mat_plus(X, Y)	. . . . .	106
mat_plus3(X, Y, Z)	. . . . .	106
mat_minus(X, Y)	. . . . .	107
mat_mul(X, Y)	. . . . .	107
mat_mul3(X, Y, Z)	. . . . .	108
mat_div(X, Y)	. . . . .	108
mat_inv(X)	. . . . .	109
mat_inv_X_mul_Y(X, Y)	. . . . .	109

mat_conj_transpose(X) . . . . .	110
A.4 Class <code>clsBlockMatrix</code> . . . . .	110
A.4.1 Initialization . . . . .	110
<code>clsBlockMatrix(dimension, file_caching, tmp_dir="", name="")</code>	110
A.4.2 Setter and Getter Methods . . . . .	111
<code>.set_block(line, col, X, clone=False)</code> . . . . .	111
<code>.get_block(line, col)</code> . . . . .	111
<code>.set_quadrant(quadrant, X, clone=False)</code> . . . . .	112
<code>.get_quadrant(quadrant, clone=False)</code> . . . . .	112
<code>.set_from_quadrants(A, B, C, D, clone=False)</code> . . . . .	113
<code>.clone(X)</code> . . . . .	113
<code>.clear()</code> . . . . .	113
A.4.3 Test if Zero . . . . .	114
<code>.is_zero()</code> . . . . .	114
<code>.block_is_zero(line, col)</code> . . . . .	114
A.4.4 Helper Properties and Methods . . . . .	114
<code>.file_caching</code> . . . . .	114
<code>.tmp_dir</code> . . . . .	114
<code>.dim</code> . . . . .	115
<code>.file_names</code> . . . . .	115
<code>.is_saved_in_tmp_file(i, j)</code> . . . . .	115
<code>.set_tmp_file_name(i, j, name)</code> . . . . .	115
<code>.get_tmp_file_name(i, j)</code> . . . . .	116
<code>.keep_tmp_files</code> . . . . .	116
<code>.keep_alive</code> . . . . .	116
A.4.5 Global Functions for Processing Block Matrices . . . . .	117
<code>bmat2_plus(X, Y)</code> . . . . .	117
<code>bmat2_minus(X, Y)</code> . . . . .	117
<code>bmat_flip_sign(X)</code> . . . . .	117
<code>bmat_mul(X, Y, p=None, msg="")</code> . . . . .	117
<code>bmat_mul3(X, Y, Z, p=None, msg="")</code> . . . . .	118
<code>bmat2_inv(X, p=None, msg="")</code> . . . . .	118
<code>bmat4_inv(X, p=None, msg="")</code> . . . . .	119
<code>bmat_conj_transpose(X)</code> . . . . .	119
<code>bmat2_M_to_S(M, p=None, msg="")</code> . . . . .	120
<code>bmat4_M_to_S(M, p=None, msg="")</code> . . . . .	120
A.5 Class <code>clsProgressPrinter</code> . . . . .	121
A.5.1 Initialization . . . . .	122
<code>clsProgressPrinter()</code> . . . . .	122
A.5.2 Main Methods . . . . .	122
<code>.push_print(msg, start_time=0)</code> . . . . .	122
<code>.print(msg)</code> . . . . .	122
<code>.pop()</code> . . . . .	123
<code>.tic_reset(goal_count, always_print_first_and_last=True, msg="")</code> . . . . .	123
<code>.tic()</code> . . . . .	123
A.5.3 Configuration Properties . . . . .	124

.min_time_to_show . . . . .	124
.max_time_betw_tic_outputs . . . . .	124
.max_time_without_tic_duration_output . . . . .	125
.indent . . . . .	125
.silent . . . . .	125
<b>A.6 Class <code>clsTaskManager</code></b> . . . . .	125
<b>A.6.1 Initialization</b> . . . . .	126
<code>clsTaskManager(simulations, steps_per_simulation, folder)</code> . . . . .	126
<b>A.6.2 Core Configuration</b> . . . . .	127
<code>.simulations</code> . . . . .	127
<code>.steps_per_simulation</code> . . . . .	127
<code>.sleep_time</code> . . . . .	127
<code>.folder</code> . . . . .	127
<b>A.6.3 Task Scheduling</b> . . . . .	127
<code>.get_next_task()</code> . . . . .	127
<code>.grab_task(sim, step)</code> . . . . .	128
<code>.end_task(sim, step)</code> . . . . .	129
<b>A.6.4 House-Keeping Utilities</b> . . . . .	129
<code>.delete_all_files()</code> . . . . .	129
<b>A.7 Class <code>clsGrid</code></b> . . . . .	130
<b>A.7.1 Initialization</b> . . . . .	131
<code>clsGrid(cavity)</code> . . . . .	131
<b>A.7.2 Grid Definition Methods</b> . . . . .	131
<code>.set_res(res_fov, res_tot, length_fov)</code> . . . . .	131
<code>.set_opt_res_based_on_sidelength(length_fov, factor, prop_dist, even)</code> . . . . .	132
<code>.set_opt_res_tot_based_on_res_fov(length_fov, res_fov, prop_dist)</code> . . . . .	133
<b>A.7.3 Grid Geometry and Field-of-View Handling</b> . . . . .	133
<code>.factor</code> . . . . .	133
<code>.res_fov</code> . . . . .	133
<code>.res_tot</code> . . . . .	134
<code>.length_fov</code> . . . . .	134
<code>.length_fov_mm</code> . . . . .	134
<code>.length_tot</code> . . . . .	134
<code>.length_tot_mm</code> . . . . .	134
<code>.even_res</code> . . . . .	134
<code>.odd_res</code> . . . . .	134
<code>.fov_pos</code> . . . . .	134
<code>.pos_offset_x</code> . . . . .	135
<code>.pos_offset_y</code> . . . . .	135
<code>.extract_FOV(E_in, k_space_in, k_space_out)</code> . . . . .	136
<code>.embed_image(E_in, k_space_in, k_space_out)</code> . . . . .	136
<code>.get_res_arr(X)</code> . . . . .	136
<code>.is_fov_res(X)</code> . . . . .	137
<code>.get_res_TR(TR)</code> . . . . .	137
<code>.get_res_vec(vec)</code> . . . . .	137
<b>A.7.4 Axes Metrics</b> . . . . .	138

.axis_fov . . . . .	138
.axis_tot . . . . .	138
.n_axis_fov . . . . .	138
.n_axis_tot . . . . .	139
.k_axis_fov . . . . .	139
.k_axis_tot . . . . .	139
.dist_to_pixels(dist) . . . . .	139
.get_ax_plot_info() . . . . .	140
A.7.5 Conversions . . . . .	140
.convert(E_in, k_space_in, k_space_out, fov_out) . . . . .	140
.arr_to_vec(X, k_space_in, column_vec=True) . . . . .	141
.vec_to_arr(X, k_space_out) . . . . .	141
fft2_phys_spatial(X) . . . . .	142
ifft2_phys_spatial(X) . . . . .	142
.convert_TR_mat_tot_to_fov(X, tics=0) . . . . .	143
.convert_TR_bmat2_tot_to_fov(X) . . . . .	143
A.7.6 Helper Methods and Properties . . . . .	144
.cavity . . . . .	144
.fourier_basis_func(nx, ny, tot, k_space_out) . . . . .	144
.empty_grid(fov_only) . . . . .	144
.stretch_x(old_array, x0, c) . . . . .	145
.stretch_y(old_array, y0, c) . . . . .	145
.get_aperture_mask(aperture, anti_alias_factor, black_value=0, consider_pos_offset=False) . . . . .	146
.get_soft_aperture_mask(aperture, epsilon, black_value) . . . . .	146
.get_angle_from_nxy(tot, nx, ny, Lambda, nr) . . . . .	147
.get_sorted_mode_numbers(fov_only, n_max, return_all_col) . . . . .	148
.mode_numbers_fov . . . . .	148
.mode_numbers_tot . . . . .	148
.mode_indices_fov . . . . .	149
.mode_indices_tot . . . . .	149
.get_row_col_idx_from_nx_ny(tot, nx, ny) . . . . .	149
.get_center_index(fov_only) . . . . .	150
.limit_mode_numbers(X, mode_limit, k_space_in, k_space_out) . . .	150
<b>B Light Fields</b> . . . . .	<b>153</b>
B.1 Light-Field Classes . . . . .	153
B.2 Base Class <code>clsLightField</code> . . . . .	154
B.2.1 Initialization . . . . .	154
<code>clsLightField(grid)</code> . . . . .	154
B.2.2 Setter and Getter Methods . . . . .	155
<code>.set_field(field, k_space)</code> . . . . .	155
<code>.set_field_vec(vec)</code> . . . . .	155
<code>.get_field_tot(k_space_out, process=0)</code> . . . . .	156
<code>.get_field_fov(k_space_out, process=0)</code> . . . . .	156
<code>.get_field_tot_vec(column_vec=True)</code> . . . . .	157
<code>.get_field_fov_vec(column_vec=True)</code> . . . . .	157
<code>.clone()</code> . . . . .	157

B.2.3	Manipulating the Light Field . . . . .	158
	.apply_TR_mat(TR) . . . . .	158
	.add(other_field) . . . . .	158
	.stretch_x(factor, center_pos=0) . . . . .	159
	.stretch_y(factor, center_pos=0) . . . . .	159
	.shift(x_shift, y_shift) . . . . .	160
B.2.4	Plotting and Analysis . . . . .	161
	.intensity_integral_fov(aperture=0) . . . . .	161
	.intensity_integral_tot(aperture=0) . . . . .	161
	.plot_field(what_to_plot, fov_only=True, save_path=None, c_map='hot', vmax_limit=None, norm=None, vmax=None) . . . . .	161
B.2.5	Properties . . . . .	163
	.grid . . . . .	163
	.name . . . . .	163
	.empty . . . . .	163
	.fov_only . . . . .	163
	.k_space . . . . .	164
	.intensity_sum_fov . . . . .	164
	.intensity_sum_tot . . . . .	164
B.3	Class clsGaussBeam . . . . .	164
B.3.1	Initialization . . . . .	165
	clsGaussBeam(grid) . . . . .	165
B.3.2	Methods . . . . .	165
	.create_beam(waist, x_offset=0, y_offset=0, x_angle_deg=0, y_angle_deg=0, z=0) . . . . .	165
	.beam_radius(z) . . . . .	166
	.radius_of_curvature(z) . . . . .	167
	.gouy_phase(z) . . . . .	168
B.4	Class clsSpeckleField . . . . .	168
B.4.1	Initialization . . . . .	169
	clsSpeckleField(grid) . . . . .	169
B.4.2	Speckle-Generation Methods . . . . .	170
	.create_field(no_of_modes, aperture, seed, fov_equiv=False, consider_pos_offset=True) . . . . .	170
	.create_field_eq_distr(no_of_modes, n_max, aperture, seed, Lambda=0, consider_pos_offset=True) . . . . .	171
B.4.3	Plotting and Analysis . . . . .	172
	.plot_angle_distribution(fov_only, no_of_modes=-1, save_path=None, Lambda=-1) . . . . .	172
B.4.4	Diagnostic and Helper Methods . . . . .	173
	.get_aperture() . . . . .	173
	.get_n_max() . . . . .	173
	.get_no_of_modes() . . . . .	174
	.get_max_angle(Lambda, nr) . . . . .	174
	.get_max_angle_deg(Lambda, nr) . . . . .	175
	.get_req_emebed_factor(dist, Lambda, nr, alpha_deg=-1) . . . . .	175
B.5	Class clsTestImage . . . . .	176
B.5.1	Initialization . . . . .	176
	clsTestImage(grid) . . . . .	176

B.5.2	Test Image Creation . . . . .	176
	.create_test_image(image, flip_horizontal=False, flip_vertical=False) . . . . .	176
B.6	Class clsPlaneWaveMixField . . . . .	177
B.6.1	Initialization . . . . .	177
	clsPlaneWaveMixField(grid) . . . . .	177
B.6.2	Add Plane Wave . . . . .	178
	.add_fourier_basis_func(nx, ny, amplitude) . . . . .	178
<b>C</b>	<b>Optical Components: Main Class</b>	<b>179</b>
C.1	Abstract Base Class clsOptComponent . . . . .	180
C.1.1	Initialization . . . . .	181
	clsOptComponent(name, cavity) . . . . .	181
	._connect_to_cavity(cavity, grid, idx) . . . . .	182
C.1.2	Properties . . . . .	182
	.name . . . . .	182
	.idx . . . . .	183
	.Lambda . . . . .	183
	.Lambda_nm . . . . .	183
	.grid . . . . .	184
	.cavity . . . . .	184
C.1.3	Helper Properties for Efficient Chaining . . . . .	184
	.k_space_in_prefer . . . . .	185
	.k_space_in_dont_care . . . . .	185
	.k_space_out_prefer . . . . .	186
	.k_space_out_dont_care . . . . .	186
C.1.4	Memory and Temporary File Management . . . . .	187
	.mem_cache_M_bmat . . . . .	187
	.file_cache_M_bmat . . . . .	187
	.clear_mem_cache() . . . . .	188
	.load_M_bmat_tot() . . . . .	188
	.load_inv_M_bmat_tot() . . . . .	189
	.save_M_bmat_tot() . . . . .	189
	.save_inv_M_bmat_tot() . . . . .	189
	.delete_M_bmat_tot() . . . . .	189
<b>D</b>	<b>2-Port Optical Components for Linear Cavities</b>	<b>191</b>
D.1	Abstract Base Class clsOptComponent2port . . . . .	191
D.1.1	Initialization . . . . .	192
	clsOptComponent2port(name, cavity) . . . . .	192
D.1.2	Light Field Propagation . . . . .	192
	.prop(E_in, k_space_in, k_space_out, direction) . . . . .	192
D.1.3	Transmission and Reflection Matrices . . . . .	193
	.T_LTR_mat_tot . . . . .	193
	.T_RTL_mat_tot . . . . .	194
	.R_L_mat_tot . . . . .	194
	.R_R_mat_tot . . . . .	194
D.1.4	Scattering and Transfer Block Matrix . . . . .	195

.S_bmat_tot . . . . .	195
.M_bmat_tot . . . . .	196
.inv_M_bmat_tot . . . . .	196
.calc_inv_M_bmat_tot() . . . . .	197
D.1.5 Other Properties . . . . .	197
.dist_phys . . . . .	197
.dist_opt . . . . .	197
.symmetric . . . . .	198
D.2 Class <code>clsPropagation</code> . . . . .	198
D.2.1 Initialization . . . . .	198
<code>clsPropagation(name, cavity)</code> . . . . .	198
D.2.2 Physical Parameter Definition . . . . .	199
.set_params( <code>dist_phys</code> , <code>n</code> ) . . . . .	199
.set_dist_opt( <code>d_opt</code> ) . . . . .	199
.set_ni_based_on_T( <code>T</code> ) . . . . .	200
D.2.3 Light Field Propagation . . . . .	201
.transfer_function . . . . .	201
.prop( <code>E_in</code> , <code>k_space_in</code> , <code>k_space_out</code> , <code>direction</code> ) . . . . .	201
D.2.4 Transmission and Reflection Matrices . . . . .	202
.T_LTR_mat_tot . . . . .	202
.T_RTL_mat_tot . . . . .	203
.R_L_mat_tot . . . . .	203
.R_R_mat_tot . . . . .	203
.calc_T_mat_tot() . . . . .	203
D.2.5 Other Properties and Methods . . . . .	204
.symmetric . . . . .	204
.n . . . . .	204
.dist_opt . . . . .	204
.clear_mem_cache() . . . . .	204
D.3 Class <code>clsSplitPropagation</code> . . . . .	205
D.3.1 Initialization . . . . .	205
<code>clsSplitPropagation(name, cavity)</code> . . . . .	205
D.3.2 Physical Parameter Definition . . . . .	206
.set_params( <code>dist_phys</code> , <code>n1</code> , <code>n2</code> , <code>dist_corr1=0</code> , <code>dist_corr2=0</code> ) . . . . .	206
.set_dist_opt( <code>d_opt</code> , <code>top_bottom</code> ) . . . . .	206
.set_ni_based_on_T( <code>T</code> , <code>top_bottom</code> ) . . . . .	207
D.3.3 Light Field Propagation . . . . .	208
.transfer_function . . . . .	208
.prop( <code>E_in</code> , <code>k_space_in</code> , <code>k_space_out</code> , <code>direction</code> ) . . . . .	208
D.3.4 Transmission and Reflection Matrices . . . . .	209
.T_LTR_mat_tot . . . . .	209
.T_RTL_mat_tot . . . . .	209
.R_L_mat_tot . . . . .	210
.R_R_mat_tot . . . . .	210
.calc_T_mat_tot() . . . . .	210
D.3.5 Other Properties and Methods . . . . .	211
.symmetric . . . . .	211

.n1 . . . . .	211
.n2 . . . . .	211
.dist_opt1 . . . . .	211
.dist_opt2 . . . . .	212
.dist_opt . . . . .	212
.clear_mem_cache() . . . . .	212
D.4 Class <code>clsThinLens</code> . . . . .	212
D.4.1 Initialization . . . . .	213
<code>clsThinLens(name, cavity)</code> . . . . .	213
D.4.2 Optical Parameters . . . . .	214
.f . . . . .	214
.f_mm . . . . .	214
.aperture . . . . .	214
.aperture_mm . . . . .	215
.set_aperture_based_on_NA(NA) . . . . .	215
.aperture_anti_alias . . . . .	216
.aperture_black_value . . . . .	216
D.4.3 Lens Type . . . . .	216
.lens_type_spherical . . . . .	216
.lens_type_perfect . . . . .	217
D.4.4 Light Field Propagation . . . . .	217
.prop(E_in, k_space_in, k_space_out, direction) . . . . .	217
.T_LTR_mat_tot . . . . .	218
.T_RTL_mat_tot . . . . .	219
.calc_T_mat_tot() . . . . .	219
.lens_mask . . . . .	219
D.4.5 Residual Reflection . . . . .	220
.set_residual_reflection(R, reflection_aperture, epsilon, black_value) . . . . .	220
.set_phys_behavior(sym_phase) . . . . .	221
.R_residual . . . . .	222
.r_residual . . . . .	222
.t_residual . . . . .	223
.reflection_aperture . . . . .	223
.reflection_aperture_black . . . . .	223
.reflection_aperture_epsilon . . . . .	224
.R_L_mat_tot . . . . .	224
.R_R_mat_tot . . . . .	225
.calc_R_mat_tot() . . . . .	225
.reflection_mask1 . . . . .	226
.reflection_mask2 . . . . .	226
.reflection_mask . . . . .	227
D.4.6 Symmetry and Space-Preferences . . . . .	227
.symmetric . . . . .	227
.k_space_in_dont_care . . . . .	227
.k_space_in_prefer . . . . .	228
.k_space_out_dont_care . . . . .	228
.k_space_out_prefer . . . . .	228

D.4.7	Memory Management and Other Settings . . . . .	228
	.clear_mem_cache() . . . . .	228
	.par_RT_calc . . . . .	229
D.5	Class <code>clsGrating</code> . . . . .	229
D.5.1	Initialization . . . . .	230
	<code>clsGrating(name, cavity)</code> . . . . .	230
D.5.2	Grating Parameters . . . . .	230
	.set_cos_grating(dx, x_max, dy, y_max, opt_dim, min_val, max_val) . . . . .	230
	.absorbtion_grating . . . . .	231
	.phase_grating . . . . .	232
	.grate_mask . . . . .	232
	.dx . . . . .	233
	.dy . . . . .	233
	.x_max . . . . .	233
	.y_max . . . . .	234
	.min_val . . . . .	234
	.max_val . . . . .	234
D.5.3	Light Field Propagation . . . . .	235
	.prop(E_in, k_space_in, k_space_out, direction) . . . . .	235
	.T_LTR_mat_tot . . . . .	235
	.T RTL mat_tot . . . . .	236
	.calc_T_mat_tot() . . . . .	236
D.5.4	Reflection Matrices . . . . .	237
	.R_L_mat_tot . . . . .	237
	.R_R_mat_tot . . . . .	237
D.5.5	Other Properties and Methods . . . . .	237
	.symmetric . . . . .	237
	.k_space_in_dont_care . . . . .	238
	.k_space_in_prefer . . . . .	238
	.k_space_out_dont_care . . . . .	238
	.k_space_out_prefer . . . . .	238
	.clear_mem_cache() . . . . .	239
D.6	Class <code>clsTransmissionTilt</code> . . . . .	239
D.6.1	Initialization . . . . .	240
	<code>clsTransmissionTilt(name, cavity)</code> . . . . .	240
D.6.2	Tilt Parameters . . . . .	240
	.x_angle . . . . .	240
	.x_angle_deg . . . . .	241
	.y_angle . . . . .	241
	.y_angle_deg . . . . .	242
	.x_zero_line . . . . .	242
	.y_zero_line . . . . .	243
	.direction . . . . .	243
D.6.3	Light Field Propagation . . . . .	243
	.prop(E_in, k_space_in, k_space_out, direction) . . . . .	243
	.T_LTR_mat_tot . . . . .	244
	.T RTL mat_tot . . . . .	244

.calc_T_mat_tot()	245
.x_mask	245
.y_mask	246
D.6.4 Reflection Matrices	246
.R_L_mat_tot	246
.R_R_mat_tot	246
D.6.5 Other Properties and Methods	247
.symmetric	247
.k_space_in_dont_care	247
.k_space_in_prefer	247
.k_space_out_dont_care	248
.k_space_out_prefer	248
.clear_mem_cache()	248
D.7 Class <code>clsSoftAperture</code>	249
D.7.1 Initialization	249
<code>clsSoftAperture(name, cavity)</code>	249
D.7.2 Aperture Parameters	249
<code>.set_params(aperture, transition_width, black_value)</code>	249
.aperture	250
.aperture_mm	250
.transition_width	251
.black_value	251
D.7.3 Light Field Propagation	252
<code>.prop(E_in, k_space_in, k_space_out, direction)</code>	252
.T_LTR_mat_tot	252
.T RTL_mat_tot	253
.calc_T_mat_tot()	253
.aperture_mask	253
D.7.4 Reflection Matrices	254
.R_L_mat_tot	254
.R_R_mat_tot	254
D.7.5 Other Properties and Methods	254
.symmetric	254
.k_space_in_dont_care	255
.k_space_in_prefer	255
.k_space_out_dont_care	255
.k_space_out_prefer	255
.clear_mem_cache()	256
D.8 Class <code>clsAmplitudeScaling</code>	256
D.8.1 Initialization	257
<code>clsAmplitudeScaling(name, cavity)</code>	257
D.8.2 Scaling Parameter	257
.amplitude_scale_factor	257
D.8.3 Light Field Propagation	257
<code>.prop(E_in, k_space_in, k_space_out, direction)</code>	257
.T_LTR_mat_tot	258
.T RTL_mat_tot	258

D.8.4	Light Field Reflection . . . . .	258
	.reflect(E_in, k_space_in, k_space_out, side) . . . . .	258
	.R_L_mat_tot . . . . .	259
	.R_R_mat_tot . . . . .	259
D.8.5	Other Properties and Methods . . . . .	259
	.symmetric . . . . .	259
	.k_space_in_dont_care . . . . .	260
	.k_space_in_prefer . . . . .	260
	.k_space_out_dont_care . . . . .	260
	.k_space_out_prefer . . . . .	260
D.9	Abstract Base Class <code>clsMirrorBase2port</code> . . . . .	260
D.9.1	Initialization . . . . .	262
	<code>clsMirrorBase2port(name, cavity)</code> . . . . .	262
D.9.2	Reflectivity and Transmissivity Parameters . . . . .	262
	.R . . . . .	262
	.T . . . . .	263
	.set_T_non_energy_conserving(T) . . . . .	263
	.set_phys_behavior(sym_phase) . . . . .	264
	.sym_phase . . . . .	265
	.r_L . . . . .	265
	.r_R . . . . .	265
	.t_LTR . . . . .	266
	.t RTL . . . . .	266
	.no_reflection_phase_shift . . . . .	267
	.left_side_relevant . . . . .	267
	.right_side_relevant . . . . .	268
D.9.3	Mirror Tilt . . . . .	268
	.rot_around_x . . . . .	268
	.rot_around_x_deg . . . . .	269
	.rot_around_y . . . . .	269
	.rot_around_y_deg . . . . .	270
	.mirror_tilted . . . . .	270
	.tilt_mask_x_L . . . . .	271
	.tilt_mask_x_R . . . . .	271
	.tilt_mask_y_L . . . . .	272
	.tilt_mask_y_R . . . . .	272
	.calc_tilt_masks_x() . . . . .	273
	.calc_tilt_masks_y() . . . . .	273
	.apply_tilt_masks . . . . .	274
D.9.4	Projection and Astigmatism Parameters . . . . .	274
	.incident_angle_y . . . . .	275
	.incident_angle_y_deg . . . . .	275
	.get_projection_factor_y1(side) . . . . .	276
	.get_projection_factor_y2(side) . . . . .	277
	.project_according_to_angles . . . . .	277
	.left_refl_size_adjust . . . . .	278
	.right_refl_size_adjust . . . . .	278
	.consider_tilt_astigmatism . . . . .	278

.left_mask_adjust . . . . .	279
.right_mask_adjust . . . . .	279
D.9.5 Transmission-Behaves-Like-Reflection Mode . . . . .	280
.LTR_transm_behaves_like_refl_left . . . . .	280
.LTR_transm_behaves_like_refl_right . . . . .	280
.LTR_transm_behaves_neutral . . . . .	281
.RTL_transm_behaves_like_refl_left . . . . .	281
.RTL_transm_behaves_like_refl_right . . . . .	281
.RTL_transm_behaves_neutral . . . . .	282
D.9.6 Memory Management . . . . .	282
.clear_mem_cache() . . . . .	282
D.10 Class <code>clsMirror</code> . . . . .	283
D.10.1 Initialization . . . . .	283
<code>clsMirror(name, cavity)</code> . . . . .	283
D.10.2 Reflection . . . . .	283
.reflect( <code>E_in, k_space_in, k_space_out, side</code> ) . . . . .	283
.R_L_mat_tot . . . . .	284
.R_R_mat_tot . . . . .	285
.calc_R_L_mat_tot() . . . . .	286
.calc_R_R_mat_tot() . . . . .	287
D.10.3 Transmission . . . . .	288
.prop( <code>E_in, k_space_in, k_space_out, direction</code> ) . . . . .	288
.T_LTR_mat_tot . . . . .	289
.T_RTL_mat_tot . . . . .	289
D.10.4 Symmetry and Space Preferences . . . . .	289
.symmetric . . . . .	289
.k_space_in_dont_care . . . . .	290
.k_space_in_prefer . . . . .	290
.k_space_out_dont_care . . . . .	290
.k_space_out_prefer . . . . .	290
D.10.5 Memory Management . . . . .	291
.clear_mem_cache() . . . . .	291
D.11 Class <code>clsCurvedMirror</code> . . . . .	291
D.11.1 Initialization . . . . .	292
<code>clsCurvedMirror(name, cavity)</code> . . . . .	292
D.11.2 Curvature and Focal-Length Parameters . . . . .	292
.f_R_L . . . . .	292
.f_R_L_mm . . . . .	292
.f_R_R . . . . .	293
.f_R_R_mm . . . . .	293
.radius_L . . . . .	293
.radius_R . . . . .	294
.is_convex(side) . . . . .	294
.is_concave(side) . . . . .	294
.f_T . . . . .	294
D.11.3 Mirror-Shape Selection . . . . .	295
.mirror_type_spherical . . . . .	295

.mirror_type_perfect . . . . .	295
D.11.4 Reflection . . . . .	296
.reflect(E_in, k_space_in, k_space_out, side) . . . . .	296
.R_L_mat_tot . . . . .	297
.R_R_mat_tot . . . . .	297
.calc_R_L_mat_tot() . . . . .	298
.calc_R_R_mat_tot() . . . . .	299
.mirror_mask_L . . . . .	300
.mirror_mask_R . . . . .	300
D.11.5 Transmission . . . . .	301
.n . . . . .	301
.prop(E_in, k_space_in, k_space_out, direction) . . . . .	302
.T_LTR_mat_tot . . . . .	302
.T RTL_mat_tot . . . . .	303
.calc_T_mat_tot() . . . . .	304
.lens_mask . . . . .	304
D.11.6 Symmetry and Space Preferences . . . . .	305
.symmetric . . . . .	305
.k_space_in_dont_care . . . . .	305
.k_space_in_prefer . . . . .	305
.k_space_out_dont_care . . . . .	306
.k_space_out_prefer . . . . .	306
D.11.7 Memory Management and Other Settings . . . . .	306
.clear_mem_cache() . . . . .	306
.par_RT_calc . . . . .	306
D.12 Class <code>clsSplitMirror</code> . . . . .	307
D.12.1 Initialization . . . . .	307
<code>clsSplitMirror(name, cavity)</code> . . . . .	307
D.12.2 Reflectivity and Transmissivity Parameters . . . . .	308
.R1 . . . . .	308
.R2 . . . . .	309
.T1 . . . . .	309
.T2 . . . . .	309
.set_T1_non_energy_conserving(T1) . . . . .	310
.set_T2_non_energy_conserving(T2) . . . . .	310
.set_phys_behavior(sym_phase) . . . . .	311
.sym_phase . . . . .	311
.r1_L . . . . .	312
.r1_R . . . . .	312
.r2_L . . . . .	312
.r2_R . . . . .	313
.t1_LTR . . . . .	313
.t1 RTL . . . . .	314
.t2_LTR . . . . .	314
.t2 RTL . . . . .	314
.left_side_relevant . . . . .	315
.right_side_relevant . . . . .	315
D.12.3 Mirror Tilt . . . . .	316

.rot_around_x_deg . . . . .	316
.rot_around_y_deg . . . . .	316
.tilt_mask_x_L . . . . .	317
.tilt_mask_x_R . . . . .	317
.tilt_mask_y_L . . . . .	318
.tilt_mask_y_R . . . . .	318
D.12.4 Reflection Matrices . . . . .	319
.R_L_mat_tot . . . . .	319
.R_R_mat_tot . . . . .	319
.calc_R_L_mat_tot() . . . . .	320
.calc_R_R_mat_tot() . . . . .	321
D.12.5 Transmission . . . . .	322
.prop(E_in, k_space_in, k_space_out, direction) . . . . .	322
.T_LTR_mat_tot . . . . .	322
.T RTL_mat_tot . . . . .	323
.calc_T_mat_tot() . . . . .	323
D.12.6 Symmetry and Space Preferences . . . . .	324
.symmetric . . . . .	324
.k_space_in_dont_care . . . . .	324
.k_space_in_prefer . . . . .	325
.k_space_out_dont_care . . . . .	325
.k_space_out_prefer . . . . .	325
D.12.7 Memory Management and Other Settings . . . . .	325
.clear_mem_cache() . . . . .	325
.par_RT_calc . . . . .	326
<b>E 4-Port Optical Components for Linear Cavities</b>	<b>327</b>
E.1 Abstract Base Class <code>clsOptComponent4port</code> . . . . .	327
E.1.1 Initialization . . . . .	329
<code>clsOptComponent4port(name, cavity)</code> . . . . .	329
E.1.2 Light Field Propagation . . . . .	329
<code>.prop(in_A, in_B, direction)</code> . . . . .	329
E.1.3 Full Transmission and Reflection Block Matrix . . . . .	330
<code>.get_T_bmat_tot(direction)</code> . . . . .	330
<code>.get_R_bmat_tot(side)</code> . . . . .	331
E.1.4 Transmission and Reflection Matrix per Path . . . . .	331
<code>.get_T_mat_tot(direction, path1, path2)</code> . . . . .	331
<code>.get_R_mat_tot(side, path1, path2)</code> . . . . .	332
E.1.5 Scattering and Transfer Block Matrices . . . . .	332
<code>.S_bmat_tot</code> . . . . .	332
<code>.M_bmat_tot</code> . . . . .	333
<code>.inv_M_bmat_tot</code> . . . . .	334
<code>.calc_inv_M_bmat_tot()</code> . . . . .	334
E.1.6 Distances . . . . .	334
<code>.get_dist_phys()</code> . . . . .	334
<code>.get_dist_opt()</code> . . . . .	335
E.2 Class <code>clsBeamSplitterMirror</code> . . . . .	335

E.2.1	Initialization . . . . .	336
	clsBeamSplitterMirror(name, cavity) . . . . .	336
E.2.2	Reflectivity and Transmissivity Parameters . . . . .	336
	.R . . . . .	336
	.T . . . . .	337
	.set_phys_behavior(sym_phase) . . . . .	337
E.2.3	Full Transmission and Reflection Block Matrix . . . . .	338
	.get_T_bmat_tot(direction) . . . . .	338
	.get_R_bmat_tot(side) . . . . .	338
E.2.4	Transmission and Reflection Matrix per Path . . . . .	339
	.get_T_mat_tot(direction, path1, path2) . . . . .	339
	.get_R_mat_tot(side, path1, path2) . . . . .	339
E.2.5	Distances . . . . .	340
	.get_dist_phys() . . . . .	340
	.get_dist_opt() . . . . .	340
E.2.6	I/O Representation Preference Flags . . . . .	340
	.k_space_in_dont_care . . . . .	340
	.k_space_in_prefer . . . . .	341
	.k_space_out_dont_care . . . . .	341
	.k_space_out_prefer . . . . .	341
E.3	Class clsOptComponentAdapter . . . . .	341
E.3.1	Initialization . . . . .	343
	clsOptComponentAdapter(name, cavity) . . . . .	343
E.3.2	Component Management . . . . .	343
	.connect_component(component, path) . . . . .	343
	.component_A . . . . .	344
	.component_B . . . . .	344
	.name . . . . .	344
E.3.3	Full Transmission and Reflection Block Matrix . . . . .	344
	.get_T_bmat_tot(direction) . . . . .	344
	.get_R_bmat_tot(side) . . . . .	345
E.3.4	Transmission and Reflection Matrix per Path . . . . .	346
	.get_T_mat_tot(direction, path1, path2) . . . . .	346
	.get_R_mat_tot(side, path1, path2) . . . . .	346
E.3.5	Distances . . . . .	347
	.get_dist_phys() . . . . .	347
	.get_dist_opt() . . . . .	347
E.3.6	I/O Representation Preference Flags . . . . .	347
	.k_space_in_dont_care . . . . .	347
	.k_space_in_prefer . . . . .	348
	.k_space_out_dont_care . . . . .	348
	.k_space_out_prefer . . . . .	348
E.4	Class clsTransmissionMixer . . . . .	348
E.4.1	Initialization . . . . .	350
	clsTransmissionMixer(name, cavity) . . . . .	350
E.4.2	Transmission Coefficients . . . . .	351
	.T_same . . . . .	351

.T_mix . . . . .	351
E.4.3 Phase-Behavior Flags . . . . .	351
.sym_phase . . . . .	351
.refl_behavior_for_path_mixing . . . . .	352
E.4.4 Full Transmission and Reflection Block Matrix . . . . .	352
.get_T_bmat_tot(direction) . . . . .	352
.get_R_bmat_tot(side) . . . . .	353
E.4.5 Transmission and Reflection Matrix per Path . . . . .	353
.get_T_mat_tot(direction, path1, path2) . . . . .	353
.get_R_mat_tot(side, path1, path2) . . . . .	354
E.4.6 Distances . . . . .	354
.get_dist_phys() . . . . .	354
.get_dist_opt() . . . . .	354
E.4.7 I/O Representation Preference Flags . . . . .	355
.k_space_in_dont_care . . . . .	355
.k_space_in_prefer . . . . .	355
.k_space_out_dont_care . . . . .	355
.k_space_out_prefer . . . . .	355
E.5 Class <code>clsReflectionMixer</code> . . . . .	356
E.5.1 Initialization . . . . .	356
<code>clsReflectionMixer(name, cavity)</code> . . . . .	356
E.5.2 Reflection Coefficients . . . . .	357
.R_same . . . . .	357
.R_mix . . . . .	357
E.5.3 Phase-Behavior Flags . . . . .	357
.sym_phase . . . . .	357
.refl_behavior_for_path_mixing . . . . .	358
E.5.4 Full Transmission and Reflection Block Matrix . . . . .	359
.get_T_bmat_tot(direction) . . . . .	359
.get_R_bmat_tot(side) . . . . .	359
E.5.5 Transmission and Reflection Matrix per Path . . . . .	360
.get_T_mat_tot(direction, path1, path2) . . . . .	360
.get_R_mat_tot(side, path1, path2) . . . . .	360
E.5.6 Distances . . . . .	360
.get_dist_phys() . . . . .	360
.get_dist_opt() . . . . .	361
E.5.7 I/O Representation Preference Flags . . . . .	361
.k_space_in_dont_care . . . . .	361
.k_space_in_prefer . . . . .	361
.k_space_out_dont_care . . . . .	361
.k_space_out_prefer . . . . .	362
<b>F Cavity Classes</b>	<b>363</b>
F.1 Abstract Base Class <code>clsCavity</code> . . . . .	363
F.1.1 Initialization . . . . .	365
<code>clsCavity(name, full_precision=True)</code> . . . . .	365
F.1.2 Core Configuration . . . . .	365

.name . . . . .	365
.grid . . . . .	366
.progress . . . . .	366
.use_bmatrix_class . . . . .	366
F.1.3 Multiprocessing . . . . .	366
.mp_pool_processes . . . . .	366
.close_mp_pool() . . . . .	367
F.1.4 Wavelength Parameters . . . . .	367
.Lambda . . . . .	367
.Lambda_nm . . . . .	367
.Lambda_ref . . . . .	368
.Lambda_ref_nm . . . . .	368
F.1.5 Folder and File Handling . . . . .	368
.folder . . . . .	368
.full_file_name(name, idx=-1, file_extension="pkl", start_with_underscore=False) . . . . .	369
.sep_char . . . . .	370
F.1.6 Reflection and Transmission Matrices – Total Resolution . . . . .	370
.R_L_mat_tot . . . . .	370
.R_R_mat_tot . . . . .	370
.T_LTR_mat_tot . . . . .	371
.T RTL_mat_tot . . . . .	371
.calc_R_L_mat_tot() . . . . .	372
.calc_R_R_mat_tot() . . . . .	372
.calc_T_LTR_mat_tot() . . . . .	372
.calc_T RTL_mat_tot() . . . . .	373
F.1.7 Reflection and Transmission – FOV Resolution . . . . .	373
.R_L_mat_fov . . . . .	373
.R_R_mat_fov . . . . .	373
.T_LTR_mat_fov . . . . .	373
.T RTL_mat_fov . . . . .	374
.convert_R_L_mat_tot_to_fov() . . . . .	374
.convert_R_R_mat_tot_to_fov() . . . . .	374
.convert_T_LTR_mat_tot_to_fov() . . . . .	374
.convert_T RTL_mat_tot_to_fov() . . . . .	375
F.1.8 Full Scattering and Transfer Block Matrices . . . . .	375
.S_bmat_tot . . . . .	375
.M_bmat_tot . . . . .	375
.calc_M_bmat_tot(idx_from=0, idx_to=999) . . . . .	376
F.1.9 Component Management . . . . .	376
.components . . . . .	376
.get_component(i) . . . . .	377
.get_last_component() . . . . .	377
.component_count . . . . .	377
F.1.10 Caching and Memory . . . . .	377
.allow_temp_mem_caching . . . . .	380
.allow_temp_file_caching . . . . .	380
.file_cache_min_calc_time . . . . .	381

.activate_component_file_cache(idx_from=0, idx_to=999) . . . . .	381
.use_swap_files_in_bmatrix_class . . . . .	382
.tmp_folder . . . . .	382
.clear_results() . . . . .	383
.clear() . . . . .	383
.delete_cached_component_files() . . . . .	384
F.1.11 File I/O Utilities . . . . .	384
.write_to_file(file_name, data) . . . . .	384
.save_mat(name, m, idx=-1, msg="") . . . . .	384
.load_mat(name, idx=-1, msg="") . . . . .	385
.file_exists(name, idx=-1) . . . . .	386
.delete_mat(name, idx=-1, msg="") . . . . .	386
.save_M_bmat() . . . . .	387
.load_M_bmat() . . . . .	387
.M_bmat_file_exists() . . . . .	387
F.1.12 Single-Step Mode . . . . .	388
.UID . . . . .	390
.pre_steps . . . . .	391
.additional_steps . . . . .	391
.total_steps . . . . .	391
.single_step(step, reverse_mul=False) . . . . .	391
.single_step_mode . . . . .	392
.pre_step(step) . . . . .	392
.additional_step(step) . . . . .	393
F.1.13 Resonance Calculations . . . . .	393
.resonance_data_simple_cavity(R_left, R_right, length_opt, sym_phase = True) . . . . .	393
.resonance_data_8f_cavity(R_left, R_center, R_right, f, sym_phase = True) . . . . .	394
F.2 Class clsCavity1path . . . . .	396
F.2.1 Initialization . . . . .	399
clsCavity1path(name, full_precision=True) . . . . .	399
F.2.2 Component Management . . . . .	399
.add_component(component) . . . . .	399
.get_dist_phys(idx1, idx2) . . . . .	400
.get_dist_opt(idx1, idx2) . . . . .	400
F.2.3 Propagation and Reflection . . . . .	401
.prop(field, idx_from, idx_to, direction) . . . . .	401
.reflect(field, idx, side) . . . . .	401
.get_T_mat_LTR(idx_from, idx_to) . . . . .	402
.get_T_mat_RTL(idx_from, idx_to) . . . . .	402
F.2.4 Round-Trip Utilities . . . . .	403
.prop_round_trip_LTR_RTL(field, idx1, idx2, incl_left_refl) . . . . .	403
.prop_mult_round_trips_LTR_RTL(no_of_trips, bulk_field_pos = 0, print_progress = False, plot_left_out_progress = False, plot_right_out_progress = False, plot_bulk_LTR_progress = False, plot_bulk_RTL_progress = False,	

idx1 = 0, idx2 = 999, x_plot_shift = 0, y_plot_shift = 0) . . . . .	403
.get_round_trip_T_mat_LTR_RTL(idx1, idx2, incl_left_refl) . . . . .	405
.get_inf_round_trip_R_L_mat(idx1, idx2) . . . . .	405
.get_round_trip_T_mat_RTL_LTR(idx1, idx2, incl_right_refl) . . . . .	407
.get_inf_round_trip_R_R_mat(idx1, idx2) . . . . .	407
F.2.5 Incident Fields . . . . .	408
.incident_field_left . . . . .	408
.incident_field_right . . . . .	409
F.2.6 Output Fields . . . . .	409
.output_field_left . . . . .	409
.calc_output_field_left() . . . . .	409
.output_field_right . . . . .	410
.calc_output_field_right() . . . . .	410
F.2.7 Intracavity Field . . . . .	411
.calc_bulk_field_from_left(idx) . . . . .	411
.calc_bulk_field_from_right(idx) . . . . .	411
.bulk_field_pos . . . . .	412
.bulk_field_LTR . . . . .	412
.bulk_field_RTL . . . . .	413
.bulk_field . . . . .	413
F.2.8 Reflection and Transmission Matrices . . . . .	413
.calc_R_L_mat_tot() . . . . .	413
.calc_R_R_mat_tot() . . . . .	414
.calc_T_LTR_mat_tot() . . . . .	414
.calc_T_RTL_mat_tot() . . . . .	415
F.2.9 Full Transfer and Scattering Block Matrix . . . . .	415
.calc_S_bmat_tot() . . . . .	415
F.2.10 Caching and Memory . . . . .	416
.clear_results() . . . . .	416
F.2.11 File I/O Utilities . . . . .	416
.save_R_L_mat_tot() . . . . .	416
.load_R_L_mat_tot() . . . . .	416
.save_R_L_mat_fov() . . . . .	416
.load_R_L_mat_fov() . . . . .	417
.save_R_R_mat_tot() . . . . .	417
.load_R_R_mat_tot() . . . . .	417
.save_R_R_mat_fov() . . . . .	417
.load_R_R_mat_fov() . . . . .	417
.save_T_LTR_mat_tot() . . . . .	417
.load_T_LTR_mat_tot() . . . . .	418
.save_T_LTR_mat_fov() . . . . .	418
.load_T_LTR_mat_fov() . . . . .	418
.save_T_RTL_mat_tot() . . . . .	418
.load_T_RTL_mat_tot() . . . . .	418
.save_T_RTL_mat_fov() . . . . .	418
.load_T_RTL_mat_fov() . . . . .	419
F.3 Class <code>clsCavity2path</code> . . . . .	419

F.3.1	Initialization . . . . .	421
	clsCavity2path(name, full_precision=True) . . . . .	421
F.3.2	Component Management . . . . .	422
	.add_4port_component(component) . . . . .	422
	.add_2port_component(A, B) . . . . .	423
F.3.3	Propagation . . . . .	423
	.prop(in_A, in_B, idx_from, idx_to, direction) . . . . .	423
	.get_dist_phys(idx1, idx2) . . . . .	424
	.get_dist_opt(idx1, idx2) . . . . .	424
F.3.4	Incident Fields . . . . .	425
	.set_incident_field(side, path, field) . . . . .	425
	.get_incident_field(side, path) . . . . .	425
F.3.5	Output Fields . . . . .	426
	.get_output_field(side, path) . . . . .	426
	._calc_output_field(side, path) . . . . .	426
F.3.6	Intracavity Field . . . . .	427
	._calc_bulk_field_from_left(idx) . . . . .	427
	._calc_bulk_field_from_right(idx) . . . . .	428
	.bulk_field_pos . . . . .	429
	.get_bulk_field_LTR(path) . . . . .	430
	.get_bulk_field RTL(path) . . . . .	430
	.get_bulk_field(path) . . . . .	431
F.3.7	Reflection and Transmission Matrices . . . . .	431
	._calc_R_L_mat_tot() . . . . .	431
	._calc_R_R_mat_tot() . . . . .	432
	._calc_T_LTR_mat_tot() . . . . .	432
	._calc_T RTL_mat_tot() . . . . .	433
F.3.8	Full Transfer and Scattering Block Matrix . . . . .	433
	._calc_S_bmat_tot() . . . . .	433
F.3.9	Caching and Memory . . . . .	434
	.clear_results() . . . . .	434

# Chapter 1

## Introduction

This manual documents the `fourier-cavity-sim` project.<sup>1</sup> I developed the library during my time as a University Assistant and PhD student at the Institute for Theoretical Physics, at TU Wien. What began as a small set of scripts for my own research needs gradually grew – benefiting from my software engineering background – into a modular library.

In short, `fourier-cavity-sim` provides building blocks for Fourier-optics simulations and Fourier-optics cavity simulations. It can simulate both linear and ring cavities, and offers component classes for flat and curved mirrors, lenses, apertures, and other elements; beyond one-direction propagation it also solves for steady-state fields, including for multiple, mutually coupled cavities. It runs on a single workstation and scales to cluster environments for parallel computing.

I used `fourier-cavity-sim` in several research projects, most notably [1, 2], which focus on Coherent Perfect Absorption (CPA). For that reason, many hands-on examples in this manual concern CPA; nevertheless, the library itself is general and not restricted to CPA use cases.

I am happy if other researchers find this library useful. If you use it in a publication, please include a reference to this manual (and the repository). I also welcome collaboration, whether by extending the library for new features or by running simulations together.

**Acknowledgements.** I thank my PhD advisor Stefan Rotter for guidance, and my co-authors and colleagues Kevin Pichler, Lena Wild, Ori Katz, Yevgeny Slobodkin, and Gil Weinberg, for valuable discussions and input that helped make this library possible.

---

<sup>1</sup>Repository: <https://github.com/HelmutHoerner/fourier-cavity-sim>



# Chapter 2

## Installation

### 2.1 Installation on a Local Computer

This project lives at: <https://github.com/HelmutHoerner/fourier-cavity-sim>.

#### 2.1.1 Prerequisites

- **Python** (version  $\geq 3.8$ ). Check with: `python --version`
- **pip**. Check with: `python -m pip --version`
- **Git** (not included with Python). Check with: `git --version`

If Git is missing:

- **Windows**: Install “Git for Windows” from <https://git-scm.com/download/win>.
- **macOS**: Install Apple Command Line Tools (`xcode-select --install`) or use Homebrew: `brew install git`.
- **Linux**: Use your package manager, e.g. Ubuntu/Debian `sudo apt install git`, Fedora `sudo dnf install git`, Arch `sudo pacman -S git`.

#### 2.1.2 Get and install Conda

You can use **Miniconda** (lightweight) or **Mambaforge** (includes the `conda-forge` channel and `mamba` for faster installs).

##### Download

- Miniconda: <https://docs.conda.io/en/latest/miniconda.html>
- Mambaforge: <https://github.com/conda-forge/miniforge>

### Install on Windows

1. Download `Miniconda3-latest-Windows-x86_64.exe` (or Mambaforge) and run the installer.
2. Accept the defaults. Allow the installer to run `conda init` when prompted.
3. Open the **(Mini)Conda Prompt** and verify:

```
conda --version
```

### Install on MacOS / Linux.

1. Download the `.sh` installer for your platform (Intel/ARM).
2. In a terminal, run (adjust filename as needed):

```
bash ~/Downloads/Miniconda3-latest-MacOSX-x86_64.sh
# or, e.g., Linux:
# bash ~/Downloads/Miniconda3-latest-Linux-x86_64.sh
```

3. Accept the license, confirm the install location, and allow `conda init`.
4. Close and reopen the terminal, then verify:

```
conda --version
```

### 2.1.3 Using Anaconda (Full Distribution)

#### When to use it

Anaconda includes a large set of scientific packages and the GUI *Anaconda Navigator*. If you already have Anaconda installed, you can use it for this project. For a lighter install, *Miniconda* or *Mambaforge* remain recommended.

#### Download and Install

Get the installer from <https://www.anaconda.com/download>. Run it with default options and let it run `conda init`.

### 2.1.4 Create Virtual Environment and Install Library

After Conda or Anaconda is installed, create and activate a dedicated environment `fourier-cavity-sim`:

```
conda create --name fourier-cavity-sim python=3.8.5
conda activate fourier-cavity-sim
conda config --add channels conda-forge
conda config --set channel_priority strict
conda install numpy blas==*=openblas
conda install spyder matplotlib scipy portalocker joblib pandas
```

*Comment:* The line `conda install numpy blas==openblas` installs the OpenBLAS-backed NumPy library, which is recommended because the default NumPy library builds are optimized for Intel; OpenBLAS can significantly improve performance on AMD CPUs.

Now go to the directory where you want the `fourier-cavity-sim` library installed, and clone and install it with the following commands:

```
git clone https://github.com/HelmutHoerner/fourier-cavity-sim.git
cd fourier-cavity-sim
pip install -e . --config-settings editable_mode=compat
```

## 2.2 Installation on a Cluster

On clusters without Conda, a reliable way to get a matching Python is to use `pyenv` with a per-user virtual environment.

cf.: <https://realpython.com/intro-to-pyenv/#creating-virtual-environments>

**1) Install `pyenv` in your home directory:**

```
curl https://pyenv.run | bash
```

**2) Initialize `pyenv` in your shell**

Append the following lines to the `.bashrc` (or `.bash_profile`) file in your home directory:

```
export PATH="$HOME/.pyenv/bin:$PATH"
eval "$(pyenv init -)"
eval "$(pyenv virtualenv-init -)"
```

Now log out, and log in again. Verify that the following command works and returns a version number:

```
pyenv --version
```

**3) Install the required Python version (3.8.5):**

```
pyenv install -v 3.8.5
```

This installs python into the hidden folder `.pyenv` in your home directory. You can test the installation with the following command:

```
pyenv versions
```

**4) Create and activate a virtual environment `pyenv_385`:**

```
pyenv virtualenv 3.8.5 pyenv_385
pyenv activate pyenv_385
```

(If no longer needed, you can deactivate later with `pyenv deactivate`)

**5) Upgrade pip:**

```
python -m pip install --upgrade pip
```

**6) Install required modules:**

```
pip install numpy
pip install scipy
pip install matplotlib
pip install portalocker
pip install joblib
pip install pandas
```

**7) Copy library files:**

Now create a folder from where you intend to run your programs. Then copy the following files from the repository into that folder:

- core.py
- cmap\_custom.csv

Then rename `core.py` to `fo_cavity_sim.py`.

If you intend to run the sample programs, now is a good time to also create the following sub-folders:

- cavity\_folder
- tmp\_folder

The python environment can now be activated with

```
~/.pyenv/versions/3.8.5/envs/pyenv_385/bin/activate
```

and a Python script `myProgram.py` could be afterwards started with

```
python -u myProgram.py
```

However, on a cluster programs are usually started in batch scripts. An detailed example is shown in chapter 6.

# Chapter 3

## Light Propagation in One Direction

### 3.1 Fourier Optics Theory

The goal of this library is to simulate *optical cavities*, i.e. resonators assembled from mirrors, lenses, beam splitters and related components. Before we can describe an entire cavity, however, we must answer a simpler question:

*Given an initial light field, how does it evolve as it propagates through free space and a sequence of optical elements?*

We tackle this problem with algorithms from **Fourier optics**. In its current form the library can

- simulate **classical, coherent light fields** with a fixed linear polarization,
- compute their **diffraction during free-space propagation**, and
- model the effect of **optical components** like lenses, mirrors, apertures, beam splitters, etc. via transmission functions that can be applied successively.

Quantum effects are *not* included, and the field is represented by a single scalar amplitude, so only one polarization component is currently handled. A future release may extend the model to a full vector treatment.

#### Field representation

Fourier optics introduces a preferred propagation axis, here chosen as the  $z$ -direction (see Figure 3.1). At a fixed axial position  $z = z_0$  the optical field is described by a complex amplitude

$$U(x, y; z_0),$$

defined on the transverse  $x$ - $y$  plane. Ideally  $U$  is a continuous function over an infinite plane; in practice the software

- samples  $U$  on an equidistant grid  $\{x_m, y_n\}$ , and
- confines the computational domain to a square “tile” of limited physical size.

Knowing  $U(x, y; z_0)$  and the intervening components, the Fourier-optics algorithms compute the field  $U(x, y; z_1)$  on a second plane at  $z = z_1$ . All higher-level routines in the cavity solvers ultimately build on this elementary propagation step.

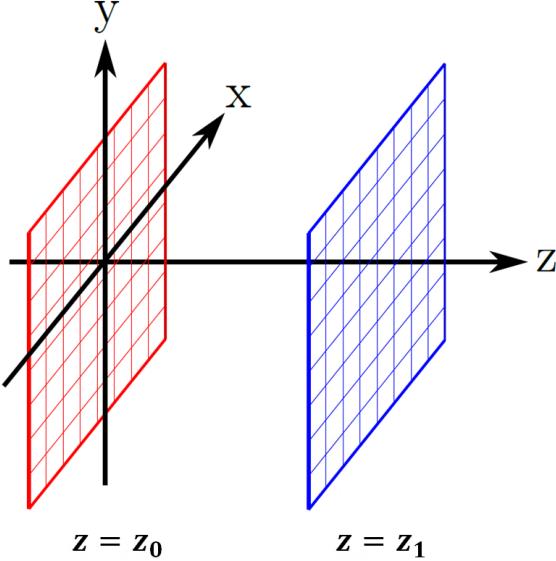


Figure 3.1: Fourier–optics picture: the light field on the red  $xy$ -plane plane at  $z = z_0$  is propagated to the blue  $xy$ -plane at  $z = z_1$ , possibly encountering optical elements on the way. In both planes the light field is represented by a complex-valued function  $U(x, y; z)$  (stored as an array in the code).

### 3.1.1 Fourier-Optics Example 1: Free-Space Propagation

#### Idea

We can decompose the monochromatic field on the input plane  $z = z_0$  into its constituent plane waves (the “angular spectrum”) and each component is then propagated individually. Formally, one proceeds as follows:

1. **Forward Fourier transform.** Starting from the complex field  $U(x, y; z_0)$  on the input plane, compute the angular spectrum

$$A(k_x, k_y; z_0) = \mathcal{F}\{U(x, y; z_0)\}, \quad (3.1)$$

where  $(k_x, k_y)$  are the transverse spatial frequencies.

2. **Plane-wave propagation.** Every spectral component represents a plane wave with axial wavenumber  $k_z = \sqrt{k^2 - k_x^2 - k_y^2}$ , where  $k = 2\pi/\lambda$ . After a free-space distance  $d = z_1 - z_0$  its amplitude picks up the phase factor

$$\exp(i k_z d).$$

Hence

$$A(k_x, k_y; z_1) = A(k_x, k_y; z_0) \exp\left[i d \sqrt{k^2 - k_x^2 - k_y^2}\right]. \quad (3.2)$$

**3. Inverse Fourier transform.** The field on the output plane is obtained from the propagated spectrum by inverse Fourier transform:

$$U(x, y; z_1) = \mathcal{F}^{-1}\{A(k_x, k_y; z_1)\}. \quad (3.3)$$

### Practical implementation

In the code the continuous variables are sampled on a finite  $N \times N$  grid:  $(x_m, y_n)$  in real space and  $(k_{x,p}, k_{y,q})$  in Fourier space. The integrals turn into discrete sums and the forward / inverse transforms are executed with fast-Fourier-transform (FFT) routines. The finite “tile” size limits both the spatial extent and the maximum angular bandwidth that can be represented, but – provided the grid is chosen adequately – this discretised scheme reproduces the continuous solution to excellent accuracy while keeping the computational cost manageable.

#### 3.1.2 Fourier-Optics Example 2: Thin-Lens Phase Mask

##### Idea

In Fourier optics a *thin lens* is treated as an infinitely thin phase plate: all refraction is squeezed into a single  $xy$ -plane. Propagation *to* and *from* that plane is still handled with the free-space formalism of the previous example; the presence of the lens merely multiplies the incident field by an  $x$ - $y$ -dependent phase factor.

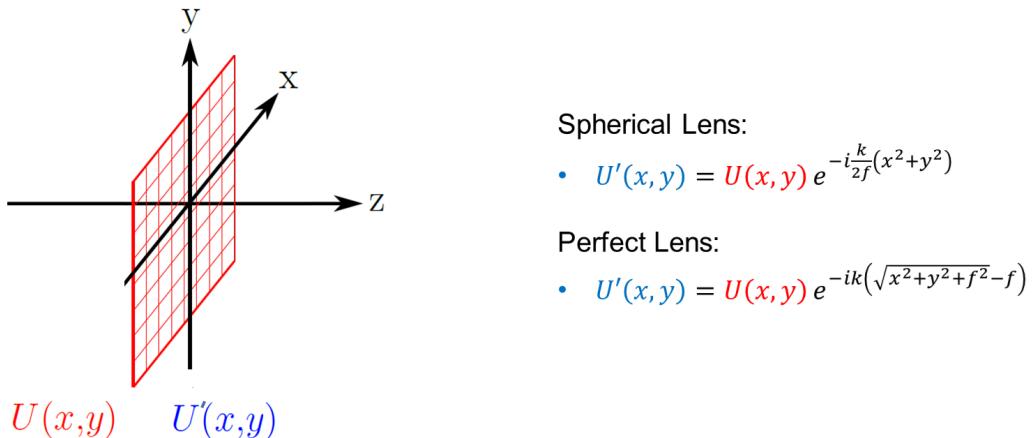


Figure 3.2: Phase modulation imposed by a thin lens with focal length  $f$ . The incident field  $U(x, y)$  on the left side of the depicted  $xy$ -plane is converted into  $U'(x, y)$  on the right side of the plane by a  $xy$ -position-dependent phase factor characteristic of the lens shape.

##### Phase functions

Let  $U(x, y)$  be the complex field immediately in front of the lens (left from the  $xy$ -plane depicted in Figure 3.2). The field immediately behind the lens, denoted  $U'(x, y)$  (right of the depicted plane in Figure 3.2)), is obtained by

$$U'(x, y) = U(x, y) P(x, y), \quad (3.4)$$

where  $P(x, y)$  is the lens’ phase mask. Two frequently used models are:

- **Spherical lens**

$$P(x, y) = \exp\left[-i \frac{k}{2f} (x^2 + y^2)\right], \quad (3.5)$$

with focal length  $f$  and wavenumber  $k = 2\pi/\lambda$ .

- **Perfect lens**

Removes the spherical aberrations; all rays meet exactly in the focal plane.

$$P(x, y) = \exp\left(-i k [\sqrt{x^2 + y^2 + f^2} - f]\right), \quad (3.6)$$

with focal length  $f$  and wavenumber  $k = 2\pi/\lambda$ .

### Practical implementation

In the library the continuous mask  $P(x, y)$  is sampled on the same  $N \times N$  grid that carries the light field. Element-wise multiplication with the pre-computed phase array therefore updates the field.

## 3.2 Code Example 1: Light Propagation Through Two Lenses

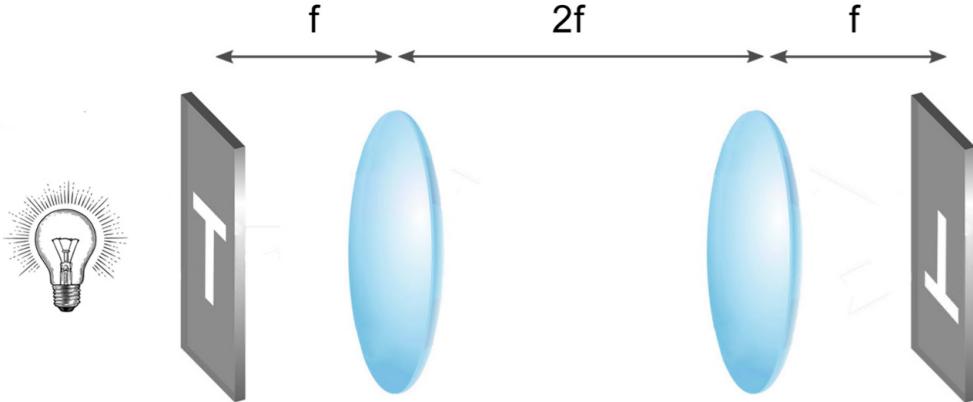


Figure 3.3: Configuration for code example 1. An input field shaped like the letter “T” (left) is propagated left-to-right through a  $4f$  telescope consisting of two thin lenses ( $f-2f-f$  spacing). Ideal Fourier-optics theory predicts an upside-down copy of the left-incident “T” in the output plane on the right side.

In this first hands-on example we want to simulate a telescopic arrangement of two lenses, consisting of two identical thin lenses of focal length  $f$ , separated by  $2f$ . A planar input field in the shape of the letter “T” is placed one focal length left of the first lens; the observation plane lies one focal length behind the second lens (Figure 3.3). After propagating the incident light field from left-to-right through this system, we expect it to produce an upside-down replica of the “T” at the right-side output plane. For simplicity the “T” is not modelled as a physical transparency illuminated by a lamp; instead, a `clsTestImage` instance is initialized directly with the desired T-shaped light field profile.

### 3.2.1 First Attempt

The following listing is taken from the project example file `examples/sample_001A.py`: it constructs the  $4f$  telescope of Figure 3.3, injects a “T”-shaped input field, and propagates that field through all components to the output plane.

```

1  from fo_cavity_sim import clsPropagation, clsThinLens, \
2      clsTestImage, clsCavity1path, Dir
3
4  if __name__ == '__main__':
5      length_fov = 0.0021 # field-of-view sidelength
6      f = 0.075           # focal length of lenses
7
8  # --- BUILD CAVITY -----
9  myCavity = clsCavity1path("my_cavity")
10 myCavity.Lambda_nm = 633
11 myCavity.grid.set_res(200, 200, length_fov)
12
13 # --- DEFINE COMPONENTS -----
14 prop = clsPropagation("f propagation", myCavity) # distance f
15 prop.set_params(f, 1)
16
17 lens = clsThinLens("lens", myCavity)             # thin lens
18 lens.lens_type_spherical = True
19 lens.f = f
20
21 # --- ASSEMBLE 4f TELESCOPE -----
22 myCavity.add_component(prop) # 0 : free space f
23 myCavity.add_component(lens) # 1 : lens f
24 myCavity.add_component(prop) # 2 : free space f
25 myCavity.add_component(prop) # 3 : free space f
26 myCavity.add_component(lens) # 4 : lens f
27 myCavity.add_component(prop) # 5 : free space f
28
29 # --- INPUT FIELD -----
30 input_field = clsTestImage(myCavity.grid)
31 input_field.create_test_image(3) # "T" pattern
32 input_field.name = "Input Field"
33 input_field.plot_field(5)
34
35 # --- PROPAGATION -----
36 output_field = myCavity.prop(input_field, 0, 5, Dir.LTR)
37 output_field.name = "Output Field"
38 output_field.plot_field(5)

```

#### Import statements (lines 1–2)

The script begins by pulling in the classes and enumerations it needs:

- `clsPropagation` – free-space propagation segment
- `clsThinLens` – ideal thin lens
- `clsTestImage` – convenience class that synthesises a “T”-shaped input field
- `clsCavity1path` – a linear “breadboard” on which we arrange the optical elements
- `Dir` – direction enum providing `.LTR` (left-to-right) and `.RTL` (right-to-left)

### Main-program guard (line 4)

Because the library accelerates heavy calculations with parallel/multicore processing, every worker process must start with a clean slate. The line `if __name__ == '__main__':` ensures exactly that: the code inside the block runs *only* in the original program you launch, not in the helper processes that the operating system spawns during parallel execution. Skipping this guard can make the workers re-execute the entire script, which may crash or hang the program on some platforms (notably Windows).

### Define parameters (lines 5–6)

- `length_fov = 0.0021` sets the side-length of the computational *xy* tile to 2.1 mm.
- `f = 0.075` assigns a focal length of 75 mm to both lenses in the telescope.

### Create cavity (lines 8–11)

- Line 8 instantiates a `clsCavity1path` object (a subclass of `clsCavity`). Although no resonator is formed yet, this object serves as a convenient “breadboard” onto which we will attach optical components.
- Line 10 sets the vacuum wavelength to 633 nm via `myCavity.Lambda_nm`.
- In Line 11, the method `myCavity.grid.set_res(...)` method is used to create a  $200 \times 200$  array that spans the previously defined 2.1 mm  $\times$  2.1 mm tile on the *xy*-plane. All subsequent field calculations share this grid geometry.

### Create optical components (lines 14–19)

- **Free-space section (lines 14–15)** – an instance of `clsPropagation` called `prop` is created and configured to simulate free space propagation over a distance of one focal length  $f$ .
- **Thin lens (lines 17–19)** – an instance of `clsThinLens` named `lens` is created and configured to simulate a thin, spherical lens with focal length  $f$ .

### Build optical system (lines 21–27)

- The components are appended to the cavity in geometric order so that the six red indices in Figure 3.4 match the zero-based indices used later in the propagation call.
- Note that *the very same optical component object can be reused*: the single `prop` instance is inserted four times (components 0, 2, 3, 5) and the single `lens` instance twice (components 1, 4). Reusing identical elements in this way is perfectly legal and saves both memory and CPU time.

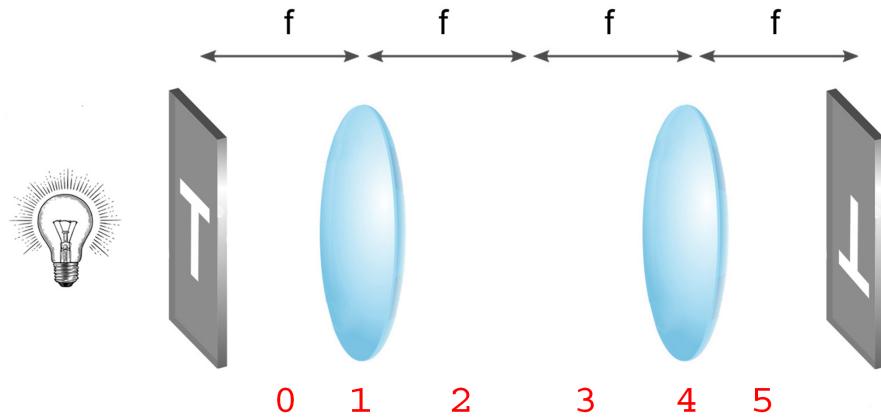


Figure 3.4: Component ordering for `sample_001A.py`. The red numbers indicate the indices assigned by `clsCavity1path.add_component(component)`.

#### Generate and display the input field (lines 29–33)

An instance of `clsTestImage` – a convenience subclass of `clsLightField` – is created, configured to produce the “T” test pattern, and visualised with `input_field.plot_field(...)`.

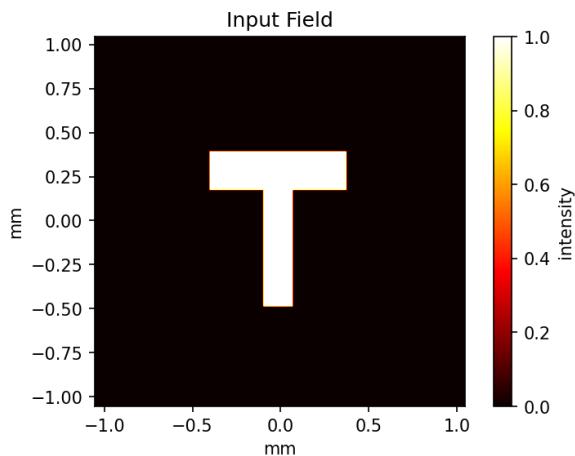


Figure 3.5: Plot of the incident field created in line 33.

#### Propagate and visualise the output field (lines 35–38)

By calling the method `myCavity.prop(...)` with `Dir.LTR`, we simulate a left-to-right propagation of the “T” input field through components 0 to 5, i.e. from the entrance plane up to the exit plane of the  $4f$  telescope. The returned `clsLightField` instance, named "Output Field", is then displayed with `output_field.plot_field(...)`.

The software creates the following output via the `clsProgressPrinter` instance stored in `myCavity.progress`:

```
processing component 0: f propagation
processing component 1: lens
processing component 2: f propagation
processing component 3: f propagation
processing component 4: lens
processing component 5: f propagation
```

and then the following plot is created in line 38:

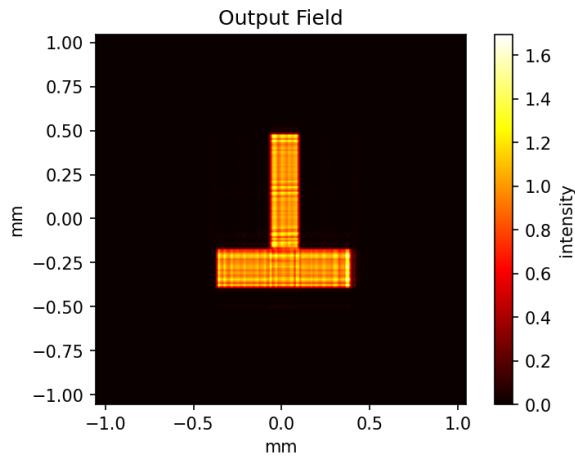


Figure 3.6: Plot of the output field created in line 38.

Although Figure 3.6 indeed shows an upside-down version of the incident “T”, the pattern is visibly distorted. Somewhere along the propagation our numerical model has introduced artefacts. What went wrong? The next subsection analyses the result and tracks down the underlying cause.

### 3.2.2 Critical Sampling Condition

The distortion in our first simulation stems from an undersampled numerical grid. For a square tile of side-length  $L$  that is to be propagated by a free-space distance  $z$  at vacuum wavelength  $\lambda$ , Voelz [3, p. 193] shows that the  $xy$ -grid with  $N \times N$  points must satisfy a *critical sampling* criterion:

$$N = \frac{L^2}{\lambda z}.$$

This means: for a given  $L$ ,  $z$ , and  $\lambda$  there is *exactly one*  $N$  so that a square  $N \times N$   $xy$ -grid yields artefact-free Fourier-optics propagation.

At first glance the critical-sampling rule appears to lock the spatial resolution once  $L$ ,  $z$  and  $\lambda$  are given. The remedy is to embed the region of interest (square side-length  $L_{\text{FOV}}$ , sampled

with  $N_{\text{FOV}} \times N_{\text{FOV}}$  points) into a larger computational tile of side-length  $L_{\text{tot}} > L_{\text{FOV}}$ . Choosing a sufficiently large  $L_{\text{tot}}$  raises the permitted pixel count  $N_{\text{tot}}$  *quadratically*; the central  $L_{\text{FOV}} \times L_{\text{FOV}}$  patch therefore inherits any desired resolution while the outer “padding” area contains zeros. Numerically this is nothing more than zero-padding before the FFT.

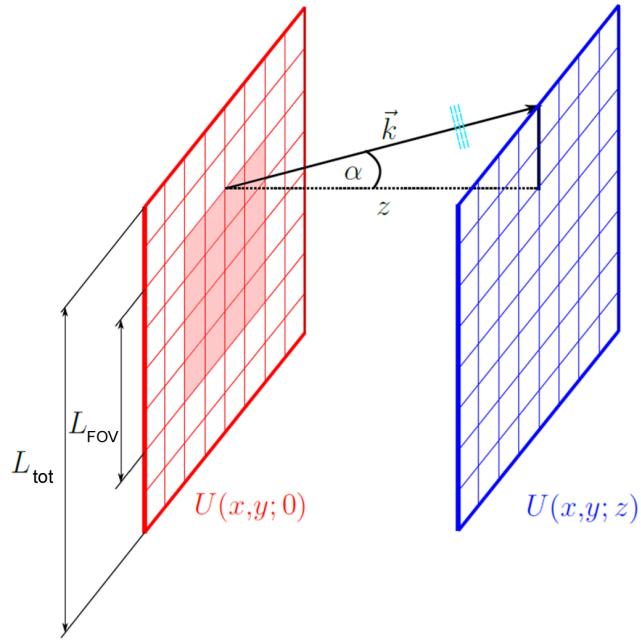


Figure 3.7: Embedding concept. The red tile of side  $L_{\text{tot}}$  satisfies the sampling condition. Only the inner ref-colored field of view ( $L_{\text{FOV}}$ ) is populated with data; the surrounding pixels are set to zero. During propagation the angular spectrum fans out (angle  $\alpha$ ), so a larger tile is required to “catch” the diffracted field on the output plane.

### Intuitive picture

A finite beam spreads transversely while traveling the distance  $z$ ; the larger computational tile simply provides room for this diffraction halo.

### 3.2.3 Second Attempt: Grid Size Chosen by the Critical-Sampling Rule

The program below is taken from the project example file `examples/sample_001B.py`. It is identical to the first attempt except for **lines 11–13**. Instead of specifying the grid manually we now call `myCavity.grid.set_opt_res_tot_based_on_res_fov(...)` to obtain a *total* grid that fulfils the critical-sampling condition for a field-of-view of  $L_{FOV} = 2.1$  mm and a propagation distance  $z = 2f$ , which is the longest *free-space* hop in the system.

```

1  from fo_cavity_sim import clsPropagation, clsThinLens, \
2                                clsTestImage, clsCavity1path, Dir
3
4  if __name__ == '__main__':
5      length_fov = 0.0021    # field-of-view side length (m)
6      f = 0.075             # focal length of lenses (m)
7
8  # --- BUILD CAVITY -----
9  myCavity = clsCavity1path("my_cavity")
10 myCavity.Lambda_nm = 633
11 # critical-sampling grid (FOV = 2.1 mm, propagation distance = 2f)
12 myCavity.grid.set_opt_res_tot_based_on_res_fov(length_fov, 100, 2*f)
13 print(f"Field-of-view: {myCavity.grid.res_fov}x{myCavity.grid.res_fov}")
14 print(f"Total grid res: {myCavity.grid.res_tot}x{myCavity.grid.res_tot}")
15
16 # --- DEFINE COMPONENTS -----
17 prop = clsPropagation("f propagation", myCavity)    # distance f
18 prop.set_params(f, 1)
19
20 lens = clsThinLens("lens", myCavity)                  # thin lens
21 lens.lens_type_spherical = True
22 lens.f = f
23
24 # --- ASSEMBLE 4f TELESCOPE -----
25 myCavity.add_component(prop)  # 0 : free space f
26 myCavity.add_component(lens) # 1 : lens f
27 myCavity.add_component(prop) # 2 : free space f
28 myCavity.add_component(prop) # 3 : free space f
29 myCavity.add_component(lens) # 4 : lens f
30 myCavity.add_component(prop) # 5 : free space f
31
32 # --- INPUT FIELD -----
33 input_field = clsTestImage(myCavity.grid)
34 input_field.create_test_image(3)  # "T" pattern
35 input_field.name = "Input Field"
36 input_field.plot_field(5)
37
38 # --- PROPAGATION -----
39 output_field = myCavity.prop(input_field, 0, 5, Dir.LTR)
40 output_field.name = "Output Field"
41 output_field.plot_field(5)

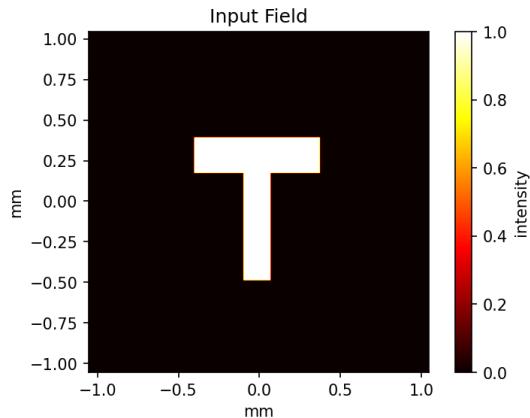
```

The `print` statements in lines 12–13 print the resulting field-of-view and total resolutions so you can verify that the automatic sizing worked as intended.

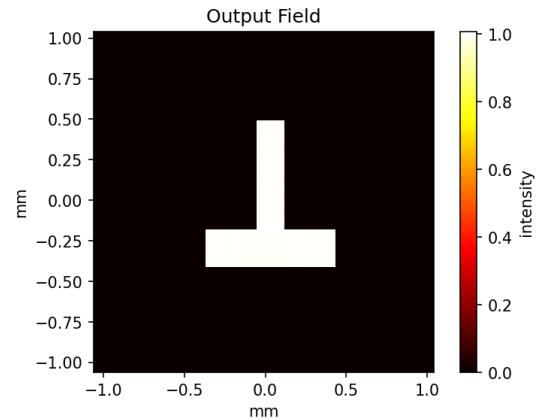
The software creates the following console output:

```
Field-of-view: 100x100
Total grid res: 216x216
processing component 0: f propagation
processing component 1: lens
processing component 2: f propagation
processing component 3: f propagation
processing component 4: lens
processing component 5: f propagation
```

The improved grid now fulfills the critical-sampling criterion, and the software produces the undistorted plots shown below:



(a) Input field ("T" pattern).



(b) Output field after the  $4f$  configuration.

Figure 3.8: With the grid sized by the critical-sampling rule the output is an undistorted, upside-down replica of the input field.

### 3.3 Code Example 2: Talbot Effect

To illustrate that the library reproduces well-known wave-optical phenomena we simulate the *Talbot effect* in the next code example.

To demonstrate this effect, we simulate the following situation: A plane-wave illumination passes a purely **phase** grating with period  $d$ . The term “phase grating” means that immediately behind the grating the intensity remains uniform, but the phase exhibits a periodic modulation. The “Talbot effect” now is this: After free-space propagation over a quarter of the *Talbot distance*

$$z_t = \frac{\lambda}{1 - \sqrt{1 - \lambda^2/d^2}}, \quad (3.7)$$

these phase modulations produce intensity modulations (and then again at distances  $\frac{1}{2}z_t$ ,  $\frac{3}{4}z_t$ ,  $z_t$ , ...).

In the following code example we propagate the field over  $\frac{1}{4}z_t$  to generate the first fractional Talbot image and compare the simulated intensity pattern with the analytical expectation.

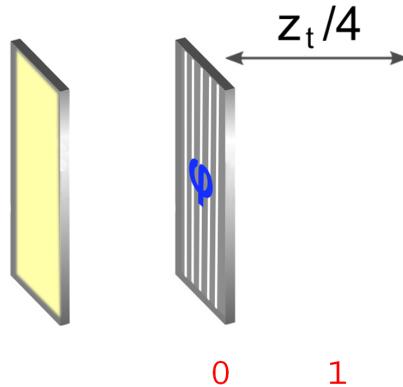


Figure 3.9: Configuration for the Talbot-effect simulation. Component 0 is the phase grating; component 1 propagates the field over one quarter of the Talbot distance  $z_T$ . The left-hand panel symbolises the incident plane-wave illumination.

The complete script for this Talbot-effect simulation is provided in the project's examples folder as `sample_001C.py`; the listing is reproduced below.

```

1 import math
2 from fo_cavity_sim import clsCavity1path, clsPropagation, \
3                               clsGrating, clsPlaneWaveMixField, Dir
4
5 if __name__ == '__main__':
6     length_fov = 0.001 # field-of-view sidelength
7
8     # --- BUILD CAVITY BREADBOARD -----
9     myCavity = clsCavity1path("my cavity")
10    myCavity.Lambda_nm = 633
11
12    # --- APPROXIMATE TALBOT DISTANCE -----
13    d_0 = 0.0001 # assumed grating period
14    zt_0 = 2 * d_0**2 / myCavity.Lambda # approximate Talbot distance
15
16    # --- CREATE OPTIMAL GRID FOR zt/4 PROPAGATION -----
17    myCavity.grid.set_opt_res_based_on_sidelength(length_fov, 1.5, zt_0/4, True)
18
19    # --- GENERATE COMPONENTS -----
20    # 0: cosine phase grating
21    grating = clsGrating("phase grating", myCavity)
22    grating.phase_grating = True
23    dx, x_max, dy, y_max = grating.set_cos_grating(d_0, length_fov, math.inf, 0,
24                                                    True, 0, math.pi/2)
25    # 1: free space propagation over 1/4 talbot-distance
26    prop = clsPropagation("zt/4 propagation", myCavity)
27    # calculate exact talbot distance based on actual phase grating
28    zt = myCavity.Lambda / (1 - math.sqrt(1-myCavity.Lambda**2/dx**2))
29    prop.set_params(zt/4, 1)
30    prop.transfer_function = 0
31
32    # --- ASSEMBLE OPTICAL SYSTEM -----
33    myCavity.add_component(grating)    # 0
34    myCavity.add_component(prop)       # 1
35
36    # --- INCIDENT PLANE WAVE -----
37    input_field = clsPlaneWaveMixField(myCavity.grid)
38    input_field.fov_only = False
39    input_field.add_fourier_basis_func(0, 0, 1)
40
41    # --- PLOT INTENSITY IMMEDIATELY AFTER GRATING -----
42    field_0 = myCavity.prop(input_field, 0, 0, Dir.LTR)
43    field_0.name = "Constant Intensity after Phase Grating"
44    field_0.plot_field(5, False)
45    # --- PLOT PHASE IMMEDIATELY AFTER GRATING -----
46    field_0.name = "Varying Phase after Phase Grating"
47    field_0.plot_field(4, False)
48
49    # --- PROPAGATE zt/4 AND PLOT INTENSITY -----
50    output_field = myCavity.prop(input_field, 0, 1, Dir.LTR)
51    output_field.name = "Varying Intensity after zt/4 Propagation"
52    output_field.plot_field(5, False)

```

### Import statements (lines 1–3)

The script starts by loading the classes and helper modules required for the Talbot-effect simulation:

- `math` – Python’s standard numerical library used for `math.inf`, `math.pi`.
- `clsCavity1path` – the “breadboard” on which the optical elements are placed.
- `clsPropagation` – models free-space propagation over an arbitrary distance.
- `clsGrating` – generates a user-defined phase or amplitude grating (here: a cosine phase mask).
- `clsPlaneWaveMixField` – convenience class that constructs plane-wave illumination from one or more Fourier-basis components.
- `Dir` – enumeration that supplies the propagation directions `.LTR` (left-to-right) and `.RTL` (right-to-left).

### Main-program guard (line 5)

The line `if __name__ == '__main__':` is essential on MS Windows, where the multiprocessing back-end used by the library must be able to import the module without accidentally re-running the top-level code in every worker process.

### Define field-of-view size (line 6)

`length_fov = 0.001` defines a side-length of 1 mm to the square field-of-view (FOV) tile on which the optical field will later be sampled.

### Build cavity breadboard (lines 8–10)

A `clsCavity1path` object named `myCavity` is instantiated and its vacuum wavelength is set to 633 nm via `myCavity.Lambda_nm`.

### Approximate Talbot distance (lines 12–14)

To define an *xy*-grid that fulfills the critical-sampling condition we need the propagation distance, which is a quarter of the Talbot distance  $z_T$ . However, the exact value of  $z_T$  depends on the exact value of the grating period  $d$  that emerges only *after* we have that very grid defined. This is the case because numerically, the initially defined period  $d_0$  “snaps” to the nearest multiple of the pixel pitch, so the exact grating period  $d_x$  becomes known *only after* the grid has been generated. To break this chicken-and-egg loop we proceed in two stages:

$$d_0 = 100 \text{ }\mu\text{m} \implies z_{T,0} = \frac{2 d_0^2}{\lambda}.$$

- The provisional  $z_{T,0}$  is sufficient to size the grid in the next step.
- Once the grid is fixed, the code recalculates the *exact* grating period  $d_x$  and the corresponding Talbot distance before the final propagation.

### Create optimal grid (lines 16–17)

`myCavity.grid.set_opt_res_based_on_sidelength(...)` is called with the desired field-of-view ( $L_{fov}$ ), a 1.5× padding factor, and the provisional propagation distance  $z_{T,0}/4$ . The

routine picks an  $N_{\text{tot}} \times N_{\text{tot}}$  grid that meets the critical-sampling condition for the full tile while embedding the smaller  $N_{\text{fov}} \times N_{\text{fov}}$  region of interest.

### Generate optical components (lines 19–30)

- **Phase grating (lines 20–23)** – An instance of `clsGrating` is created, switched to `.phase_grating=True`, and initialized via `grating.set_cos_grating(...)`. The call returns the *actual* period `dx`, which may differ slightly from the nominal `d_0` because the grating “snaps” to the pixel grid.
- **$z_T/4$  propagation (lines 24–30)** – A `clsPropagation` segment is created for the free-space distance  $z_T/4$ . The exact Talbot distance is recalculated with the final `dx` and passed to `prop.set_params(...)`. Finally, `prop.transfer_function=0` selects the Rayleigh–Sommerfeld transfer function (physically exact), which is required to reproduce the Talbot effect correctly, instead of the default Fresnel approximation (1).

### Assemble optical system (lines 32–34)

The two components are registered with the “breadboard” cavity in the same geometric order that appears in Figure 3.9: first the phase grating (component 0), then the  $z_T/4$  free-space section (component 1). Both calls use `myCavity.add_component(component)`.

### Define incident plane wave (lines 36–39)

An instance of `clsPlaneWaveMixField` – a convenience subclass of `clsLightField` – is created. Calling `input_field.add_fourier_basis_func(...)` with the parameters `(0, 0, 1)` makes the light field consisting of the single Fourier basis function with  $(k_x, k_y) = (0, 0)$  and unit amplitude, i.e. a uniform plane wave of constant phase and intensity propagating in the positive  $z$ -direction.

### Visualise field directly behind the grating (lines 41–47)

Calling `myCavity.prop(...)` with the parameters `(input_field, 0, 0, Dir.LTR)` in line 42 propagates the incident plane wave through *only* component 0 (the phase grating) and returns the resulting field in `field_0`. Two separate plots are produced:

- `.plot_field(5, False)` plots the *intensity* and shows that the intensity remains perfectly uniform immediately behind the grating, matching the expectation for pure phase modulation.
- `.plot_field(4, False)` visualizes the periodic *phase* pattern imprinted by the grating.

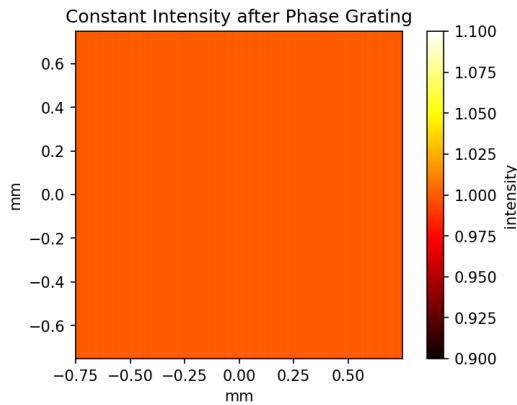
### Propagate over $\frac{1}{4}z_t$ and display Talbot image (lines 49–52)

Calling `myCavity.prop(...)` with the parameters `(input_field, 0, 1, Dir.LTR)` now propagates the plane wave through *both* components – the phase grating and the subsequent free-space section of length  $z_T/4$ . The returned field `output_field` is visualized in intensity-mode with `.plot_field(5, False)`. As predicted by Talbot theory, the formerly invisible phase modulation has converted into a clearly resolved intensity pattern that reproduces the grating period within the total computational tile.

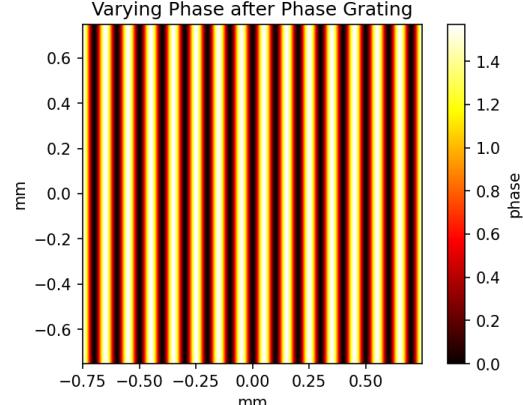
When the script is executed the following output is created:

- 1. First propagation step (lines 41–47)** — The field is propagated only through component 0 (the phase grating). The shared `clsProgressPrinter` instance reports the processing progress and two plots are produced that visualize the field *immediately* behind the grating:

```
processing component 0: phase grating
```



(a) Intensity: still flat, no energy modulation.



(b) Phase: clear cosine modulation.

Figure 3.10: Light-field diagnostics produced by lines 41–47.

- 2. Propagation over  $z_T/4$  (lines 49–52)** — Next, the same input plane wave is propagated through *both* components: the phase grating *and* the free-space section of length  $z_T/4$ . The progress printer now shows two more messages and a single plot appears that reveals how the pure phase modulation has converted into an intensity grating – the Talbot image:

```
processing component 0: phase grating
processing component 1: zt/4 propagation
```

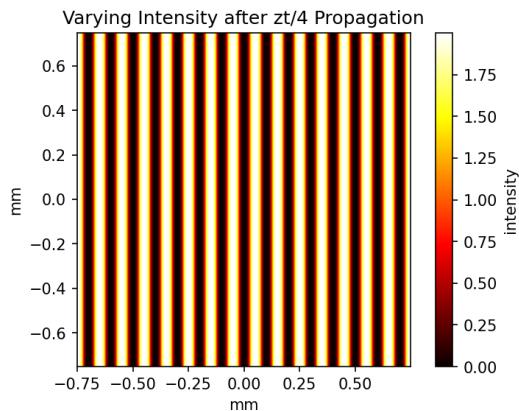


Figure 3.11: Intensity after a propagation of  $z_t/4$ . The Talbot effect is clearly visible.

# Chapter 4

## Linear Cavities

So far we have propagated light only in one direction. In a *linear cavity*, however, mirrors send the field back and forth ( $LTR \leftrightarrow RTL$ ), and the observable response at either side is the coherent sum of contributions from *infinitely many* round trips – every pass adds yet another term. This makes the problem fundamentally more involved. The following sections explain how the library deals with this situation and how to obtain the steady-state fields and reflection/transmission characteristics of such cavities.

### 4.1 Multiple Round-Trips Method

#### 4.1.1 Theory

Consider the simple linear cavity sketched in Figure 4.1: an input coupler with (power) reflectivity  $R_1$ , some intra-cavity optics (lenses, absorber), and an end mirror with reflectivity  $R_2$ . We are interested in the **left reflection** produced by a given left-incident field. A straightforward – and very intuitive – way to approximate the result is to **explicitly sum a finite number of round trips**.

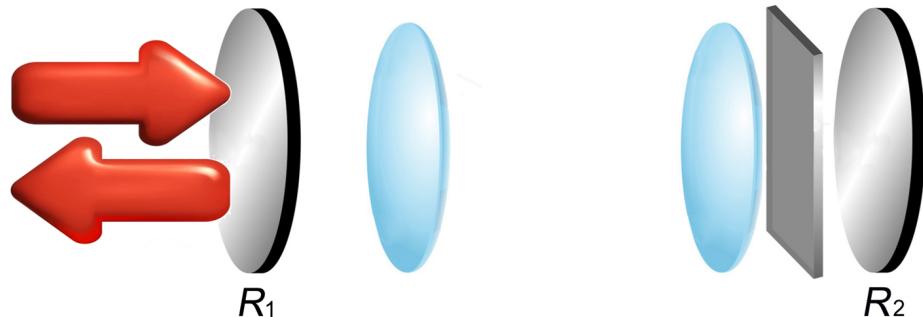


Figure 4.1: Model used to explain the multiple-round-trips method: an input coupler  $R_1$ , two lenses, an absorber, and an end mirror  $R_2$ . Each round trip loses a fraction of power by leaking through  $R_1$  and  $R_2$  and by absorption, so the series of contributions to the left output converges.

1. The incident field partially enters the cavity through the left input coupler.
2. It performs one round trip through the component stack.
3. A fraction  $1 - R_1$  leaks back through the input coupler and contributes to the left output; the remaining part stays in the cavity and performs yet another round trip.
4. Steps 2–3 are repeated and all contributions are coherently added.

Because any realistic cavity exhibits losses (output coupling, absorption, . . .), the intracavity power decays geometrically, and the sum converges rapidly. In the example of Figure 4.1 the dominant round-trip power factor is

$$\eta \equiv T_{\text{SR}} \approx R_1 R_2 T_{\text{abs}}^2, \quad (4.1)$$

where  $T_{\text{abs}}$  is the absorber’s one-way power transmissivity (lenses are assumed lossless here). After  $m$  round trips the remaining intra-cavity power is roughly  $\eta^m$ ; hence the residual contribution beyond the  $M$ -th round trip is of order  $\eta^M$ . A practical choice for the required number of round trips is therefore

$$M \gtrsim \frac{\ln \varepsilon}{\ln \eta}, \quad (4.2)$$

where  $\varepsilon$  is the desired relative accuracy (e.g.  $10^{-6}$ ).

#### 4.1.2 Code Example: MAD-CPA, Multiple Roundtrips Method

Figure 4.2 shows the linear cavity that we are going to simulate with the multiple-round-trips (MRT) method. It is a *massively-degenerate coherent perfect absorber* (MAD-CPA): at the appropriate resonance frequency the structure absorbs *any* incident transverse field that is injected from the left [2]. The red digits underneath the optics indicate the component indices (0...9) that we will use when adding the elements to the simulation; they must be appended in this exact order.

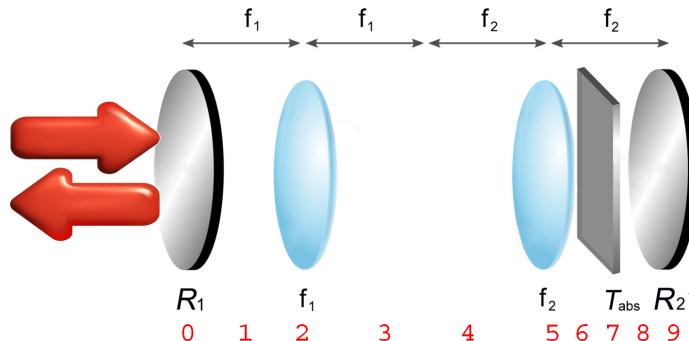


Figure 4.2: Optical layout of the MAD-CPA cavity to be simulated. A partially reflective input coupler  $R_1$  is followed by a  $4f$ -relay (lenses with focal lengths  $f_1$  and  $f_2$ ), an absorbing plate with power transmissivity  $T_{\text{abs}}$ , and a highly reflective end mirror  $R_2$ . Red numbers (0...9) label the components in the order they are added to the `clsCavity1path` instance.

**Full example.** The multiple-round-trips simulation of the MAD-CPA cavity, contained in examples/sample\_002A.py, is listed below.

```

1 import math
2 from fo_cavity_sim import clsCavity1path, clsMirror, clsPropagation, \
3     clsThinLens, clsSpeckleField, get_sample_vec
4
5 if __name__ == '__main__':
6     # --- SIMULATION PARAMETERS -----
7     R_left = 0.7                      # reflectivity left mirror
8     R_right = 0.999                    # reflectivity right mirror
9     d_absorb = 0.0006                  # thickness of absorber
10    T_abs = math.sqrt(R_left/R_right) # optimal transmittivity of absorber
11    pos_absorb = 0.005                # distance of absorber after left mirror
12    nr = 1.5                         # real part of absorber's refractive index
13    f1 = 0.075                       # focal length first lens
14    f2 = f1 - d_absorb/2*(nr-1/nr)   # second lens
15    length_fov = 0.0021              # field-of-view sidelength
16    epsilon = 1e-6                   # accuracy of steady-state approximation
17    TSR = R_left * R_right * T_abs**2 # single roundtrip transmittivity
18    no_of_RT = int(round(math.log(epsilon)/math.log(TSR))) # roundtrips
19
20    # --- CREATE CAVITY -----
21    myCavity = clsCavity1path("4f cavity")
22    myCavity.use_swap_files_in_bmatrix_class = False
23    myCavity.folder = "cavity_folder"
24    myCavity.Lambda_nm = 633
25
26    # --- DETERMINE RESONANCE WAVELENGTH, CREATE OPTIMAL GRID -----
27    lambda_c, k_c, n_c, lambda_period = \
28        myCavity.resonance_data_simple_cavity(R_left, R_right, 4*f1)
29    myCavity.Lambda = lambda_c
30    myCavity.grid.set_opt_res_tot_based_on_res_fov(length_fov, 100, 2*f1)
31    print(f"Field-of-view: {myCavity.grid.res_fov}x{myCavity.grid.res_fov}")
32    print(f"Total grid res: {myCavity.grid.res_tot}x{myCavity.grid.res_tot}")
33    print("Number of roundtrips:", no_of_RT)
34
35    # --- GENERATE COMPONENTS -----
36    # left input coupling mirror
37    mirror1 = clsMirror("left mirror", myCavity)
38    mirror1.R = R_left
39
40    # propagation over distance f1
41    prop_f1 = clsPropagation("f propagation", myCavity)
42    prop_f1.set_params(f1, 1)
43
44    # lens with focal length f
45    lens_f1 = clsThinLens("lens 1", myCavity)
46    lens_f1.f = f1
47
48    # propagation over distance f2
49    prop_f2 = clsPropagation("f2 propagation", myCavity)
50    prop_f2.set_params(f2, 1)
51
52    # lens with focal length f2
53    lens_f2 = clsThinLens("lens 2", myCavity)
54    lens_f2.f = f2
55
56    # propagation between second lens and absorber
57    prop_lens_absorb = clsPropagation("propagation lens<>absorber", myCavity)

```

```

58 prop_lens_absorb.set_params(f2-pos_absorb-d_absorb/nr, 1)
59
60 # absorber with thickness d_absorb
61 absorber = clsPropagation("absorber", myCavity)
62 absorber.set_params(d_absorb, nr)
63 absorber.set_ni_based_on_T(T_abs)
64
65 # propagation between absorber and right mirror
66 prop_abs_mirr = clsPropagation("propagation absorber<>mirror2", myCavity)
67 prop_abs_mirr.set_params(pos_absorb, 1)
68
69 # right mirror
70 mirror2 = clsMirror("right mirror", myCavity)
71 mirror2.R = R_right
72
73 # --- ASSEMBLE 4F CAVITY -----
74 myCavity.add_component(mirror1)          # 0
75 myCavity.add_component(prop_f1)         # 1
76 myCavity.add_component(lens_f1)          # 2
77 myCavity.add_component(prop_f1)         # 3
78 myCavity.add_component(prop_f2)          # 4
79 myCavity.add_component(lens_f2)          # 5
80 myCavity.add_component(prop_lens_absorb) # 6
81 myCavity.add_component(absorber)         # 7
82 myCavity.add_component(prop_abs_mirr)    # 8
83 myCavity.add_component(mirror2)          # 9
84
85 # --- INPUT: RANDOM SPECKLE FIELD -----
86 inp = clsSpeckleField(myCavity.grid)
87 inp.create_field_eq_distr(100, 20, 0.6*myCavity.grid.length_fov, 0)
88 inp.name = "Input Field"
89 inp.plot_field(5)
90 myCavity.incident_field_left = inp
91
92 # --- CALCULATE CAVITY'S LEFT OUTPUT FOR 3 DIFFERENT WAVELENGTHS -----
93 lambda_vec = get_sample_vec(3, lambda_period/120, 0, False, 1)
94 for index, dLambda in enumerate(lambda_vec):
95     # SET CURRENT WAVELENGTH
96     dL_pm = dLambda * 10**12 # delta lambda in pm
97     print("")
98     print(f"***  delta lambda = {dL_pm:.3f} pm  ***")
99     myCavity.Lambda = lambda_c + dLambda
100
101 # CALCULATE OUTPUT FIELD WITH "MULTIPLE ROUNDTRIPS" METHOD"
102 myCavity.prop_mult_round_trips_LTR_RTL(no_of_RT ,0, 9)
103 out = myCavity.output_field_left
104 out.name = f"delta lambda = {dL_pm:.3f} pm"
105 out.plot_field(5, vmax_limit=0.03, c_map = "custom")
106
107 # CALCULATE REFLECTION COEFFICIENT OVER FOV AND SAVE RESULT TO FILE
108 r = out.intensity_integral_fov() / inp.intensity_integral_fov()
109 print(f"Reflectivity: {r}")
110 data = [index, dL_pm, r]
111 myCavity.write_to_file("result.csv",data)

```

### Import statements (lines 1–3)

The script starts by loading the classes and helper modules required for the multiple-round-trips simulation of the MAD–CPA cavity:

- `math` – Python’s standard numerical library, used here for `math.sqrt`, `math.log`.
- `clsCavity1path` – The “breadboard” onto which the cavity is assembled.
- `clsMirror` – Simulates the left and right cavity mirrors.
- `clsPropagation` – Handles free-space propagation between optical elements and simulates propagation through the absorber.
- `clsThinLens` – Used to simulate the two lenses with focal lengths  $f_1$  and  $f_2$ .
- `clsSpeckleField` – subclass of `clsLightField` used for generating the incident light field. It produces a speckle pattern by superimposing random Fourier modes of equal amplitude and random phases.
- `get_sample_vec(...)` – Helper function returning sampling points distributed around a given center value, used here to generate a vector of wavelengths around the cavity’s resonance wavelength.

### Main-program guard (line 5)

The line `if __name__ == '__main__':` is essential on MS Windows, where the multiprocessing back-end used by the library must be able to import the module without accidentally re-running the top-level code in every worker process.

### Simulation parameters (lines 6–18)

This block defines all physical and numerical constants that characterise the MAD–CPA cavity and the accuracy of the multiple-round-trips (MRT) summation:

- `R_left = 0.7` and `R_right = 0.999`: power reflectivities of the left (input coupler) and right end mirror.
- `d_absorb = 0.6 mm`: physical thickness of the absorber.
- `T_abs =  $\sqrt{R_{\text{left}}/R_{\text{right}}}$` : “optimal” one-pass power transmissivity of the absorber that realises coherent perfect absorption for the chosen mirror reflectivities.
- `pos_absorb = 5 mm`: distance of the absorber from the left mirror.
- `nr = 1.5`: real part of the absorber’s refractive index.
- `f1 = 75 mm`: focal length of the first lens.
- `f2 := f1 - d_absorb/2(n_r - 1/n_r)`: effective focal length of the second lens, corrected to avoid refraction aberrations due to the absorber.
- `length_fov = 2.1 mm`: side length of the field-of-view tile.
- `epsilon =  $10^{-6}$` : target relative accuracy for truncating the infinite round-trip series.
- `TSR = R_left R_right T_{abs}^2`: single round-trip *power* transmission factor (dominant loss/gain factor of one cavity loop).

- `no_of_RT = ⌊ln(ε)/ln(TSR)⌋`: number of round trips to be explicitly summed (cf.  $M \gtrsim \ln \epsilon / \ln \eta$  in the theory section).

### Create cavity (lines 20–24)

These lines instantiate the working cavity object (`myCavity`, labeled “4f cavity”), disable disk-backed swap files so that all block matrices stay in RAM (faster as long as memory allows), set the output directory to `cavity_folder`, and fix the vacuum wavelength to 633 nm.

### Determine resonance wavelength and optimal grid size (lines 26–33)

First, the method `myCavity.resonance_data_simple_cavity(...)` is called with the according mirror reflectivities  $R_{\text{left}}, R_{\text{right}}$  and the cavity length  $4f_1$ . It returns the resonance wavelength  $\lambda_c$  closest to the initial wavelength, plus some auxiliary values  $k_c, n_c$ , and  $\lambda_{\text{period}}$ , the latter being the distance between resonance wavelengths. The simulation wavelength is then set to the resonance wavelength  $\lambda_c$ . Next, the spatial grid is created to satisfy the critical-sampling condition for a field-of-view of side length `length_fov` with  $100 \times 100$  pixels. The subsequent `print` statements report the chosen FOV resolution, total grid resolution, and the previously estimated number of round trips (`no_of_RT`) for a quick sanity check.

### Generate components (lines 35–71)

The following optical elements are instantiated and parameterised:

- `mirror1` – left input-coupling mirror, an instance of `clsMirror` with power reflectivity  $R_1 = R_{\text{left}}$ .
- `prop_f1` – free-space propagation over a distance  $f_1$ , implemented with `clsPropagation`; the distance  $f_1$  and refractive index 1 are set via `.set_params(...)`.
- `lens_f1` – first thin lens with focal length  $f_1$ , realised by `clsThinLens` and configured through its attribute `.f = f1`.
- `prop_f2` – free-space propagation over the distance  $f_2$ , again with `clsPropagation` and `.set_params(...)`.
- `lens_f2` – second thin lens with focal length  $f_2$ , another `clsThinLens` instance.
- `prop_lens_absorb` – propagation from the second lens to the absorber, modelled as `clsPropagation` over the distance  $f_2 - pos_{\text{absorb}} - d_{\text{absorb}}/n_r$ .
- `absorber` – the finite-thickness absorbing slab, also represented by `clsPropagation` with `.set_params(...)` setting the propagation distance to the absorber thickness  $d_{\text{absorb}}$  and the refractive index to  $n_r$ . Its imaginary refractive-index component is chosen via `.set_ni_based_on_T(...)` so that its one-way power transmissivity equals the designed value  $T_{\text{abs}}$ .
- `prop_abs_mirr` – propagation from the absorber to the right mirror over the distance `pos_absorb`.
- `mirror2` – right end mirror, a `clsMirror` with reflectivity  $R_2 = R_{\text{right}}$ .

### Assemble 4f cavity (lines 73–83)

The components are appended to the cavity in the exact geometric order shown in Figure 4.2; the indices (0–9) match the red labels in the figure:

```

0 mirror1 (input coupler)
1 prop_f1 (free space  $f_1$ )
2 lens_f1 (thin lens,  $f_1$ )
3 prop_f1 (reused, free space  $f_1$ )
4 prop_f2 (free space  $f_2$ )
5 lens_f2 (thin lens,  $f_2$ )
6 prop_lens_absorb (lens → absorber)
7 absorber (finite thickness, transmissivity  $T_{\text{abs}}$ )
8 prop_abs_mirr (absorber → right mirror)
9 mirror2 (end mirror)

```

Note that objects can be reused (e.g. `prop_f1` serves as both components 1 and 3). `myCavity.add_component(component)` merely registers the elements; no optical computation is triggered at this stage.

#### **Input: random speckle field (lines 85–90)**

A random speckle field is synthesized by instantiating `clsSpeckleField` and calling `inp.create_field_eq_distr(...)`. This method call generates a speckle pattern consisting of 100 random modes, whose populated plane-wave angles are approximately evenly distributed. The resulting instance `inpt` is named “Input Field” and visualised via `inp.plot_field(...)`. Finally, it is assigned to the cavity as the left incident field through the property `myCavity.incident_field.left`.

#### **Scan three wavelengths around resonance (lines 92–99)**

`get_sample_vec(...)` returns three wavelength offsets  $\Delta\lambda$  symmetrically distributed around zero. The sampling points lie in the interval  $[-\lambda_{\text{period}}/120, \lambda_{\text{period}}/120]$ . The subsequent `for`-loop iterates over these offsets, converts each  $\Delta\lambda$  to picometres for readable console output, and then sets the actual simulation wavelength to

$$\lambda = \lambda_c + \Delta\lambda.$$

This prepares the cavity for computing the left reflection at each detuning from resonance.

#### **Compute steady-state approximation via multiple round trips (lines 101–105)**

The call `myCavity.prop_mult_round_trips_LTR_RTL(...)` with the parameters `(no_of_RT, 0, 9)` is where the MRT (multiple-round-trips) approximation is executed. Starting from the `left` incident field at the left interface of component 0 (the left mirror), the routine propagates the field back and forth between components 0 and 9 for `no_of_RT` cycles, coherently accumulating all leaking contributions. When it finishes, the effective steady-state outputs are cached in `myCavity.output_field_left` and `myCavity.output_field_right`. Lines 103–105 then read the left output, label it with the current detuning  $\Delta\lambda$  (in pm), and display its intensity using a fixed colour scale (`vmax_limit = 0.03`).

### Compute and log reflectivity (lines 107–111)

The scalar cavity reflectivity over the field of view is evaluated as the ratio of the output and input *integrated* intensities,

$$r = \frac{\text{out.intensity\_integral\_fov}(\dots)}{\text{inp.intensity\_integral\_fov}(\dots)}. \quad (4.3)$$

The value is printed and, together with the loop counter `index` and the current detuning  $\Delta\lambda$  (in pm), appended to `result.csv` via `myCavity.write_to_file(...)`. The stored row is `[index, dL_pm, r]`.

#### 4.1.3 Simulation Results

The software creates the following console output:

```
Field-of-view: 100x100
Total grid res: 216x216
Number of roundtrips: 23

*** delta lambda = -0.006 pm ***
Calculating multiple LTR-RTL round-trips between left mirror and right mirror
roundtrip 23 of 23
Reflectivity: 0.020811645344743137

*** delta lambda = 0.000 pm ***
deleting lens phase mask of 'lens 1' from memory cache
deleting lens phase mask of 'lens 2' from memory cache
Calculating multiple LTR-RTL round-trips between left mirror and right mirror
roundtrip 23 of 23
Reflectivity: 6.09447765904176e-08

*** delta lambda = 0.006 pm ***
deleting lens phase mask of 'lens 1' from memory cache
deleting lens phase mask of 'lens 2' from memory cache
Calculating multiple LTR-RTL round-trips between left mirror and right mirror
roundtrip 23 of 23
Reflectivity: 0.020811648199204325
```

#### Generated CSV File.

Also, a file `result.csv` is created in the `cavity_folder`, containing the following three lines:

0, -0.0055651226349117955, 0.020811645344743137
1, 0.0, 6.09447765904176e-08
2, 0.0055651226349117955, 0.020811648199204325

### Plots.

The software also produces some plots. As a reference, it first shows the intensity of the left-incident speckle field that is injected into the cavity (Figure 4.3)

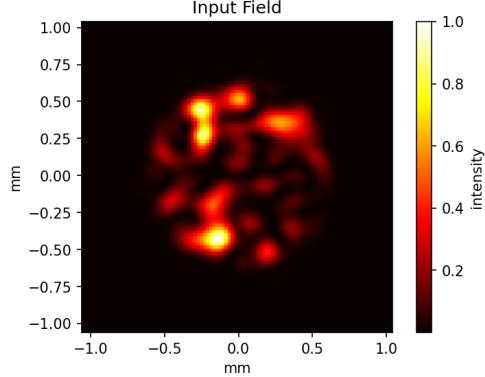


Figure 4.3: Incident speckle field injected from the left.

It then plots the three reflected fields for the different wavelength detunings: detunings are plotted.

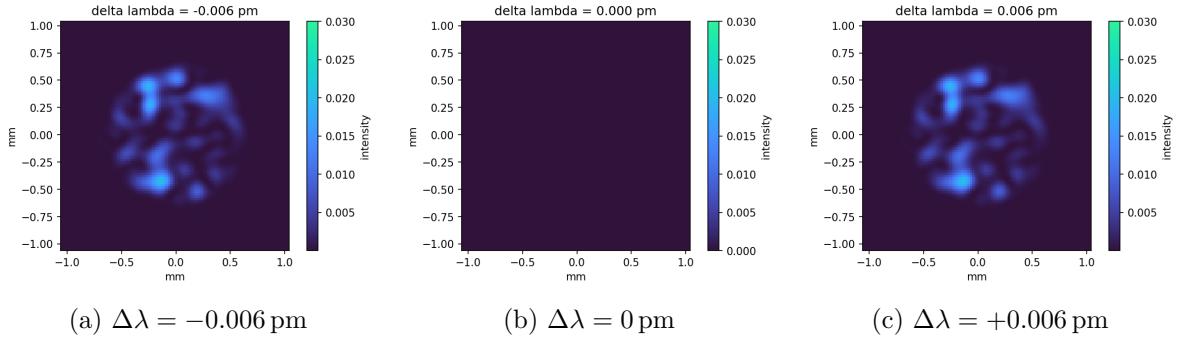


Figure 4.4: Left-output intensity for three wavelength detunings around the cavity resonance, computed with the multiple-round-trips method. At exact resonance the cavity acts as a coherent perfect absorber and the reflected field vanishes.

As expected for a MAD-CPA, the reflected power essentially vanishes at exact resonance ( $\Delta\lambda = 0$ ), whereas small detunings of only  $\pm 6$  pm already yield a finite reflectivity of about  $2.08 \times 10^{-2}$ . The off-resonant frames in Figure 4.4 show residual speckle-like structure, while the on-resonance frame is (numerically) dark.

## 4.2 Geometric Series Method

### 4.2.1 Theory: Transmission and Reflection Matrices

Up to now we have described light fields on a transverse plane at fixed  $z$  by their sampled complex amplitudes  $U(x, y; z)$  on an  $N \times N$  grid. For the geometric-series approach it is

necessary to switch to a *modal (coefficient) representation*: we expand the field into the discrete set of Fourier basis functions supported by that grid and stack their complex weights into a column vector.

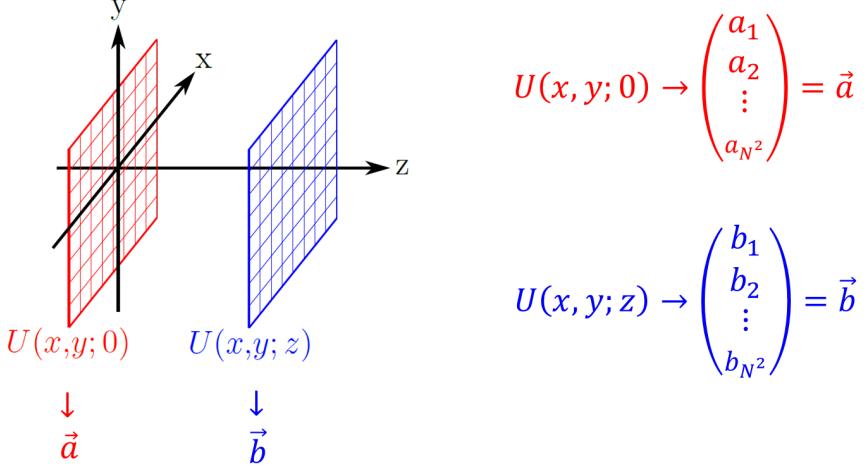


Figure 4.5: Field vectorization. The sampled fields on the input (red) and output (blue) planes,  $U(x,y;0)$  and  $U(x,y;z)$ , are expanded into the discrete Fourier basis of the  $N \times N$  grid. Their coefficients are collected in the vectors  $\mathbf{a} \in \mathbb{C}^{N^2}$  and  $\mathbf{b} \in \mathbb{C}^{N^2}$ , respectively.

Concretely, let  $\{\phi_i(x,y)\}_{i=1}^{N^2}$  denote the orthonormal discrete Fourier modes on the tile. Then

$$U(x,y;0) = \sum_{i=1}^{N^2} a_i \phi_i(x,y), \quad U(x,y;z) = \sum_i b_i \phi_i(x,y), \quad (4.4)$$

and we collect the  $N^2$  coefficients into vectors

$$\mathbf{a} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_{N^2} \end{pmatrix}, \quad \mathbf{b} = \begin{pmatrix} b_1 \\ b_2 \\ \vdots \\ b_{N^2} \end{pmatrix}. \quad (4.5)$$

Because all optical elements we consider are linear, the mapping from the input coefficients to the output coefficients is *linear* as well, and can therefore be written as a matrix–vector product

$$\mathbf{b} = \mathbf{T} \mathbf{a}, \quad (4.6)$$

where  $\mathbf{T} \in \mathbb{C}^{N^2 \times N^2}$  is the (modal) *transmission matrix* of the whole component stack between the two planes. Whenever reflections are involved (e.g. at partially reflective mirrors) we will analogously describe the mapping from incident to reflected modal coefficients by *reflection matrices*  $\mathbf{R}$ .

### How do we obtain the matrices $\mathbf{T}$ and $\mathbf{R}$ with Fourier optics?

Propagation through any optical element (or stack of elements) is *linear*. Therefore, if we know how the element transforms each basis pattern, we know how it acts on *any* input field (by superposition). Take the  $M = N^2$  discrete Fourier modes on our grid and label the

corresponding unit coefficient vectors by  $\{\mathbf{e}_j\}_{j=1}^M$  (e.g.  $\mathbf{e}_1 = (1, 0, 0, \dots)^\top$ ,  $\mathbf{e}_2 = (0, 1, 0, \dots)^\top$ ,  $\dots$ ). We now “poke” the system with each  $\mathbf{e}_j$  in turn and record the output. Concretely, for every  $j$ :

1. Interpret  $\mathbf{e}_j$  as a modal coefficient vector on the *input* plane and synthesise the corresponding spatial field  $U_j(x, y; z_0)$ .
2. Propagate  $U_j$  through the optical component (or the entire stack) with the usual Fourier–optics machinery to obtain  $U_j(x, y; z_1)$  on the *output* plane.
3. Project  $U_j(x, y; z_1)$  back onto the Fourier basis to get the output coefficient vector  $\mathbf{b}^{(j)} \in \mathbb{C}^M$ .

Placing these output vectors next to each other yields the *transmission matrix*

$$\mathbf{T} = [\mathbf{b}^{(1)} \ \mathbf{b}^{(2)} \ \dots \ \mathbf{b}^{(M)}] \in \mathbb{C}^{M \times M}, \quad (4.7)$$

because, by construction, its  $j$ -th column is exactly the response to the  $j$ -th basis input.

The very same recipe produces reflection matrices. If the basis mode  $\mathbf{e}_j$  is sent towards, say, the left boundary and the field that returns to the left is analysed into coefficients  $\mathbf{r}_L^{(j)}$ , then

$$\mathbf{R}_L = [\mathbf{r}_L^{(1)} \ \mathbf{r}_L^{(2)} \ \dots \ \mathbf{r}_L^{(M)}]. \quad (4.8)$$

Analogously one obtains  $\mathbf{R}_R$ ,  $\mathbf{T}_{LTR}$ , and  $\mathbf{T}_{RTL}$  for two-sided cavities.

In summary: *probe each basis mode once, collect the resulting coefficient vectors as columns, and you have the full matrix description of the element or stack.*

#### 4.2.2 Geometric-series method

To avoid explicitly summing a large number of round trips, we can work directly with transfer and reflection matrices and sum the resulting *geometric series* in closed form. The key ingredient is the *single round-trip matrix*  $\mathbf{T}_{RT}$ , which maps a field located *just to the right* of the input coupler, through the entire cavity to the end mirror, back to the same plane after reflection at the right mirror (LTR propagation to the right, reflection, RTL propagation back). This situation is sketched in Figure 4.6.

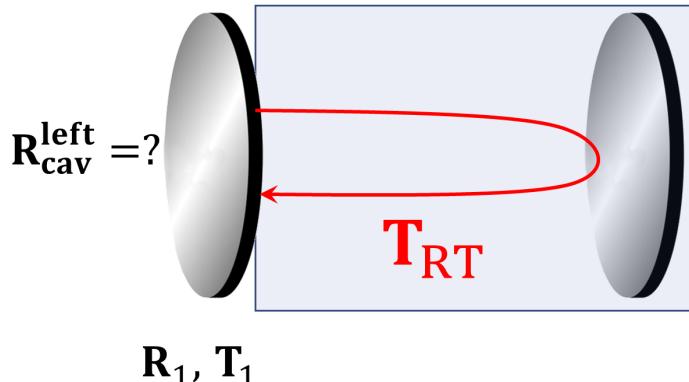


Figure 4.6: Single round trip in the cavity. Starting at the right-hand side of the input mirror  $R_1$ , the field propagates through an arbitrary sequence of intracavity elements (light-blue box), reaches the end mirror  $R_2$ , reflects, and returns to the starting plane. Knowing the corresponding matrix  $\mathbf{T}_{\text{RT}}$  allows one to sum all higher round trips via a geometric series to obtain the total left reflection matrix  $\mathbf{R}_{\text{cav}}^{\text{left}}$ .

#### Left reflection matrix from a geometric series.

Let  $\mathbf{R}_1$  denote the *internal* right-hand side reflection matrix of the input coupler, and  $\mathbf{T}_1$  the transmission matrix of the input coupler (left mirror). Then the cavity's net left reflection matrix can be written as an infinite sum of leakage terms:

$$\begin{aligned} \mathbf{R}_{\text{cav}}^{\text{left}} &= \underbrace{\mathbf{R}_1}_{\text{prompt reflection}} + \underbrace{\mathbf{T}_1 \mathbf{T}_{\text{RT}} \mathbf{T}_1}_{\text{1st RT leaks out}} + \underbrace{\mathbf{T}_1 \mathbf{T}_{\text{RT}} \mathbf{R}_1 \mathbf{T}_{\text{RT}} \mathbf{T}_1}_{\text{2nd RT leaks out}} + \cdots \\ &= \mathbf{R}_1 + \mathbf{T}_1 \mathbf{T}_{\text{RT}} \left[ \sum_{m=0}^{\infty} (\mathbf{R}_1 \mathbf{T}_{\text{RT}})^m \right] \mathbf{T}_1. \end{aligned} \quad (4.9)$$

This series can be summed in closed form to yield

$$\boxed{\mathbf{R}_{\text{cav}}^{\text{left}} = \mathbf{R}_1 + \mathbf{T}_1 \mathbf{T}_{\text{RT}} (\mathbb{1} - \mathbf{R}_1 \mathbf{T}_{\text{RT}})^{-1} \mathbf{T}_1} \quad (4.10)$$

In the library, this workflow is encapsulated in `.get_inf_round_trip_R_L_mat(...)`, which performs all steps above and returns  $\mathbf{R}_{\text{cav}}^{\text{left}}$  directly.

#### 4.2.3 Code Example: MAD-CPA, Geometric Series Method

We now demonstrate how to implement the geometric-series method to simulate the same MAD-CPA configuration shown in Figure 4.2.

The full source listing is given below and is also available as `examples/sample_002C.py`.

```

1 import math
2 from fo_cavity_sim import clsCavity1path, clsMirror, clsPropagation, \
3     clsThinLens, clsSpeckleField, get_sample_vec
4
5 if __name__ == '__main__':
6     # --- SIMULATION PARAMETERS -----
7     R_left = 0.7          # reflectivity left mirror
8     R_right = 0.999        # reflectivity right mirror
9     d_absorb = 0.0006      # thickness of absorber
10    T_abs = math.sqrt(R_left/R_right) # optimal transmittivity of absorber
11    pos_absorb = 0.005       # distance of absorber after left mirror
12    nr = 1.5              # real part of absorber's refractive index
13    f1 = 0.075             # focal length first lens
14    f2 = f1 - d_absorb/2*(nr-1/nr) # second lens
15    length_fov = 0.0021     # field-of-view sidelength
16
17    # --- CREATE CAVITY -----
18    myCavity = clsCavity1path("4f cavity")
19    myCavity.use_swap_files_in_bmatrix_class = True
20    myCavity.folder = "cavity_folder"
21    myCavity.tmp_folder = "tmp_folder"
22    myCavity.Lambda_nm = 633
23
24    # --- DETERMINE RESONANCE WAVELENGTH, CREATE OPTIMAL GRID -----
25    lambda_c, k_c, n_c, lambda_period = \
26        myCavity.resonance_data_simple_cavity(R_left, R_right, 4*f1)
27    myCavity.Lambda = lambda_c
28    myCavity.grid.set_opt_res_tot_based_on_res_fov(length_fov, 100, 2*f1)
29    print(f"Field-of-view: {myCavity.grid.res_fov}x{myCavity.grid.res_fov}")
30    print(f"Total grid res: {myCavity.grid.res_tot}x{myCavity.grid.res_tot}")
31
32    # --- GENERATE COMPONENTS -----
33
34    # left input coupling mirror
35    mirror1 = clsMirror("left mirror", myCavity)
36    mirror1.R = R_left
37
38    # propagation over distance f1
39    prop_f1 = clsPropagation("f propagation", myCavity)
40    prop_f1.set_params(f1, 1)
41
42    # lens with focal length f
43    lens_f1 = clsThinLens("lens 1", myCavity)
44    lens_f1.f = f1
45
46    # propagation over distance f2
47    prop_f2 = clsPropagation("f2 propagation", myCavity)
48    prop_f2.set_params(f2, 1)
49
50    # lens with focal length f2
51    lens_f2 = clsThinLens("lens 2", myCavity)
52    lens_f2.f = f2
53
54    # propagation between second lens and absorber
55    prop_lens_absorb = clsPropagation("propagation lens<>absorber", myCavity)
56    prop_lens_absorb.set_params(f2-pos_absorb-d_absorb/nr, 1)
57
58    # absorber with thickness d_absorb

```

```

59     absorber = clsPropagation("absorber", myCavity)
60     absorber.set_params(d_absorb, nr)
61     absorber.set_ni_based_on_T(T_abs)
62
63     # propagation between absorber and right mirror
64     prop_abs_mirr = clsPropagation("propagation absorber<>mirror2", myCavity)
65     prop_abs_mirr.set_params(pos_absorb, 1)
66
67     # right mirror
68     mirror2 = clsMirror("right mirror", myCavity)
69     mirror2.R = R_right
70
71     # --- ASSEMBLE 4F CAVITY -----
72     myCavity.add_component(mirror1)           # 0
73     myCavity.add_component(prop_f1)          # 1
74     myCavity.add_component(lens_f1)          # 2
75     myCavity.add_component(prop_f1)          # 3
76     myCavity.add_component(prop_f2)          # 4
77     myCavity.add_component(lens_f2)          # 5
78     myCavity.add_component(prop_lens_absorb) # 6
79     myCavity.add_component(absorber)         # 7
80     myCavity.add_component(prop_abs_mirr)    # 8
81     myCavity.add_component(mirror2)          # 9
82
83     # --- INPUT: RANDOM SPECKLE FIELD -----
84     inp = clsSpeckleField(myCavity.grid)
85     inp.create_field_eq_distr(100, 20, 0.6*myCavity.grid.length_fov, 0)
86     inp.name = "Input Field"
87     inp.plot_field(5)
88
89     # --- CALCULATE CAVITY'S LEFT OUTPUT FOR 3 DIFFERENT WAVELENGTHS -----
90     lambda_vec = get_sample_vec(3, lambda_period/120, 0, False, 1)
91     for index, dLambda in enumerate(lambda_vec):
92         # SET CURRENT WAVELENGTH
93         dL_pm = dLambda * 10**12 # delta lambda in pm
94         print("")
95         print(f"***  delta lambda = {dL_pm:.3f} pm  ***")
96         myCavity.Lambda = lambda_c + dLambda
97
98         # CALCULATE OUTPUT FIELD WITH "GEOMETRIC SERIES" METHOD"
99         R = myCavity.get_inf_round_trip_R_L_mat(0,9)
100        out = inp.apply_TR_mat(R)
101        out.name = f"delta lambda = {dL_pm:.3f} pm"
102        out.plot_field(5, vmax_limit=0.03, c_map = "custom")
103
104        # CALCULATE REFLECTION COEFFICIENT OVER FOV AND SAVE RESULT TO FILE
105        r = out.intensity_integral_fov() / inp.intensity_integral_fov()
106        print(f"Reflectivity: {r}")
107        data = [index, dL_pm, r]
108        myCavity.write_to_file("result.csv",data)

```

### How this listing differs from the MRT version

This listing is almost the same as the multiple-round-trips (MRT) example on p. 25; therefore we only describe the differences:

- **No round-trip counter in the simulation parameters (lines 6–15).** The geometric-series approach evaluates the infinite sum in closed form, so there is no need to estimate the required number of round trips (no `epsilon`, `TSR`, or `no_of_RT`).

- **On-disk caching for block matrices (lines 19–21).** File-backed caching is enabled via `myCavity.use_swap_files_in_bmatrix_class = True`, and the temporary directory is set with `myCavity.tmp_folder = "tmp_folder"`. This reduces RAM pressure.
- **Input field is not attached to the cavity (lines 83–87).** The speckle field `inp` is created and plotted as before, but it is *not* assigned to `myCavity.incident_field_left`. In the geometric-series workflow we apply the cavity's left-reflection operator directly to the incident field `inp`.
- **Geometric-series evaluation of the left reflection (lines 99–101).** The left reflection matrix for the whole cavity, (components 0 through 9) is obtained via `R = myCavity.get_inf_round_trip_R_L_mat(...)`, and the reflected field is computed by applying this reflection matrix directly to the input field by calling `out = inp.apply_TR_mat(...)`.

All remaining steps are identical as in the multiple-round-trips-listing on p. 25f.

#### 4.2.4 Simulation Results

The software creates the following console output:

```
Field-of-view: 100x100
Total grid res: 216x216

*** delta lambda = -0.006 pm ***
calculating left reflection matrix after infinite roundtrips between left mirror and right mirror
Calculating single LTR-RTL roundtrip between left mirror and right mirror
left-to-right propagation:
    processing component 1: f propagation
    processing component 2: lens 1
        calculating transmission matrix T
        done (177.1 seconds)
    done (205.3 seconds)
    processing component 3: f propagation
    done (72.6 seconds)
    processing component 4: f2 propagation
    done (8.2 seconds)
    processing component 5: lens 2
        calculating transmission matrix T
        done (182.4 seconds)
    done (842.2 seconds)
    processing component 6: propagation lens<>absorber
    done (71.5 seconds)
    processing component 7: absorber
    done (8.3 seconds)
    processing component 8: propagation absorber<>mirror2
    done (8.6 seconds)
done (1216.7 seconds)
calculating reflection at right mirror
done (7.7 seconds)
right-to-left propagation:
    processing component 8: propagation absorber<>mirror2
    done (8.4 seconds)
    processing component 7: absorber
    done (7.6 seconds)
    processing component 6: propagation lens<>absorber
    done (8.4 seconds)
    processing component 5: lens 2
    done (696.2 seconds)
    processing component 4: f2 propagation
    done (71.1 seconds)
```

```

processing component 3: f propagation
done (8.0 seconds)
processing component 2: lens 1
done (722.0 seconds)
processing component 1: f propagation
done (73.5 seconds)
done (1595.2 seconds)
done (2819.6 seconds)
calculating infinite roundtrip T-matrix
done (1090.4 seconds)
finalizing calculation
done (809.2 seconds)
done (4726.4 seconds)
Reflectivity: 0.020873309562584166

*** delta lambda = 0.000 pm ***
deleting transmission matrix T of 'f propagation' from memory cache
deleting lens phase mask of 'lens 1' from memory cache
deleting transmission matrix T of 'lens 1' from memory cache
deleting transmission matrix T of 'f2 propagation' from memory cache
deleting lens phase mask of 'lens 2' from memory cache
deleting transmission matrix T of 'lens 2' from memory cache
deleting transmission matrix T of 'propagation lens<>absorber' from memory cache
deleting transmission matrix T of 'absorber' from memory cache
deleting transmission matrix T of 'propagation absorber<>mirror2' from memory cache
calculating left reflection matrix after infinite roundtrips between left mirror and right mirror
Calculating single LTR-RTL roundtrip between left mirror and right mirror
left-to-right propagation:
  processing component 1: f propagation
  processing component 2: lens 1
    calculating transmission matrix T
    done (150.4 seconds)
  done (178.5 seconds)
  processing component 3: f propagation
  done (72.0 seconds)
  processing component 4: f2 propagation
  done (8.1 seconds)
  processing component 5: lens 2
    calculating transmission matrix T
    done (179.6 seconds)
  done (909.4 seconds)
  processing component 6: propagation lens<>absorber
  done (73.4 seconds)
  processing component 7: absorber
  done (7.9 seconds)
  processing component 8: propagation absorber<>mirror2
  done (8.5 seconds)
done (1257.8 seconds)
calculating reflection at right mirror
done (7.7 seconds)
right-to-left propagation:
  processing component 8: propagation absorber<>mirror2
  done (8.1 seconds)
  processing component 7: absorber
  done (7.6 seconds)
  processing component 6: propagation lens<>absorber
  done (8.0 seconds)
  processing component 5: lens 2
  done (754.7 seconds)
  processing component 4: f2 propagation
  done (71.3 seconds)
  processing component 3: f propagation
  done (7.7 seconds)
  processing component 2: lens 1
  done (768.8 seconds)
  processing component 1: f propagation
  done (70.7 seconds)
done (1696.8 seconds)
done (2962.3 seconds)

```

```

calculating infinite roundtrip T-matrix
done (1102.0 seconds)
finalizing calculation
done (817.2 seconds)
done (4887.4 seconds)
Reflectivity: 8.505302809310465e-09

*** delta lambda = 0.006 pm ***
deleting transmission matrix T of 'f propagation' from memory cache
deleting lens phase mask of 'lens 1' from memory cache
deleting transmission matrix T of 'lens 1' from memory cache
deleting transmission matrix T of 'f2 propagation' from memory cache
deleting lens phase mask of 'lens 2' from memory cache
deleting transmission matrix T of 'lens 2' from memory cache
deleting transmission matrix T of 'propagation lens<>absorber' from memory cache
deleting transmission matrix T of 'absorber' from memory cache
deleting transmission matrix T of 'propagation absorber<>mirror2' from memory cache
calculating left reflection matrix after infinite roundtrips between left mirror and right mirror
Calculating single LTR-RTL roundtrip between left mirror and right mirror
left-to-right propagation:
processing component 1: f propagation
processing component 2: lens 1
    calculating transmission matrix T
    done (157.9 seconds)
done (185.3 seconds)
processing component 3: f propagation
done (69.0 seconds)
processing component 4: f2 propagation
done (7.6 seconds)
processing component 5: lens 2
    calculating transmission matrix T
    done (179.8 seconds)
done (924.6 seconds)
processing component 6: propagation lens<>absorber
done (71.8 seconds)
processing component 7: absorber
done (8.4 seconds)
processing component 8: propagation absorber<>mirror2
done (7.7 seconds)
done (1274.5 seconds)
calculating reflection at right mirror
done (7.1 seconds)
right-to-left propagation:
processing component 8: propagation absorber<>mirror2
done (7.8 seconds)
processing component 7: absorber
done (8.1 seconds)
processing component 6: propagation lens<>absorber
done (8.3 seconds)
processing component 5: lens 2
done (765.5 seconds)
processing component 4: f2 propagation
done (70.6 seconds)
processing component 3: f propagation
done (7.9 seconds)
processing component 2: lens 1
done (780.7 seconds)
processing component 1: f propagation
done (70.8 seconds)
done (1719.7 seconds)
done (3001.3 seconds)
calculating infinite roundtrip T-matrix
done (1110.5 seconds)
finalizing calculation
done (821.8 seconds)
done (4939.6 seconds)
Reflectivity: 0.020873312416837986

```

### Interpreting the console output.

We see that, for each wavelength detuning  $\Delta\lambda$ , the solver first (re)computes the single LTR→RTL round-trip by propagating through each component; it then assembles the infinite-round-trip operator  $(\mathbb{1} - \mathbf{R}_1 \mathbf{T}_{RT})^{-1}$ , and finally builds the left-reflection matrix. Because all transmission matrices depend on wavelength, cached phase masks and transmission matrices are purged and rebuilt at each  $\Delta\lambda$  (cf. the “deleting ... from memory cache” lines). This route is computationally heavier than the explicit multiple-round-trips summation, but it returns the steady-state response in closed form (i.e. without truncation error from a finite round-trip count), limited only by numerical precision.

### Generated CSV File.

For each wavelength the script appends one line to `result.csv` with three comma-separated columns:

```
0, -0.0055651226349117955, 0.020873309562584166
1, 0.0, 8.505302809310465e-09
2, 0.0055651226349117955, 0.020873312416837986
```

### Plots.

As in the MRT run (cf. Figure 4.3), the code first displays the intensity of the left-incident speckle field (Fig. 4.7). It is effectively identical to the earlier case.

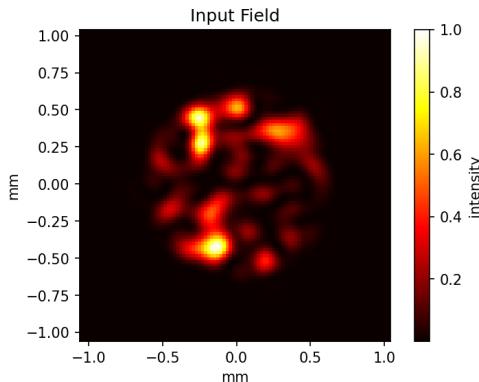


Figure 4.7: Incident speckle field injected from the left.

Subsequently, the reflected fields for the three detunings are shown in Figure 4.8. They closely match the MRT results in Figure 4.4: at exact resonance the left output is numerically dark, while a detuning of  $\pm 6$  pm yields a weak speckle-like residual. Minor pixel-level differences are attributable to independent numerical tolerances and caching between the two methods.

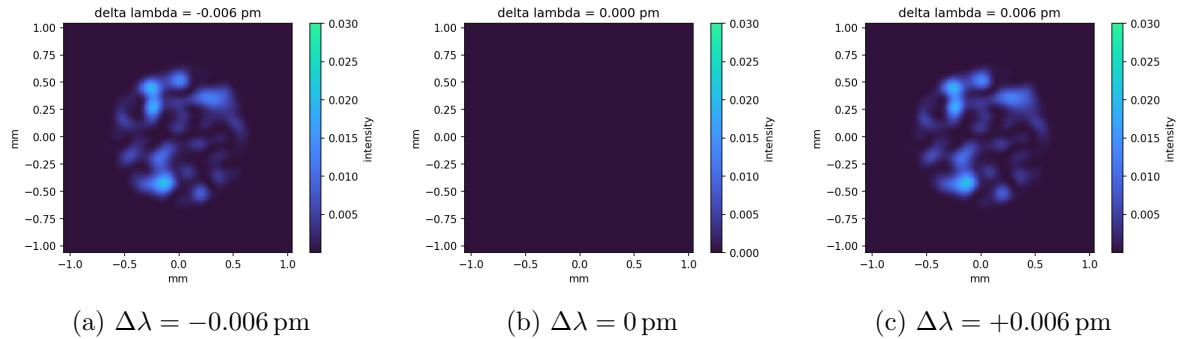


Figure 4.8: Left-output intensity for three wavelength detunings around the cavity resonance, computed with the geometric series method. At exact resonance the cavity acts as a coherent perfect absorber and the reflected field vanishes.

### 4.3 Scattering and Transfer Matrix Approach

Many practical systems consist of *coupled* or *cascaded* cavities with optical drives from both sides, and our goal is to predict the steady-state fields leaving the structure on the left and on the right (see Figure 4.9). For such bidirectional, multi-section layouts it is convenient to switch to a framework involving *scattering matrices* and *transfer matrices* (not to be confused with the *transmission matrices* used earlier). This approach naturally accommodates simultaneous left/right drives, yields global reflection and transmission in a single linear-algebra step, and also allows one to recover the intracavity field at an arbitrary interface or “probe” plane within the stack.

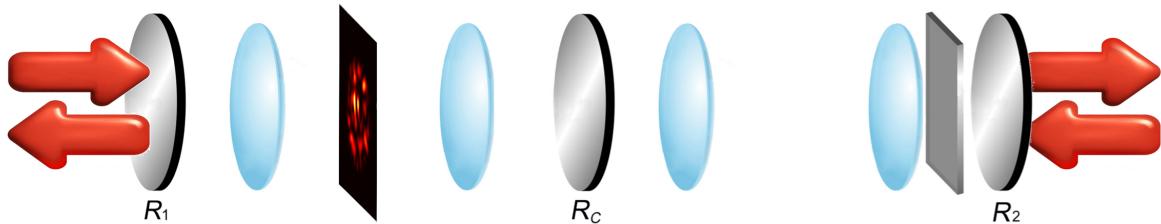


Figure 4.9: Coupled cavities with possible incident fields from both sides; besides the left/right outputs, the scattering-matrix framework also enables evaluation of the intracavity field at designated planes.

In the following sub-section we explain the theory behind this approach, and show how they are used to compute both the global left/right responses and selected intracavity fields.

### 4.3.1 Theory: Scattering- and Transfer-Matrices

#### Scattering Matrices

With the reflection and transmission matrices available, we can easily assemble for each optical element a so-called *scattering matrix*  $\mathbf{S}$  by placing the element's transfer and reflection matrices into the quadrants as shown in Figure 4.10.

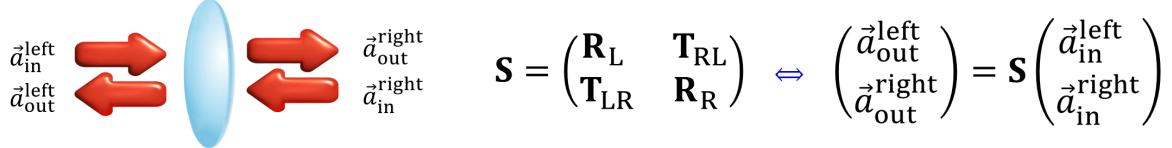


Figure 4.10: Scattering-matrix convention used in this library for any two-port optical element. The block entries are the element's reflection matrices  $\mathbf{R}_L, \mathbf{R}_R$  and transmission matrices  $\mathbf{T}_{LR}, \mathbf{T}_{RL}$ .

The scattering matrix  $\mathbf{S}$  is a block matrix composed of four sub-matrices:  $\mathbf{R}_L$  and  $\mathbf{R}_R$  are the reflection matrices for light incident from the left and right, respectively, while  $\mathbf{T}_{LR}$  and  $\mathbf{T}_{RL}$  are the transmission matrices for light passing from left-to-right and right-to-left, respectively. Conventions differ in the literature; in our software library we use the the scattering matrix definition as shown in Figure 4.10, which means that the scattering matrix sets the incident and outgoing fields into relation as follows:

$$\begin{pmatrix} \mathbf{a}_{out}^{left} \\ \mathbf{a}_{out}^{right} \end{pmatrix} = \mathbf{S} \begin{pmatrix} \mathbf{a}_{in}^{left} \\ \mathbf{a}_{in}^{right} \end{pmatrix} = \begin{pmatrix} \mathbf{R}_L & \mathbf{T}_{RL} \\ \mathbf{T}_{LR} & \mathbf{R}_R \end{pmatrix} \begin{pmatrix} \mathbf{a}_{in}^{left} \\ \mathbf{a}_{in}^{right} \end{pmatrix}. \quad (4.11)$$

Here  $\mathbf{a}_{out}^{left}$ ,  $\mathbf{a}_{out}^{right}$ ,  $\mathbf{a}_{in}^{left}$ , and  $\mathbf{a}_{in}^{right}$  are the coefficient vectors of the discrete Fourier basis with size  $N^2$ . Consequently, each block  $\mathbf{R}_L, \mathbf{R}_R, \mathbf{T}_{LR}, \mathbf{T}_{RL}$  is a  $N^2 \times N^2$  matrix, and the whole scattering matrix  $\mathbf{S}$  is of size  $2N^2 \times 2N^2$ .

With the reflection and transmission blocks in hand, it is tempting to “stack” scattering matrices for a chain of elements. Unfortunately, scattering matrices do not compose by a simple product. A convenient workaround is to convert each element's scattering matrix  $\mathbf{S}$  into a *transfer matrix*  $\mathbf{M}$ , which *does* cascade by ordinary matrix multiplication.

#### Transfer Matrices

In our software we use the following convention for transfer matrices  $\mathbf{M}$ :

$$\begin{pmatrix} \mathbf{a}_{out}^{left} \\ \mathbf{a}_{in}^{left} \end{pmatrix} = \mathbf{M} \begin{pmatrix} \mathbf{a}_{in}^{right} \\ \mathbf{a}_{out}^{right} \end{pmatrix} \quad (4.12)$$

A transfer matrix  $\mathbf{M}$  maps the right-side fields to the left-side fields of an element. For an optical component with a scattering matrix

$$\mathbf{S} = \begin{pmatrix} \mathbf{S}_{11} & \mathbf{S}_{12} \\ \mathbf{S}_{21} & \mathbf{S}_{22} \end{pmatrix}, \quad (4.13)$$

the corresponding transfer matrix can be computed via the following relation [4]:

$$\mathbf{M}(\mathbf{S}) = \begin{pmatrix} \mathbf{S}_{12} - \mathbf{S}_{11} \mathbf{S}_{21}^{-1} \mathbf{S}_{22} & \mathbf{S}_{11} \mathbf{S}_{21}^{-1} \\ -\mathbf{S}_{21}^{-1} \mathbf{S}_{22} & \mathbf{S}_{21}^{-1} \end{pmatrix}. \quad (4.14)$$

### Cascading

If the optical elements  $1, 2, \dots, N$  are placed from left to right, their transfer matrices combine as

$$\mathbf{M}_{\text{tot}} = \mathbf{M}_1 \mathbf{M}_2 \cdots \mathbf{M}_N, \quad (4.15)$$

i.e. by a simple matrix product in spatial order.

### Retrieving the total scattering matrix

To recover the global scattering matrix  $\mathbf{S}_{\text{tot}}$  of the full stack (from which the overall reflection and transmission blocks can be read off directly), we use the back-conversion [4]

$$\mathbf{S}_{\text{tot}} = \begin{pmatrix} \mathbf{R}_L & \mathbf{T}_{RL} \\ \mathbf{T}_{LR} & \mathbf{R}_R \end{pmatrix} = \begin{pmatrix} \mathbf{M}_{12} \mathbf{M}_{22}^{-1} & \mathbf{M}_{11} - \mathbf{M}_{12} \mathbf{M}_{22}^{-1} \mathbf{M}_{21} \\ \mathbf{M}_{22}^{-1} & -\mathbf{M}_{22}^{-1} \mathbf{M}_{21} \end{pmatrix} \quad (4.16)$$

#### 4.3.2 Algorithm for calculating left and right output fields

The scattering/transfer-matrix framework leads to a simple, reproducible workflow for computing the steady-state outputs on both sides of an arbitrary multi-element cavity (cf. Figure 4.11).

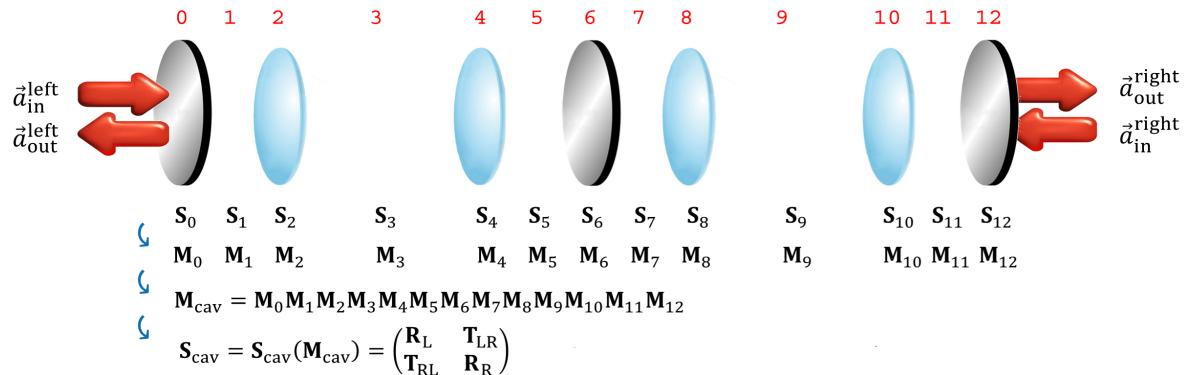


Figure 4.11: Visualization of the scattering and transfer matrix algorithm: (i) Scattering matrices  $\mathbf{S}_i$  are generated for each element; (ii) then converted into transfer matrices  $\mathbf{M}_i$ ; (iii) which are cascaded to calculate  $\mathbf{M}_{\text{cav}}$ ; (iv) which is finally back-converted into the global scattering matrix  $\mathbf{S}_{\text{cav}}$ .

### Steps

#### 1. Calculate the reflection and transmission matrices for each component.

For each component  $i$ , the reflection and transmission matrices  $\mathbf{R}_L$ ,  $\mathbf{R}_R$ ,  $\mathbf{T}_{LR}$ ,  $\mathbf{T}_{RL}$  are computed using Fourier optics.

**2. Assemble the scattering matrix for each component.**

For each optical component  $i$ , the scattering block matrix  $\mathbf{S}_i$  is assembled:

$$\mathbf{S}_i = \begin{pmatrix} \mathbf{R}_L & \mathbf{T}_{RL} \\ \mathbf{T}_{LR} & \mathbf{R}_R \end{pmatrix}. \quad (4.17)$$

**3. Convert to transfer matrices.**

For each optical component  $i$ , the scattering block matrix  $\mathbf{S}_i$  is converted to the corresponding transfer matrix  $\mathbf{M}_i$ .

**4. Cascade transfer matrices.**

Multiply the  $\mathbf{M}_i$  in spatial order (left→right) to get  $\mathbf{M}_{\text{cav}} = \mathbf{M}_1 \mathbf{M}_2 \cdots \mathbf{M}_N$ .

**5. Back-convert.**

Convert  $\mathbf{M}_{\text{cav}}$  to the global scattering matrix  $\mathbf{S}_{\text{cav}}$  via (4.16).

**6. Apply to inputs.**

Given incident coefficient vectors  $\mathbf{a}_{\text{in}}^{\text{left}}$  and  $\mathbf{a}_{\text{in}}^{\text{right}}$ , compute the outputs in one linear step:

$$\begin{pmatrix} \mathbf{a}_{\text{out}}^{\text{left}} \\ \mathbf{a}_{\text{out}}^{\text{right}} \end{pmatrix} = \mathbf{S}_{\text{cav}} \begin{pmatrix} \mathbf{a}_{\text{in}}^{\text{left}} \\ \mathbf{a}_{\text{in}}^{\text{right}} \end{pmatrix}. \quad (4.18)$$

Equivalently, in block form,

$$\mathbf{a}_{\text{out}}^{\text{left}} = \mathbf{R}_L \mathbf{a}_{\text{in}}^{\text{left}} + \mathbf{T}_{RL} \mathbf{a}_{\text{in}}^{\text{right}}, \quad \mathbf{a}_{\text{out}}^{\text{right}} = \mathbf{R}_R \mathbf{a}_{\text{in}}^{\text{right}} + \mathbf{T}_{LR} \mathbf{a}_{\text{in}}^{\text{left}}. \quad (4.19)$$

### 4.3.3 Intracavity field from the scattering/transfer matrices

Before computing intracavity fields we assume that the steady-state *exterior* fields have already been obtained with the scattering-matrix approach, i.e. the incident vectors  $\mathbf{a}_{\text{in}}^{\text{left}}$ ,  $\mathbf{a}_{\text{in}}^{\text{right}}$  and the corresponding outputs  $\mathbf{a}_{\text{out}}^{\text{left}}$ ,  $\mathbf{a}_{\text{out}}^{\text{right}}$  are known.

#### Goal

Recover the directional intracavity fields  $\mathbf{a}_{\text{RTL}}^{\text{bulk}}$  and  $\mathbf{a}_{\text{LTR}}^{\text{bulk}}$  at the interface *to the left of* component 6 (i.e. at the right side of component 5 in Figure 4.12). The same procedure works symmetrically from the right.

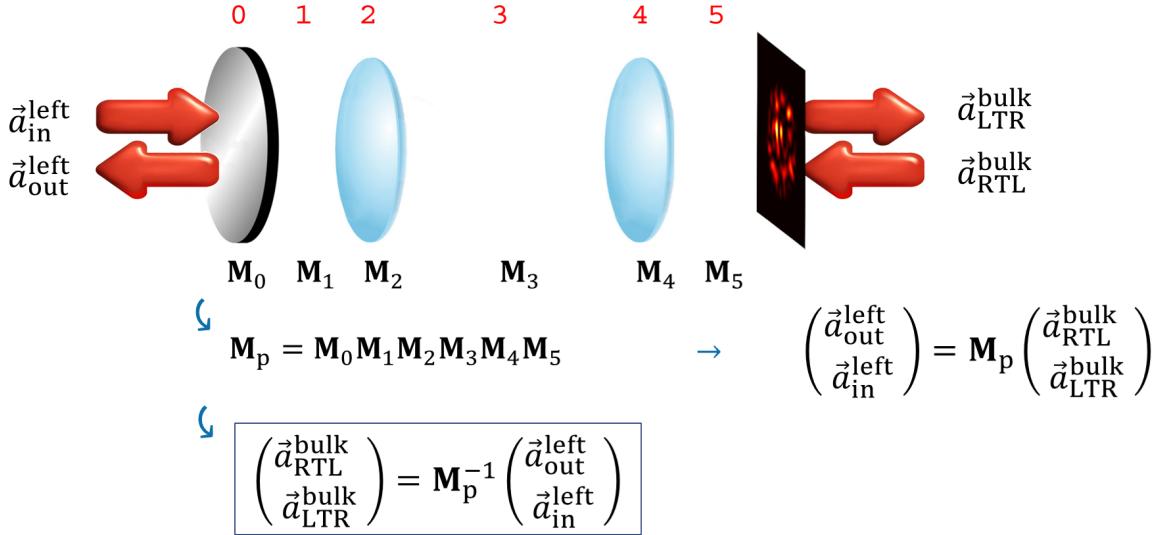


Figure 4.12: Intracavity reconstruction at an internal interface. Shown is the left part of the cavity up to (and including) component 5. The partial transfer matrix  $\mathbf{M}_p = \mathbf{M}_0 \mathbf{M}_1 \cdots \mathbf{M}_5$  maps the directional intracavity vectors at the interface ( $\mathbf{a}_{\text{RTL}}^{\text{bulk}}, \mathbf{a}_{\text{LTR}}^{\text{bulk}}$ ) to the exterior pair on the left ( $\mathbf{a}_{\text{out}}^{\text{left}}, \mathbf{a}_{\text{in}}^{\text{left}}$ ).

### Algorithm

#### 1. Form the partial transfer matrix.

Multiply the per-element transfer matrices from the left boundary up to the target interface:

$$\mathbf{M}_p = \mathbf{M}_0 \mathbf{M}_1 \mathbf{M}_2 \mathbf{M}_3 \mathbf{M}_4 \mathbf{M}_5. \quad (4.20)$$

#### 2. Use the transfer relation at that interface.

We know that by our convention the following relation is true:

$$\begin{pmatrix} \mathbf{a}_{\text{out}}^{\text{left}} \\ \mathbf{a}_{\text{in}}^{\text{left}} \end{pmatrix} = \mathbf{M}_p \begin{pmatrix} \mathbf{a}_{\text{RTL}}^{\text{bulk}} \\ \mathbf{a}_{\text{LTR}}^{\text{bulk}} \end{pmatrix}. \quad (4.21)$$

Therefore, we can easily determine the intracavity field vectors  $\mathbf{a}_{\text{RTL}}^{\text{bulk}}$  and  $\mathbf{a}_{\text{LTR}}^{\text{bulk}}$  by inverting  $\mathbf{M}_p$  and solving:

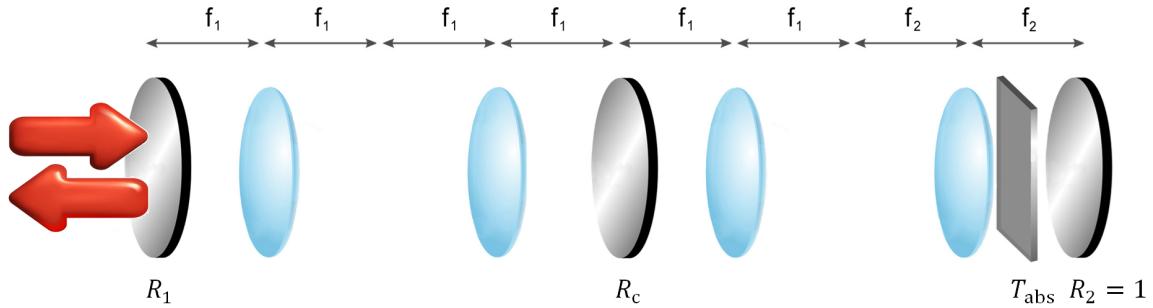
$$\begin{pmatrix} \mathbf{a}_{\text{RTL}}^{\text{bulk}} \\ \mathbf{a}_{\text{LTR}}^{\text{bulk}} \end{pmatrix} = \mathbf{M}_p^{-1} \begin{pmatrix} \mathbf{a}_{\text{out}}^{\text{left}} \\ \mathbf{a}_{\text{in}}^{\text{left}} \end{pmatrix}. \quad (4.22)$$

### From the right side

To reconstruct the field at the same interface “coming from the right,” build the partial matrix from the right boundary down to the interface and use the analogous transfer relation with  $(\mathbf{a}_{\text{in}}^{\text{right}}, \mathbf{a}_{\text{out}}^{\text{right}})^\top$ .

#### 4.3.4 Code Example: EP–MAD–CPA, Scattering-Matrix Method

In this example we simulate an *exceptional-point massively degenerate coherent perfect absorber* (EP–MAD–CPA) [1], shown in Figure 4.13.



$$R_1 = \sqrt{T_{\text{abs}}} \quad R_c = \frac{4R_1}{(1+R_1)^2} \quad f_2 = f_1 - d \left( n_r - \frac{1}{n_r} \right) \quad \lambda = \lambda_{\text{res}}$$

Figure 4.13: EP–MAD–CPA layout used in the example.

The device consists of two coupled EP–CPAs arranged in a  $4f$ – $4f$  chain with a central coupler. When the parameters satisfy the relations indicated below the figure (e.g.  $R_1 = \sqrt{T_{\text{abs}}}$ ,  $R_c = \frac{4R_1}{(1+R_1)^2}$ ,  $f_2 = f_1 - d \left( n_r - \frac{1}{n_r} \right)$ ,  $R_2 = 1$ ) and the wavelength is set to  $\lambda = \lambda_{\text{res}}$ , **any** left-incident field is critically coupled and fully absorbed (spatial degeneracy). Moreover, operating at the exceptional point broadens the absorption versus wavelength compared to a standard MAD–CPA (spectral degeneracy). We will compute this behavior using the scattering-matrix algorithm introduced above.

### Listing.

The full EP–MAD–CPA example using the scattering-matrix approach is shown below; the source file is `examples/sample_005A.py`.

```

1 import math
2 from fo_cavity_sim import clsCavity1path, clsMirror, clsPropagation,
3     clsThinLens, clsSpeckleField, get_sample_vec
4
5 if __name__ == '__main__':
6     # --- SIMULATION PARAMETERS -----
7     R_left = 0.7                      # right mirror
8     R_right = 0.999                    # left mirror
9     R_center = 4*R_left/((1+R_left)**2) # center mirror
10    d_absorb = 0.0006                 # thickness of absorber
11    T_abs = math.sqrt(R_left/R_right)  # optimal absorption
12    pos_absorb = 0.005                # distance absorber after left mirror
13    nr = 1.5                         # absorber's refractive index
14    f1 = 0.015                       # focal length first lens
15    f2 = f1 - d_absorb/2*(nr-1/nr)   # focal length second lens
16    length_fov = 0.00081             # field-of-view sidelength
17
18    # --- CREATE CAVITY -----
19    myCavity = clsCavity1path("8f_cavity")
20    myCavity.use_swap_files_in_bmatrix_class = True
21    myCavity.folder = "cavity_folder"
22    myCavity.tmp_folder = "tmp_folder"
23    myCavity.Lambda_nm = 633
24
```

```

25 # --- DETERMINE RESONANCE WAVELENGTH, CREATE GRID -----
26 lambda_c, k_c, n_c, lambda_period, l2_corr = \
27     myCavity.resonance_data_8f_cavity(R_left, R_center, R_right, f1)
28 l2_corr *= 1.212
29 myCavity.Lambda = lambda_c
30 myCavity.grid.set_opt_res_based_on_sidelength(length_fov, 2, 2*f1, True)
31 print(f"Field-of-view: {myCavity.grid.res_fov}x{myCavity.grid.res_fov}")
32 print(f"Total grid res: {myCavity.grid.res_tot}x{myCavity.grid.res_tot}")
33
34 # --- GENERATE COMPONENTS -----
35 # input coupling mirror
36 mirror1 = clsMirror("left mirror", myCavity)
37 mirror1.R = R_left
38
39 # f1 propagation
40 prop_f1 = clsPropagation("f1 propagation", myCavity)
41 prop_f1.set_params(f1, 1)
42
43 # lens f1
44 lens_f1 = clsThinLens("lens 1", myCavity)
45 lens_f1.f = f1
46
47 # center mirror
48 center_mirror = clsMirror("center mirror", myCavity)
49 center_mirror.R = R_center
50
51 # f2 propagation
52 prop_f2 = clsPropagation("f2 propagation", myCavity)
53 prop_f2.set_params(f2, 1)
54
55 # lens f2
56 lens_f2 = clsThinLens("lens 2", myCavity)
57 lens_f2.f = f2
58
59 # propagation between second lens and absorber
60 prop_lens_absorb = clsPropagation("propagation lens<>absorber", myCavity)
61 prop_lens_absorb.set_params(f2-pos_absorb-d_absorb/nr, 1)
62
63 # absorber
64 absorber = clsPropagation("absorber", myCavity)
65 absorber.set_params(d_absorb, nr)
66 absorber.set_ni_based_on_T(T_abs)
67
68 # propagation between absorber and right mirror
69 prop_absorb_mirr = clsPropagation("propagation absorber<>mirror2", myCavity)
70 prop_absorb_mirr.set_params(pos_absorb + l2_corr, 1)
71
72 # right mirror
73 mirror2 = clsMirror("right mirror", myCavity)
74 mirror2.R = R_right
75
76 # --- ASSEMBLE CAVITY -----
77 myCavity.add_component(mirror1)           # 0 *****
78 myCavity.add_component(prop_f1)          # 1
79 myCavity.add_component(lens_f1)          # 2
80 myCavity.add_component(prop_f1)          # 3
81 myCavity.add_component(prop_f1)          # 4
82 myCavity.add_component(lens_f1)          # 5
83 myCavity.add_component(prop_f1)          # 6

```

```

84 myCavity.add_component(center_mirror)      # 7 *****
85 myCavity.add_component(prop_f1)           # 8
86 myCavity.add_component(lens_f1)           # 9
87 myCavity.add_component(prop_f1)           # 10
88 myCavity.add_component(prop_f2)           # 11
89 myCavity.add_component(lens_f2)           # 12
90 myCavity.add_component(prop_lens_absorb)  # 13
91 myCavity.add_component(absorber)          # 14
92 myCavity.add_component(prop_absorb_mirr)  # 15
93 myCavity.add_component(mirror2)          # 16 *****
94
95 # --- INPUT: RANDOM SPECKLE FIELD -----
96 inp = clsSpeckleField(myCavity.grid)
97 inp.create_field_eq_distr(100, 20, 0.6*myCavity.grid.length_fov, 0)
98 inp.name = "Input Field"
99 inp.plot_field(5)
100 myCavity.incident_field_left = inp
101
102 # --- CALCULATE LEFT OUTPUT AND BULK FIELD FOR 3 DIFFERENT WAVELENGTHS ---
103 lambda_vec = get_sample_vec(3, lambda_period/120, 0, False, 1)
104 for index, dLambda in enumerate(lambda_vec):
105     # SET CURRENT WAVELENGTH
106     dL_pm = dLambda * 10**12 # delta lambda in pm
107     print("")
108     print(f"***  delta lambda = {dL_pm:.3f} pm   ***")
109     myCavity.Lambda = lambda_c + dLambda
110
111     # OUTPUT FIELD LEFT
112     out = myCavity.output_field_left
113     out.name = f"LEFT OUTPUT dL = {dL_pm:.3f} pm"
114     out.plot_field(5, vmax_limit=0.001, c_map = "custom")
115
116     # BULK FIELD LEFT OF CENTRAL MIRROR
117     myCavity.calc_bulk_field_from_left(6)
118     bulk = myCavity.bulk_field
119     bulk.name = f"BULK, DL = {dL_pm:.3f} pm"
120     bulk.plot_field(5)
121
122     # CALCULATE REFLECTION COEFFICIENT OVER FOV AND SAVE RESULT TO FILE
123     R = out.intensity_integral_fov() / inp.intensity_integral_fov()
124     print(f"Reflectivity: {R}")
125     data = [index, dL_pm, R]
126     myCavity.write_to_file("result.csv", data)

```

### Import Statements and main-program guard (lines 1–5)

As in the previous examples, lines 1–5 import all required classes and include the main-program guard `if __name__ == '__main__':` to ensure the library's parallel processing behaves reliably across platforms.

### Simulation parameters (lines 6–16)

This block sets the EP–MAD–CPA simulation parameters. In particular, the center-mirror reflectivity  $R_c$  and the absorber transmissivity  $T_{\text{abs}}$  are chosen to satisfy the EP–CPA resonance conditions; the second focal length  $f_2$  is corrected for the absorber slab ; and, to keep memory and runtime moderate, a smaller field of view than before is used ( $L_{\text{FOV}} = 0.81 \text{ mm}$ ).

### Create cavity (lines 18–23)

As in the previous examples, we create a `clsCavity1path` instance `myCavity`, enable on-disk caching, set the working folders, and choose 633 nm as the wavelength.

### Resonance condition and grid size (lines 25–32)

We call `myCavity.resonance_data_8f_cavity(...)` to obtain the resonance wavelength nearest to 633 nm and an estimate of the right-subcavity length correction  $\ell_{2,\text{corr}}$  (which is fine-tuned in line 28). With  $\lambda$  set to this resonance, the simulation grid is created via `myCavity.grid.set_opt_res_based_on_sidelength(...)`. For runtime speed reasons we intentionally use a propagation distance of a single  $f_1$  (rather than  $2f_1$ ) in the critical-sampling formula, yielding a coarser – yet sufficiently accurate – grid.

### Instantiate optical components (lines 34–74)

All required elements of the EP-MAD-CPA are created. Several of them (the  $f_1$  lens and propagation section, and the  $f_2$  lens and propagation section) are instantiated once and later reused multiple times when assembling the cavity.

### Assemble cavity (lines 76–93)

The components are placed on `myCavity` (the “breadboard”) in strict left-to-right order to realize the EP-MAD-CPA layout shown in Figure 4.13. Reused elements (`prop_f1`, `lens_f1`, `prop_f2`, `lens_f2`) appear multiple times. The calls to `myCavity.add_component(component)` merely register the elements; no optics is computed at this stage.

### Define incident field (lines 95–100)

An instance of `clsSpeckleField` (a convenience subclass of `clsLightField`) is created, configured to generate a random speckle pattern, and plotted for reference with `.plot_field(...)`. The resulting field is assigned to the left input port through `myCavity.incident_field_left`; no right-incident field is specified, so the right drive remains zero.

### Sweep over wavelength detunings (lines 102–109)

A short vector of three detunings  $\Delta\lambda$  – symmetrically distributed around zero – is generated with `get_sample_vec(...)`. The loop then converts each  $\Delta\lambda$  to picometres for logging and sets the working wavelength for that iteration to  $\lambda = \lambda_c + \Delta\lambda$  via `myCavity.Lambda`. Subsequent computations (output and bulk fields) are performed for each detuning in turn.

### Left output field (lines 111–114)

For each detuning, simply accessing `myCavity.output_field_left` triggers the computation and caching of the steady-state field leaving the cavity on the left at the current wavelength. The returned `clsLightField` is stored in `out`, given a descriptive name, and plotted via `out.plot_field(...)`.

### Intracavity (bulk) field left of the central mirror (lines 116–120)

Calling `myCavity.calc_bulk_field_from_left(...)` with parameter (6) computes the steady-state field *immediately to the right of component 6* – i.e. just left of the central mirror (component 7) – using the previously determined incident/output fields. The coherent superposition of the left-to-right and right-to-left bulk fields is then obtained via `myCavity.bulk_field`, assigned a descriptive name, and visualised with `bulk.plot_field(...)`.

### Compute and record reflectivity (lines 122–126)

As in the previous examples, the cavity's FOV reflectivity for the current detuning is obtained as the ratio of the output and input FOV-integrated intensities, `out.intensity_integral_fov(...)` / `inp.intensity_integral_fov(...)`. The value is printed and appended (together with the loop index and  $\Delta\lambda$  in pm) to `result.csv` via `myCavity.write_to_file(...)`. This accumulates one row per wavelength setting for later inspection or plotting.

#### 4.3.5 Simulation Results

The software creates the following console output:

```

Field-of-view: 70x70
Total grid res: 138x138

*** delta lambda = -0.028 pm ***
calculating cavity's reflection matrix R_L from cavity's transfer matrix M
calculating cavity's transfer matrix M

processing component 0: left mirror
    calculating transfer matrix M
    processing transfer matrix M

processing component 1: f propagation
    component will be used again: activating memory caching for component
    calculating transfer matrix M
        deleting transmission matrix T of 'f propagation' from memory cache
    processing transfer matrix M

processing component 2: lens 1
    component will be used again: activating memory caching for component
    calculating transfer matrix M
        calculating transmission matrix T
        done (29.7 seconds)
        deleting lens phase mask of 'lens 1' from memory cache
        deleting transmission matrix T of 'lens 1' from memory cache
        converting S-matrix to M-matrix
            20.0% done (step 1 of 5, 83.6 seconds)
            80.0% done (step 4 of 5, 8.0 seconds)
            100.0% done (step 5 of 5, 4.0 seconds)
        done (99.6 seconds)
    done (147.9 seconds)
processing transfer matrix M
    performing matrix multiplication
        25.0% done (step 2 of 8)
        50.0% done (step 4 of 8)
        62.5% done (step 5 of 8)
        100.0% done (step 8 of 8)
    done (59.5 seconds)
done (59.6 seconds)
done (207.4 seconds)

processing component 3: f propagation
    component will be used again: activating memory caching for component
    processing transfer matrix M
        performing matrix multiplication
            12.5% done (step 1 of 8)
            25.0% done (step 2 of 8)
            50.0% done (step 4 of 8)
            62.5% done (step 5 of 8)
            75.0% done (step 6 of 8)
            100.0% done (step 8 of 8)
        done (94.6 seconds)
    done (97.3 seconds)
done (97.3 seconds)

processing component 4: f propagation
    component will be used again: activating memory caching for component
    processing transfer matrix M
        performing matrix multiplication
            12.5% done (step 1 of 8)
            25.0% done (step 2 of 8)
            50.0% done (step 4 of 8)
            75.0% done (step 6 of 8)
            100.0% done (step 8 of 8)
        done (72.4 seconds)
    done (74.8 seconds)
done (74.8 seconds)

processing component 5: lens 1
    component will be used again: activating memory caching for component
    processing transfer matrix M

```

```

performing matrix multiplication
  12.5% done (step 1 of 8)
  25.0% done (step 2 of 8)
  50.0% done (step 4 of 8)
  62.5% done (step 5 of 8)
  75.0% done (step 6 of 8)
  100.0% done (step 8 of 8)
done (260.8 seconds)
done (263.5 seconds)
done (263.5 seconds)

processing component 6: f propagation
  component will be used again: activating memory caching for component
  processing transfer matrix M
    performing matrix multiplication
      12.5% done (step 1 of 8)
      25.0% done (step 2 of 8)
      50.0% done (step 4 of 8)
      62.5% done (step 5 of 8)
      75.0% done (step 6 of 8)
      100.0% done (step 8 of 8)
    done (91.7 seconds)
done (94.3 seconds)
done (94.3 seconds)

processing component 7: center mirror
  calculating transfer matrix M
  processing transfer matrix M
    performing matrix multiplication
      25.0% done (step 2 of 8)
      50.0% done (step 4 of 8)
      75.0% done (step 6 of 8)
      100.0% done (step 8 of 8)
    done (105.9 seconds)
done (108.6 seconds)
done (109.0 seconds)

processing component 8: f propagation
  component will be used again: activating memory caching for component
  processing transfer matrix M
    performing matrix multiplication
      12.5% done (step 1 of 8)
      25.0% done (step 2 of 8)
      50.0% done (step 4 of 8)
      75.0% done (step 6 of 8)
      100.0% done (step 8 of 8)
    done (72.5 seconds)
done (74.9 seconds)
done (74.9 seconds)

processing component 9: lens 1
  component was used before and will not be used again: preparing memory cache to be deleted
  deleting transfer matrix M of 'lens 1' from memory cache
  processing transfer matrix M
    performing matrix multiplication
      12.5% done (step 1 of 8)
      25.0% done (step 2 of 8)
      50.0% done (step 4 of 8)
      62.5% done (step 5 of 8)
      75.0% done (step 6 of 8)
      100.0% done (step 8 of 8)
    done (263.2 seconds)
done (266.0 seconds)
done (266.0 seconds)

processing component 10: f propagation
  component was used before and will not be used again: preparing memory cache to be deleted
  deleting transfer matrix M of 'f propagation' from memory cache
  processing transfer matrix M
    performing matrix multiplication
      12.5% done (step 1 of 8)
      25.0% done (step 2 of 8)
      50.0% done (step 4 of 8)
      62.5% done (step 5 of 8)
      75.0% done (step 6 of 8)
      100.0% done (step 8 of 8)
    done (92.8 seconds)
done (95.5 seconds)
done (95.5 seconds)

processing component 11: f2 propagation
  calculating transfer matrix M
  deleting transmission matrix T of 'f2 propagation' from memory cache
  processing transfer matrix M
    performing matrix multiplication
      25.0% done (step 2 of 8)
      50.0% done (step 4 of 8)
      75.0% done (step 6 of 8)
      100.0% done (step 8 of 8)
    done (72.3 seconds)
done (75.0 seconds)
done (75.8 seconds)

```

```

processing component 12: lens 2
  calculating transfer matrix M
    calculating transmission matrix T
    done (29.7 seconds)
  deleting lens phase mask of 'lens 2' from memory cache
  deleting transmission matrix T of 'lens 2' from memory cache
  converting S-matrix to M-matrix
    20.0% done (step 1 of 5, 82.9 seconds)
    80.0% done (step 4 of 5, 8.0 seconds)
    100.0% done (step 5 of 5, 4.1 seconds)
  done (98.9 seconds)
done (147.1 seconds)
processing transfer matrix M
  performing matrix multiplication
    12.5% done (step 1 of 8)
    25.0% done (step 2 of 8)
    50.0% done (step 4 of 8)
    62.5% done (step 5 of 8)
    75.0% done (step 6 of 8)
    100.0% done (step 8 of 8)
  done (265.1 seconds)
done (267.8 seconds)
done (415.0 seconds)

processing component 13: propagation lens<>absorber
  calculating transfer matrix M
    deleting transmission matrix T of 'propagation lens<>absorber' from memory cache
  processing transfer matrix M
    performing matrix multiplication
      12.5% done (step 1 of 8)
      25.0% done (step 2 of 8)
      50.0% done (step 4 of 8)
      62.5% done (step 5 of 8)
      75.0% done (step 6 of 8)
      100.0% done (step 8 of 8)
    done (92.3 seconds)
done (95.0 seconds)
done (95.8 seconds)

processing component 14: absorber
  calculating transfer matrix M
    deleting transmission matrix T of 'absorber' from memory cache
  processing transfer matrix M
    performing matrix multiplication
      25.0% done (step 2 of 8)
      50.0% done (step 4 of 8)
      62.5% done (step 5 of 8)
      75.0% done (step 6 of 8)
      100.0% done (step 8 of 8)
    done (72.9 seconds)
done (75.5 seconds)
done (76.3 seconds)

processing component 15: propagation absorber<>mirror2
  calculating transfer matrix M
    deleting transmission matrix T of 'propagation absorber<>mirror2' from memory cache
  processing transfer matrix M
    performing matrix multiplication
      12.5% done (step 1 of 8)
      25.0% done (step 2 of 8)
      50.0% done (step 4 of 8)
      75.0% done (step 6 of 8)
      100.0% done (step 8 of 8)
    done (73.2 seconds)
done (76.0 seconds)
done (76.8 seconds)

processing component 16: right mirror
  calculating transfer matrix M
  processing transfer matrix M
    performing matrix multiplication
      12.5% done (step 1 of 8)
      25.0% done (step 2 of 8)
      50.0% done (step 4 of 8)
      75.0% done (step 6 of 8)
      100.0% done (step 8 of 8)
    done (105.4 seconds)
done (108.1 seconds)
done (108.5 seconds)
done (2133.5 seconds)

performing matrix division
  50.0% done (step 1 of 2, 75.0 seconds)
  100.0% done (step 2 of 2, 48.9 seconds)
done (123.9 seconds)
done (2257.4 seconds)
calculating bulk field right of component 6, coming from the left side
processing component 6: f propagation
  calculating inverse transfer matrix M
    calculating transfer matrix M
    deleting transmission matrix T of 'f propagation' from memory cache
processing component 5: lens 1
  calculating inverse transfer matrix M

```

```

calculating transfer matrix M
calculating transmission matrix T
done (31.7 seconds)
deleting lens phase mask of 'lens 1' from memory cache
deleting transmission matrix T of 'lens 1' from memory cache
converting S-matrix to M-matrix
  20.0% done (step 1 of 5, 84.2 seconds)
  80.0% done (step 4 of 5, 7.9 seconds)
  100.0% done (step 5 of 5, 3.9 seconds)
done (100.1 seconds)
done (149.9 seconds)
done (166.5 seconds)
done (204.5 seconds)
processing component 4: f propagation
  calculating inverse transfer matrix M
  calculating transfer matrix M
    deleting transmission matrix T of 'f propagation' from memory cache
done (48.7 seconds)
processing component 3: f propagation
  calculating inverse transfer matrix M
  calculating transfer matrix M
    deleting transmission matrix T of 'f propagation' from memory cache
done (37.9 seconds)
processing component 2: lens 1
  calculating inverse transfer matrix M
  calculating transfer matrix M
  calculating transmission matrix T
done (31.2 seconds)
deleting lens phase mask of 'lens 1' from memory cache
deleting transmission matrix T of 'lens 1' from memory cache
converting S-matrix to M-matrix
  20.0% done (step 1 of 5, 84.0 seconds)
  80.0% done (step 4 of 5, 8.1 seconds)
  100.0% done (step 5 of 5, 4.0 seconds)
done (100.0 seconds)
done (150.7 seconds)
done (167.5 seconds)
done (306.0 seconds)
processing component 1: f propagation
  calculating inverse transfer matrix M
  calculating transfer matrix M
    deleting transmission matrix T of 'f propagation' from memory cache
done (48.3 seconds)
processing component 0: left mirror
  calculating inverse transfer matrix M
  calculating transfer matrix M
done (59.7 seconds)
calculating right-to-left bulk field.
done (8.1 seconds)
calculating left-to-right bulk field.
done (8.0 seconds)
done (722.2 seconds)
Reflectivity: 0.0018769870150859773

*** delta lambda = 0.000 pm ***
calculating cavity's reflection matrix R_L from cavity's transfer matrix M
...

```

## Interpreting the console output

Before discussing the detailed progress messages, note that this scattering-/transfer-matrix run is substantially heavier than the earlier multiple-round-trips and geometric-series examples: it typically takes more wall-clock time and requires considerably more memory (we recommend at least 64 GB of RAM for this script at the stated grid size). The library then becomes deliberately “chatty” during long computations to keep you informed about progress. In the excerpt above (shown only for the first wavelength-detuning to save space), you see:

- **Per-component status:** lines such as “processing component 2: lens 1” indicate where the calculation is in the optical stack.
- **Matrix construction:** messages like “calculating transmission matrix T”, “converting S-matrix to M-matrix”, and “performing matrix multiplication” document the creation and cascading of per-element scattering/transformation matrices.
- **Caching behavior:** notices such as “component will be used again: activating memory caching” and “deleting ... from memory cache” show when caches are created or released to balance speed and memory usage.

- **Global assembly:** “calculating cavity’s transfer matrix  $M$ ” followed by “performing matrix division” corresponds to building the full  $\mathbf{M}_{\text{tot}}$  and extracting  $\mathbf{R}_L$  via the block-matrix relations.

Only the first of the three wavelength iterations is printed here; the second and third follow the same sequence of steps.

### Controlling verbosity

If you prefer fewer status lines, you can:

- suppress all progress messages by setting `myCavity.progress.silent=True`, or
- throttle updates by increasing `myCavity.progress.max_time_betw_tic_outputs`, which enforces a minimum number of seconds between consecutive progress outputs.

### Generated CSV File.

For each wavelength the script appends one line to `result.csv` with three comma-separated columns (index, detuning, reflectivity):

```
0, -0.02782531961676226, 0.0018769870150859773
1, 0.0, 3.986410952999991e-06
2, 0.02782531961676226, 0.0018772592095200897
```

### Plots.

The code first displays the intensity of the left-incident speckle field (Fig. 4.14).

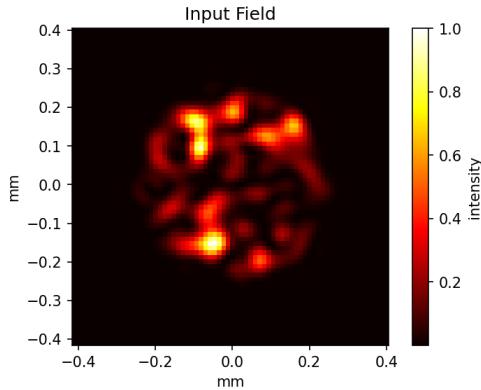


Figure 4.14: Incident speckle field injected from the left.

Subsequently, the reflected fields for the three detunings are plotted in each loop (line 114), as shown in Figure 4.15. As expected, at exact resonance the left output is numerically dark; a small detuning produces a weak, speckle-like residual.

Also, the intracavity fields immediately to the left of the central mirror are plotted in each loop (line 120), as shown in Figure 4.16. Note that even when the external reflection vanishes at exact resonance, the *intracavity* field remains finite and structured.

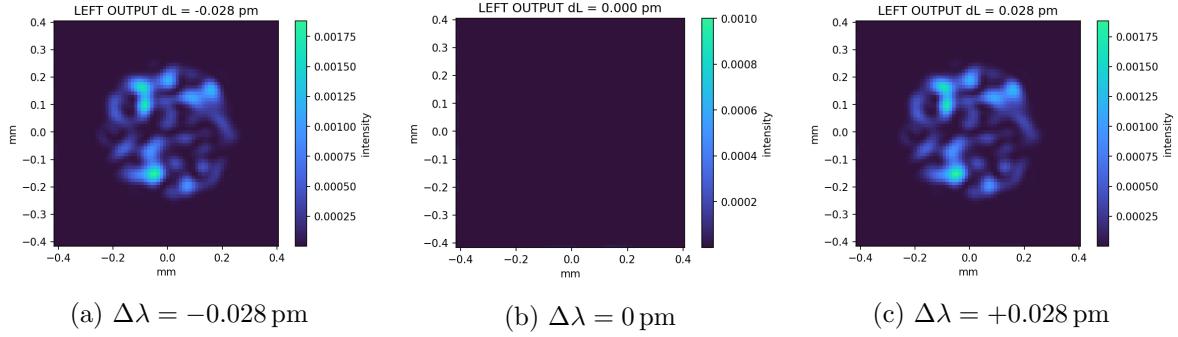


Figure 4.15: Left-output intensity plots for three wavelength detunings around the EP-MAD-CPA resonance. At exact resonance the cavity acts as a coherent perfect absorber and the reflected field vanishes.

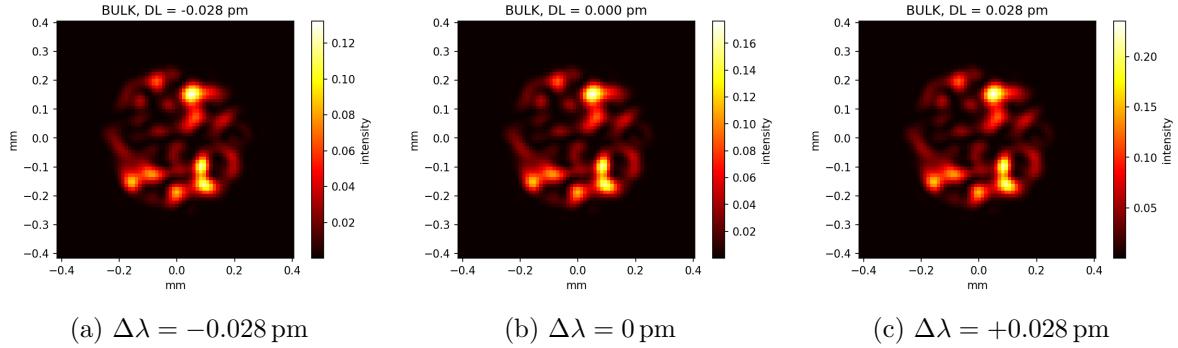


Figure 4.16: Intracavity intensity plots for three wavelength detunings around the EP-MAD-CPA resonance. The plots show the field immediately left of the central mirror.



# Chapter 5

## Ring Cavities

Before introducing the theory behind ring cavities, we first outline the setting. Unlike linear cavities, a ring cavity routes light around a closed loop with *two distinct paths* between the couplers (the two arms of the ring). In each arm light can propagate in *both* directions at the same time. The cavity may be driven from both couplers and from either side, yielding up to four independent driving (incident) fields and four corresponding outgoing fields. At each coupler the counter-propagating fields from the two paths mix. Figure 5.1 shows the notation used in this chapter.

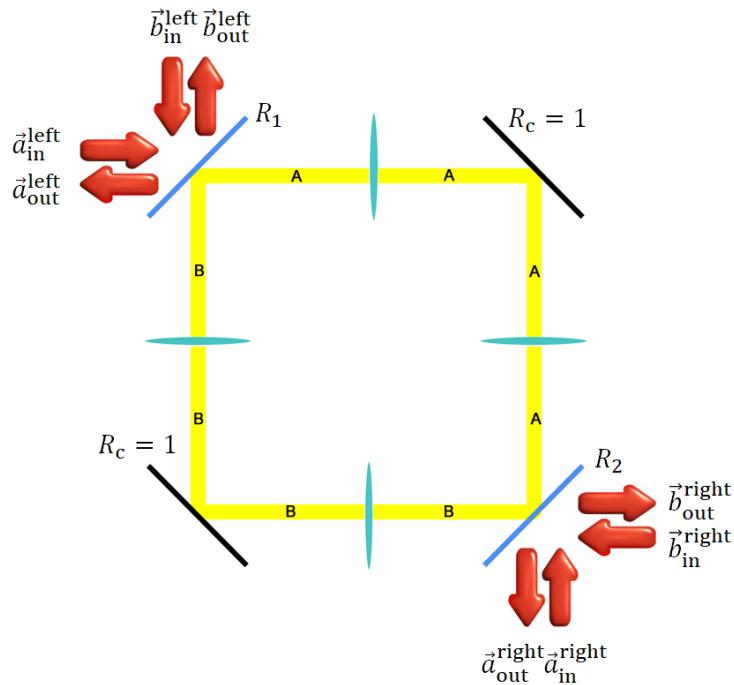


Figure 5.1: Ring cavity with two external couplers  $R_1$  and  $R_2$ . Arrows indicate the four possible driving and outgoing fields at the ports. Between the couplers the loop offers two paths (arms); fields may propagate along either path in both directions and are mixed at each coupler.

## 5.1 Theory

### 5.1.1 Four-Port Scattering Matrices

To simulate ring cavities (Fig. 5.1) we use scattering and transfer matrices for *four-port* components, see Fig. 5.2. Each component provides two physical paths (arms “A” and “B”) that can be accessed from the left and from the right, giving four ports in total. A standard example is a beam splitter (the couplers  $R_1$  and  $R_2$  in Fig. 5.1), where an incident field may either remain in its original path or be cross-coupled into the opposite path. In this library, such components are implemented as subclasses of `clsOptComponent4port`.

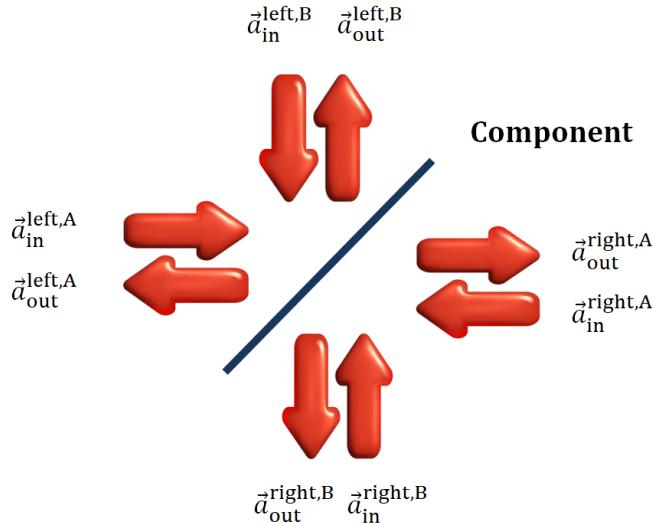


Figure 5.2: Conceptual view of a 4-port component. Each side supports two independent spatial paths (A and B). Consequently, four input amplitude vectors ( $\mathbf{a}_{in}^{left,A}, \mathbf{a}_{in}^{left,B}, \mathbf{a}_{in}^{right,A}, \mathbf{a}_{in}^{right,B}$ ) are mapped to four output vectors ( $\mathbf{a}_{out}^{left,A}, \mathbf{a}_{out}^{left,B}, \mathbf{a}_{out}^{right,A}, \mathbf{a}_{out}^{right,B}$ ) by a  $4 \times 4$  block matrix.

#### Block-matrix conventions

For a 2-port component the scattering relation is expressed by Eq. (4.11). In the 4-port case each side carries two paths, so every “vector” in Eq. (4.11) becomes a *vector of vectors*. Specifically we expand

$$\mathbf{a}_{in}^{left} = \begin{pmatrix} \mathbf{a}_{in}^{left,A} \\ \mathbf{a}_{in}^{left,B} \end{pmatrix}, \quad \mathbf{a}_{in}^{right} = \begin{pmatrix} \mathbf{a}_{in}^{right,A} \\ \mathbf{a}_{in}^{right,B} \end{pmatrix}, \quad \mathbf{a}_{out}^{left} = \begin{pmatrix} \mathbf{a}_{out}^{left,A} \\ \mathbf{a}_{out}^{left,B} \end{pmatrix}, \quad \mathbf{a}_{out}^{right} = \begin{pmatrix} \mathbf{a}_{out}^{right,A} \\ \mathbf{a}_{out}^{right,B} \end{pmatrix}. \quad (5.1)$$

With this notation the scattering equation becomes

$$\begin{pmatrix} \mathbf{a}_{out}^{left} \\ \mathbf{a}_{out}^{right} \end{pmatrix} = \mathbf{S} \begin{pmatrix} \mathbf{a}_{in}^{left} \\ \mathbf{a}_{in}^{right} \end{pmatrix}, \quad \mathbf{S} = \begin{pmatrix} \mathbf{R}_L & \mathbf{T}_{RTL} \\ \mathbf{T}_{LTR} & \mathbf{R}_R \end{pmatrix}, \quad (5.2)$$

where each block (e.g.  $\mathbf{R}_L$ ) is itself a  $2 \times 2$  block matrix that resolves the two paths:

$$\mathbf{R}_L = \begin{pmatrix} \mathbf{R}_L^{A \rightarrow A} & \mathbf{R}_L^{B \rightarrow A} \\ \mathbf{R}_L^{A \rightarrow B} & \mathbf{R}_L^{B \rightarrow B} \end{pmatrix}, \quad \mathbf{T}_{RTL} = \begin{pmatrix} \mathbf{T}_{RTL}^{A \rightarrow A} & \mathbf{T}_{RTL}^{B \rightarrow A} \\ \mathbf{T}_{RTL}^{A \rightarrow B} & \mathbf{T}_{RTL}^{B \rightarrow B} \end{pmatrix}, \quad \text{etc.} \quad (5.3)$$

### Sub-matrix naming scheme.

Every sub-matrix in  $\mathbf{S}$  carries (i) a side or direction tag and (ii) a path-to-path superscript, for example:

- $\mathbf{R}_L^{A \rightarrow B}$  – “reflect at the *left* interface, taking light that started in path  $A$  and sending it back out in path  $B$ .”
- $\mathbf{T}_{RTL}^{B \rightarrow A}$  – “transmit *right-to-left*, converting a field that came in on the right in path  $B$  into a field that exits on the left in path  $A$ .”

With this notation the full  $4 \times 4$  scattering relation reads

$$\begin{pmatrix} \mathbf{a}_{out}^{left,A} \\ \mathbf{a}_{out}^{left,B} \\ \mathbf{a}_{out}^{right,A} \\ \mathbf{a}_{out}^{right,B} \end{pmatrix} = \begin{pmatrix} \mathbf{R}_L^{A \rightarrow A} & \mathbf{R}_L^{B \rightarrow A} & \mathbf{T}_{RTL}^{A \rightarrow A} & \mathbf{T}_{RTL}^{B \rightarrow A} \\ \mathbf{R}_L^{A \rightarrow B} & \mathbf{R}_L^{B \rightarrow B} & \mathbf{T}_{RTL}^{A \rightarrow B} & \mathbf{T}_{RTL}^{B \rightarrow B} \\ \mathbf{T}_{LTR}^{A \rightarrow A} & \mathbf{T}_{LTR}^{B \rightarrow A} & \mathbf{R}_R^{A \rightarrow A} & \mathbf{R}_R^{B \rightarrow A} \\ \mathbf{T}_{LTR}^{A \rightarrow B} & \mathbf{T}_{LTR}^{B \rightarrow B} & \mathbf{R}_R^{A \rightarrow B} & \mathbf{R}_R^{B \rightarrow B} \end{pmatrix} \begin{pmatrix} \mathbf{a}_{in}^{left,A} \\ \mathbf{a}_{in}^{left,B} \\ \mathbf{a}_{in}^{right,A} \\ \mathbf{a}_{in}^{right,B} \end{pmatrix}. \quad (5.4)$$

Equation (5.4) makes explicit how every incoming amplitude vector couples to every outgoing one through the scattering block-matrix.

A concrete example of a four-port component is a beam splitter, such as  $R_1$  and  $R_2$  in Fig. 5.1. Here, reflection always swaps the two spatial paths  $A$  and  $B$ , while transmission preserves the path. Consequently, the full  $4 \times 4$  block scattering matrix contains only eight nonzero submatrices (each block is  $N^2 \times N^2$ ;  $0$  denotes the corresponding zero block):

$$\mathbf{S} = \begin{pmatrix} 0 & \mathbf{R}_L^{B \rightarrow A} & \mathbf{T}_{RTL}^{A \rightarrow A} & 0 \\ \mathbf{R}_L^{A \rightarrow B} & 0 & 0 & \mathbf{T}_{RTL}^{B \rightarrow B} \\ \mathbf{T}_{LTR}^{A \rightarrow A} & 0 & 0 & \mathbf{R}_R^{B \rightarrow A} \\ 0 & \mathbf{T}_{LTR}^{B \rightarrow B} & \mathbf{R}_R^{A \rightarrow B} & 0 \end{pmatrix} \quad (5.5)$$

In this library, such a beam splitter is provided by `clsBeamSplitterMirror`. By contrast, for the components labeled  $R_c$  in Fig. 5.1 the paths  $A$  and  $B$  do *not* mix: the light field is simply transmitted within the same path in either direction. This behavior can be modeled with `clsTransmissionMixer`, configured to expose the scattering matrix

$$\mathbf{S} = \begin{pmatrix} 0 & 0 & \mathbf{T}_{RTL}^{A \rightarrow A} & 0 \\ 0 & 0 & 0 & \mathbf{T}_{RTL}^{B \rightarrow B} \\ \mathbf{T}_{LTR}^{A \rightarrow A} & 0 & 0 & 0 \\ 0 & \mathbf{T}_{LTR}^{B \rightarrow B} & 0 & 0 \end{pmatrix} \quad (5.6)$$

*Remark.* In this four-port convention, what is *physically* a specular reflection at the fold mirrors  $R_c$  is represented by the *transmission* sub-blocks (e.g.  $\mathbf{T}_{LTR}^{A \rightarrow A}$ ,  $\mathbf{T}_{RTL}^{B \rightarrow B}$ ). The reason is purely bookkeeping: the field is routed from the left ports to the right ports while staying in the same geometric path, so it does not “return” to the same side. Consequently, the corresponding reflection blocks are zero, and the nonzero  $\mathbf{T}$ -blocks faithfully mimic the physical reflection behavior without path mixing.

### 5.1.2 Four-Port Transfer Matrices

The transfer-matrix convention is generalized analogously to the scattering matrices:

$$\begin{pmatrix} \mathbf{a}_{\text{out}}^{\text{left}} \\ \mathbf{a}_{\text{in}}^{\text{left}} \end{pmatrix} = \mathbf{M} \begin{pmatrix} \mathbf{a}_{\text{in}}^{\text{right}} \\ \mathbf{a}_{\text{out}}^{\text{right}} \end{pmatrix}. \quad (5.7)$$

The  $4 \times 4$  *transfer* relation is written in the same expanded form:

$$\begin{pmatrix} \mathbf{a}_{\text{out}}^{\text{left},A} \\ \mathbf{a}_{\text{out}}^{\text{left},B} \\ \mathbf{a}_{\text{in}}^{\text{left},A} \\ \mathbf{a}_{\text{in}}^{\text{left},B} \end{pmatrix} = \begin{pmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} & \mathbf{M}_{13} & \mathbf{M}_{14} \\ \mathbf{M}_{21} & \mathbf{M}_{22} & \mathbf{M}_{23} & \mathbf{M}_{24} \\ \mathbf{M}_{31} & \mathbf{M}_{32} & \mathbf{M}_{33} & \mathbf{M}_{34} \\ \mathbf{M}_{41} & \mathbf{M}_{42} & \mathbf{M}_{43} & \mathbf{M}_{44} \end{pmatrix} \begin{pmatrix} \mathbf{a}_{\text{in}}^{\text{right},A} \\ \mathbf{a}_{\text{in}}^{\text{right},B} \\ \mathbf{a}_{\text{out}}^{\text{right},A} \\ \mathbf{a}_{\text{out}}^{\text{right},B} \end{pmatrix}. \quad (5.8)$$

Here each  $\mathbf{M}_{ij}$  is itself a square matrix of size  $n \times n$  that captures the channel-resolved coupling between the corresponding input and output amplitude vectors.

### 5.1.3 Assembling a ring cavity from left to right

A convenient way to “build” a ring cavity is to rotate the layout so that all ports lie horizontally and then add components from left to right in the order in which the fields encounter them. We start with the left beam-splitter mirror  $R_1$ , a genuine four-port element that mixes the two spatial paths  $A$  and  $B$  upon reflection while leaving them unchanged upon transmission. Its scattering matrix is denoted  $\mathbf{S}_0$  (see Figure 5.3).

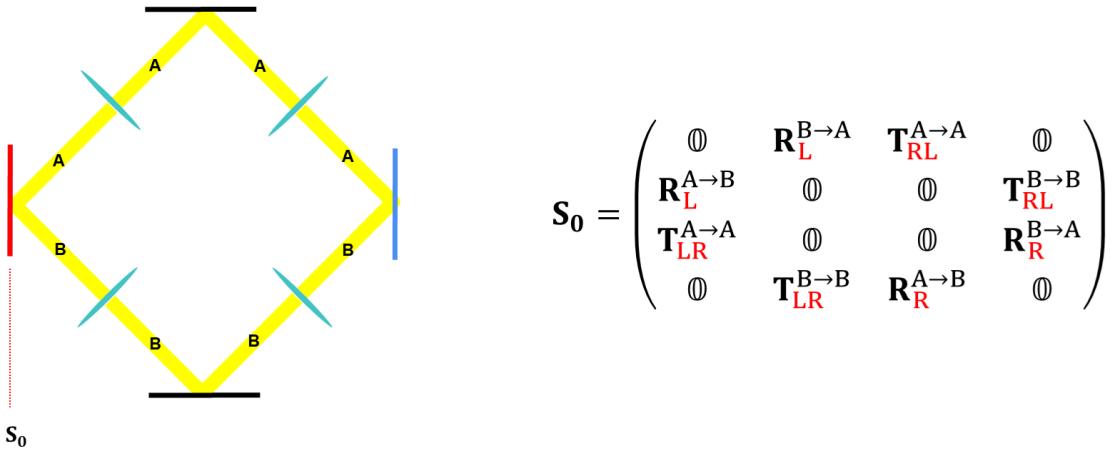


Figure 5.3: Start of the assembly: the left beam-splitter mirror  $R_1$  is represented by the four-port scattering matrix  $\mathbf{S}_0$ . Reflection swaps the paths ( $A \leftrightarrow B$ ), transmission keeps them ( $A \rightarrow A$ ,  $B \rightarrow B$ ).

Next come the two parallel segments (one in path  $A$ , one in path  $B$ ) consisting of free-space sections and, later, lenses. Each such *two-port* element is wrapped into a four-port representation by placing its transmission matrix into the appropriate block(s) of a  $4 \times 4$  scattering matrix, while all other blocks are zero. This yields  $\mathbf{S}_1$  as sketched in Figure 5.4. In code this wrapping is handled by `clsOptComponentAdapter`.

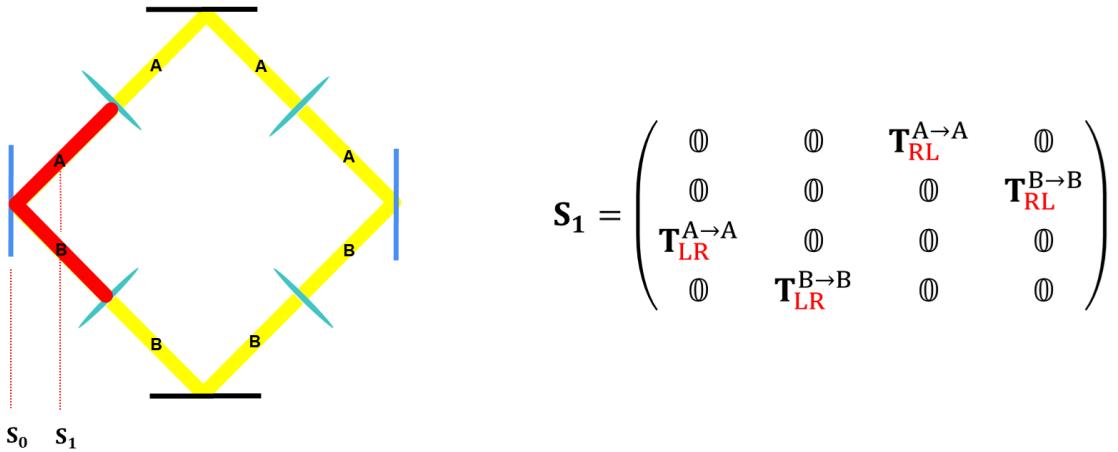


Figure 5.4: Adding the parallel path segments after  $R_1$ : the resulting scattering matrix  $\mathbf{S}_1$  contains only the nonzero transmission blocks that carry  $A \rightarrow A$  and  $B \rightarrow B$  across the segment; all mixing blocks are zero.

Proceed in the same manner for all subsequent two-port components (free-space sections, lenses, absorbers) in each arm: place their  $A \rightarrow A$  and  $B \rightarrow B$  transmission and reflection blocks into the four-port scattering matrix at the correct positions. The fold mirrors  $R_c$  are true four-port components without path mixing and are modeled by `clsTransmissionMixer`. The right beam-splitter mirror is again a `clsBeamSplitterMirror`.

#### 5.1.4 Algorithm for calculating left and right output fields

As with linear cavities (Sec. 4.3.2), the ring-cavity workflow is based on scattering and transfer matrices; the only difference is that each component now has four ports (two paths, each accessible from left and right).

1. **Build component scattering matrices.** For each four-port element (or a group of two-port elements in arm  $A$  and/or arm  $B$ ), assemble a  $4 \times 4$  scattering matrix  $\mathbf{S}_i$ . When  $\mathbf{S}_i$  is constructed from two-port elements, the sections in arms  $A$  and  $B$  need not be at the same physical position within the cavity (their path lengths may differ); they only have to be inserted in the correct left-to-right order of the component sequence. You can also add an element to just one arm; the unused arm then behaves as a neutral element, i.e. its transmission blocks  $\mathbf{T}_{\text{LTR}}$  and  $\mathbf{T}_{\text{RTL}}$  are identity matrices and its reflection blocks  $\mathbf{R}_L$  and  $\mathbf{R}_R$  are zero matrices.
2. **Convert to transfer matrices.** Convert each  $\mathbf{S}_i$  to the corresponding transfer matrix  $\mathbf{M}_i$  using the four-port generalisation of Eq. (4.14).
3. **Cascade in spatial order.** Form the cavity transfer matrix

$$\mathbf{M}_{\text{cav}} = \mathbf{M}_0 \mathbf{M}_1 \cdots \mathbf{M}_N.$$

4. **Back-convert to a global scattering matrix.** Obtain the overall  $4 \times 4$  scattering matrix  $\mathbf{S}_{\text{cav}}$  via the four-port generalization of Eq. (4.16).
5. **Extract blocks and compute outputs.** Interpret the  $4 \times 4$  scattering block matrix as a  $2 \times 2$  block matrix, where each block is itself a  $2 \times 2$  block matrix  $\mathbf{R}_L$ ,  $\mathbf{R}_R$ ,  $\mathbf{T}_{\text{LTR}}$ ,

and  $\mathbf{T}_{\text{RTL}}$  (see 5.1). Apply these to the incident fields to obtain the desired outputs:

$$\mathbf{a}_{\text{out}}^{\text{L,A}} = \mathbf{R}_L^{(1,1)} \mathbf{a}_{\text{in}}^{\text{L,A}} + \mathbf{R}_L^{(1,2)} \mathbf{a}_{\text{in}}^{\text{L,B}} + \mathbf{T}_{\text{RTL}}^{(1,1)} \mathbf{a}_{\text{in}}^{\text{R,A}} + \mathbf{T}_{\text{RTL}}^{(1,2)} \mathbf{a}_{\text{in}}^{\text{R,B}}$$

$$\mathbf{a}_{\text{out}}^{\text{L,B}} = \mathbf{R}_L^{(2,1)} \mathbf{a}_{\text{in}}^{\text{L,A}} + \mathbf{R}_L^{(2,2)} \mathbf{a}_{\text{in}}^{\text{L,B}} + \mathbf{T}_{\text{RTL}}^{(2,1)} \mathbf{a}_{\text{in}}^{\text{R,A}} + \mathbf{T}_{\text{RTL}}^{(2,2)} \mathbf{a}_{\text{in}}^{\text{R,B}}$$

$$\mathbf{a}_{\text{out}}^{\text{R,A}} = \mathbf{T}_{\text{LTR}}^{(1,1)} \mathbf{a}_{\text{in}}^{\text{L,A}} + \mathbf{T}_{\text{LTR}}^{(1,2)} \mathbf{a}_{\text{in}}^{\text{L,B}} + \mathbf{R}_R^{(1,1)} \mathbf{a}_{\text{in}}^{\text{R,A}} + \mathbf{R}_R^{(1,2)} \mathbf{a}_{\text{in}}^{\text{R,B}}$$

$$\mathbf{a}_{\text{out}}^{\text{R,B}} = \mathbf{T}_{\text{LTR}}^{(2,1)} \mathbf{a}_{\text{in}}^{\text{L,A}} + \mathbf{T}_{\text{LTR}}^{(2,2)} \mathbf{a}_{\text{in}}^{\text{L,B}} + \mathbf{R}_R^{(2,1)} \mathbf{a}_{\text{in}}^{\text{R,A}} + \mathbf{R}_R^{(2,2)} \mathbf{a}_{\text{in}}^{\text{R,B}}$$

Block indices such as  $\mathbf{R}_L^{(i,j)}$  refer to the  $i$ -th row and  $j$ -th column submatrix within the  $2 \times 2$  block. The vectors  $\mathbf{a}_{\text{in}}^{\text{L,A}}$ ,  $\mathbf{a}_{\text{in}}^{\text{L,B}}$ ,  $\mathbf{a}_{\text{in}}^{\text{R,A}}$ , and  $\mathbf{a}_{\text{in}}^{\text{R,B}}$  collect the Fourier coefficients of the incident fields for the indicated side and path; the corresponding outputs are  $\mathbf{a}_{\text{out}}^{\text{L,A}}$ ,  $\mathbf{a}_{\text{out}}^{\text{L,B}}$ ,  $\mathbf{a}_{\text{out}}^{\text{R,A}}$ , and  $\mathbf{a}_{\text{out}}^{\text{R,B}}$ .

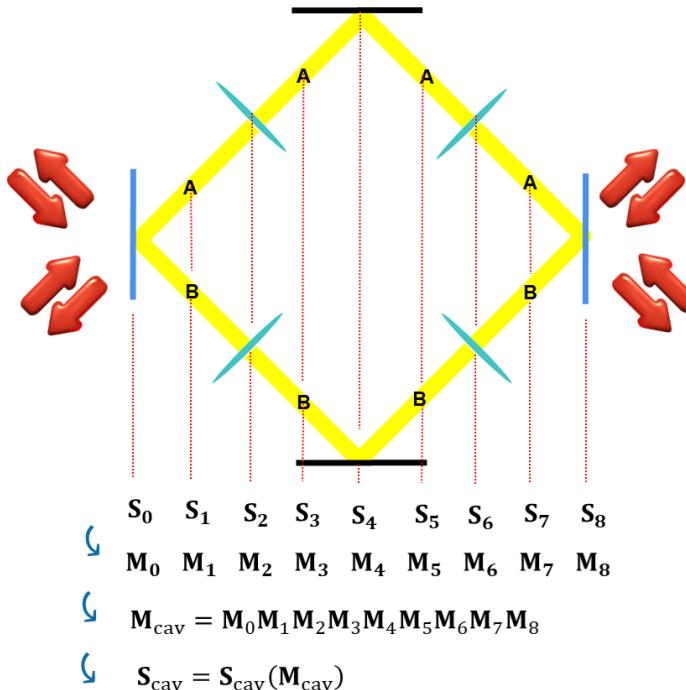


Figure 5.5: Scattering/transfer-matrix workflow for ring cavities with two paths (A,B). Each four-port component (or grouped two-port section) is represented by a  $4 \times 4$  scattering matrix  $\mathbf{S}_i$ , converted to a transfer matrix  $\mathbf{M}_i$ , cascaded, and finally back-converted to obtain the global reflection/transmission blocks.

### 5.1.5 Algorithm for calculating intracavity fields

In extension of the algorithm for linear cavities (Sec. 4.3.3), it is also possible to calculate the intracavity fields immediately right of index position  $i$  in arm  $A$  and  $B$  with the following algorithm:

1. **Inverse transfer matrix stack.** Calculate the following cumulative product of inverse transfer matrices:

$$\mathbf{M}_{\Sigma}^{-1} = \mathbf{M}_i^{-1} \mathbf{M}_{i-1}^{-1} \dots \mathbf{M}_1^{-1},$$

where each  $\mathbf{M}_k^{-1}$  is the inverse  $4 \times 4$  transfer block matrix of component  $k$ . This product aggregates the inverse transfer matrices from the left boundary up to the interface immediately right of component  $i$ .

- 2. Field reconstruction.** The block matrix  $\mathbf{M}_{\Sigma}^{-1}$  consists of sixteen sub-blocks  $\mathbf{M}_{(p,q)}^{-1}$  ( $p, q \in \{1, 2, 3, 4\}$ ). Applying the first-row blocks to the left output vectors and the second-row blocks to the left incident vectors yields the RTL contributions; the third and fourth rows analogously yield the LTR contributions for paths A and B:

$$\begin{aligned}\mathbf{a}_{\text{RTL}}^A &= \mathbf{M}_{(1,1)}^{-1} \mathbf{a}_{\text{out}}^{\text{L},A} + \mathbf{M}_{(1,2)}^{-1} \mathbf{a}_{\text{out}}^{\text{L},B} + \mathbf{M}_{(1,3)}^{-1} \mathbf{a}_{\text{in}}^{\text{L},A} + \mathbf{M}_{(1,4)}^{-1} \mathbf{a}_{\text{in}}^{\text{L},B} \\ \mathbf{a}_{\text{RTL}}^B &= \mathbf{M}_{(2,1)}^{-1} \mathbf{a}_{\text{out}}^{\text{L},A} + \mathbf{M}_{(2,2)}^{-1} \mathbf{a}_{\text{out}}^{\text{L},B} + \mathbf{M}_{(2,3)}^{-1} \mathbf{a}_{\text{in}}^{\text{L},A} + \mathbf{M}_{(2,4)}^{-1} \mathbf{a}_{\text{in}}^{\text{L},B} \\ \mathbf{a}_{\text{LTR}}^A &= \mathbf{M}_{(3,1)}^{-1} \mathbf{a}_{\text{out}}^{\text{L},A} + \mathbf{M}_{(3,2)}^{-1} \mathbf{a}_{\text{out}}^{\text{L},B} + \mathbf{M}_{(3,3)}^{-1} \mathbf{a}_{\text{in}}^{\text{L},A} + \mathbf{M}_{(3,4)}^{-1} \mathbf{a}_{\text{in}}^{\text{L},B} \\ \mathbf{a}_{\text{LTR}}^B &= \mathbf{M}_{(4,1)}^{-1} \mathbf{a}_{\text{out}}^{\text{L},A} + \mathbf{M}_{(4,2)}^{-1} \mathbf{a}_{\text{out}}^{\text{L},B} + \mathbf{M}_{(4,3)}^{-1} \mathbf{a}_{\text{in}}^{\text{L},A} + \mathbf{M}_{(4,4)}^{-1} \mathbf{a}_{\text{in}}^{\text{L},B}.\end{aligned}$$

### 5.1.6 Code Example: Ring Cavity, Scattering-Matrix Method

In this code example we simulate the ring cavity shown in Figure 5.6.

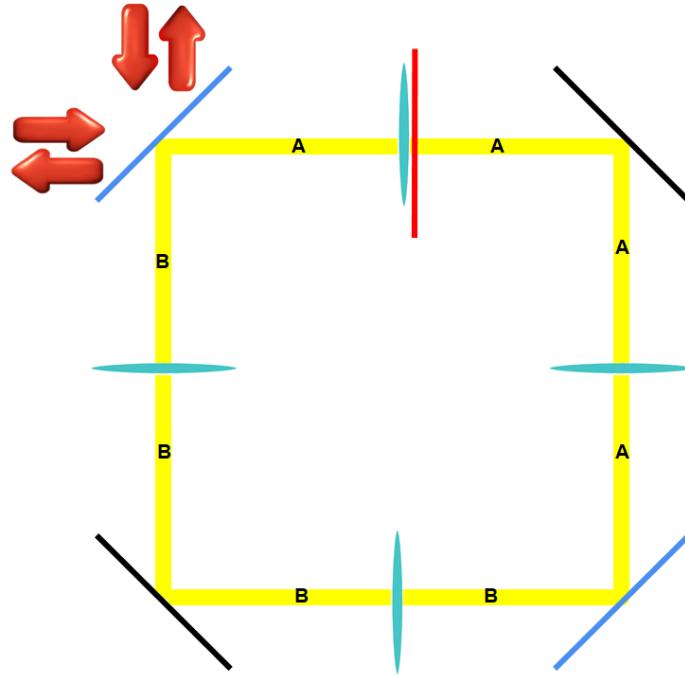


Figure 5.6: Simulated ring cavity. Top left: beamsplitter coupler; all other mirrors are perfectly reflecting corner mirrors ( $R_c = 1$ ). Each lens is placed one focal length  $f$  away from the neighboring mirror. The two intracavity paths are labelled A (top/right) and B (left/bottom). The red line marks the plane at which the intracavity field in path A is probed.

### Configuration.

Two driving fields are injected at the *top-left* beamsplitter: a horizontal input on path *A* and a vertical input on path *B*. All other mirrors are perfect reflectors. Each lens is placed one focal length  $f$  away from the neighboring mirror. In each full round-trip, the light field goes through an  $8f$ -configuration. Consequently, a field launched on the horizontal path *A* is imaged (upright) onto the *vertical* output port of the same coupler, and vice versa for a field launched on path *B*. In addition, we probe the intracavity field in path *A* at the plane indicated by the red line.

### Assembly order (left to right).

With the rotated layout in Fig. 5.7, we visualize the exact left-to-right build sequence used in the simulation:

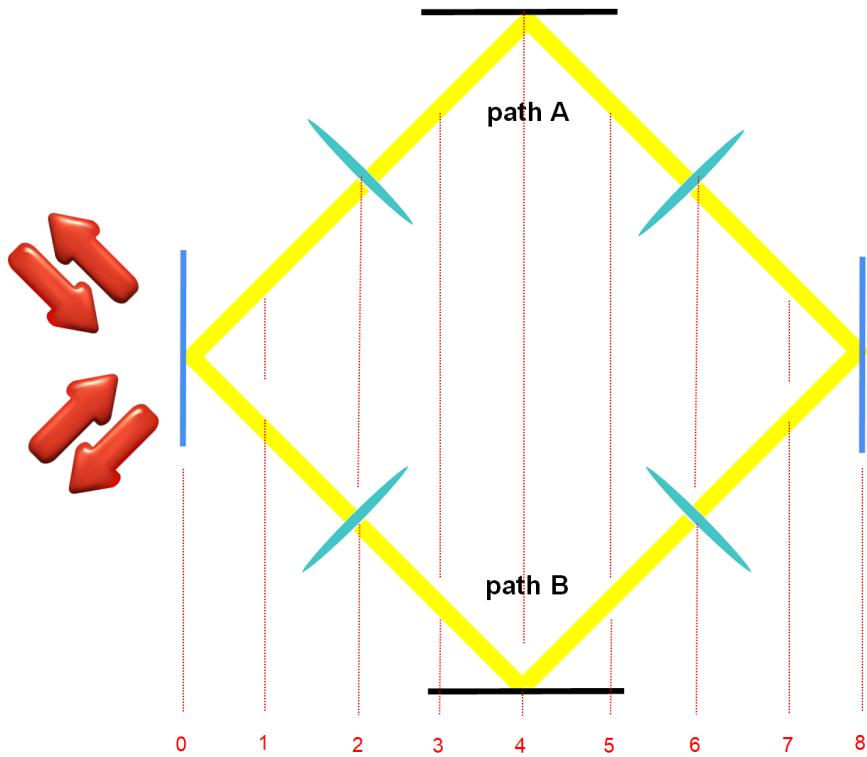


Figure 5.7: Ring cavity rotated to emphasize the left-to-right assembly order used in the simulation. The red dashed lines mark the insertion positions (0)–(8) of the successive four-port components.

- (0) **Beamsplitter mirror** (coupler at the left, four-port).
- (1) **Propagation in paths A and B** (grouped, four-port block).
- (2) **Lens in paths A and B** (grouped, four-port).
- (3) **Propagation in paths A and B** (grouped, four-port).
- (4) **Corner mirrors implemented with `clsTransmissionMixer`.**
- (5) **Propagation in paths A and B** (grouped, four-port).
- (6) **Lens in paths A and B** (grouped, four-port).
- (7) **Propagation in paths A and B** (grouped, four-port).
- (8) **Beamsplitter mirror** (coupler at the right, four-port).

**Listing.**

The full ring-cavity example using the scattering-matrix approach is shown below; the source file is `examples/sample_006A.py`.

```

1  from fo_cavity_sim import (
2      clsCavity2path, clsBeamSplitterMirror,
3      clsPropagation, clsThinLens, clsTransmissionMixer,
4      clsSpeckleField, clsTestImage, Side, Path
5  )
6
7  if __name__ == '__main__':
8      # --- SIMULATION PARAMETERS -----
9      R_left = 0.7          # left mirror
10     R_right = 0.999       # right mirror
11     f = 0.05             # focal length of lenses
12     length_fov = 0.002   # field-of-view width
13
14     # --- GENERATE CAVITY -----
15     myCavity = clsCavity2path("ring cavity")
16     myCavity.folder = "cavity_folder"
17     myCavity.tmp_folder = "tmp_folder"
18     myCavity.Lambda_nm = 633
19     myCavity.use_swap_files_in_bmatrix_class = True
20
21     # --- GENERATE GRID -----
22     myCavity.grid.set_opt_res_based_on_sidelength(length_fov, 1.5, 2 * f, True)
23     print(f"Field-of-view: {myCavity.grid.res_fov}x{myCavity.grid.res_fov}")
24     print(f"Total grid res: {myCavity.grid.res_tot}x{myCavity.grid.res_tot}")
25
26     # --- GENERATE COMPONENTS -----
27     # left input coupling mirror
28     mirror1 = clsBeamSplitterMirror("left mirror", myCavity)
29     mirror1.R = R_left
30
31     # f propagation
32     prop_f = clsPropagation("f propagation", myCavity)
33     prop_f.set_params(f, 1)
34
35     # lens
36     lens_f = clsThinLens("lens", myCavity)
37     lens_f.f = f
38
39     # top-right and bottom-left mirrors (no path mixing; pure "corner" mirrors)
40     corner_mirrors = clsTransmissionMixer("corner mirrors", myCavity)
41     corner_mirrors.T_same = 1
42     corner_mirrors.refl_behavior_for_path_mixing = False
43
44     # right mirror
45     mirror2 = clsBeamSplitterMirror("right mirror", myCavity)
46     mirror2.R = R_right
47
48     # --- BUILD RING CAVITY -----
49     myCavity.add_4port_component(mirror1)           # 0 left mirror
50     myCavity.add_2port_component(prop_f, prop_f)    # 1
51     myCavity.add_2port_component(lens_f, lens_f)    # 2
52     myCavity.add_2port_component(prop_f, prop_f)    # 3
53     myCavity.add_4port_component(corner_mirrors)    # 4 corner mirrors
54     myCavity.add_2port_component(prop_f, prop_f)    # 5

```

```

55 myCavity.add_2port_component(lens_f, lens_f)           # 6
56 myCavity.add_2port_component(prop_f, prop_f)          # 7
57 myCavity.add_4port_component(mirror2)                 # 8 right mirror
58
59 # --- INCIDENT FIELD PATH A (HORIZONTAL) -----
60 in_hor = clsTestImage(myCavity.grid)
61 in_hor.create_test_image(3)
62 in_hor.name = "Input (Horizontal)"
63 in_hor.plot_field(5)
64 myCavity.set_incident_field(Side.LEFT, Path.A, in_hor)
65
66 # --- INCIDENT FIELD PATH B (VERTICAL) -----
67 in_ver = clsSpeckleField(myCavity.grid)
68 in_ver.create_field(500, 0.6 * length_fov, 0)
69 in_ver.name = "Input (Vertical)"
70 in_ver.plot_field(5)
71 myCavity.set_incident_field(Side.LEFT, Path.B, in_ver)
72
73 # --- CALCULATE OUTPUT FIELDS LEFT SIDE -----
74 out_L_B = myCavity.get_output_field(Side.LEFT, Path.B)
75 out_L_B.plot_field(5)
76 out_L_A = myCavity.get_output_field(Side.LEFT, Path.A)
77 out_L_A.plot_field(5)
78
79 # --- FIELD INSIDE THE CAVITY RIGHT OF FIRST LENS -----
80 myCavity.calc_bulk_field_from_left(2)
81
82 bulk_field_A_LTR = myCavity.get_bulk_field_LTR(Path.A)
83 bulk_field_A_LTR.name = "Left-to-Right Bulk Field (Path A)"
84 bulk_field_A_LTR.plot_field(5)
85
86 bulk_field_A_RTL = myCavity.get_bulk_field_RTL(Path.A)
87 bulk_field_A_RTL.name = "Right-to-Left Bulk Field (Path A)"
88 bulk_field_A_RTL.plot_field(5)
89
90 bulk_field_A = myCavity.get_bulk_field(Path.A)
91 bulk_field_A.name = "Total Bulk Field (Path A)"
92 bulk_field_A.plot_field(5)

```

### Import statements (lines 1–5)

As in the earlier listings, lines 1–5 pull in all required classes. New here is `clsCavity2path`, the two-arm “breadboard” for ring cavities. We also import the four-port components `clsBeamSplitterMirror` (couplers) and `clsTransmissionMixer` (corner mirrors without path mixing), the two-port elements `clsPropagation` and `clsThinLens`, the light field sources `clsSpeckleField` and `clsTestImage`, and the enums `Side` and `Path` to address ports (`Side.LEFT`, `Side.RIGHT`) and arms (`Path.A`, `Path.B`).

### Main-program guard (line 7)

The line `if __name__ == '__main__':` is to ensure the library’s parallel processing behaves reliably across platforms.

### Simulation parameters (lines 8–12)

Sets the mirror reflectivities, the lens focal length  $f$ , and the field-of-view width  $L_{FOV}$  used in this example.

### Generate grid (lines 14–19)

Creates a simulation grid that satisfies the critical-sampling condition for a free propagation distance of  $2f$ , which is appropriate for a  $4f$  and  $8f$  configuration. The two `print` statements report the chosen field-of-view resolution (`res_fov`) and the total padded resolution (`res_tot`).

### Generate components (lines 26–46)

This block instantiates all optical components used in the ring:

- **Left coupler** – `clsBeamSplitterMirror` (`mirror1`) with power reflectivity  $R_{\text{left}} = 1$ .
- **Free-space section** – `clsPropagation` (`prop_f`) configured for a distance  $f$  in vacuum.
- **Thin lens** – `clsThinLens` (`lens_f`) with focal length  $f$ .
- **Corner mirrors** – `clsTransmissionMixer` (`corner_mirrors`) set to `.T_same=1` and `.refl_behavior_for_path_mixing=False`, i.e. the light field gets perfectly transmitted and has reflection-like phase behavior within the same paths.
- **Right coupler** – `clsBeamSplitterMirror` (`mirror2`) with power reflectivity  $R_{\text{right}} = 0.999$ .

### Build ring cavity (lines 48–57)

The components are placed from left to right following the rotated layout in Figure 5.7: the left beamsplitter, a propagation section, a lens section, another propagation section, the two corner mirrors, a second propagation section, a second lens section, a final propagation section, and the right beamsplitter. Four-port elements are added with `myCavity.add_4port_component(component)`, while paired two-port elements are added with `myCavity.add_2port_component(A, B)`.

The method `myCavity.add_2port_component(A, B)` registers up to two independent two-port elements – one assigned to `Path.A` and one to `Path.B` – at the *same list position* in the cavity’s component sequence. The actual physical distances traveled in paths A and B may differ; only the ordering index is shared. Internally, the pair is wrapped in a `clsOptComponentAdapter` so the cavity still sees a single component entry. Reusing the same instances (`prop_f`, `lens_f`) across multiple positions is supported and benefits from caching.

### Define incident field in path A (lines 59–64)

An instance of `clsTestImage` is created and configured to produce the “T” pattern, named and plotted for reference, and then registered as the left-incident field on path A via `myCavity.set_incident_field(...)` with parameters (`Side.LEFT`, `Path.A`).

### Define incident field in path B (lines 66–71)

An instance of `clsSpeckleField` is created to generate a random speckle pattern. It is named, plotted for reference, and then registered as the left-incident field on path B via `myCavity.set_incident_field(...)` with parameters (`Side.LEFT`, `Path.B`).

### Compute and plot left-side outputs (lines 73–77)

Calling `myCavity.get_output_field(...)(Side.LEFT, Path.B)` in line 74 triggers the internal computation of *all* output fields and caches the results. The subsequent call in line 76 for path A then reuses the cached data and returns instantly. Each returned `clsLightField` is plotted via `.plot_field(...)`.

### Probe intracavity field near the first lens (lines 79–92)

Line 79 calls `myCavity.calc_bulk_field_from_left(...)` with parameter (2), which computes the steady-state intracavity field *immediately to the right* of component 2 (the first lens), using the left incident–output pair. Lines 81–92 then retrieve and plot, for path A, the left-to-right part via `.get_bulk_field_LTR(...)`, the right-to-left part via `.get_bulk_field_RTL(...)`, and their coherent sum via `.get_bulk_field(...)`, each followed by `.plot_field(...)`.

## 5.2 Simulation Results

The software creates the following console output:

```

Field-of-view: 96x96
Total grid res: 142x142
re-using component 1 as component 3
re-using component 1 as component 5
re-using component 2 as component 6
re-using component 1 as component 7

calculating cavity's reflection 2x2 block matrix R_L from cavity's transfer 4x4 block matrix M
calculating cavity's transfer matrix M

processing component 0: left mirror
    calculating transfer matrix M
        constructing scattering matrix S
    processing transfer matrix M

processing component 1: f propagation in path A, f propagation in path B
    component will be used again: activating memory caching for component
    calculating transfer matrix M
        constructing scattering matrix S
    processing transfer matrix M
done (5.2 seconds)
done (7.5 seconds)

processing component 2: lens in path A, lens in path B
    component will be used again: activating memory caching for component
    calculating transfer matrix M
        calculating transmission matrix T
    done (30.6 seconds)
    constructing scattering matrix S
    converting S-matrix to M-matrix
        3.2% done (step 1 of 31, 94.9 seconds)
        9.7% done (step 3 of 31, 95.4 seconds)
        32.3% done (step 10 of 31, 3.8 seconds)
        71.0% done (step 22 of 31, 17.0 seconds)
        90.3% done (step 28 of 31, 3.8 seconds)
        100.0% done (step 31 of 31, 0.2 seconds)
    done (238.6 seconds)
done (306.7 seconds)
processing transfer matrix M
performing matrix multiplication
    1.6% done (step 1 of 64)
    4.7% done (step 3 of 64)
    9.4% done (step 6 of 64)
    25.0% done (step 16 of 64)
    34.4% done (step 22 of 64)
    37.5% done (step 24 of 64)
    42.2% done (step 27 of 64)
    50.0% done (step 32 of 64)
    59.4% done (step 38 of 64)
    60.9% done (step 39 of 64)
    67.2% done (step 43 of 64)
    75.0% done (step 48 of 64)
    76.6% done (step 49 of 64)
    79.7% done (step 51 of 64)
    84.4% done (step 54 of 64)
    100.0% done (step 64 of 64)
done (232.6 seconds)
done (232.7 seconds)
done (539.3 seconds)

processing component 3: f propagation in path A, f propagation in path B
    component will be used again: activating memory caching for component
    processing transfer matrix M
        performing matrix multiplication
            1.6% done (step 1 of 64)
            3.1% done (step 2 of 64)

```

```

6.2% done (step 4 of 64)
14.1% done (step 9 of 64)
25.0% done (step 16 of 64)
34.4% done (step 22 of 64)
35.9% done (step 23 of 64)
37.5% done (step 24 of 64)
42.2% done (step 27 of 64)
43.8% done (step 28 of 64)
53.1% done (step 34 of 64)
59.4% done (step 38 of 64)
60.9% done (step 39 of 64)
62.5% done (step 40 of 64)
67.2% done (step 43 of 64)
68.8% done (step 44 of 64)
76.6% done (step 49 of 64)
78.1% done (step 50 of 64)
81.2% done (step 52 of 64)
89.1% done (step 57 of 64)
100.0% done (step 64 of 64)
done (337.8 seconds)
done (343.9 seconds)
done (343.9 seconds)

processing component 4: corner mirrors
    calculating transfer matrix M
    constructing scattering matrix S
processing transfer matrix M
    performing matrix multiplication
        1.6% done (step 1 of 64)
        4.7% done (step 3 of 64)
        6.2% done (step 4 of 64)
        14.1% done (step 9 of 64)
        25.0% done (step 16 of 64)
        34.4% done (step 22 of 64)
        35.9% done (step 23 of 64)
        40.6% done (step 26 of 64)
        42.2% done (step 27 of 64)
        46.9% done (step 30 of 64)
        54.7% done (step 35 of 64)
        59.4% done (step 38 of 64)
        60.9% done (step 39 of 64)
        65.6% done (step 42 of 64)
        67.2% done (step 43 of 64)
        71.9% done (step 46 of 64)
        76.6% done (step 49 of 64)
        79.7% done (step 51 of 64)
        81.2% done (step 52 of 64)
        89.1% done (step 57 of 64)
        100.0% done (step 64 of 64)
done (290.4 seconds)
done (295.6 seconds)
done (297.0 seconds)

processing component 5: f propagation in path A, f propagation in path B
    component will be used again: activating memory caching for component
processing transfer matrix M
    performing matrix multiplication
        1.6% done (step 1 of 64)
        4.7% done (step 3 of 64)
        6.2% done (step 4 of 64)
        14.1% done (step 9 of 64)
        25.0% done (step 16 of 64)
        34.4% done (step 22 of 64)
        35.9% done (step 23 of 64)
        40.6% done (step 26 of 64)
        42.2% done (step 27 of 64)
        46.9% done (step 30 of 64)
        54.7% done (step 35 of 64)
        59.4% done (step 38 of 64)
        60.9% done (step 39 of 64)
        65.6% done (step 42 of 64)
        67.2% done (step 43 of 64)
        71.9% done (step 46 of 64)
        76.6% done (step 49 of 64)
        79.7% done (step 51 of 64)
        81.2% done (step 52 of 64)
        89.1% done (step 57 of 64)
        100.0% done (step 64 of 64)
done (324.5 seconds)
done (330.2 seconds)
done (330.2 seconds)

processing component 6: lens in path A, lens in path B
    component was used before and will not be used again: preparing memory cache to be deleted
    deleting transfer matrix M of 'lens in path A, lens in path B' from memory cache
    deleting lens phase mask of 'lens' from memory cache
    deleting transmission matrix T of 'lens' from memory cache
processing transfer matrix M
    performing matrix multiplication
        1.6% done (step 1 of 64, 63.7 seconds)
        4.7% done (step 3 of 64, 9.3 seconds)
        6.2% done (step 4 of 64, 12.7 seconds)
        12.5% done (step 8 of 64, 3.7 seconds)

```

```

18.8% done (step 12 of 64, 3.8 seconds)
25.0% done (step 16 of 64, 64.2 seconds)
29.7% done (step 19 of 64, 3.7 seconds)
34.4% done (step 22 of 64, 64.5 seconds)
35.9% done (step 23 of 64, 13.3 seconds)
40.6% done (step 26 of 64, 3.7 seconds)
42.2% done (step 27 of 64, 66.1 seconds)
46.9% done (step 30 of 64, 3.7 seconds)
51.6% done (step 33 of 64, 3.8 seconds)
59.4% done (step 38 of 64, 67.4 seconds)
60.9% done (step 39 of 64, 14.1 seconds)
62.5% done (step 40 of 64, 10.4 seconds)
67.2% done (step 43 of 64, 66.5 seconds)
68.8% done (step 44 of 64, 10.9 seconds)
75.0% done (step 48 of 64, 3.7 seconds)
76.6% done (step 49 of 64, 67.5 seconds)
79.7% done (step 51 of 64, 9.3 seconds)
81.2% done (step 52 of 64, 13.0 seconds)
87.5% done (step 56 of 64, 3.6 seconds)
93.8% done (step 60 of 64, 3.7 seconds)
100.0% done (step 64 of 64, 67.0 seconds)
done (762.2 seconds)
done (768.0 seconds)
done (768.8 seconds)

processing component 7: f propagation in path A, f propagation in path B
component was used before and will not be used again: preparing memory cache to be deleted
deleting transfer matrix M of 'f propagation in path A, f propagation in path B' from memory cache
deleting transmission matrix T of 'f propagation' from memory cache
processing transfer matrix M
performing matrix multiplication
1.6% done (step 1 of 64)
4.7% done (step 3 of 64)
6.2% done (step 4 of 64)
14.1% done (step 9 of 64)
25.0% done (step 16 of 64)
34.4% done (step 22 of 64)
35.9% done (step 23 of 64)
40.6% done (step 26 of 64)
42.2% done (step 27 of 64)
43.8% done (step 28 of 64)
53.1% done (step 34 of 64)
59.4% done (step 38 of 64)
60.9% done (step 39 of 64)
62.5% done (step 40 of 64)
67.2% done (step 43 of 64)
68.8% done (step 44 of 64)
76.6% done (step 49 of 64)
78.1% done (step 50 of 64)
81.2% done (step 52 of 64)
89.1% done (step 57 of 64)
100.0% done (step 64 of 64)
done (346.6 seconds)
done (351.7 seconds)
done (351.7 seconds)

processing component 8: right mirror
calculating transfer matrix M
constructing scattering matrix S
processing transfer matrix M
performing matrix multiplication
1.6% done (step 1 of 64)
3.1% done (step 2 of 64)
4.7% done (step 3 of 64)
6.2% done (step 4 of 64)
14.1% done (step 9 of 64)
20.3% done (step 13 of 64)
23.4% done (step 15 of 64)
25.0% done (step 16 of 64)
34.4% done (step 22 of 64)
35.9% done (step 23 of 64)
40.6% done (step 26 of 64)
42.2% done (step 27 of 64)
43.8% done (step 28 of 64)
53.1% done (step 34 of 64)
59.4% done (step 38 of 64)
60.9% done (step 39 of 64)
65.6% done (step 42 of 64)
67.2% done (step 43 of 64)
68.8% done (step 44 of 64)
76.6% done (step 49 of 64)
78.1% done (step 50 of 64)
79.7% done (step 51 of 64)
81.2% done (step 52 of 64)
89.1% done (step 57 of 64)
95.3% done (step 61 of 64)
98.4% done (step 63 of 64)
100.0% done (step 64 of 64)
done (414.8 seconds)
done (420.4 seconds)
done (421.8 seconds)
done (3061.7 seconds)

```

```

7.7% done (step 1 of 13, 91.1 seconds)
23.1% done (step 3 of 13, 92.4 seconds)
69.2% done (step 9 of 13, 68.8 seconds)
76.9% done (step 10 of 13, 68.2 seconds)
84.6% done (step 11 of 13, 10.0 seconds)
100.0% done (step 13 of 13, 3.8 seconds)
done (3407.3 seconds)
calculating bulk field right of component 2, coming from the left side
processing component 2: lens in path A, lens in path B
    calculating inverse transfer matrix M
        calculating transfer matrix M
            calculating transmission matrix T
            done (31.2 seconds)
            constructing scattering matrix S
            converting S-matrix to M-matrix
                3.2% done (step 1 of 31, 97.5 seconds)
                9.7% done (step 3 of 31, 97.0 seconds)
                32.3% done (step 10 of 31, 3.4 seconds)
                71.0% done (step 22 of 31, 15.6 seconds)
                90.3% done (step 28 of 31, 3.6 seconds)
                100.0% done (step 31 of 31, 0.2 seconds)
                done (238.6 seconds)
done (306.2 seconds)
inverting M
    10.0% done (step 1 of 10, 97.3 seconds)
    30.0% done (step 3 of 10, 98.2 seconds)
    60.0% done (step 6 of 10, 99.3 seconds)
    80.0% done (step 8 of 10, 98.5 seconds)
    100.0% done (step 10 of 10, 0.0 seconds)
    done (393.3 seconds)
done (703.4 seconds)
done (703.4 seconds)
processing component 1: f propagation in path A, f propagation in path B
    calculating inverse transfer matrix M
        calculating transfer matrix M
            constructing scattering matrix S
            done (3.1 seconds)
done (163.0 seconds)
processing component 0: left mirror
    calculating inverse transfer matrix M
        calculating transfer matrix M
            constructing scattering matrix S
            done (4.4 seconds)
done (235.4 seconds)
calculating right-to-left bulk field in path A
done (7.5 seconds)
calculating right-to-left bulk field in path B
done (7.8 seconds)
calculating left-to-right bulk field in path A
done (7.9 seconds)
calculating left-to-right bulk field in path B
done (7.9 seconds)
done (1132.9 seconds)

```

### Runtime and verbosity.

This simulation is substantially heavier than the earlier linear–cavity cases and both runtime and memory demands increase accordingly. We recommend at least 64 GB of RAM for this script. During long runs the library becomes deliberately “chatty” to keep you informed about progress.

If you prefer fewer status lines, you can:

- suppress all progress messages by setting `myCavity.progress.silent=True`, or
- throttle updates by increasing `myCavity.progress.max_time_btwn_tic_outputs`, which enforces a minimum number of seconds between consecutive progress outputs.

### Left-side incident fields.

The two incident fields are shown in Figure 5.8: the “T” image enters *horizontally* on path A, while the speckle field enters *vertically* on path B (cf. Figure 5.6).

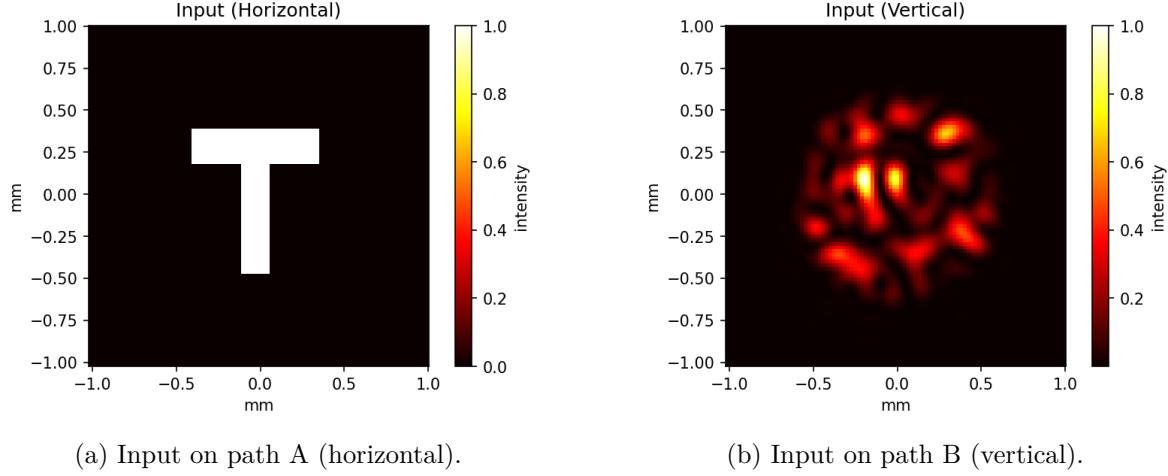


Figure 5.8: Incident fields used in the ring–cavity simulation (see Figure 5.6).

### Left-side output fields.

At the left port the two outputs appear *crossed*: the horizontal “T” injected on path A exits on path B, and the vertical speckle injected on path B exits on path A. This is consistent with the beamsplitter coupling and the  $8f$  relay formed by the four lenses, which transports each input once around the ring and delivers an upright image to the opposite path.

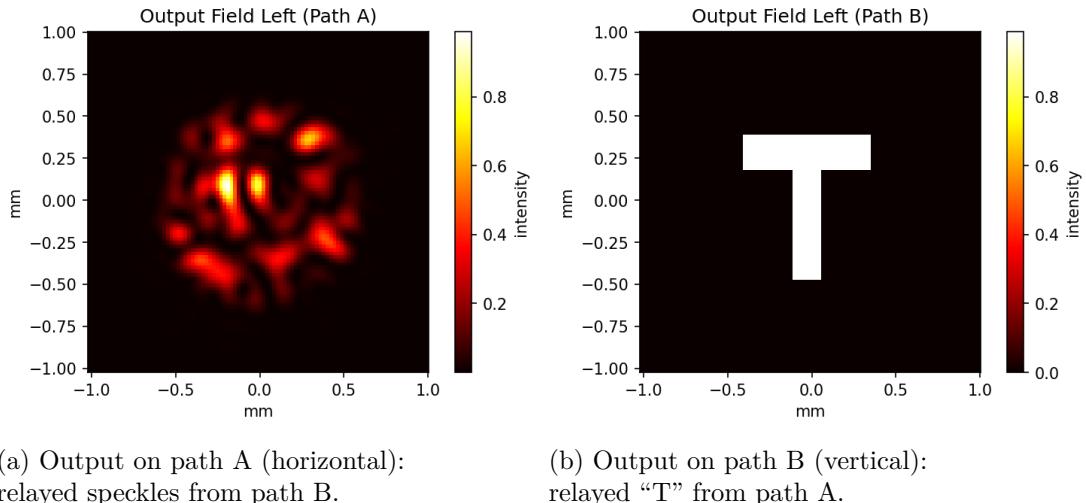


Figure 5.9: Left-side outputs of the ring cavity. The “T” traverses the ring clockwise and appears upright on path B; the speckle field traverses counter-clockwise and appears on path A.

### Intracavity fields at the probe position.

The next three panels show the bulk fields *in path A* at the probe plane marked in red in Figure 5.6 (immediately to the right of the top lens). The first is the light field propagating left-to-right at that position (i.e. clockwise around the ring), the second is the light field propagating right-to-left at that position (i.e. counter-clockwise), and the third is their coherent sum. In this geometry the LTR wave corresponds mainly to the “T” image after roughly  $1f$  of relay (plus contributions from multiple round-trips), whereas the RTL wave corresponds to the speckle pattern transported the other way (about  $7f$  path before reaching the probe plus contributions from multiple round-trips).

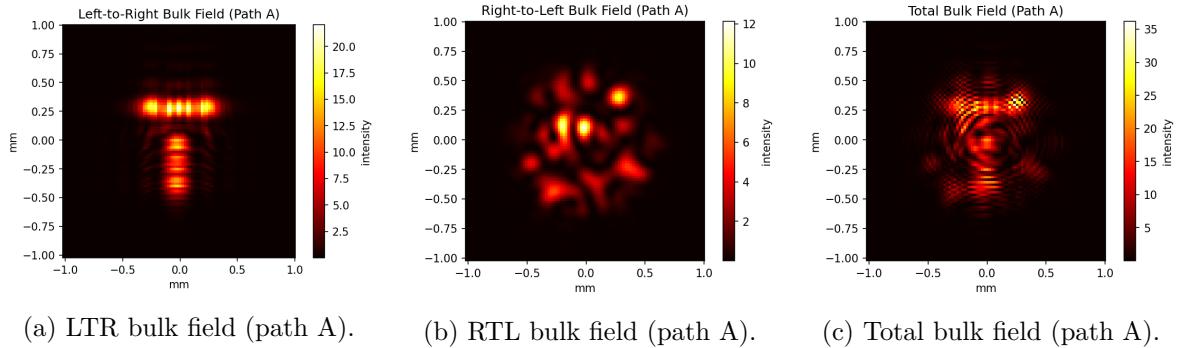


Figure 5.10: Intracavity intensity at the probe plane in path A: (a) “T” field propagating left-to-right (clockwise); (b) speckle field propagating right-to-left (counterclockwise); (c) coherent superposition of both directions.



# Chapter 6

## Running Parallel Tasks on a Cluster

### 6.1 Single-Step Mode

For large, high-resolution systems the construction of the global transfer matrix  $\mathbf{M}_{\text{cav}} = \mathbf{M}_0 \mathbf{M}_1 \mathbf{M}_2 \dots$  can take many hours. *Single-step mode* decomposes this job into discrete, restartable steps that must run in order for a given wavelength, but can run concurrently across different wavelengths – ideal for cluster environments.

Assume a single wavelength. On its first invocation, a single-step program builds only the first component’s transfer matrix  $\mathbf{M}_0$  and writes it to disk as the current partial product. On the next invocation, it reloads that partial product, computes  $\mathbf{M}_1$ , updates the product to  $\mathbf{M}_0 \mathbf{M}_1$ , and stores it again. This proceeds component by component until all `N =clsCavity.component_count` elements have been incorporated, yielding the full  $\mathbf{M}_{\text{cav}}$ . One or more post-processing steps then use  $\mathbf{M}_{\text{cav}}$  e.g., to evaluate output fields, and write the final simulation results.

In practice you may want to sweep many wavelengths (e.g., 17 points around resonance). Single-step execution remains *sequential per wavelength*, but *independent across wavelengths*. Thus, while one worker is still building  $\mathbf{M}_0$  at  $\lambda_1$ , another can already build  $\mathbf{M}_0$  at  $\lambda_2$ , and a third might have progressed to  $\mathbf{M}_0 \mathbf{M}_1$  at  $\lambda_3$ . Each wavelength maintains its own on-disk partial products, so multiple instances of the same program can run safely in parallel on a cluster, each advancing a different wavelength.

In the remainder of this chapter we present a compact example that orchestrates single-step execution with `clsTaskManager`. It hands each parallel worker the next unfinished  $\langle \text{wavelength}, \text{step} \rangle$  pair, so multiple instances of the same program advance different wavelengths safely and deterministically, with partial products saved on-disk enabling restarts without duplicate work.

#### 6.1.1 Code Example: EP–MAD–CPA, Single-Step Cluster Simulation

In this example we revisit the EP–MAD–CPA layout from subsection 4.3.4, now using a different focal length and a slightly higher grid resolution to demonstrate the single-step workflow on a cluster.

**Listing.**

The full single-step EP-MAD-CPA example is shown below; the source file can be found at examples/sample\_007A.py.

```

1 import math
2 from fo_cavity_sim import clsCavity1path, clsMirror, clsPropagation,
3     clsThinLens, clsSpeckleField, get_sample_vec, clsTaskManager
4
5 if __name__ == '__main__':
6     # --- SIMULATION PARAMETERS -----
7     R_left = 0.7                      # right mirror
8     R_right = 0.999                    # left mirror
9     R_center = 4*R_left/((1+R_left)**2) # center mirror
10    d_absorb = 0.0006                 # thickness of absorber
11    T_abs = math.sqrt(R_left/R_right) # optimal absorption
12    pos_absorb = 0.005                # distance absorber after left mirror
13    nr = 1.5                         # absorber's refractive index
14    f1 = 0.025                       # focal length first lens
15    f2 = f1 - d_absorb/2*(nr-1/nr)   # focal length second lens
16    length_fov = 0.00081             # field-of-view sidelength
17    folder = "cavity_folder"
18    tmp_folder = "tmp_folder"
19
20    # --- CAVITY SUB-CLASS CONTAINS FINAL SIMULATION STEP -----
21    class clsMyCavity(clsCavity1path):
22        def __init__(self, name):
23            super().__init__(name)
24
25        def additional_step(self, step: int):
26            """
27                Final steps after calculating M_bmat
28            """
29            # --- INPUT: RANDOM SPECKLE FIELD -----
30            self.progress.print("Generating incident field")
31            inp = clsSpeckleField(myCavity.grid)
32            inp.create_field_eq_distr(100, 20, 0.6*myCavity.grid.length_fov, 0)
33            inp.name = "Input Field"
34            self.incident_field_left = inp
35
36            # --- SAVE INCIDENT FIELD (ONLY ONCE) -----
37            if self.UID==0:
38                file_name = self._full_file_name("in", file_extension = "png")
39                inp.plot_field(5, True, file_name)
40
41            # --- GET AND SAVE OUTPUT FIELD -----
42            self.progress.print("")
43            self.progress.print("applying reflection matrix to incident field")
44            out = self.output_field_left
45            file_name = self._full_file_name("out", file_extension = "png")
46            out.plot_field(5, True, file_name, c_map="custom")
47
48            # --- CALCULATE AND SAVE REFLECTIVITY -----
49            R = out.intensity_integral_fov() / inp.intensity_integral_fov()
50            self.progress.print(f"Reflectivity: {R}")
51            data = [self.UID, self.Lambda_nm,
52                    1000*(self.Lambda_nm-self.Lambda_ref_nm), R]
53            self.write_to_file("result.csv",data)
54

```

```

55      # --- DELETE TMP FILES -----
56      self.M_bmat_tot.keep_tmp_files = False
57      self.M_bmat_tot.clear()
58      self.M_bmat_tot = None
59      self.progress.pop()
60
61      # --- CREATE CAVITY -----
62      myCavity = clsMyCavity("8f_cavity")
63      myCavity.additional_steps=1
64      myCavity.allow_temp_file_caching = True
65      myCavity.use_swap_files_in_bmatrix_class = True
66      myCavity.folder = folder
67      myCavity.tmp_folder = tmp_folder
68      myCavity.Lambda_nm = 633
69
70      # --- DETERMINE RESONANCE WAVELENGTH, CREATE GRID -----
71      lambda_c, k_c, n_c, lambda_period, l2_corr = \
72          myCavity.resonance_data_8f_cavity(R_left, R_center, R_right, f1)
73      l2_corr *= 1.212
74      myCavity.Lambda = lambda_c
75      myCavity.grid.set_opt_res_based_on_sidelength(length_fov, 2, f1, True)
76      print(f"Field-of-view: {myCavity.grid.res_fov}x{myCavity.grid.res_fov}")
77      print(f"Total grid res: {myCavity.grid.res_tot}x{myCavity.grid.res_tot}")
78
79      # --- GENERATE COMPONENTS -----
80      # input coupling mirror
81      mirror1 = clsMirror("left mirror", myCavity)
82      mirror1.R = R_left
83
84      # f1 propagation
85      prop_f1 = clsPropagation("f1 propagation", myCavity)
86      prop_f1.set_params(f1, 1)
87
88      # lens f1
89      lens_f1 = clsThinLens("lens 1", myCavity)
90      lens_f1.f = f1
91
92      # center mirror
93      center_mirror = clsMirror("center mirror", myCavity)
94      center_mirror.R = R_center
95
96      # f2 propagation
97      prop_f2 = clsPropagation("f2 propagation", myCavity)
98      prop_f2.set_params(f2, 1)
99
100     # lens f2
101     lens_f2 = clsThinLens("lens 2", myCavity)
102     lens_f2.f = f2
103
104     # propagation between second lens and absorber
105     prop_lens_absorb = clsPropagation("propagation lens<>absorber", myCavity)
106     prop_lens_absorb.set_params(f2-pos_absorb-d_absorb/nr, 1)
107
108     # absorber
109     absorber = clsPropagation("absorber", myCavity)
110     absorber.set_params(d_absorb, nr)
111     absorber.set_ni_based_on_T(T_abs)
112
113     # propagation between absorber and right mirror

```

```

114 prop_absorb_mirr = clsPropagation("propagation absorber<>mirror2", myCavity)
115 prop_absorb_mirr.set_params(pos_absorb + 12_corr, 1)
116
117 # right mirror
118 mirror2 = clsMirror("right mirror", myCavity)
119 mirror2.R = R_right
120
121 # --- ASSEMBLE CAVITY -----
122 myCavity.add_component(mirror1)           # 0 *****
123 myCavity.add_component(prop_f1)          # 1
124 myCavity.add_component(lens_f1)          # 2
125 myCavity.add_component(prop_f1)          # 3
126 myCavity.add_component(prop_f1)          # 4
127 myCavity.add_component(lens_f1)          # 5
128 myCavity.add_component(prop_f1)          # 6
129 myCavity.add_component(center_mirror)    # 7 *****
130 myCavity.add_component(prop_f1)          # 8
131 myCavity.add_component(lens_f1)          # 9
132 myCavity.add_component(prop_f1)          # 10
133 myCavity.add_component(prop_f2)          # 11
134 myCavity.add_component(lens_f2)          # 12
135 myCavity.add_component(prop_lens_absorb) # 13
136 myCavity.add_component(absorber)         # 14
137 myCavity.add_component(prop_absorb_mirr) # 15
138 myCavity.add_component(mirror2)          # 16 *****
139
140 #####
141 # Main Program
142 #####
143 myCavity.Lambda_ref = lambda_c
144 points_to_simulate = 17
145 lambda_vec = get_sample_vec(points_to_simulate, lambda_period/25, 0, True, 1)
146 steps = myCavity.total_steps
147 task_manager = clsTaskManager(points_to_simulate, steps, tmp_folder)
148 task_manager.sleep_time = 600
149
150 sim, step = task_manager.get_next_task()
151 if not sim == -1:
152     dLambda = lambda_vec[sim]
153     dL_pm = dLambda * 1e12
154     print("")
155     print("*****")
156     print(f"UID = {sim}, step = {step}, delta lambda = {dL_pm:.4f} pm")
157     print("*****")
158     myCavity.Lambda = lambda_c + dLambda
159     myCavity.UID = sim
160     myCavity.single_step(step)
161     task_manager.end_task(sim, step)

```

### Import statements (lines 1–3)

As in the earlier listings, lines 1–3 import all required classes. New here is `clsTaskManager`, which orchestrates the single-step tasks across multiple worker processes so the program can run many wavelengths in parallel while advancing each one step by step.

### Main-program guard (line 5)

The line `if __name__ == '__main__':` is to ensure the library's parallel processing behaves reliably across platforms.

### Simulation parameters (lines 6–18)

This block collects all physics-relevant settings for the EP–MAD–CPA (mirror reflectivities, absorber thickness and position, lens focal lengths, field-of-view size, etc.) as well as the name of the output directory and temporary directory (`folder`, `tmp_folder`). Both directories must already exist as subfolders of the program's working directory before the program is started.

### Subclass for post-processing (lines 20–59)

In single-step mode the method `clsCavity.single_step(...)` (invoked on each run of this program; see line 161) automatically calls `clsCavity.additional_step(...)` once the cavity's transfer matrix  $\mathbf{M}_{\text{cav}}$  has been assembled. The base implementation is empty; you must override it with code that *uses*  $\mathbf{M}_{\text{cav}}$  (e.g. to compute and save output fields and metrics). Therefore we define a small subclass `clsMyCavity(clsCavity1path)` and call `super().__init__(name)`; the overridden method `.additional_step(...)` is provided next in the listing. We explain this method here in one place, even though execution happens later.

#### `.additional_step(...)` (lines 25–59)

##### Creating and saving incident field (lines 25–39)

This code-block creates the left-incident speckle field and assigns it to the cavity property `self.incident_field_left`. For documentation, the input field is written to disk at the first wavelength job (`self.UID == 0`). The helper `self.full_file_name(...)` composes a UID-aware file name, so each wavelength run stores its images under a unique name. The UID itself is set in the main program in line 159: `myCavity.UID = sim`. The plot is saved with `clsLightField.plot_field(...)`.

##### Compute and save left output (lines 41–46)

Accessing `self.output_field_left` triggers the construction of the left reflection matrix from  $\mathbf{M}_{\text{cav}}$  and applies it to the registered incident field to obtain the left output field. The result is then saved: a UID-aware file name is created with `self.full_file_name(...)` and the image is written via `out.plot_field(...)`.

##### Compute and record reflectivity (lines 48–53)

The reflectivity over the field of view is computed as the ratio of total output to input power,  $R = \text{out.intensity\_integral\_fov}(\dots)/\text{inp.intensity\_integral\_fov}(\dots)$  and appended as a CSV row using `self.write_to_file(...)`. Each row stores the job ID `self.UID`, the absolute wavelength `self.Lambda_nm`, the detuning in picometres ( $1000 \cdot (\text{self.Lambda\_nm} - \text{self.Lambda\_ref\_nm})$ ), and  $R$ .

##### Cleanup (lines 55–59)

After the results are written, the transfer matrix  $\mathbf{M}_{\text{cav}}$  is discarded to free RAM and disk space.

### Create cavity and configure runtime (lines 61–68)

Before any of the lines 25–59 are executed, an instance of the subclass `clsMyCavity` is created and configured for single-step runs: `myCavity.additional_steps=1` enables one post-processing pass after  $\mathbf{M}_{\text{cav}}$  is built; `myCavity.allow_temp_file_caching` and

`myCavity.use_swap_files_in_bmatrix_class` turn on disk-backed caching to reduce peak RAM; the output and temporary directories are set via `myCavity.folder` and `myCavity.tmp_folder`; finally, the vacuum wavelength is set to 633 nm.

### Resonance and grid (lines 70–77)

The call to `myCavity.resonance_data_8f_cavity(...)` returns the resonance wavelength  $\lambda_c$  (closest to 633 nm), the free-spectral data ( $k_c$ ,  $n_c$ ,  $\lambda_{\text{period}}$ ), and a small path-length correction  $l2_{\text{corr}}$  for the right sub-cavity. After applying a calibration factor to  $l2_{\text{corr}}$ , the simulation wavelength is set to  $\lambda_c$ . Using `myCavity.grid.set_opt_res_based_on_sidelength(...)`, the grid is then sized to satisfy the critical-sampling condition. Finally, the chosen FOV and total grid resolutions are printed.

### Generate components (lines 79–119)

All optical elements for the EP–MAD–CPA are instantiated (input coupler, free-space sections  $f_1$  and  $f_2$ , thin lenses  $f_1$  and  $f_2$ , absorber section with index  $n_r$ , and set transmissivity, spacing propagations, and the end mirror). This matches the setup in subsection 4.3.4; there are no differences specific to single-step mode.

### Assemble cavity (lines 121–138)

Identical to subsection 4.3.4: the components are added left-to-right to build the EP–MAD–CPA layout. Several instances (`prop_f1`, `lens_f1`, etc.) are reused. These calls only register the elements; no optics is computed yet.

### Main program (lines 140–161)

This block drives the single-step workflow. First, it

- sets the cavity’s reference wavelength `.Lambda_ref` to the expected resonance (later used in lines 51–52 to compute  $\Delta\lambda$ );
- creates a vector of 17 wavelength samples around resonance, using `get_sample_vec(...)`;
- determines the total number of steps for *one* wavelength via `.total_steps`;
- instantiates `clsTaskManager` to coordinate work distribution; and
- sets `task_manager.sleep_time` to 600 seconds. This is the time to pause when `task_manager.get_next_task()` is unable to assign a new task.

Then, in line 150, the program requests a task with `task_manager.get_next_task()`. The first return value `sim` selects the wavelength (index into `lambda_vec`); the second, `step`, selects the sequential step for that wavelength. If `sim == -1`, either all work is finished *or* no task is currently available because other processes are handling all remaining wavelengths. In the latter case the `.get_next_task()` method waits for `sleep_time` seconds before returning.

If a task could be assigned, the code sets `myCavity.Lambda` and `myCavity.UID`, and executes `myCavity.single_step(...)`. If there are  $N$  optical components, invoking the method `myCavity.single_step(...)` with `step= 0, 1, \dots, N - 1` sequentially constructs the cavity transfer matrix  $\mathbf{M}_{\text{cav}}$  by multiplying the component matrices. On the final call with `step= N`, `.single_step(...)` hands control to the overridden method `myCavity.additional_step(...)` (defined in lines 25–59), which then uses  $\mathbf{M}_{\text{cav}}$  (e.g., to compute and save fields).

Finally, the program registers completion via `task_manager.end_task(...)`. Multiple program instances can run concurrently; `clsTaskManager` ensures that each `(sim, step)` is done exactly once and that per-wavelength steps are executed in order.

### 6.1.2 Slurm File Configuration

This is a template showing an exemplary slurm-file configuration `sample_007A.slurm` to run above program on a cluster.

```

1 #!/bin/bash
2 #
3 #SBATCH -J S007A           # job name
4 #SBATCH -N 1               # number of nodes
5 #SBATCH --ntasks-per-node=48      # here we use all cores of the node
6 #SBATCH --ntasks-per-core=1
7 #SBATCH --threads-per-core=1
8 #SBATCH --partition=*****
9 #SBATCH --qos=*****
10 #SBATCH --account=*****
11 #SBATCH --mail-type=BEGIN,END,FAIL
12 #SBATCH --mail-user=*****.*****@*****.**
13 #SBATCH --time=01:00:00          # run time unit=minutes
14 #SBATCH --array=1-330%17
15
16 source /home/*****/.pyenv/versions/3.8.5/envs/pyenv_385/bin/activate
17 which python
18 time python -u sample_007A.py

```

#### Assumptions.

We assume (i) a Python virtual environment is available and can be activated on the compute nodes (like in line 16); (ii) the files `fo_cavity_sim.py`, `cmap_custom.py`, the driver script `sample_007A.py`, and the SLURM script `sample_007A.slurm` are all placed in the working directory from which the job is submitted; and (iii) the subfolders `cavity_folder` and `tmp_folder` already exist.

#### SLURM header (lines 1–13).

Configure these directives to match your cluster: job name, node/CPU layout, partition, QoS/account, mail options, and wall time. The values shown are placeholders.

#### Job array (line 14).

`#SBATCH --array=1-330%17` launches up to 330 array tasks with at most 17 running concurrently. In this example the simulation spans 17 wavelengths and, per wavelength, 18 single-step stages (17 components plus one post-processing step), i.e.  $17 \times 18 = 306$  required invocations. The array length is set to 330 to provide slack for scheduling contention near the end of the run; the concurrency cap `%17` matches the number of wavelengths so that at most one task per wavelength proceeds at a time. Adjust these numbers if you change `points_to_simulate` or the component count.

#### Environment activation (lines 16–17).

Line 16 activates the desired Python environment (adapt the path/name to your setup by replacing the `*****` characters accordingly). Line 17 prints the interpreter path so you can verify the environment on the node.

### Program launch (line 18).

`python -u sample_007A.py` starts the driver script in *unbuffered* mode (`-u`), which ensures timely, reliable console output from the library's parallel sections. Each array task runs an identical program instance; work distribution across wavelengths/steps is coordinated at runtime by `clsTaskManager`, not by SLURM environment variables.

### Submitting the job

To launch the batch, change into the directory containing `sample_007A.slrn` and submit:

```
sbatch sample_007A.slrn
```

SLURM prints a job ID (e.g. 123456). You can monitor and manage the job with:

<code>squeue -u \${USER}</code>	list your running/pending jobs,
<code>squeue -j 123456</code>	inspect this job's state,
<code>tail -f slurm-123456.out</code>	follow the job's stdout/stderr,
<code>scancel 123456</code>	cancel the job if needed.

#### 6.1.3 Simulation Results

The `stdout/stderr` of each SLURM array task is written to its own `.out` file. With the array size chosen above this produces one file per task. Below we show the logs for the *central* wavelength run ( $\Delta\lambda = 0$ ).

```
/home/********/.pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 0, delta lambda = 0.0000 pm
*****
UID: 0, single step 0
processing component 0: left mirror
    calculating transfer matrix M
processing transfer matrix M
    saving transfer block matrix of cavity

real      1m52.131s
user       0m4.117s
sys        0m13.794s
```

```
/home/********/.pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 1, delta lambda = 0.0000 pm
*****
UID: 0, single step 1
processing component 1: f1 propagation
    loading transfer block matrix of cavity
    component will be used again: activating file caching for component
    calculating transfer matrix M
        deleting transmission matrix T of 'f1 propagation' from memory cache
processing transfer matrix M
    saving transfer block matrix of cavity

real      1m33.077s
user       0m4.261s
sys        0m13.855s
```

```

/home/********/.pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 2, delta lambda = 0.0000 pm
*****
UID: 0, single step 2
    processing component 2: lens 1
        loading transfer block matrix of cavity
        component will be used again: activating file caching for component
        calculating transfer matrix M
            calculating transmission matrix T
            done (25.9 seconds)
            deleting lens phase mask of 'lens 1' from memory cache
            deleting transmission matrix T of 'lens 1' from memory cache
            converting S-matrix to M-matrix
                20.0% done (step 1 of 5, 251.3 seconds)
                40.0% done (step 2 of 5, 35.6 seconds)
                80.0% done (step 4 of 5, 37.3 seconds)
                100.0% done (step 5 of 5, 36.0 seconds)
                done (360.1 seconds)
            done (473.8 seconds)
            saving transfer matrix M of 'lens 1'
            processing transfer matrix M
                performing matrix multiplication
                    12.5% done (step 1 of 8)
                    25.0% done (step 2 of 8)
                    50.0% done (step 4 of 8)
                    62.5% done (step 5 of 8)
                    75.0% done (step 6 of 8)
                    100.0% done (step 8 of 8)
                    done (279.2 seconds)
            done (279.3 seconds)
            saving transfer block matrix of cavity
        done (753.1 seconds)
    done (753.1 seconds)

real      12m34.815s
user      192m19.390s
sys       5m15.712s

```

```

/home/********/.pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 3, delta lambda = 0.0000 pm
*****
UID: 0, single step 3
    processing component 3: f1 propagation
        loading transfer block matrix of cavity
        component will be used again: activating file caching for component
        calculating transfer matrix M
            deleting transmission matrix T of 'f1 propagation' from memory cache
        processing transfer matrix M
            performing matrix multiplication
                12.5% done (step 1 of 8)
                25.0% done (step 2 of 8, 74.7 seconds)
                37.5% done (step 3 of 8, 33.6 seconds)
                50.0% done (step 4 of 8, 54.3 seconds)
                62.5% done (step 5 of 8, 56.5 seconds)
                75.0% done (step 6 of 8, 81.1 seconds)
                87.5% done (step 7 of 8, 40.6 seconds)
                100.0% done (step 8 of 8, 62.2 seconds)
                done (443.3 seconds)
            done (443.3 seconds)
            saving transfer block matrix of cavity
        done (443.5 seconds)
    done (443.5 seconds)

real      8m19.865s
user      0m30.464s
sys       0m59.396s

```

```
/home/********/.pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 4, delta lambda = 0.0000 pm
*****
UID: 0, single step 4
processing component 4: f1 propagation
loading transfer block matrix of cavity
component will be used again: activating file caching for component
calculating transfer matrix M
deleting transmission matrix T of 'f1 propagation' from memory cache
processing transfer matrix M
performing matrix multiplication
12.5% done (step 1 of 8)
25.0% done (step 2 of 8, 81.0 seconds)
37.5% done (step 3 of 8, 34.3 seconds)
50.0% done (step 4 of 8, 38.9 seconds)
62.5% done (step 5 of 8, 44.7 seconds)
75.0% done (step 6 of 8, 62.3 seconds)
87.5% done (step 7 of 8, 39.0 seconds)
100.0% done (step 8 of 8, 48.1 seconds)
done (400.1 seconds)
done (400.2 seconds)
saving transfer block matrix of cavity
done (400.4 seconds)
done (400.4 seconds)

real      7m6.668s
user      0m24.069s
sys       0m47.351s
```

```
/home/********/.pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 5, delta lambda = 0.0000 pm
*****
UID: 0, single step 5
processing component 5: lens 1
loading transfer block matrix of cavity
component will be used again: activating file caching for component
loading transfer matrix M of 'lens 1'
deleting transfer matrix M of 'lens 1' from memory cache
processing transfer matrix M
performing matrix multiplication
12.5% done (step 1 of 8, 149.2 seconds)
25.0% done (step 2 of 8, 114.6 seconds)
37.5% done (step 3 of 8, 59.9 seconds)
50.0% done (step 4 of 8, 186.4 seconds)
62.5% done (step 5 of 8, 127.4 seconds)
75.0% done (step 6 of 8, 31.6 seconds)
87.5% done (step 7 of 8, 20.2 seconds)
100.0% done (step 8 of 8, 133.5 seconds)
done (823.0 seconds)
done (823.0 seconds)
saving transfer block matrix of cavity
done (823.3 seconds)
done (823.3 seconds)

real      14m37.222s
user      319m41.438s
sys       15m25.663s
```

```
/home/*********/pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 6, delta lambda = 0.0000 pm
*****
UID: 0, single step 6
    processing component 6: f1 propagation
        loading transfer block matrix of cavity
        component will be used again: activating file caching for component
        calculating transfer matrix M
        deleting transmission matrix T of 'f1 propagation' from memory cache
    processing transfer matrix M
        performing matrix multiplication
            12.5% done (step 1 of 8)
            25.0% done (step 2 of 8)
            37.5% done (step 3 of 8)
            50.0% done (step 4 of 8)
            62.5% done (step 5 of 8)
            75.0% done (step 6 of 8)
            87.5% done (step 7 of 8)
            100.0% done (step 8 of 8)
        done (298.4 seconds)
    done (298.5 seconds)
        saving transfer block matrix of cavity
    done (298.7 seconds)
done (298.7 seconds)

real      5m21.986s
user       0m30.440s
sys       0m49.899s
```

```
/home/*********/pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 7, delta lambda = 0.0000 pm
*****
UID: 0, single step 7
    processing component 7: center mirror
        loading transfer block matrix of cavity
        calculating transfer matrix M
    processing transfer matrix M
        performing matrix multiplication
            12.5% done (step 1 of 8)
            25.0% done (step 2 of 8, 72.6 seconds)
            37.5% done (step 3 of 8, 33.1 seconds)
            50.0% done (step 4 of 8, 55.6 seconds)
            62.5% done (step 5 of 8, 32.8 seconds)
            75.0% done (step 6 of 8, 58.3 seconds)
            87.5% done (step 7 of 8, 32.1 seconds)
            100.0% done (step 8 of 8, 70.7 seconds)
        done (387.1 seconds)
    done (387.2 seconds)
        saving transfer block matrix of cavity
    done (387.3 seconds)
done (387.3 seconds)

real      7m22.335s
user       0m31.912s
sys       1m2.094s
```

```

/home/********/.pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 8, delta lambda = 0.0000 pm
*****
UID: 0, single step 8
processing component 8: f1 propagation
loading transfer block matrix of cavity
component will be used again: activating file caching for component
calculating transfer matrix M
deleting transmission matrix T of 'f1 propagation' from memory cache
processing transfer matrix M
performing matrix multiplication
 12.5% done (step 1 of 8)
 25.0% done (step 2 of 8, 68.9 seconds)
 37.5% done (step 3 of 8, 32.1 seconds)
 50.0% done (step 4 of 8, 29.3 seconds)
 62.5% done (step 5 of 8, 33.5 seconds)
 75.0% done (step 6 of 8, 59.1 seconds)
 87.5% done (step 7 of 8, 23.8 seconds)
100.0% done (step 8 of 8, 33.5 seconds)
done (319.4 seconds)
done (319.5 seconds)
saving transfer block matrix of cavity
done (319.7 seconds)
done (319.7 seconds)

real      5m56.889s
user       0m24.987s
sys        0m50.470s

```

```

/home/********/.pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 9, delta lambda = 0.0000 pm
*****
UID: 0, single step 9
processing component 9: lens 1
loading transfer block matrix of cavity
component was used before and will not be used again:
- activating cached file to be loaded
- preparing cache file to be deleted after use
loading transfer matrix M of 'lens 1'
deleting transfer matrix M of 'lens 1' from memory cache
processing transfer matrix M
performing matrix multiplication
 12.5% done (step 1 of 8, 144.5 seconds)
 25.0% done (step 2 of 8, 48.7 seconds)
 37.5% done (step 3 of 8, 35.0 seconds)
 50.0% done (step 4 of 8, 194.2 seconds)
 62.5% done (step 5 of 8, 200.9 seconds)
 75.0% done (step 6 of 8, 60.6 seconds)
 87.5% done (step 7 of 8, 21.7 seconds)
100.0% done (step 8 of 8, 136.5 seconds)
done (842.0 seconds)
deleting transfer matrix file of lens 1
done (842.1 seconds)
saving transfer block matrix of cavity
done (842.2 seconds)
done (842.2 seconds)

real      14m32.619s
user       31m0.642s
sys        13m58.462s

```

```
/home/********/.pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 10, delta lambda = 0.0000 pm
*****
UID: 0, single step 10
    processing component 10: f1 propagation
        loading transfer block matrix of cavity
        component was used before and will not be used again:
        - activating cached file to be loaded
        - preparing cache file to be deleted after use
        calculating transfer matrix M
            deleting transmission matrix T of 'f1 propagation' from memory cache
    processing transfer matrix M
        performing matrix multiplication
            12.5% done (step 1 of 8)
            25.0% done (step 2 of 8)
            37.5% done (step 3 of 8)
            50.0% done (step 4 of 8)
            62.5% done (step 5 of 8)
            75.0% done (step 6 of 8, 63.5 seconds)
            87.5% done (step 7 of 8, 32.4 seconds)
            100.0% done (step 8 of 8, 42.2 seconds)
        done (286.2 seconds)
        saving transfer block matrix of cavity
    done (286.4 seconds)
done (286.4 seconds)

real      5m9.767s
user      0m30.849s
sys       0m59.344s
```

```
/home/********/.pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 11, delta lambda = 0.0000 pm
*****
UID: 0, single step 11
    processing component 11: f2 propagation
        loading transfer block matrix of cavity
        calculating transfer matrix M
            deleting transmission matrix T of 'f2 propagation' from memory cache
    processing transfer matrix M
        performing matrix multiplication
            12.5% done (step 1 of 8)
            25.0% done (step 2 of 8, 84.9 seconds)
            37.5% done (step 3 of 8, 27.9 seconds)
            50.0% done (step 4 of 8, 26.1 seconds)
            62.5% done (step 5 of 8, 36.0 seconds)
            75.0% done (step 6 of 8, 62.0 seconds)
            87.5% done (step 7 of 8, 31.7 seconds)
            100.0% done (step 8 of 8, 37.4 seconds)
        done (350.2 seconds)
    done (350.2 seconds)
    saving transfer block matrix of cavity
done (350.4 seconds)
done (350.4 seconds)

real      6m32.754s
user      0m23.739s
sys       0m46.372s
```

```

/home/********/.pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 12, delta lambda = 0.0000 pm
*****
UID: 0, single step 12
    processing component 12: lens 2
        loading transfer block matrix of cavity
        calculating transfer matrix M
            calculating transmission matrix T
            done (23.1 seconds)
            deleting lens phase mask of 'lens 2' from memory cache
            deleting transmission matrix T of 'lens 2' from memory cache
            converting S-matrix to M-matrix
                20.0% done (step 1 of 5, 261.4 seconds)
                40.0% done (step 2 of 5, 27.4 seconds)
                80.0% done (step 4 of 5, 27.2 seconds)
                100.0% done (step 5 of 5, 28.6 seconds)
            done (344.6 seconds)
        done (442.5 seconds)
    processing transfer matrix M
        performing matrix multiplication
            12.5% done (step 1 of 8, 145.7 seconds)
            25.0% done (step 2 of 8, 67.8 seconds)
            37.5% done (step 3 of 8, 32.7 seconds)
            50.0% done (step 4 of 8, 187.7 seconds)
            62.5% done (step 5 of 8, 206.4 seconds)
            75.0% done (step 6 of 8, 140.8 seconds)
            87.5% done (step 7 of 8, 46.3 seconds)
            100.0% done (step 8 of 8, 153.1 seconds)
        done (980.4 seconds)
    done (980.4 seconds)
    saving transfer block matrix of cavity
done (1422.9 seconds)
done (1422.9 seconds)

real      24m2.315s
user      515m24.975s
sys       20m10.395s

```

```

/home/********/.pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 13, delta lambda = 0.0000 pm
*****
UID: 0, single step 13
    processing component 13: propagation lens<>absorber
        loading transfer block matrix of cavity
        calculating transfer matrix M
            deleting transmission matrix T of 'propagation lens<>absorber' from memory cache
    processing transfer matrix M
        performing matrix multiplication
            12.5% done (step 1 of 8)
            25.0% done (step 2 of 8, 83.0 seconds)
            37.5% done (step 3 of 8, 32.4 seconds)
            50.0% done (step 4 of 8, 47.2 seconds)
            62.5% done (step 5 of 8, 45.8 seconds)
            75.0% done (step 6 of 8, 69.6 seconds)
            87.5% done (step 7 of 8, 35.3 seconds)
            100.0% done (step 8 of 8, 65.0 seconds)
        done (422.4 seconds)
    done (422.4 seconds)
    saving transfer block matrix of cavity
done (422.6 seconds)
done (422.6 seconds)

real      7m52.997s
user      0m32.823s
sys       0m49.985s

```

```
/home/********/.pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 14, delta lambda = 0.0000 pm
*****
UID: 0, single step 14
    processing component 14: absorber
        loading transfer block matrix of cavity
        calculating transfer matrix M
            deleting transmission matrix T of 'absorber' from memory cache
    processing transfer matrix M
        performing matrix multiplication
            12.5% done (step 1 of 8)
            25.0% done (step 2 of 8, 82.5 seconds)
            37.5% done (step 3 of 8, 47.6 seconds)
            50.0% done (step 4 of 8, 50.6 seconds)
            62.5% done (step 5 of 8, 51.1 seconds)
            75.0% done (step 6 of 8, 79.1 seconds)
            87.5% done (step 7 of 8, 40.7 seconds)
            100.0% done (step 8 of 8, 42.6 seconds)
        done (442.6 seconds)
    done (442.7 seconds)
        saving transfer block matrix of cavity
    done (442.9 seconds)
done (442.9 seconds)

real      8m25.601s
user       0m23.758s
sys       0m45.921s
```

```
/home/********/.pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 15, delta lambda = 0.0000 pm
*****
UID: 0, single step 15
    processing component 15: propagation absorber<>mirror2
        loading transfer block matrix of cavity
        calculating transfer matrix M
            deleting transmission matrix T of 'propagation absorber<>mirror2' from memory cache
    processing transfer matrix M
        performing matrix multiplication
            12.5% done (step 1 of 8)
            25.0% done (step 2 of 8, 65.0 seconds)
            37.5% done (step 3 of 8, 30.3 seconds)
            50.0% done (step 4 of 8, 42.6 seconds)
            62.5% done (step 5 of 8, 50.8 seconds)
            75.0% done (step 6 of 8, 66.5 seconds)
            87.5% done (step 7 of 8, 45.0 seconds)
            100.0% done (step 8 of 8, 39.2 seconds)
        done (394.1 seconds)
    done (395.1 seconds)
        saving transfer block matrix of cavity
    done (395.4 seconds)
done (395.4 seconds)

real      7m25.222s
user       0m23.366s
sys       0m46.245s
```

```
/home/********/.pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 16, delta lambda = 0.0000 pm
*****
UID: 0, single step 16
    processing component 16: right mirror
        loading transfer block matrix of cavity
        calculating transfer matrix M
        processing transfer matrix M
            performing matrix multiplication
                12.5% done (step 1 of 8)
                25.0% done (step 2 of 8, 77.1 seconds)
                37.5% done (step 3 of 8, 59.5 seconds)
                50.0% done (step 4 of 8, 171.5 seconds)
                62.5% done (step 5 of 8, 102.9 seconds)
                75.0% done (step 6 of 8, 110.3 seconds)
                87.5% done (step 7 of 8, 40.2 seconds)
                100.0% done (step 8 of 8, 86.8 seconds)
            done (691.1 seconds)
        done (691.2 seconds)
        saving transfer block matrix of cavity
        done (691.3 seconds)
    done (691.3 seconds)

real      12m11.295s
user      0m32.311s
sys       1m2.963s
```

```
/home/********/.pyenv/versions/3.8.5/envs/pyenv_385/bin/python
Field-of-view: 84x84
Total grid res: 166x166

*****
UID = 0, step = 17, delta lambda = 0.0000 pm
*****
UID: 0, single step 17
    processing additional step 0
        loading transfer block matrix of cavity
        Generating incident field

        applying reflection matrix to incident field
        calculating cavity's reflection matrix R_L from cavity's transfer matrix M
        performing matrix division
            50.0% done (step 1 of 2, 245.6 seconds)
            100.0% done (step 2 of 2, 116.5 seconds)
        done (362.1 seconds)
    done (362.1 seconds)
    Reflectivity: 3.241568113683991e-07
    done (367.3 seconds)
done (367.3 seconds)

real      6m59.428s
user      264m35.869s
sys       6m50.368s
```

### Interpreting the logs.

From the excerpts above you can see that a *single* wavelength is executed as 18 single-step jobs (steps 0–17): steps 0–16 successively build the cavity transfer matrix by multiplying in one component per step, and the final step 17 performs the post-processing via `additional_step` (loading the accumulated block matrix, computing  $\mathbf{R}_L$ , generating/saving the output field, and logging the reflectivity). The “chatty” progress lines report when cached block matrices are loaded, when expensive sub-operations (e.g. transmission-matrix generation or matrix multiplications/divisions) run, and the wall-clock time per step. This is precisely the intended single-step decomposition that enables sequential execution per wavelength and parallelization *across* wavelengths.

**Runtime note.**

In this example, the slowest per-wavelength step was step 12 (“lens 2”), which took about  $\approx 24$  min of wall time. Setting the Slurm time limit to `#SBATCH --time=01:00:00` per array task is therefore conservative and safe. (Leaving extra headroom is advisable to accommodate filesystem contention or node-to-node performance variability.) For larger grids and higher resolutions, the execution time per single step can easily grow to many hours, so adjust `--time` accordingly.

**Generated CSV file.** For each wavelength, the script appends one line to `result.csv` (index, wavelength in nm, detuning in pm, reflectivity).

```

0, 632.9997729682652, 0.0, 3.241568113683991e-07
1, 632.9997829854437, 0.010017178510679514, 0.00024355023650065848
2, 632.9997629510867, -0.010017178510679514, 0.00024357180913094903
3, 632.9997930026221, 0.02003435690767219, 0.0038752771361340913
5, 632.9998030198007, 0.030051535418351705, 0.019297180738447276
4, 632.9997529339083, -0.02003435690767219, 0.0038754496920477173
6, 632.9997429167298, -0.030051535418351705, 0.019297750659437646
7, 632.999813036979, 0.04006871381534438, 0.05848316533124665
8, 632.9997328995513, -0.04006871392903122, 0.05848442416521731
9, 632.9998230541576, 0.050085892326023895, 0.1315107629477487
10, 632.9997228823728, -0.05008589243971073, 0.13151288670747238
11, 632.999833071336, 0.06010307072301657, 0.2386321461672914
12, 632.9997128651944, -0.06010307083670341, 0.23863502322501887
13, 632.9998430885145, 0.07012024923369609, 0.36685518945216006
14, 632.9997028480159, -0.07012024934738292, 0.36685842747494846
15, 632.999853105693, 0.0801374277443756, 0.4964808538317171
16, 632.9996928308374, -0.08013742785806244, 0.4964839870044258

```

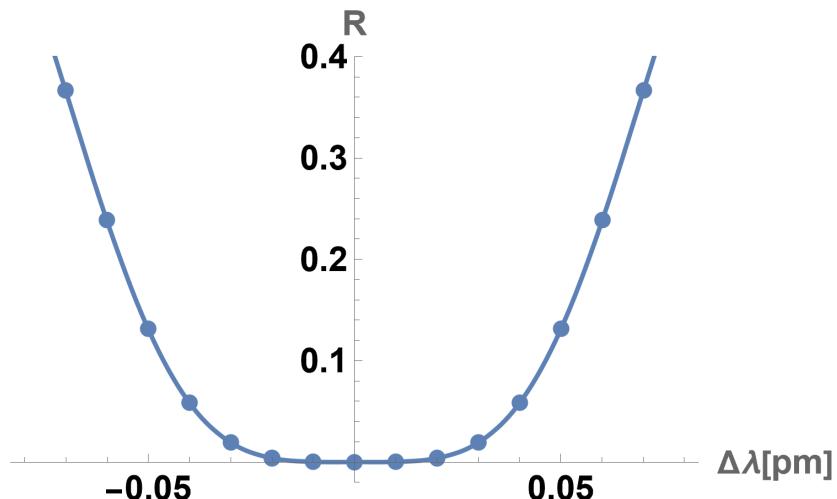


Figure 6.1: Reflectivity  $R$  of the EP-MAD-CPA as a function of detuning  $\Delta\lambda$  (pm).

Because the computation is parallelized across many subtasks and nodes, the rows are *not guaranteed to be ordered by the main index*: some wavelength evaluations finish earlier and are written sooner. The wavelength detunings produced by `get_sample_vec(...)` (line 145 in the source code) are *stacked symmetrically* about the center, i.e., points alternate to the left and right of the central resonance wavelength. Concurrent writes are handled safely by `.write_to_file(...)`, which serializes access so that simultaneous processes do not collide. After sorting the records by index, we plot the reflectivity versus detuning; see Fig. 6.1.

## 6.2 Generated Images

The script produces one input-field image `0_in.png` and a set of output-field images `0_out.png` through `16_out.png`. The input image is created in line 39 of the source code and shows the incident field; the output images are written in line 46.

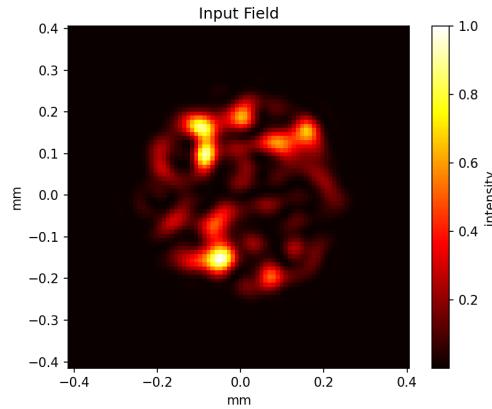


Figure 6.2: Incident field (`0_in.png`).

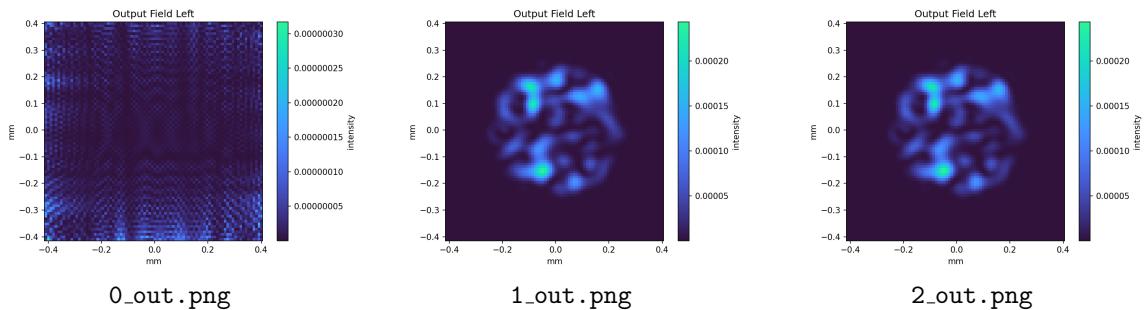


Figure 6.3: First three output fields placed side by side. Here, `0_out.png` corresponds to  $\Delta\lambda = 0$  pm, while `1_out.png` and `2_out.png` correspond to  $+0.01$  pm and  $-0.01$  pm, respectively.

# Chapter 7

## Component Overview

### 7.1 Light Field Classes

#### `clsLightField`

The class `clsLightField` is the central container class for optical fields in the library. Internally it stores a 2D NumPy array that may reside in field-of-view (FOV) or total resolution and in either position space or  $k$ -space; the current representation is hidden from the user. The class exposes a uniform interface so the field can be retrieved in any desired representation, manipulated (shift, stretch, add, clone, apply transmission/reflection matrices), plotted, and analysed (energy integrals, peak intensities, etc.).

#### Subclasses

Several subclasses inherit all functionality of `clsLightField` and can generate specific types of optical fields:

- `clsGaussBeam` – Generates a fundamental Gaussian beam with user-defined waist, curvature, and phase.
- `clsSpeckleField` – Produces a fully developed speckle pattern by superimposing random Fourier modes of equal amplitude and random phase.
- `clsTestImage` – Creates simple test patterns. Useful for algorithm verification.
- `clsPlaneWaveMixField` – Allows manual construction of a field by specifying an arbitrary set of plane-wave components (i.e. selected Fourier modes) with chosen amplitudes and phases.

### 7.2 Optical Components

Optical components (including free space propagation) are represented by sub-classes of `clsOptComponent`. They provide a modular toolkit for constructing complex optical systems. Two-port components live in a single spatial path and chain together in `clsCavity1path`, while four-port components couple two paths and are assembled in `clsCavity2path`. The

adapter class `clsOptComponentAdapter` bridges both worlds, ensuring that two-port elements can also be added to arms of a `clsCavity2path` instance.

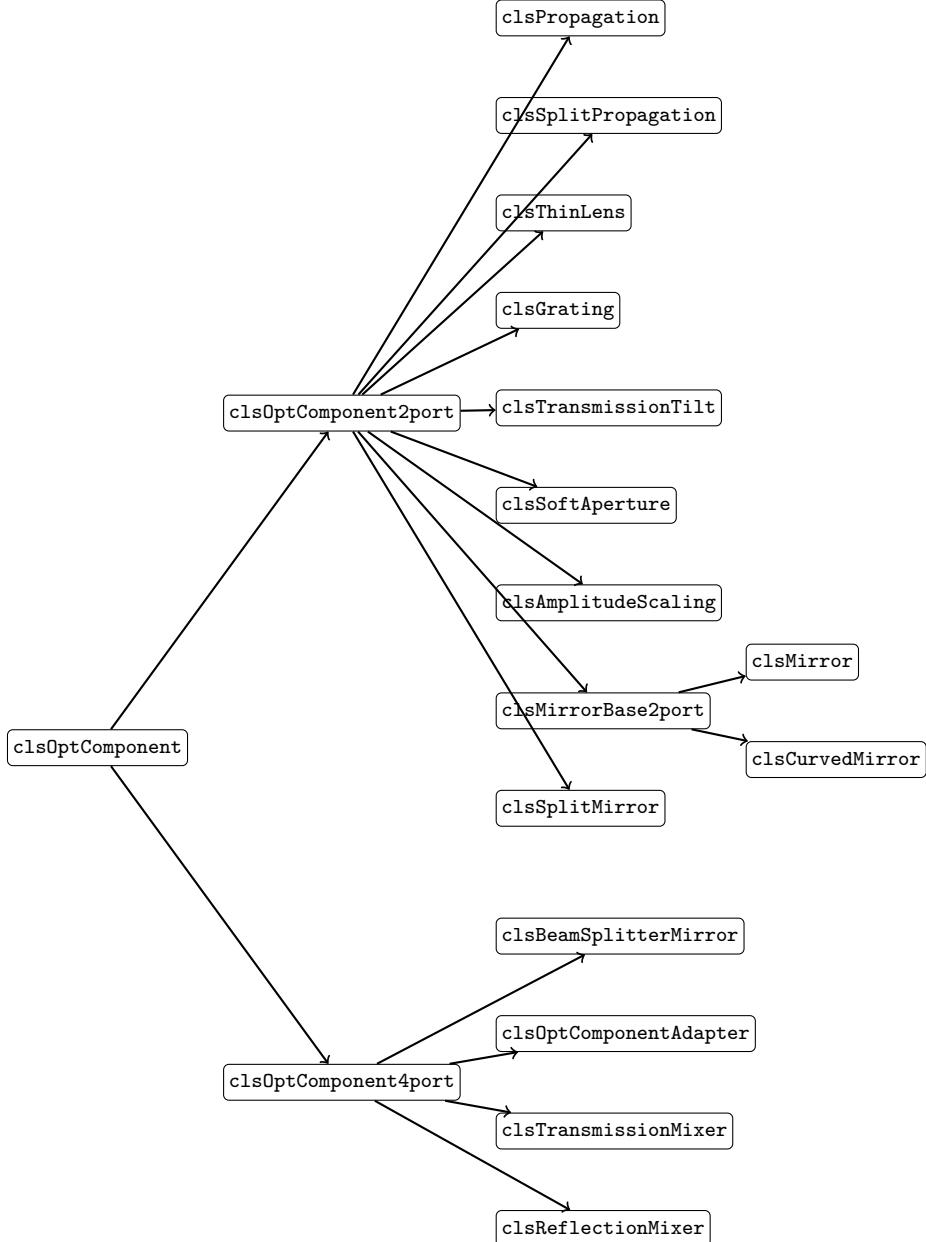


Figure 7.1: Class hierarchy of optical components. The root `clsOptComponent` splits into two branches: 2-port and 4-port components with their subclasses.

### 7.2.1 2-Port Component Classes

The majority of familiar elements (lenses, mirrors, gratings, soft apertures, etc.) can be modeled as *two-port* components: they possess a **left** side and a **right** side. A linear arrangement of such elements is built on top of `clsCavity1path`. This class acts like an optical “breadboard”: you may append components in the order they appear along a single beam path

using `.add_component(component)`. In this way one can model a single resonator, a chain of coupled cavities separated by partially reflective mirrors, or any other one-dimensional optical sequence.

- **`clsPropagation`**

The class `clsPropagation` models free-space or material propagation over a specified distance. There are two parameters to configure: The physical distance and the complex-valued refractive index  $n$ . The real part of  $n$  sets the phase velocity (optical path length), while the imaginary part models absorption (exponential decay).

Two transfer-function methods are available:

- **Rayleigh–Sommerfeld transfer function** (physically exact);
- **Fresnel transfer function** (paraxial approximation, valid for small angles)

- **`clsSplitPropagation`**

The class `clsSplitPropagation` models free-space or material propagation over a specified distance, with *different refractive indices in the top and bottom halves* of the optical field (relative to the  $y = 0$  plane). It behaves similarly to `clsPropagation`, but introduces spatial asymmetry in the propagation medium.

Separate complex-valued refractive indices  $n_1$  (for  $y > 0$ ) and  $n_2$  (for  $y < 0$ ) can be defined. This makes the class suitable for simulating situations where only part of the field propagates through an absorber or a different material (e.g. no absorber above the optical axis, absorber below).

As in `clsPropagation`, two transfer-function methods are available:

- **Rayleigh–Sommerfeld transfer function** (physically exact);
- **Fresnel transfer function** (paraxial approximation).

The propagation distance and refractive indices can be controlled independently. For the two field halves, different optical distances and absorption effects can therefore be realized within the same component.

- **`clsThinLens`**

The class `clsThinLens` models an idealised *thin lens* that may optionally include a circular pupil (aperture) and residual facet reflections.

**Key features:**

- **Optical parameters** – focal length, numerical-aperture-based pupil sizing, and controllable edge attenuation for residual reflections.
- **Lens type** – switch between a spherical thin-lens approximation and a perfect aspherical profile.
- **Residual reflections** – specify facet reflectivity and an aperture that smoothly fades the reflection to zero outside the clear pupil; reflection and transmission coefficients obey energy conservation.

- **clsGrating**

The class `clsGrating` models a *thin, spatially periodic grating*. The component multiplies the incident field in position space by a *grating mask* – either an **absorption grating** (periodic amplitude modulation) or a **phase grating** (periodic phase modulation).

**Key features**

- **1-D or 2-D cosine grating.** Periods and finite half-widths can be defined.
- **Absorption vs. phase grating.**
  - \* *Absorption grating:* The mask values oscillate between `min_val` and `max_val`, modelling a periodic attenuation profile.
  - \* *Phase grating:* First the same cosine profile is generated in the range `[min_val, max_val]`; this real-valued pattern is then interpreted as a phase  $\phi(x, y)$  and converted to a complex phase factor  $\exp(i\phi(x, y))$ . Hence `min_val` and `max_val` correspond to the minimum and maximum phase shift imparted by the grating.

- **clsTransmissionTilt**

The class `clsTransmissionTilt` represents an *ideal, loss-free phase element* that imposes a user-defined angular tilt on a light field by multiplying the field with separable phase masks in the  $x$ - and/or  $y$ -direction.

**Key features:**

- Independent tilt angles in  $x$ - and  $y$ -direction.
- Optional *zero lines* (`.x_zero_line`, `.y_zero_line`) that define where the phase ramp crosses zero.
- A `.direction` property that determines whether the tilt is applied for left-to-right (LTR) or right-to-left (RTL) propagation.

- **clsSoftAperture**

The class `clsSoftAperture` models an *ideal, thin amplitude element* that limits a light field by a circular aperture with a configurable soft transition at its rim. Unlike a hard-edged aperture, the soft aperture implements a smooth radial transmission profile.

**Key features:**

- Adjustable aperture diameter in meters (`.aperture`) or millimeters (`.aperture_mm`); both properties keep each other in sync.
- A *transition width* (`.transition_width`) that defines the  $1/e$ -width of the edge roll-off.
- A *black value* (`.black_value`) that sets the residual transmission in the fully blocked region (0 means fully opaque, 1 means fully transparent).

- **clsAmplitudeScaling**

The class `clsAmplitudeScaling` models an ideal thin optical element that uniformly scales the amplitude of the light field passing through it.

- **clsMirror**

The class `clsMirror` represents an *ideal, flat, thin* mirror with arbitrary power reflectivity and transmissivity. It derives from `clsMirrorBase2port`, which in turn derives from `clsOptComponent2port`. Consequently, `clsMirror` inherits every method and property documented for those base classes – most notably the full set of reflection/transmission coefficients (`.R`, `.T`, `.r_L`, `.t_LTR`, etc.) and the rich machinery for mirror tilt, incidence angle, projection factors, and “transmission-behaves-like-reflection” modes. The class also offers optional rotation about the *x*- and *y*-axes. This functionality is inherited from `clsMirrorBase2port`.

- **clsCurvedMirror**

The class `clsCurvedMirror` represents an ideal thin *curved* mirror (partially transparent or fully reflective) and derives from `clsMirrorBase2port`, which itself derives from `clsOptComponent2port`. It therefore inherits all generic two-port functionality (reflectivity/transmissivity coefficients, mirror-tilt handling, “Transmission-Behaves-Like-Reflection” flags, projection/astigmatism logic, ...) and adds the following curved-mirror-specific features:

- **Independent curvature on each facet.** Separate focal lengths `.f_R_L` / `.f_R_R` allow convex, concave, or flat surfaces on either side.
- **Spherical vs. “perfect” (paraxially exact) phase profiles.** The shape selector `.mirror_type_spherical` / `.mirror_type_perfect` chooses between a quadratic phase mask (spherical mirror approximation) and the exact hyperbolic phase appropriate for a perfect focusing surface.
- **Lens action for transmitted light.** A partially transparent curved mirror behaves like a thin lens with finite absorption. When both facets participate in transmission, their curvatures together with the substrate refractive index `.n` determine an effective focal length `.f_T`.
- **Consistent integration with mirror tilt/projection/astigmatism framework.** All curved-surface masks, tilt masks, and projection factors are combined in the correct sequence for both direct reflection and “reflection-via-transmission” emulation modes.
- **Optional rotation about the *x*- and *y*-axes.** This functionality is inherited from `clsMirrorBase2port`.

- **clsSplitMirror**

The class `clsSplitMirror` represents a *flat, thin* mirror whose optical behaviour is **different above and below the horizontal mid-plane** ( $y = 0$ ). The upper half of the aperture is characterized by the power coefficients  $(R_1, T_1)$  and the lower half by  $(R_2, T_2)$ .

#### Key features

- Independent reflectivity / transmissivity in the *upper* ( $y > 0$ ) and *lower* ( $y < 0$ ) half-planes.
- Optional rotation about the *x*- and *y*-axes.

### 7.2.2 4-Port Component Classes

Beamsplitters and similar devices require a richer abstraction: a *four-port* component has two physical faces, each of which couples *two* distinct beam paths. The class `clsCavity2path` allows you to stack such four-port components – e.g. multiple beamsplitter mirrors – to form interferometers or coupled resonator networks.

Crucially, `clsCavity2path` can also host ordinary two-port elements using the class `clsOptComponentAdapter`: a lens or mirror can e.g. be placed in one (or both) arms of an interferometer without breaking the four-port formalism.

- **`clsBeamSplitterMirror`**

The class `clsBeamSplitterMirror` models an ideal, non-polarising beam-splitter. It derives from `clsOptComponent4port`, which itself extends the common base class `clsOptComponent`; therefore, every method and property defined for those super-classes is available here without modification. Reflection always swaps the two spatial paths *A* and *B* while transmission never mixes them. Consequently, the full  $4 \times 4$  scattering matrix contains only eight non-zero sub-matrices.

- **`clsOptComponentAdapter`**

The class `clsOptComponentAdapter` converts one or two *two-port* elements into a fully fledged *four-port* component by plugging them into the individual spatial paths (*A* and *B*) of the 4-port model. The class derives from `clsOptComponent4port` and therefore inherits the complete scattering/transfer-matrix interface.

Using `clsOptComponentAdapter` you can insert arbitrary chains of two-port components between genuine four-port elements such as the beam-splitter. For every pair (or single) of two-port devices attached to paths *A* and *B*, the adapter constructs a block-diagonal  $4 \times 4$  scattering matrix. Each section of the cavity - regardless of whether it originated as a two-port or four-port element - thus exposes a uniform four-port interface, enabling `clsCavity2path` to treat all segments in exactly the same way.

- **`clsTransmissionMixer`**

The class `clsTransmissionMixer` provides a four-port component that *mixes* the two spatial paths without any reflection. It is conceptually the transmission-only counterpart of a beam-splitter.

A common application is illustrated in Fig. 5.1. In this ring cavity, the blue beam-splitter on the top-left is modeled by `clsBeamSplitterMirror`, which feeds horizontal incident light into path *A* (top arm), and vertical incident light into path *B* (left arm). However, the two black  $90^\circ$  turning mirrors on the top-right and bottom-left side merely redirect the beams while keeping their logical path labels unchanged. Physically they behave like reflections, but in the simulation they must appear as *unit-transmission, phase-shifting* elements. Assigning a `clsTransmissionMixer` with `.refl_behavior_for_path_mixing=False` to simulate both of these corners accomplishes exactly that: Same-path channels ( $A \rightarrow A$ ,  $B \rightarrow B$ ) receive the reflection-like phase, while cross-path channels ( $A \rightarrow B$ ,  $B \rightarrow A$ ) remain zero. Note that a `clsBeamSplitterMirror` would mix paths *A* and *B* via its off-diagonal reflection blocks, which is not what the corner mirrors do; hence we use `clsTransmissionMixer` instead.

- **clsReflectionMixer**

The class `clsReflectionMixer` is the reflection-only counterpart to the class `clsTransmissionMixer`. Where the latter couples the two spatial paths purely through transmission, `clsReflectionMixer` couples them exclusively through reflection. Like its transmission-only sibling, `clsReflectionMixer` inherits the full four-port interface from `clsOptComponent4port`, providing on-demand scattering and transfer block matrices through the inherited properties.



# Appendix A

## Helper Functions and Classes

### A.1 Enumeration Types

A few simple `Enum` types are defined and used throughout the library to make the code more readable and expressive.

#### Usage

These enums are typically used as argument values in method calls. They improve code readability and avoid the use of "magic numbers."

#### Import

The `Enum` types are defined in the main library module and can be imported as:

```
from fourier_cavity_sim import Dir, Dir2, Side, Res, Path
```

#### Dir

Propagation direction (single-direction):

- `Dir.LTR` – left-to-right
- `Dir.RTL` – right-to-left

#### Dir2

Propagation direction (allowing both directions):

- `Dir2.BOTH` – both directions
- `Dir2.LTR` – left-to-right
- `Dir2.RTL` – right-to-left

**Side**

Lateral side:

- `Side.LEFT` – left side
- `Side.RIGHT` – right side

**Res**

Resolution selector:

- `Res.FOV` – field-of-view resolution
- `Res.TOT` – total resolution

**Path**

Arm selection in 4-port cavities:

- `Path.A` – path A (e.g. horizontal)
- `Path.B` – path B (e.g. vertical)

## A.2 Helper Functions

The library also provides a few standalone helper functions that simplify common simulation tasks. These are not methods associated with specific classes, but independent functions.

```
get_sample_vec(NoOfPoints, LR_range, center, stacked, p=1)
```

### Function.

Returns a vector of `NoOfPoints` sampling points distributed around a given `center` value, with density controlled by parameter `p`. The returned points lie in the interval  $[center - LR\_range, center + LR\_range]$ , with higher density around the center for larger `p`.

### Parameters

- `NoOfPoints` – Total number of points to generate. If even, automatically rounded up to the next odd number.
- `LR_range` – Range of sampling interval around the center (half-width of the interval).
- `center` – Center of the sampling interval.
- `stacked` – If `True`, the returned vector is reordered so that points alternate symmetrically around the center.
- `p` – Controls how densely points are clustered near the center. `p=1` gives linear spacing; larger values cluster more strongly around the center.

**Returns**

- `numpy.ndarray` – Vector of sampling points.

**Typical use case**

Useful for parameter sweeps where fine resolution is desired around a specific point – for example, scanning wavelength or cavity length near resonance.

```
delta_f_to_delta_lambda(delta_f_hz, lambda_center_m, n=1.0)
```

**Function.**

Converts a frequency spacing  $\Delta f$  (in Hz) to an equivalent wavelength spacing  $\Delta\lambda$  (in meters), using the specified center wavelength  $\lambda$  instead of center frequency.

**Parameters**

- `delta_f_hz` – Frequency spacing  $\Delta f$  in Hz.
- `lambda_center_m` – Center wavelength  $\lambda$  in meters.
- `n` – Refractive index of the medium (default is 1.0 for vacuum).

**Returns**

- `float` – Corresponding wavelength spacing  $\Delta\lambda$  in meters.

**Typical use case**

Allows converting between spectral resolutions expressed in frequency units and in wavelength units – for example, when mapping between a cavity's free spectral range (FSR) in Hz and its corresponding  $\Delta\lambda$  spacing near a given resonance.

```
polar_interpolation(z1, z2, w1=0.5)
```

**Function.**

Performs a weighted interpolation between two complex numbers, interpolating both amplitude and phase in polar form.

**Parameters**

- `z1` – First complex number.
- `z2` – Second complex number.
- `w1` – Weight for `z1` (default 0.5). The second weight is implicitly  $w_2 = 1 - w_1$ .

**Returns**

- `complex` – Interpolated complex value.

**How it works**

Interpolates the amplitudes of `z1` and `z2` linearly. Interpolates the phases by computing a weighted circular mean of the two input phases, ensuring correct interpolation even across phase discontinuities.

**Typical use case**

Useful when interpolating fields, mode amplitudes, or other complex-valued quantities

where both amplitude and phase must be treated properly in a physically meaningful way.

### `find_best_match(ax, x)`

#### Function.

Finds and returns the element in vector `ax` that is closest to the given target value `x`.

#### Parameters

- `ax` – 1D array-like object (vector of values).
- `x` – Target value to match.

#### Returns

- Same type as elements of `ax` – The element of `ax` whose value is closest to `x`.

#### Typical use case

Useful for snapping continuous values to the nearest available grid point, discrete parameter value, or measurement sample – for example, selecting the closest valid mode number or wavelength in a predefined set.

## A.3 Matrix Representation and Operations

In this library, matrix objects are used to represent optical transfer, reflection, or propagation matrices. To minimise memory usage and optimize performance, the following compact representations are supported:

- **Full matrix:** Represented as a standard dense `numpy.ndarray` of shape  $(N, N)$ .
- **Diagonal matrix:** Represented as a sparse diagonal `spmatrix` (SciPy sparse matrix).
- **Scalar:** If a scalar value (e.g. 0, 1, or any other number) is used, it is interpreted as the scalar times the identity matrix.

#### Purpose

This flexible convention allows efficient storage and manipulation of matrices arising in large-scale simulations, where most matrices are diagonal or simple scalings of the identity.

#### Supported operations

To ensure consistent behaviour across all combinations of matrix types, the library provides a set of special functions for matrix operations:

- Addition and subtraction
- Matrix Multiplication and Division
- Inversion
- etc.

These functions automatically handle all valid combinations of the above representations, allowing users to write concise and efficient simulation code without having to explicitly convert between dense, sparse, or scalar forms.

### `is_matrix(X)`

#### **Function.**

Distinguishes between an explicit matrix (full or sparse) and a scalar. Returns `True` if `X` is represented as a full matrix (`numpy.ndarray`) or a sparse diagonal matrix (`spmatrix`); returns `False` if `X` is a scalar or any other type.

#### **Parameters**

- `X` – Object to check.

#### **Returns**

- `bool` – `True` if `X` is a full or sparse matrix; `False` if `X` is a scalar or other non-matrix object.

#### **Typical use case**

Used internally and in user code to detect whether an object represents an explicit matrix (requiring full matrix handling) or a scalar (to be interpreted as scalar times identity).

### `mat_is_zero(X)`

#### **Function.**

Checks whether `X` is identically zero. Works for full matrices, diagonal sparse matrices, and scalars.

#### **Parameters**

- `X` – Matrix or scalar.

#### **Returns**

- `bool` – `True` if `X` is zero; `False` otherwise.

#### **Behavior**

- If `X` is a full matrix (`numpy.ndarray`): returns `True` if all entries are zero.
- If `X` is a sparse diagonal matrix (`spmatrix`): returns `True` if all diagonal entries are zero.
- If `X` is a scalar: returns `True` if the scalar is zero.

#### **Typical use case**

Used to efficiently test whether a matrix (or scalar) is zero, independent of its representation.

**`mat_plus(X, Y)`****Function.**

Adds two matrices  $X$  and  $Y$ , allowing either or both arguments to be scalars. If either argument is a scalar, it is interpreted as a scalar times the identity matrix.

**Parameters**

- $X$  – First matrix or scalar.
- $Y$  – Second matrix or scalar.

**Returns**

- `numpy.ndarray` or `spmatrix` or scalar – Sum of  $X$  and  $Y$ , with scalars promoted to identity matrices as needed.

**Behavior**

- If both  $X$  and  $Y$  are matrices: returns their sum.
- If only  $X$  is a matrix and  $Y$  is a scalar: returns  $X + 1 \cdot Y$ .
- If only  $Y$  is a matrix and  $X$  is a scalar: returns  $1 \cdot X + Y$ .
- If both  $X$  and  $Y$  are scalars: returns  $X + Y$ .

**Typical use case**

Provides a uniform interface for matrix addition in the library's flexible matrix representation scheme, handling scalars and matrices transparently.

**`mat_plus3(X, Y, Z)`****Function.**

Adds three matrices  $X$ ,  $Y$ , and  $Z$ , allowing any of them to be scalars. If a scalar is given, it is interpreted as a scalar times the identity matrix.

**Parameters**

- $X$  – Matrix or scalar.
- $Y$  – Matrix or scalar.
- $Z$  – Matrix or scalar.

**Returns**

- `numpy.ndarray` or `spmatrix` or scalar – Result of  $X + Y + Z$ , with scalars promoted to identity matrices as needed.

**Behavior**

- Any combination of scalars and matrices is supported.
- The operation is performed by recursively applying `mat_plus(...)`.

**Typical use case**

Convenience function for adding three matrices or scalars in one call, avoiding nested explicit calls to `mat_plus(...)`.

**mat\_minus(X, Y)****Function.**

Subtracts two matrices X and Y, allowing either or both arguments to be scalars. If either argument is a scalar, it is interpreted as a scalar times the identity matrix.

**Parameters**

- X – First matrix or scalar.
- Y – Second matrix or scalar.

**Returns**

- `numpy.ndarray` or `spmatrix` or scalar – Result of  $X - Y$ , with scalars promoted to identity matrices as needed.

**Behavior**

- If both X and Y are matrices: returns their difference.
- If only X is a matrix and Y is a scalar: returns  $X - 1 \cdot Y$ .
- If only Y is a matrix and X is a scalar: returns  $1 \cdot X - Y$ .
- If both X and Y are scalars: returns  $X - Y$ .

**Typical use case**

Provides a uniform interface for matrix subtraction in the library's flexible matrix representation scheme, handling scalars and matrices transparently.

**mat\_mul(X, Y)****Function.**

Performs matrix multiplication  $X \cdot Y$ , allowing either or both arguments to be scalars. If a scalar is given, it is interpreted as a scalar times the identity matrix.

**Parameters**

- X – Matrix or scalar.
- Y – Matrix or scalar.

**Returns**

- `numpy.ndarray` or `spmatrix` or scalar – Result of  $X \cdot Y$ , with scalars promoted to identity matrices as needed.

**Behavior**

- If both X and Y are matrices: returns their matrix product.
- If only one of X or Y is a scalar: returns scalar multiplication of the matrix.
- If both X and Y are scalars: returns scalar product.

**Typical use case**

Provides a uniform interface for matrix multiplication in the library's flexible matrix representation scheme, handling scalars and matrices transparently.

**`mat_mul3(X, Y, Z)`****Function.**

Performs matrix multiplication  $X \cdot Y \cdot Z$ , allowing any of the arguments to be scalars. If a scalar is given, it is interpreted as a scalar times the identity matrix.

**Parameters**

- $X$  – Matrix or scalar.
- $Y$  – Matrix or scalar.
- $Z$  – Matrix or scalar.

**Returns**

- `numpy.ndarray` or `spmatrix` or scalar – Result of  $X \cdot Y \cdot Z$ , with scalars promoted to identity matrices as needed.

**Behavior**

- Any combination of scalars and matrices is supported.
- The operation is performed by recursively applying `mat_mul(...)`.

**Typical use case**

Convenience function for multiplying three matrices or scalars in one call, avoiding nested explicit calls to `mat_mul(...)`.

**`mat_div(X, Y)`****Function.**

Performs matrix division  $X/Y$ , allowing either or both arguments to be scalars. Matrix division is implemented as  $X \cdot Y^{-1}$ .

**Parameters**

- $X$  – Matrix or scalar (numerator).
- $Y$  – Matrix or scalar (denominator).

**Returns**

- `numpy.ndarray` or `spmatrix` or scalar – Result of  $X/Y$ , with scalars and matrices handled transparently.

**Behavior**

- If both  $X$  and  $Y$  are dense matrices: uses `numpy.linalg.solve` to compute  $X \cdot Y^{-1}$  efficiently.
- If either  $X$  or  $Y$  is a sparse (diagonal) matrix: uses `mat_inv(...)` and `mat_mul(...)`.
- If one or both arguments are scalars: treated as scalar times identity matrix, with appropriate promotion.

**Typical use case**

Provides a uniform interface for matrix division in the library's flexible matrix repre-

smentation scheme, handling scalars and matrices transparently.

### `mat_inv(X)`

#### **Function.**

Computes the inverse of  $X$ . Works for full matrices, diagonal sparse matrices, and scalars.

#### **Parameters**

- $X$  – Matrix or scalar to invert.

#### **Returns**

- `numpy.ndarray` or `spmatrix` or scalar – Inverse of  $X$ .

#### **Behavior**

- If  $X$  is a full matrix (`numpy.ndarray`): returns the dense inverse computed via `numpy.linalg.inv`.
- If  $X$  is a sparse diagonal matrix (`spmatrix`): returns a new sparse diagonal matrix with inverted diagonal entries.
- If  $X$  is a scalar: returns the reciprocal  $1/X$ .

**Typical use case** Provides a uniform interface for matrix inversion in the library's flexible matrix representation scheme, handling scalars and matrices transparently.

### `mat_inv_X_mul_Y(X, Y)`

#### **Function.**

Efficiently computes  $X^{-1} \cdot Y$ , the product of the inverse of  $X$  with  $Y$ . Uses an optimised solver when both arguments are dense matrices.

#### **Parameters**

- $X$  – Matrix or scalar to invert.
- $Y$  – Matrix or scalar to multiply with `inv(X)`.

#### **Returns**

- `numpy.ndarray` or `spmatrix` or scalar – Result of `inv(X) · Y`, with scalars and matrices handled transparently.

#### **Behavior**

- If both  $X$  and  $Y$  are dense matrices: uses `numpy.linalg.solve` for efficiency and numerical stability.
- Otherwise: computes `mat_inv(X)` followed by `mat_mul` with  $Y$ .

#### **Typical use case**

Provides an efficient and robust way to compute `inv(X) · Y`, especially when  $X$  is a large dense matrix and explicit inversion should be avoided.

**`mat_conj_transpose(X)`****Function.**

Computes the conjugate transpose (Hermitian adjoint) of `X`. Works for full matrices, diagonal sparse matrices, and scalars.

**Parameters**

- `X` – Matrix or scalar.

**Returns**

- `numpy.ndarray` or `spmatrix` or scalar – Conjugate transpose of `X`.

**Behavior**

- If `X` is a full matrix (`numpy.ndarray`): returns  $X^\dagger = (X^T)^*$ , the conjugate transpose.
- If `X` is a sparse diagonal matrix (`spmatrix`): returns a sparse diagonal matrix with conjugated entries (transpose is trivial).
- If `X` is a scalar: returns its complex conjugate.

**Typical use case**

Provides a uniform interface for computing the conjugate transpose in the library's flexible matrix representation scheme, handling scalars and matrices transparently.

## A.4 Class `clsBlockMatrix`

In our library, the scattering and transfer matrices for linear cavities are represented as  $2 \times 2$  block matrices, and for ring cavities even as  $4 \times 4$  block matrices. To handle the resulting memory demand efficiently, we introduce the class `clsBlockMatrix`.

This class provides two key features: First, if enabled, each block of the matrix is transparently stored in a temporary file and only loaded into memory when accessed, significantly reducing the memory footprint. Second, the internal representation of each block is optimized based on its structure:

- A general (dense) block is stored as a `numpy.ndarray`.
- A diagonal block is stored as a `scipy.sparse` matrix of type `spmatrix`.
- A block that is identically zero, an identity matrix, or a diagonal matrix with constant entries can be represented by a simple scalar.

### A.4.1 Initialization

**`clsBlockMatrix(dimension, file_caching, tmp_dir="", name "")`****Constructor.**

This constructor initializes a new instance of the `clsBlockMatrix` class.

- **`dimension`**

Must be either 2 or 4, corresponding to a  $2 \times 2$  or  $4 \times 4$  block matrix, respectively.

- **`file_caching`**

Boolean flag that enables or disables file-based caching of the blocks. If `True`, each non-scalar block is automatically written to a temporary file, which persists for the lifetime of the instance. This significantly reduces RAM usage – for example, a full  $4 \times 4$  block matrix uses only about  $\frac{1}{16}$  of the memory compared to in-memory storage.

- **`tmp_dir`**

Optional. A string specifying the directory in which temporary block files are stored. If not provided, the files are created in the current working directory.

- **`name`**

Optional. A string that assigns a human-readable name to the instance. This can be useful for debugging or logging purposes.

#### A.4.2 Setter and Getter Methods

`.set_block(line, col, X, clone=False)`

**Method.**

This method assigns a block to the specified position in the matrix.

- **`line, col`**

Zero-based indices specifying the row and column of the block. For example, in a  $2 \times 2$  matrix, valid values are 0 or 1.

- **`X`**

The block to be inserted. This can be a NumPy array, a sparse matrix, or a scalar. All of these may contain complex values.

- **`clone`**

This parameter is only relevant if `file_caching` is disabled. If `clone=True`, the block `X` is copied into the internal RAM storage of the instance. If `False`, only a reference to `X` is stored.

`.get_block(line, col)`

**Method.**

This method returns the block at the specified position in the matrix.

- **`line, col`**

Zero-based indices specifying the row and column of the desired block.

- **Return value**

The returned object can be a NumPy array, a sparse matrix, or a scalar. Complex-valued entries are supported in all cases.

`.set_quadrant(quadrant, X, clone=False)`

**Method.**

This method assigns a  $2 \times 2$  block matrix `X` to one of the four quadrants of a  $4 \times 4$  block matrix.

- `quadrant`

An integer from 1 to 4, specifying which quadrant is to be set:

- 1 – top left
- 2 – top right
- 3 – bottom left
- 4 – bottom right

- `X`

A  $2 \times 2$  block matrix consisting of NumPy arrays, sparse matrices, or scalars. These are copied into the corresponding blocks of the selected quadrant. Complex-valued entries are fully supported.

- `clone`

Controls how data is stored, depending on the file caching mode:

- If `file_caching=False` and `clone=True`, all blocks in `X` are fully copied into RAM.
- If `file_caching=True` and `clone=False`, the blocks are shared via references to the existing temporary files.

This method can also be used with a  $2 \times 2$  block matrix. In that case, `X` must be a single block, and the function behaves identically to `set_block`. It does not offer any additional functionality in this case.

`.get_quadrant(quadrant, clone=False)`

**Method.**

This method returns one of the four quadrants of a  $4 \times 4$  block matrix as a separate  $2 \times 2$  block matrix instance.

- `quadrant`

An integer from 1 to 4, specifying which quadrant to extract:

- 1 – top left
- 2 – top right
- 3 – bottom left
- 4 – bottom right

This function can only be used with  $4 \times 4$  block matrices.

- `clone`

Controls how the block data is returned:

- If `file_caching=False` and `clone=True`, the data is fully copied into the returned instance.

- If `file_caching=True` and `clone=False`, the returned instance shares references to the existing temporary files.

```
.set_from_quadrants(A, B, C, D, clone=False)
```

### Method.

This method sets the content of the current matrix using the four quadrant inputs `A`, `B`, `C`, and `D`. The interpretation depends on the size of the matrix:

- **$4 \times 4$  block matrix:**

`A`, `B`, `C`, and `D` must each be  $2 \times 2$  block matrix instances. They are assigned to the four quadrants as follows:

- `A` – top left
- `B` – top right
- `C` – bottom left
- `D` – bottom right

- **$2 \times 2$  block matrix:**

`A`, `B`, `C`, and `D` must be individual blocks (NumPy arrays, sparse matrices, or scalars), which are assigned to the four positions of the matrix in the same order.

`clone` behaves as in other functions:

- If `file_caching=False` and `clone=True`, the block data is fully copied into memory.
- If `file_caching=True` and `clone=False`, existing temporary files are shared with the input data.

```
.clone(X)
```

### Method.

This method copies all blocks from another matrix `X` into the current instance. After calling `Y.clone(X)`, the matrix `Y` contains the same values as `X`, but is fully independent – modifying one will not affect the other. This operation is only valid if both matrices have the same block dimensions (e.g., both are  $2 \times 2$  or both are  $4 \times 4$ ).

```
.clear()
```

### Method.

This method clears all data associated with the block matrix instance. It is automatically called when the instance is destroyed.

- All references to NumPy arrays and sparse matrices held in RAM are deleted.
- If file caching is active, all associated temporary files are also deleted – with the following exceptions to avoid premature deletion of shared resources:
  1. The current  $2 \times 2$  block matrix was created using `.get_quadrant(...)` from a  $4 \times 4$  block matrix.

2. The current  $2 \times 2$  block matrix was used to populate a quadrant of a  $4 \times 4$  matrix via `.set_quadrant(...)`.
3. The flags `.keep_tmp_files = True` or `.keep_alive= True` were manually set on the instance.

These safeguards ensure that temporary files shared between multiple instances are not deleted prematurely.

### A.4.3 Test if Zero

`.is_zero()`

#### Method.

This method checks whether all blocks in the matrix are identically zero.

#### Return value

Returns `True` if every block in the matrix is zero (including scalar, NumPy array, or sparse matrix blocks); otherwise, returns `False`.

`.block_is_zero(line, col)`

#### Method.

This method checks whether a specific block in the matrix is identically zero.

- `line, col`

Zero-based indices specifying the position of the block to be tested.

#### Return value

Returns `True` if the specified block is zero (regardless of whether it is a scalar, NumPy array, or sparse matrix); otherwise, returns `False`.

### A.4.4 Helper Properties and Methods

`.file_caching`

This read-only property indicates whether file caching is enabled for the instance.

#### Return value

Returns `True` if file caching was enabled during initialization, and `False` otherwise.

`.tmp_dir`

#### Read-only property.

This read-only property returns the path to the directory used for temporary storage of block files when file caching is enabled.

#### Return value:

A string representing the directory path where temporary files are stored. If an explicit path was provided during initialization, that path is returned. If an empty string was

given, the current working directory is used.

### `.dim`

#### **Read-only property.**

This read-only property returns the dimension of the block matrix.

#### **Return value**

Returns 2 for a  $2 \times 2$  block matrix and 4 for a  $4 \times 4$  block matrix.

### `.file_names`

#### **Read-only property.**

This read-only property returns the list of file paths corresponding to the temporary files associated with each block.

#### **Return value**

A list of 4 or 16 strings, depending on the block matrix size. Each entry contains the full path of the temporary file used for caching the corresponding block, if file caching is enabled. If a specific temporary directory was provided during initialization via `tmp_dir`, the returned file paths include that directory.

### `.is_saved_in_tmp_file(i, j)`

#### **Method.**

This method checks whether the specified block is currently stored in a temporary file or in RAM.

- `i, j`

Zero-based row and column indices of the block to be checked.

#### **Return value**

Returns `True` if the block at position `(i, j)` is stored in a temporary file, and `False` if it is stored in RAM.

### `.set_tmp_file_name(i, j, name)`

#### **Method.**

This method assigns a custom file name to the temporary file used for caching the block at position `(i, j)`.

- `i, j`

Zero-based row and column indices of the block.

- `name`

A string specifying the full file name (including path, if desired) to be used for storing the block. This overrides any automatically generated file name.

**Note**

This method has no effect unless file caching is active *and* the specified block is not a scalar value.

`.get_tmp_file_name(i, j)`

**Method.**

This method returns the file name currently assigned to the temporary file used for caching the block at position `(i, j)`.

- `i, j`

Zero-based row and column indices of the block.

**Return value**

A string containing the full file path used for caching the block. If file caching is not active, the return value is `None`.

**Note**

The returned file name is the path assigned for caching, but it does not imply that the file currently exists. The file is only created if and when the block is actually written to disk.

`.keep_tmp_files`

**Read/write property.**

This property can be used to prevent automatic deletion of temporary files associated with this instance.

**Type**

Boolean. Defaults to `False`.

**Effect**

If set to `True`, temporary files are preserved when the instance is cleared or destroyed.

**See also**

See also `.clear()` for details on the conditions under which temporary files are deleted.

`.keep_alive`

**Read/write property.**

This property can be used to prevent automatic deletion of temporary files when the instance is destroyed.

**Type**

Boolean. Defaults to `False`.

**Effect**

If set to `True`, temporary files will be preserved even when the instance is cleared or destroyed.

**Note**

This behaves similarly to `.keep_tmp_files`, and both flags are checked in `.clear()` before temporary files are deleted.

#### A.4.5 Global Functions for Processing Block Matrices

`bmat2_plus(X, Y)`

**Function.**

This function computes the sum of two  $2 \times 2$  block matrices.

- X, Y

Two block matrix instances of dimension  $2 \times 2$ . The matrices may use different storage modes (RAM or file-cached).

**Return value:**

A new  $2 \times 2$  block matrix representing the result of  $X + Y$ .

`bmat2_minus(X, Y)`

**Function.**

This function computes the difference between two  $2 \times 2$  block matrices.

- X, Y

Two block matrix instances of dimension  $2 \times 2$ . The matrices may use different storage modes (RAM or file-cached).

**Return value:**

A new  $2 \times 2$  block matrix representing the result of  $X - Y$ .

`bmat_flip_sign(X)`

**Function.**

This function flips the sign of all blocks in a block matrix.

- X

A block matrix of dimension  $2 \times 2$  or  $4 \times 4$ .

**Return value:**

A new block matrix of the same dimension as X, with each block is multiplied by  $-1$ .

`bmat_mul(X, Y, p=None, msg="")`

**Function.**

This function computes the matrix product of two block matrices of the same dimension ( $2 \times 2$  or  $4 \times 4$ ) and returns the result.

- X, Y

Block matrix instances of the same dimension. They may use different storage modes. The storage mode of the output is inherited from `X`.

- `p`  
Optional. If set to an instance of `clsProgressPrinter`, progress output is generated during the matrix multiplication – useful for very large block matrices.
- `msg`  
Optional. A string that will be printed along with the progress output (only if `p` is provided). If `msg` is an empty string (default), no message is printed.

**Return value:**

A new block matrix representing the matrix product `X * Y`.

The function implements the following algorithm for the multiplication of two  $n \times n$  block matrices `X` and `Y`:

$$(\mathbf{XY})_{ij} = \sum_{k=1}^n \mathbf{X}_{ik} \mathbf{Y}_{kj}$$

`bmat_mul3(X, Y, Z, p=None, msg="")`

**Function.**

This function computes the matrix product of three block matrices of the same dimension ( $2 \times 2$  or  $4 \times 4$ ) and returns the result.

- `X, Y, Z`  
Block matrix instances of the same dimension. They may use different storage modes. The storage mode of the output is inherited from `X`.
- `p`  
Optional. If set to an instance of `clsProgressPrinter`, progress output is generated during the multiplication – useful for very large block matrices.
- `msg`  
Optional. A string that will be printed together with the progress output (only if `p` is provided). If `msg` is an empty string (default), no message is printed.

**Return value:**

A new block matrix representing the matrix product `X * Y * Z`.

`bmat2_inv(X, p=None, msg="")`

**Function.**

This function computes the matrix inverse of a  $2 \times 2$  block matrix.

- `X`  
A  $2 \times 2$  block matrix. The blocks must be invertible and compatible for matrix inversion.
- `p`  
Optional. If set to an instance of `clsProgressPrinter`, progress output is generated during the computation.

- `msg`

Optional. A string that will be printed together with the progress output (only if `p` is provided). If `msg` is an empty string (default), no message is printed.

#### Return value:

A new  $2 \times 2$  block matrix representing the matrix inverse of  $\mathbf{X}$ .

Following [5, p. 184] and [6], the matrix inversion is performed as follows: Let  $\mathbf{X}$  be a  $2 \times 2$  block matrix with the components  $\mathbf{X}_{11} = \mathbf{A}$ ,  $\mathbf{X}_{12} = \mathbf{B}$ ,  $\mathbf{X}_{21} = \mathbf{C}$ , and  $\mathbf{X}_{22} = \mathbf{D}$ , each being matrices themselves. Then

$$\mathbf{X}^{-1} = \begin{cases} \begin{pmatrix} \mathbf{0} & \mathbf{C}^{-1} \\ \mathbf{B}^{-1} & \mathbf{0} \end{pmatrix}, & \text{for } \mathbf{A} = \mathbf{0} \wedge \mathbf{D} = \mathbf{0} \\ \begin{pmatrix} \mathbf{A}^{-1} & \mathbf{0} \\ \mathbf{0} & \mathbf{D}^{-1} \end{pmatrix}, & \text{for } \mathbf{B} = \mathbf{0} \wedge \mathbf{C} = \mathbf{0} \\ \begin{pmatrix} (\mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1} & -\mathbf{A}^{-1}\mathbf{B}(\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1} \\ -\mathbf{D}^{-1}\mathbf{C}(\mathbf{A} - \mathbf{B}\mathbf{D}^{-1}\mathbf{C})^{-1} & (\mathbf{D} - \mathbf{C}\mathbf{A}^{-1}\mathbf{B})^{-1} \end{pmatrix}, & \text{for } \mathbf{A}, \mathbf{D} \text{ nonsingular} \end{cases}$$

`bmat4_inv(X, p=None, msg="")`

#### Function.

This function computes the matrix inverse of a  $4 \times 4$  block matrix.

- `X`

A  $4 \times 4$  block matrix. The blocks must be invertible and compatible for matrix inversion.

- `p`

Optional. If set to an instance of `clsProgressPrinter`, progress output is generated during the computation.

- `msg`

Optional. A string that will be printed together with the progress output (only if `p` is provided). If `msg` is an empty string (default), no message is printed.

#### Return value:

A new  $4 \times 4$  block matrix representing the matrix inverse of  $\mathbf{X}$ .

The matrix inversion is performed with the same algorithm as described in `bmat2_inv(...)`, but with  $\mathbf{A}$ ,  $\mathbf{B}$ ,  $\mathbf{C}$ , and  $\mathbf{D}$  now being  $2 \times 2$  block matrices themselves.

`bmat_conj_transpose(X)`

#### Function.

This function computes the conjugate transpose (Hermitian adjoint) of a block matrix.

- **X**

A  $2 \times 2$  or  $4 \times 4$  block matrix instance of type `clsBlockMatrix`. Each block may contain complex-valued data.

**Return value:**

A new block matrix representing the conjugate transpose of `X`, where each block is transposed and complex-conjugated, and the block positions are mirrored accordingly.

**bmat2\_M\_to\_S(M, p=None, msg="")**

**Function.**

This function converts a  $2 \times 2$  transfer block matrix  $M$  into the corresponding scattering block matrix  $S$ .

- **M**

A  $2 \times 2$  block matrix representing a transfer matrix, as used for linear cavities in this library.

- **p**

Optional. If set to an instance of `clsProgressPrinter`, progress output is generated during the computation.

- **msg**

Optional. A message string that will be printed with the progress output (only if `p` is provided). If empty, no message is printed.

**Return value:**

A  $2 \times 2$  block matrix representing the corresponding scattering matrix  $S$ .

**bmat4\_M\_to\_S(M, p=None, msg="")**

**Function.**

This function converts a  $4 \times 4$  transfer block matrix  $M$  into the corresponding scattering block matrix  $S$ .

- **M**

A  $4 \times 4$  block matrix representing a transfer matrix, as used for ring cavities in this library.

- **p**

Optional. If set to an instance of `clsProgressPrinter`, progress output is generated during the computation.

- **msg**

Optional. A message string that will be printed with the progress output (only if `p` is provided). If empty, no message is printed.

**Return value:**

A  $4 \times 4$  block matrix representing the corresponding scattering matrix  $S$ .

## A.5 Class `clsProgressPrinter`

The class `clsProgressPrinter` provides a simple mechanism for generating readable progress output during long-running or nested computations. It is designed to structure output hierarchically, making it easy to understand which sub-task is currently running and how long each part of the computation takes.

The core mechanism is based on the methods `.push_print(...)` and `.pop()`, which work like a stack: whenever a new sub-task begins, `.push_print(...)` is called with a short message indicating the task. At the end of the sub-task, `.pop()` is called. The class then prints a neatly indented output including timing information.

```
Starting First Task
  Starting Second Task
    Starting Third Task
      done (7.0 seconds)
      done (16.5 seconds)
    done (25.3 seconds)
```

The indentation indicates the nesting depth of the tasks, and each `done` line includes the duration of the corresponding sub-task. This makes `clsProgressPrinter` particularly useful for debugging or logging complex procedures with multiple nested steps.

In addition to these hierarchical start/finish messages, `clsProgressPrinter` can monitor tasks that consist of many sub-steps whose individual duration is not known a priori. To use this mode, you first call `.tic_reset(...)` with the total number of steps (e.g. 100), and optionally a message that describes the task. When it later becomes necessary to print progress output, this message will be printed once to mark the start of the task. You then invoke `.tic()` once per completed step. The printer keeps an internal timer and emits a progress line only when

- (i) more than `.max_time_betw_tic_outputs` seconds have elapsed since the last update (default: 10), or (ii) the very last step is reached and the option `always_print_first_and_last` (set in `.tic_reset(...)`) requests an explicit closing line.

This “adaptive throttling” means that a loop of 100 very quick iterations can finish completely silently, whereas a slower loop of the same length will print a handful of lines such as

```
Starting Interesting Task
  12.0% done (step 12 of 100, 9.2 seconds)
  37.0% done (step 37 of 100, 8.3 seconds)
  64.0% done (step 64 of 100, 9.1 seconds)
  100.0% done (step 100 of 100, 10.2 seconds)
done (36.8 seconds)
```

Several parameters described further down let you tailor the behavior. Taken together, the hierarchical stack of messages and the adaptive step counter make `clsProgressPrinter` a lightweight yet flexible tool for keeping users informed.

### A.5.1 Initialization

#### `clsProgressPrinter()`

##### **Constructor.**

Creates a new instance of `clsProgressPrinter` with all internal counters and timing settings initialized to their default values.

### A.5.2 Main Methods

#### `.push_print(msg, start_time=0)`

This method starts a new sub-task and prints a message to indicate that it has begun. It also increases the current indentation level, so that all following output appears visually nested.

- `msg`  
A short message describing the task that has just started. This message is printed immediately (unless output is disabled).
- `start_time`  
Optional. A custom start time (in seconds, e.g. from `time.perf_counter()`) may be passed. If left at 0 (default), the current time is used internally to later compute the elapsed time when `.pop()` is called.

Every call to `.push_print(...)` increases the internal nesting level. The indentation applied to all following output is based on this depth (see also `.indent`). The corresponding task must later be closed by a matching call to `.pop()`, which prints a “done” line including the elapsed time.

#### `.print(msg)`

##### **Method.**

Prints a message with indentation corresponding to the current nesting level. Unlike `.push_print(...)`, this method does not affect the internal task stack and does not influence timing or progress tracking.

- `msg`  
A string to be printed. The message will appear with indentation matching the current depth (i.e., the number of active tasks started via `.push_print(...)`).

This method is useful for displaying informational output related to the current task without opening or closing a new sub-task level.

### `.pop()`

#### **Method.**

Closes the most recently opened task level and prints a `done` message including the duration of the sub-task. This method should be used to match a previous call to `.push_print(...)`. If no task is currently open (i.e., the internal stack is empty), the method returns silently.

#### **Behaviour.**

- Removes the most recent start time from the internal stack.
- Computes the elapsed time since that task was started.
- If the elapsed time exceeds `.min_time_to_show`, prints `done (X seconds)` with indentation matching the current depth.

**Return value:** Returns the elapsed time (in seconds) since the corresponding `.push_print(...)` call. This can be useful if further processing depends on how long the sub-task took. If progress output was disabled via `.silent`, no message is printed, but the timing and return value still work as expected.

### `.tic_reset(goal_count, always_print_first_and_last=True, msg= "")`

#### **Method.**

This method initializes the internal progress counter for use with `.tic()`. It sets the total number of steps expected and resets all associated timing variables.

- `goal_count`  
The total number of steps expected for the task.
- `always_print_first_and_last`  
Optional. If set to `True` (default), the first and final progress lines are always printed, even if the task finishes quickly. If `False`, progress output is only triggered by elapsed time thresholds.
- `msg`  
Optional. A short message describing the task. If progress output is triggered later on, this message will be printed once – via `.push_print(...)` – to mark the beginning of the task.

This method should be called before starting a loop or process that consists of a known number of steps. After calling `.tic_reset(...)`, invoke `.tic()` after each step to update the progress output.

### `.tic()`

#### **Method.**

This method advances the internal progress counter by one step and may print a progress update if the configured timing conditions are met. It is intended to be used after calling `.tic_reset(...)`, which sets the total number of steps. After that,

`.tic()` should be called once after each completed step.

**Printing logic:** A progress line is printed only if

- more than `.max_time_betw_tic_outputs` seconds have passed since the last update, or
- the final step is reached and the flag `always_print_first_and_last` (passed to `.tic_reset(...)`) is set to `True`.

If output is triggered and a task message was passed to `.tic_reset(...)` via the `msg` parameter, this message is printed once using `.push_print(...)` to mark the start of the task.

The printed progress line includes:

- the percentage completed and the current step index;
- optionally, the time elapsed since the last call to `.tic()`, if this exceeds `.max_time_without_tic_duration_output`.

#### Finalization.

When the final step is reached, `.tic()` automatically calls `.pop()` to print the final `done (... s)` line and close the task level. As a result, short tasks may produce no output at all, while longer tasks yield just enough feedback to keep the user informed.

### A.5.3 Configuration Properties

#### `.min_time_to_show`

**Read/write property.**

A read/write property that defines the minimum duration (in seconds) a sub-task must run for its final `done (... seconds)` message to be printed. If a sub-task finishes faster than this threshold, the `.pop()` call returns the duration, but does not print a `done` line. This helps avoid cluttering the output with trivial timing information for fast operations.

**Default:** 3.0 seconds

#### `.max_time_betw_tic_outputs`

**Read/write property.**

A read/write property that controls how often progress lines are printed during step-based tasks. This value defines the minimum number of seconds that must pass between two consecutive progress lines produced by `.tic()`. If the elapsed time since the last output is less than this threshold, no line will be printed.

**Default:** 10 seconds

**`.max_time_without_tic_duration_output`****Read/write property.**

A read/write property that controls when `.tic()` begins printing the duration of each step. If more than the specified number of seconds have passed since the last progress update, the output line will include the time spent on the current step (e.g., “12.0% done … 1.2 seconds”). This helps the user estimate performance over time without overwhelming the output during fast tasks.

**Default:** 60 seconds

**`.indent`****Read-only property.**

A read-only property that returns a string of spaces used to indent messages based on the current nesting level. Each level of indentation corresponds to a call to `.push_print(...)`. The resulting indent is applied automatically to all output generated by `.push_print(...)`, `.print(...)`, and `.tic()`. This ensures that output reflects the logical structure of nested tasks.

**Default:** 3 spaces per level

**`.silent`****Read/write property.**

A read/write property that enables or disables all console output generated by this instance. If set to `True`, no progress messages will be printed – including those from `.push_print(...)`, `.pop()`, `.print(...)`, and `.tic()`. This can be useful for suppressing output during automated runs or testing. Internally, all timing and tracking still occurs even when silent mode is enabled.

**Default:** `False`

## A.6 Class `clsTaskManager`

`clsTaskManager` can be used to coordinate the *step-wise* execution of many cavity simulations in parallel. Each simulation runs in *single-step mode* (see Sec. F.1.12): `step = 0` builds the first component’s transfer matrix  $\mathbf{M}_0$ ; `step = 1` computes  $\mathbf{M}_1$  and multiplies it with  $\mathbf{M}_0$  to advance the product  $\mathbf{M}_0\mathbf{M}_1$ ; … For  $s$  components there are typically  $s + 1$  steps: one for each component, and one “additional” step that uses the cavity-wide transfer matrix to e.g. evaluate the output fields. Hence one worker may process e.g. *step 2 of simulation A* while another simultaneously already starts *step 0 of simulation B*.

There are in total `.simulations` distinct simulations, each consisting of `.steps_per_simulation` steps. For every pair  $(sim, step)$  the task manager maintains two tiny marker files in `.folder` whose names encode both indices `[sim]_[step].a.tmp` and

`[sim]_[step].b.tmp`:

- “`[sim]_[step].a.tmp`” → (*sim, step*) is *in progress*,
- “`[sim]_[step].b.tmp`” → (*sim, step*) is *finished*.

A worker process typically

1. creates an instance of `clsTaskManager` by calling the constructor `clsTaskManager(...)` with the correct parameters (number of simulations and steps, and file folder);
2. calls `.get_next_task()` (which creates the matching “`_a.tmp`” file) to claim the next available task identified by (*sim, step*);
3. if no task is currently free, `.get_next_task()` returns `(-1, -1)` after pausing for `.sleep_time` seconds, so the caller can simply retry or exit;
4. otherwise, performs the corresponding cavity step;
5. notifies completion via `.end_task(...)`, which creates the “`_b.tmp`” file.

The actual computation then terminates; a subsequent worker (possibly on a different node) picks up the next task. File locking is handled by `portalocker`, eliminating race conditions, and incomplete tasks left by crashed workers are safely reclaimed by the next process that starts.

### A.6.1 Initialization

`clsTaskManager(simulations, steps_per_simulation, folder)`

#### Constructor.

Creates a task-manager instance that can coordinate the distributed, step-wise execution of multiple cavity simulations.

#### Parameters

- `simulations` – Total number of independent simulations (*sim index*). Values  $\leq 0$  are automatically set to 1.
- `steps_per_simulation` – Number of single-step calculation stages that constitute *one simulation* (*step index*). Values  $\leq 0$  are automatically set to 1.
- `folder` – Path in which the tiny `_a.tmp` / `_b.tmp` marker files are created. An empty string uses the current working directory.

#### Internal state

- Sets `.sleep_time = 10s` – the pause used by `.get_next_task()` when no task is currently free.

#### Typical usage

Instantiate a single `clsTaskManager` in a shared network folder accessible to all worker jobs; each job then calls `.get_next_task()` / `.end_task(...)` to claim and release tasks without central supervision.

### A.6.2 Core Configuration

#### `.simulations`

##### **Read/write property.**

Specifies the number of distinct simulations that will be coordinated by the task manager.

##### **Setter behavior**

If a value smaller than zero is assigned, it is automatically adjusted to 1.

#### `.steps_per_simulation`

##### **Read/write property.**

Specifies the number of execution steps each simulation is divided into.

##### **Setter behavior**

If a value smaller than zero is assigned, it is automatically adjusted to 1.

#### `.sleep_time`

##### **Read/write property.**

Defines the number of seconds to pause when `.get_next_task()` is unable to assign a new task. If no available task can be grabbed, the `.get_next_task()` method prints a message and waits for `.sleep_time` seconds before returning (-1, -1).

##### **Default value**

10 (seconds)

#### `.folder`

##### **Read/write property.**

Specifies the directory used to store task marker files that coordinate the parallel execution of simulation steps. For each task (*sim, step*), two files are created in this folder to indicate whether the task is currently being processed or has already been completed (see `.grab_task(...)` and `.end_task(...)`).

If set to an empty string, the current working directory is used.

##### **Typical usage**

Set `.folder` to a shared directory accessible by all participating processes (e.g., a common network-mounted filesystem).

### A.6.3 Task Scheduling

#### `.get_next_task()`

##### **Method.**

Returns the next simulation task (*sim, step*) that can be executed by the calling process. Internally calls `.grab_task(...)` to determine whether the task is available. If so, it

is marked as “in processing” and returned. If all tasks are already finished or currently in progress, the method pauses execution for `.sleep_time` seconds and returns `(-1, -1)` to signal that no task could be assigned.

### Behavior

1. Iterates over all simulations ( $0 \dots .simulations - 1$ ) and for each simulation over all steps ( $0 \dots .steps\_per\_simulation - 1$ ).
2. For each pair  $(sim, step)$  it calls `.grab_task(...)`.
3. If a task is successfully *grabbed*, returns  $(sim, step)$ .
4. If the task is currently in progress, *skips to the next simulation*  $sim + 1$ .
5. If no task can be grabbed in any simulation, prints “*Could not grab a task. Will sleep for ... s.*”, waits for `.sleep_time`, and returns `(-1, -1)`.

### Return value

- A tuple  $(sim, step)$  identifying the next task, or
- `(-1, -1)` if no task is currently available.

### Typical usage

Each worker process calls this method once, executes the assigned task (if any), then exits. A subsequent process can resume by calling `.get_next_task()` again.

## `.grab_task(sim, step)`

### Method.

Attempts to *claim* the task identified by the integer pair  $(sim, step)$ . Tasks are tracked by two tiny files in `.folder`:

```
“[sim]_[step].a.tmp” : marker “in processing”
“[sim]_[step].b.tmp” : marker “done”
```

### Parameters

- `sim` – Simulation index ( $0 \leq sim < .simulations$ ).
- `step` – Step index within that simulation ( $0 \leq step < .steps\_per\_simulation$ ).

### Behavior

1. Introduces a small random delay between 0 and 1 second to desynchronize parallel workers.
2. If `sim` or `step` is out of range, returns `(False, False)`.
3. Constructs the two marker filenames that correspond to the given  $(sim, step)$  and (if present) prepends `.folder`.
4. If the “done” file exists: returns `(False, False)` – task already completed.
5. Else if the “processing” file exists: returns `(False, True)` – task is currently being handled by another worker; caller should move on.

6. Otherwise the task is free. Creates the “processing” file (atomically marks the claim) and returns (`True`, `True`).

#### Return value

A two-boolean tuple (`grabbed`, `processing`):

- `grabbed = True` - caller has successfully taken ownership.  
`processing = True` in this case.
- `grabbed = False`, `processing = True` - another worker is on it.
- `grabbed = False`, `processing = False` - task already finished (see step 4).

`.end_task(sim, step)`

#### Method.

Marks the task identified by  $(sim, step)$  as *completed*. A tiny “done” file “[`sim`]\_[`step`].b.tmp” is written to `.folder`; its mere existence signals to all workers that the task no longer needs processing.

#### Parameters

- `sim` - Simulation index ( $0 \leq sim < .simulations$ ). Out-of-range values are ignored.
- `step` - Step index within that simulation ( $0 \leq step < .steps_per_simulation$ ). Out-of-range values are ignored.

#### Behavior

1. Validates the indices; returns immediately if either is out of range.
2. Builds the filename “[`sim`]\_[`step`].b.tmp”, prepending `.folder` when non-empty.
3. Writes the tmp file to disk, thereby flagging the task as finished.

#### Typical call sequence

```
sim, step = task_mgr.get_next_task()
if sim >= 0:
    ... perform the work ...
    task_mgr.end_task(sim, step)
```

### A.6.4 House-Keeping Utilities

`.delete_all_files()`

#### Method.

Removes *all* marker files created by `.grab_task(...)` and `.end_task(...)` for every simulation/step combination, thereby resetting the task-manager’s state.

#### Typical usage

Call once before launching a fresh batch of worker processes to ensure that no stale marker files from previous runs remain.

## A.7 Class `clsGrid`

In Fourier optics a scalar, a *polarized* light field can be described in **position space**<sup>1</sup> by the complex amplitude

$$U(x, y; z),$$

or – after a two-dimensional Fourier transform – in *k-space* by

$$A(k_x, k_y; z),$$

where  $z$  is the main propagation axis. On a computer both representations must be sampled on a finite, *square*  $N \times N$  grid. What was once a continuous Fourier transform is now an FFT, and an infinitely extended plane becomes a finite tile of side  $L$ .

For a chosen field-of-view (FOV) of side-length  $L_{\text{fov}}$ , a target resolution, and a propagation distance  $z$ , the grid size is not arbitrary. For a wavelength  $\lambda$  the *critical-sampling condition* [3, p. 193] requires

$$N_{\text{tot}} = \frac{L_{\text{tot}}^2}{\lambda z},$$

where  $N_{\text{tot}}$  is the number of pixels per side and  $L_{\text{tot}}$  is the side-length of the *total* square grid. Because  $L_{\text{fov}}$  is usually smaller than  $L_{\text{tot}}$ , the desired FOV must be *embedded* in a larger  $N_{\text{tot}} \times N_{\text{tot}}$  grid that fulfils the sampling condition; the inner  $L_{\text{fov}} \times L_{\text{fov}}$  region then contains the physically relevant data, while the surrounding pixels act as a numerical padding to prevent artefacts.

The class `clsGrid` collects all routines needed to design and work with such square grids:

- **Grid initialization methods**

The class `clsGrid` has methods like `.set_opt_res_based_on_sidelength(...)` and `.set_opt_res_tot_based_on_res_fov(...)` which can compute the total grid size required to host a desired FOV while satisfying the sampling condition for a given propagation distance.

- **FOV extraction / embedding**

A smaller FOV can be embedded in, or extracted from, the larger total grid with `.embed_image(...)` and `.extract_FOV(...)`.

- **Coordinate conversions**

The `axis_*/k_axis_*/n_axis_*` properties and other helper functions translate between array indices and physical coordinates in position space, *k*-space, or mode-number space.

- **Array–vector reshaping**

`.arr_to_vec(...)` and `.vec_to_arr(...)` convert a 2-D field array to a 1-D vector (and back) for matrix algebra.

- **Fourier transforms**

`fft2_phys_spatial(...)` and `.ifft2_phys_spatial(...)` perform forward and inverse FFTs using the grid's normalization and sampling conventions.

---

<sup>1</sup>We use the term *position space* (often called *real space* in the literature) to emphasize that the coordinates are  $(x, y)$  rather than spatial frequencies.

- **Miscellaneous utilities**

In addition to the core methods, `clsGrid` offers many helper functions that are frequently invoked behind the scenes by higher-level classes. Although these helpers are often wrapped by other interfaces, they remain available for direct use whenever low-level control is required.

Every `clsGrid` instance is linked to a cavity object `clsCavity1path` or `clsCavity2path`. The constructor initializes a small placeholder grid ( $10 \times 10$  pixels); it is the user's responsibility to call the sizing functions early in a simulation to obtain the properly sampled grid that the cavity requires. Overall, besides these grid initialization methods, `clsGrid` provides the low-level infrastructure on which higher-level classes such as `clsLightField` build.

### A.7.1 Initialization

`clsGrid(cavity)`

**Constructor.**

Creates a new `clsGrid` instance and binds it to an optical cavity.

- `cavity`

An instance of `clsCavity1path` or `clsCavity2path`. The grid keeps a reference to this object via the read-only property `.cavity`.

**Initial state.**

The constructor allocates a placeholder square array of  $10 \times 10$  pixels and initializes all internal parameters accordingly. These default dimensions *do not* satisfy the critical-sampling condition; the user is expected to call one of the sizing helpers – e.g. the method `.set_opt_res_based_on_sidelength(...)` or the method `.set_opt_res_tot_based_on_res_fov(...)` early in the simulation to create a grid that matches the desired field-of-view, resolution, and propagation distance.

### A.7.2 Grid Definition Methods

`.set_res(res_fov, res_tot, length_fov)`

**Method.**

Manually sets the field-of-view (FOV) side length and the resolutions of both the FOV and total simulation grid.

- `res_fov` – Resolution (in pixels) of the FOV region.
- `res_tot` – Resolution (in pixels) of the total simulation grid.
- `length_fov` – Side length of the FOV in physical units (meters).

This method directly sets the physical dimensions of the grid and should only be used if the user has a good understanding of the critical sampling condition. In typical use cases, it is safer to use helper methods like `.set_opt_res_based_on_sidelength(...)` or `.set_opt_res_tot_based_on_res_fov(...)`, which compute compatible grid sizes automatically.

**Automatic corrections:**

- If `res_fov` is smaller than 10, it is set to 10 and a warning is printed.
- If `res_tot` is smaller than `res_fov`, it is increased to match.
- If the FOV and total grid have inconsistent even/odd parity, `res_tot` is incremented to match.
- If `length_fov` is smaller than  $1 \mu\text{m}$ , it is set to  $1 \mu\text{m}$ .

After these checks, the total side length  $L_{\text{tot}}$  is computed as:

$$L_{\text{tot}} = L_{\text{fov}} \cdot \frac{N_{\text{tot}}}{N_{\text{fov}}}$$

and all axis arrays are reinitialized.

```
.set_opt_res_based_on_sidelength(length_fov, factor, prop_dist, even)
```

**Method.**

Automatically computes suitable grid parameters that fulfill the critical-sampling condition for a given propagation distance.

- `length_fov` – Desired side length of the physically meaningful field-of-view (FOV) region (in meters).
- `factor` – Multiplier defining how much larger the total simulation grid should be relative to the FOV grid.
- `prop_dist` – Propagation distance in  $z$ -direction (in meters).
- `even` – If `True`, both FOV and total grid resolution will be even; if `False`, both will be odd.

This method derives a total grid size large enough to embed the FOV with padding, using the formula

$$N_{\text{tot}} \approx \frac{L_{\text{tot}}^2}{\lambda z}, \quad \text{where } L_{\text{tot}} = \text{length_fov} \cdot \text{factor}$$

The result ensures the simulation grid is compatible with the critical sampling condition.

**Behavior and corrections:**

- If `factor` is less than 1, it is set to 1 with a warning.
- If `length_fov` or `prop_dist` is less than  $1 \mu\text{m}$ , it is clipped to  $1 \mu\text{m}$  with a warning.
- The resulting total resolution `res_tot` is rounded to the nearest valid integer and parity-adjusted according to the `even` flag.
- The FOV resolution `res_fov` is derived from `res_tot` and `factor`, and adjusted for even/odd parity as well.

```
.set_opt_res_tot_based_on_res_fov(length_fov, res_fov,
prop_dist)
```

### Method.

Automatically determines the optimal resolution of the total simulation grid based on a given field-of-view (FOV) side length, the desired resolution, and the propagation distance.

- `length_fov` – Side length of the desired field-of-view region (in meters).
- `res_fov` – Resolution of the field-of-view in pixels.
- `prop_dist` – Propagation distance in  $z$ -direction (in meters).

This method calculates the resolution `res_tot` of the larger simulation grid such that the critical sampling condition is satisfied. The formula is based on:

$$N_{\text{tot}} \approx \left( \frac{N_{\text{fov}}}{L_{\text{fov}}} \right)^2 \cdot \lambda z$$

### Behavior and corrections:

- If `length_fov` is less than  $1 \mu\text{m}$ , it is clipped to  $1 \mu\text{m}$  with a warning.
- If `res_fov` is smaller than 10, it is set to 10 with a warning.
- The resulting `res_tot` is parity-adjusted to match the even/odd status of `res_fov`.

## A.7.3 Grid Geometry and Field-of-View Handling

```
.factor
```

### Read-only property.

Returns the ratio between the side length (or resolution) of the total grid and that of the field-of-view (FOV) grid:

$$\text{factor} = \frac{\text{res\_tot}}{\text{res\_fov}}$$

This value indicates how much larger the embedding simulation grid is compared to the active field-of-view. It is typically greater than or equal to 1 and reflects the “guard region” needed for diffraction effects in propagation.

```
.res_fov
```

### Read-only property.

Returns the resolution of the field-of-view (FOV) grid per side, in pixels. The total number of pixels in the FOV is therefore `res_fov`  $\times$  `res_fov`. This is the portion of the grid where actual field data is typically defined.

**.res\_tot****Read-only property.**

Returns the total resolution of the simulation grid per side, including padding (guard region), in pixels. The total number of pixels is `res_tot`  $\times$  `res_tot`. This larger grid ensures that the critical sampling condition is fulfilled for the given propagation distance.

**.length\_fov****Read-only property.**

Returns the side length of the field-of-view (FOV) grid in meters. This is the physical width and height covered by the active simulation region in position space.

**.length\_fov\_mm****Read-only property.**

Returns the side length of the field-of-view (FOV) grid in millimeters.

**.length\_tot****Read-only property.**

Returns the side length of the total simulation grid (including padding) in meters. This is the physical width and height of the full grid used to satisfy the sampling conditions.

**.length\_tot\_mm****Read-only property.**

Returns the side length of the total simulation grid (including padding) in millimeters.

**.even\_res****Read-only property.**

Returns `True` if the field-of-view (FOV) resolution `res_fov` is an even number of pixels, otherwise `False`.

**.odd\_res****Read-only property.**

Returns `True` if the field-of-view (FOV) resolution `res_fov` is an odd number of pixels, otherwise `False`.

**.fov\_pos****Read/write property.**

Determines the position of the field-of-view (FOV) within the total grid.

The value must be an integer from 0 to 4:

- 0 – Center (default)
- 1 – Top
- 2 – Bottom
- 3 – Left
- 4 – Right

#### **Position meaning (default case).**

0 centers the FOV in both directions. For the other values the FOV remains centered in one axis and is shifted along the other so that its *center* sits midway between the overall grid center and the chosen edge:

- 1 (Top) – FOV center is in the middle of the upper half of the grid. (Only if the FOV is taller than half the grid, it is clamped to touch the top edge.)
- 2 (Bottom) – Mirrored to the lower half. (Clamped to the bottom edge if necessary.)
- 3 (Left) – FOV center is in the middle of the left half. (Clamped to the left edge if necessary.)
- 4 (Right) – Mirrored to the right half. (Clamped to the right edge if necessary.)

The resulting pixel offsets are stored in `.pos_offset_x` and `.pos_offset_y`.

Changing this value recalculates the internal pixel offsets `.pos_offset_x` and `.pos_offset_y` to align the FOV accordingly within the total padded grid. If an invalid value is given (less than 0 or greater than 4), it is automatically clamped to the nearest valid value.

#### **`.pos_offset_x`**

##### **Read-only property.**

Indicates how many pixels the center of the FOV is shifted along the *x*-axis relative to a centered position in the total grid. The value is automatically recalculated when `.fov_pos` is changed.

A positive value means the FOV is shifted to the right, a negative value means it is shifted to the left, and zero indicates that it is centered. This offset is used internally for FOV extraction and embedding.

#### **`.pos_offset_y`**

##### **Read-only property.**

Indicates how many pixels the center of the FOV is shifted along the *y*-axis relative to a centered position in the total grid. The value is automatically recalculated when `.fov_pos` is changed.

A positive value means the FOV is shifted downward, a negative value means it is shifted upward, and zero indicates that it is centered. This offset is used internally for FOV extraction and embedding.

**.extract\_FOV(E\_in, k\_space\_in, k\_space\_out)**

Extracts the field-of-view (FOV) from a larger input array `E_in`. The FOV is defined according to the current grid settings (see `.fov_pos`).

**Parameters:**

- `E_in` – A 2D array representing the electric field (either in position space or Fourier space).
- `k_space_in` – If `True`, the input `E_in` is interpreted as being in Fourier space and will internally be transformed to position space before cropping.
- `k_space_out` – If `True`, the extracted FOV is transformed to Fourier space before returning.

**Return value**

The extracted FOV as a 2D array. Its dimensions correspond to the resolution `.res_fov`. Whether the result is in position or Fourier space depends on `k_space_out`.

**.embed\_image(E\_in, k\_space\_in, k\_space\_out)**

Embeds a field-of-view (FOV) array `E_in` into a larger zero-padded array of size `.res_tot × .res_tot`, according to the current FOV position (see `.fov_pos`).

**Parameters:**

- `E_in` – A 2D array representing the electric field in the FOV, either in position space or Fourier space.
- `k_space_in` – If `True`, the input is interpreted as being in Fourier space and is internally transformed to position space before embedding.
- `k_space_out` – If `True`, the embedded result is transformed to Fourier space before returning.

**Return value**

A 2D array of size `res_tot × res_tot`, with the FOV embedded into the larger grid. The output is in position or Fourier space depending on `k_space_out`.

**.get\_res\_arr(X)****Method.**

Determines which resolution the array `X` corresponds to.

**Parameters:**

- `X` – A NumPy array (typically a 2D field array) or a scalar.

**Return value**

- 0 – Unknown resolution (does not match either FOV or total resolution).
- 1 – Field-of-view resolution (`.res_fov × .res_fov`).
- 2 – Total resolution (`.res_tot × .res_tot`).

If `X` is a scalar, the function assumes full resolution and returns 2.

### `.is_fov_res(X)`

#### **Method.**

Returns `True` if the input array `X` matches the field-of-view resolution (`.res_fov`), otherwise returns `False`.

#### **Parameters:**

- `X` – A NumPy array or scalar.

#### **Return value**

- `True` if `X` has shape `(res_fov, res_fov)`,
- `False` otherwise.

Internally, this method uses `.get_res_arr(...)`.

### `.get_res_TR(TR)`

#### **Method.**

Determines the resolution type associated with a transmission or reflection matrix `TR`.

#### **Parameters:**

- `TR` – A square matrix (NumPy 2D array), or a scalar.

#### **Return value**

- 0 if the resolution could not be identified,
- 1 if the matrix corresponds to field-of-view resolution, meaning it is of size `.res_fov2 × .res_fov2`,
- 2 if the matrix corresponds to the total resolution, meaning it is of size `.res_tot2 × .res_tot2`.

#### **Explanation:**

A light field with  $N \times N$  pixels is represented in Fourier space by an array with  $N \times N$  entries. This implies that the number of modal basis functions is  $N^2$ , and therefore a corresponding transmission or reflection matrix must have  $N^2 \times N^2$  elements. This method compares the shape of `TR` with `.res_fov2` and `.res_tot2` to determine the matching resolution.

If `TR` is not a matrix (e.g., a scalar), it is assumed to represent total resolution.

### `.get_res_vec(vec)`

#### **Method.**

Determines the corresponding resolution of a vectorized light field `vec`. This vector is assumed to originate from a 2D array of Fourier coefficients which has been flattened into a one-dimensional vector by `.arr_to_vec(...)`. Its length must therefore equal the square of the original resolution.

**Return value**

- 1 if the vector corresponds to field-of-view resolution with a length of `.res_fov`<sup>2</sup>
- 2 if the vector corresponds to total resolution with a length of `.res_tot`<sup>2</sup>
- 0 if the resolution cannot be determined

**A.7.4 Axes Metrics****`.axis_fov`****Read-only property.**

Returns a 1D NumPy array containing the position-space coordinates for the field-of-view (FOV) grid along one dimension (either  $x$  or  $y$ ). The array has length `.res_fov` and contains real-space positions centered around zero, consistent with the defined `.length_fov`.

**Index-to-coordinate formula.**

Let  $N$  be the side resolution (here:  $N = \text{.res\_fov}$ ) and  $L$  be the side length of the grid (here:  $L = \text{length\_fov}$ ).

Let  $X[i, j]$  be a 2D array representing the light field in position space, where row index  $i$  corresponds to the  $y$ -direction and column index  $j$  to the  $x$ -direction, with  $i, j \in \{0, \dots, N - 1\}$ . Then the physical coordinate corresponding to each column index  $j$  is:

$$x_j = \left( j - \frac{N}{2} \right) \cdot \frac{L}{N}$$

and analogously, the coordinate in  $y$ -direction is:

$$y_i = \left( i - \frac{N}{2} \right) \cdot \frac{L}{N}$$

**`.axis_tot`****Read-only property.**

Returns a 1D NumPy array containing the position-space coordinate values along one axis (either  $x$  or  $y$ ) of the total simulation grid. Like `.axis_fov`, the coordinates are centered around zero and evenly spaced over the full total grid side length given by `.length_tot`. The array length is `.res_tot`. The  $x_j$  and  $y_i$  coordinates are calculated as before, just with  $N = \text{.res\_tot}$  and  $L = \text{length\_tot}$ .

**`.n_axis_fov`****Read-only property.**

Returns a 1D NumPy array of signed integer mode numbers corresponding to the discrete spatial frequencies for a field-of-view grid. These values are arranged in descending order and represent the Fourier mode indices.

The number of elements equals `.res_fov`, and the values range symmetrically around zero. For even  $N$ , the sequence runs from  $N/2$  to  $-N/2+1$ ; for odd  $N$ , from  $(N-1)/2$  to  $-(N-1)/2$ .

In a 2D array `A[i, j]` representing the light field in Fourier space (k-space), the row index  $i$  corresponds to the vertical mode number  $n_y$ , and the column index  $j$  corresponds to the horizontal mode number  $n_x$ . Each Fourier coefficient `A[i, j]` is thus associated with a  $(n_x, n_y)$  pair from the respective axis arrays.

#### `.n_axis_tot`

##### **Read-only property.**

Same as `.n_axis_fov`, but for the total grid instead of the field-of-view. The number of elements equals `.res_tot`, and each index corresponds to a mode number for the full grid resolution.

#### `.k_axis_fov`

##### **Read-only property.**

Returns a 1D NumPy array of physical wave vector components  $k_x$  or  $k_y$  (in radians per meter) corresponding to the Fourier-mode indices given in `.n_axis_fov`. These values represent the discrete spatial frequencies for the field-of-view grid.

Each  $k$ -value is calculated from its corresponding mode number  $n$  using:

$$k = \frac{2\pi n}{L}$$

where  $L$  is the nominal side-length of the field-of-view grid (`length_fov`). The length of the returned array is `.res_fov`, and the values are centered around zero.

#### `.k_axis_tot`

##### **Read-only property.**

Returns a 1D NumPy array of physical wave vector components  $k_x$  or  $k_y$  (in radians per meter) corresponding to the Fourier-mode indices given in `.n_axis_tot`. These values represent the discrete spatial frequencies for the full-resolution (total) grid.

Each  $k$ -value is calculated from its corresponding mode number  $n$  using:

$$k = \frac{2\pi n}{L}$$

where  $L$  is the nominal side-length of the total grid (`length_tot`). The array has length `.res_tot`, and the values are symmetrically distributed around zero.

#### `.dist_to_pixels(dist)`

##### **Method.**

Converts a physical distance `dist` (in meters) to an integer number of pixels.

**Parameter**

- `dist` – Physical distance in meters.

**Return value**

An integer representing the number of pixels corresponding to the given distance in the total-resolution grid (`.res_tot`).

The conversion uses the current pixel density of the total grid:

$$\text{pixels} = \text{round} \left( \text{dist} \cdot \frac{\text{res\_tot}}{\text{length\_tot}} \right)$$

`.get_ax_plot_info()`**Method.**

Returns convenient plot-axis limits (in a human-friendly unit) for both the field-of-view (FOV) and the total grid.

**Return value**

$$(x_{\min}^{\text{FOV}}, x_{\max}^{\text{FOV}}, x_{\min}^{\text{tot}}, x_{\max}^{\text{tot}}, \text{unit})$$

- $x_{\min}^{\text{FOV}}, x_{\max}^{\text{FOV}}$  – lower and upper limits of the FOV axis,
- $x_{\min}^{\text{tot}}, x_{\max}^{\text{tot}}$  – lower and upper limits of the total grid axis,
- `unit` – one of "nm", "um", "mm", "cm", or "m".

The routine first computes the half-pixel-shifted boundaries of both axes:

$$\begin{aligned} x_{\min}^{\text{FOV}} &= x_0 - \frac{\Delta x_{\text{FOV}}}{2}, & x_{\max}^{\text{FOV}} &= x_{N-1} + \frac{\Delta x_{\text{FOV}}}{2}, \\ x_{\min}^{\text{tot}} &= x_0 - \frac{\Delta x_{\text{tot}}}{2}, & x_{\max}^{\text{tot}} &= x_{N-1} + \frac{\Delta x_{\text{tot}}}{2}. \end{aligned}$$

It then chooses the most readable length unit: nm → um → mm → cm → m, scales all four limits accordingly, and returns the scaled numbers together with the chosen unit string. The function is handy when setting tick labels for images or surface plots.

**A.7.5 Conversions**`.convert(E_in, k_space_in, k_space_out, fov_out)`**Method.**

Converts the light field array `E_in` between position space and Fourier space, and optionally between total and field-of-view (FOV) resolution.

**Parameters**

- `E_in` – input array representing the light field. Can be either FOV or total resolution; the resolution is automatically detected. `E_in` may also be a scalar (interpreted as a constant-valued field in total resolution), or `None` (interpreted as a black field).

- `k_space_in` – set to `True` if `E_in` is in Fourier space; `False` if it is in position space.
- `k_space_out` – set to `True` to return the result in Fourier space; `False` for position space.
- `fov_out` – set to `True` to return the result in FOV resolution; `False` for total resolution.

**Returns value**

An array in the desired space and resolution, obtained by applying the appropriate FFT or inverse FFT operations and, if necessary, cropping or embedding using `.extract_FOV(...)` and `.embed_image(...)`.

`.arr_to_vec(X, k_space_in, column_vec=True)`

**Method.**

Converts a light field array `X` into a one-dimensional vector of Fourier coefficients, ordered according to the mode numbers as returned by `.mode_numbers_fov` or `.mode_numbers_tot`, depending on the resolution of `X`.

If `X` is not already in Fourier space, it is first transformed using a 2D FFT.

**Parameters**

- `X` – input 2D NumPy array representing the light field in either position or Fourier space.
- `k_space_in` – set to `True` if `X` is already in Fourier space; `False` if it is in position space.
- `column_vec` – if `True` (default), returns the output as a column vector (shape  $(N^2, 1)$ ); otherwise, returns a flat 1D array (shape  $(N^2,)$ ).

**Return value**

A vector of Fourier coefficients extracted from the input array. The coefficients are ordered according to the corresponding `mode_numbers` array, which arranges all  $(n_x, n_y)$  pairs by increasing angle of the corresponding  $k$ -vectors relative to the  $z$ -axis.

`.vec_to_arr(X, k_space_out)`

**Method.**

Converts a one-dimensional vector of Fourier coefficients into a 2D array representing the light field in either position space or Fourier space, depending on the `k_space_out` flag. This method reverses the operation performed by `.arr_to_vec(...)`.

The input vector entries are expected to be placed in the appropriate positions of the 2D output array using the mode indices provided by `.mode_indices_fov` or `.mode_indices_tot`, depending on the length of the input vector.

**Parameters**

- `X` – a one-dimensional NumPy vector of Fourier coefficients.
- `k_space_out` – if `True`, the output will be a 2D array in Fourier space; if `False`,

it will be in position space (inverse FFT is applied).

#### Returns value

A 2D NumPy array of shape ( $N, N$ ) representing the light field in the selected space.

### `fft2_phys_spatial(X)`

#### Method.

Performs a 2-D Fast Fourier Transform (FFT) of a position-space array  $X$  using the physics spatial Fourier-transform convention. The appropriate coordinate axis (either `.axis_fov` or `.axis_tot`) is selected automatically based on the shape of  $X$ .

#### Parameter

- $X$  – 2-D NumPy array containing the complex field  $U(x, y)$  in position space.

#### Return value

A 2-D NumPy array of complex Fourier coefficients  $A(k_x, k_y)$  representing the same field in  $k$ -space.

#### Energy-preserving normalization

If the input field is normalized such that its total power

$$\iint |U(x, y)|^2 dx dy = 1,$$

then the output coefficients are scaled so that their Euclidean norm fulfills

$$\|A\|_2 = \sqrt{\sum_{k_x, k_y} |A(k_x, k_y)|^2} = 1.$$

### `ifft2_phys_spatial(X)`

#### Method.

Performs a 2-D inverse Fast Fourier Transform (FFT) of a  $k$ -space array  $X$ , using the physics spatial inverse Fourier-transform convention, returning the corresponding field in position space. The appropriate coordinate axis (either `.axis_fov` or `.axis_tot`) is selected automatically based on the shape of  $X$ .

#### Parameter

- $X$  – 2-D NumPy array of complex Fourier coefficients  $A(k_x, k_y)$  representing the field in  $k$ -space.

#### Return value

A 2-D NumPy array containing the complex field  $U(x, y)$  in position space.

#### Energy-preserving normalization

If the input coefficients are normalized such that their Euclidean norm satisfies

$$\|A\|_2 = \sqrt{\sum_{k_x,k_y} |A(k_x, k_y)|^2} = 1,$$

then the output field is scaled such that the total power

$$\iint |U(x, y)|^2 dx dy = 1.$$

```
.convert_TR_mat_tot_to_fov(X, tics=0)
```

### Method.

Converts a transmission or reflection matrix `X` that acts on a total-resolution light field into an equivalent matrix that acts on a field with field-of-view (FOV) resolution.

This operation is relatively expensive. For each mode number pair  $(n_x, n_y)$  in the FOV resolution, the method reconstructs the corresponding Fourier basis function, embeds it into the total grid, applies the full-resolution matrix `X`, and extracts the resulting field in FOV resolution. Each column of the resulting matrix is built this way.

The method performs `.res_fov`<sup>2</sup> such operations. To allow progress monitoring, it makes use of the `progress` attribute of the associated cavity, an instance of `clsProgressPrinter`. The number of expected progress steps is normally `res_fov`<sup>2</sup>, but it can be increased via the optional parameter `tics` if additional time-consuming operations follow immediately afterward.

### Parameters

- `X` – Square NumPy array representing a transfer or reflection matrix of total resolution.
- `tics` – Optional integer setting the number of total progress steps (default: `res_fov`<sup>2</sup>).

### Return value

A NumPy array representing the corresponding transfer or reflection matrix acting on FOV-resolution fields.

```
.convert_TR_bmat2_tot_to_fov(X)
```

### Method.

Converts a  $2 \times 2$  block matrix `X` representing a total-resolution transmission or reflection operator in a ring cavity into a block matrix acting on field-of-view (FOV) resolution fields.

Each of the four subblocks of `X` is treated separately using the same procedure as in `.convert_TR_mat_tot_to_fov(...)`, and the results are reassembled into a new  $2 \times 2$  matrix acting on FOV-resolution vectorized light fields.

### Important

This method has not yet been refactored for compatibility with

`clsBlockMatrix`. It only works when the property `.use_bmatrix_class` of `clsCavity2path` is set to `False`, which is deprecated and not recommended.

#### Parameter

- `x` — A  $2 \times 2$  block matrix composed of four NumPy arrays (each square), acting on total-resolution Fourier coefficient vectors.

#### Return value

A  $2 \times 2$  NumPy array of square matrices, each acting on FOV-resolution vectorized light fields.

### A.7.6 Helper Methods and Properties

#### `.cavity`

##### Read-only property.

Returns the `clsCavity1path` or `clsCavity2path` instance to which this `clsGrid` instance is linked. This reference provides access to relevant cavity parameters, such as the wavelength `Lambda`. The cavity reference is defined and stored during construction of the `clsGrid(...)` object.

#### `.fourier_basis_func(nx, ny, tot, k_space_out)`

##### Method.

Generates a normalized 2D Fast-Fourier basis function corresponding to the mode numbers `nx`, `ny`, either in position space or in  $k$ -space.

##### Parameters

- `nx` — Mode number in the  $x$ -direction.
- `ny` — Mode number in the  $y$ -direction.
- `tot` — If `True`, the basis function is defined on the full-resolution grid (`.res_tot`); otherwise, on the field-of-view grid (`.res_fov`).
- `k_space_out` — If `True`, the function is returned in  $k$ -space (spatial frequency domain); otherwise, in position space.

##### Return value

A 2D NumPy array representing the  $(n_x, n_y)$  Fourier basis function. The function is normalized in  $k$ -space. If `k_space_out = False`, it is inverse-transformed into the corresponding position-space field.

#### `.empty_grid(fov_only)`

##### Method.

Returns a 2D NumPy array of complex zeros, representing an empty light field grid.

##### Parameter

- `fov_only` — If `True`, the returned grid has field-of-view resolution (`.res_fov × .res_fov`); if `False`, the full resolution (`.res_tot × .res_tot`) is used.

**Return value** A square NumPy array filled with complex zeros and appropriate resolution.

### `.stretch_x(old_array, x0, c)`

#### **Method.**

Stretches or compresses the contents of a 2D array representing a light field in position space along the horizontal (x-)axis. The transformation is centered around a physical coordinate  $x_0$ , and the amount of stretch or compression is controlled by the factor  $c$ .

A stretch factor  $c > 1$  expands the content, while  $c < 1$  compresses it. The vertical (y-axis) structure of the input array remains unchanged. The total energy (intensity) is preserved through amplitude normalization.

#### **Parameters**

- `old_array` – 2D NumPy array representing a light field in position space.
- `x0` – Physical x-coordinate (in meters) around which the horizontal axis is stretched or compressed.
- `c` – Stretch factor. Values  $c > 1$  stretch the image, values  $c < 1$  compress it.

#### **Return value**

A new array of the same shape as `old_array`, resampled along the x-axis using linear interpolation with smooth complex phase handling. The interpolation is performed separately for amplitude and phase: the absolute value is interpolated linearly, while the phase is interpolated on the unit circle to avoid discontinuities and phase jumps. This is achieved using the helper function `polar_interpolation(...)`.

### `.stretch_y(old_array, y0, c)`

#### **Method.**

Stretches or compresses the contents of a 2D array representing a light field in position space along the vertical (y-)axis. The transformation is centered around a physical coordinate  $y_0$ , and the amount of stretch or compression is controlled by the factor  $c$ .

A stretch factor  $c > 1$  expands the content vertically, while  $c < 1$  compresses it. The horizontal (x-axis) structure of the input array remains unchanged. The total energy (intensity) is preserved through amplitude normalization.

#### **Parameters**

- `old_array` – 2D NumPy array representing a light field in position space.
- `y0` – Physical y-coordinate (in meters) around which the vertical axis is stretched or compressed.
- `c` – Stretch factor. Values  $c > 1$  stretch the image, values  $c < 1$  compress it.

#### **Return value**

A new array of the same shape as `old_array`, resampled along the y-axis using linear interpolation with smooth complex phase handling. The interpolation is performed separately for amplitude and phase: the absolute value is interpolated linearly, while

the phase is interpolated on the unit circle to avoid discontinuities and phase jumps. This is achieved using the helper function `polar_interpolation(...)`.

```
.get_aperture_mask(aperture, anti_alias_factor, black_value=0,
                   consider_pos_offset=False)
```

### Method.

Returns a circular aperture mask sampled on the total grid. Pixels whose centres lie inside the pupil radius are set to 1; pixels outside are set to `black_value` (0 by default). Optional supersampling (`anti_alias_factor=2` or `4`) produces a grey-edged mask that reduces aliasing when the pupil edge falls between pixel centers.

### Parameters

- `aperture` – Pupil diameter in meters; the radius is  $r = \text{aperture}/2$ .
- `anti_alias_factor` – 1 (no anti-aliasing), 2 ( $2\times$  supersampling), or 4 ( $4\times$  supersampling). Supersampling builds the mask at higher resolution and averages back to the native grid, giving smoother edges.
- `black_value` – Value assigned to pixels outside the pupil (default 0).
- `consider_pos_offset` – If `True`, the mask is shifted by the current FOV pixel offsets (`.pos_offset_x`, `.pos_offset_y`) so that the aperture stays centred on the FOV even when the FOV itself is decentered within the total grid.

### Return value

A NumPy array of shape `(.res_tot, .res_tot)` containing the aperture mask.

```
.get_soft_aperture_mask(aperture, epsilon, black_value)
```

### Method.

Generates a circular aperture mask on the *total* grid with a controllable, smoothly tapered edge. Inside the core radius the mask equals 1, outside a slightly larger radius it equals `black_value`, and in the transition region it falls smoothly from 1 to `black_value` using a cubic smooth-step profile.

### Parameters

- `aperture` – Full pupil diameter in metres; the nominal radius is  $a = \text{aperture}/2$ .
- `epsilon` – Half-width of the transition zone in metres. The mask equals 1 for  $r \leq a - \epsilon$  and `black_value` for  $r \geq a + \epsilon$ .
- `black_value` – Value assigned to pixels outside the pupil (often 0 or `np.nan`).

### Return value

A NumPy array of shape `(.res_tot, .res_tot)` representing the soft-edged aperture.

### Implementation detail

For every pixel radius  $r$  the mask value  $M(r)$  is

$$M(r) = \begin{cases} 1, & r \leq a - \epsilon, \\ (1 - (3t^2 - 2t^3))(1 - \text{black\_value}) + \text{black\_value}, & a - \epsilon < r < a + \epsilon, \\ \text{black\_value}, & r \geq a + \epsilon, \end{cases}$$

$$t = \frac{r - (a - \epsilon)}{2\epsilon}.$$

The cubic polynomial  $3t^2 - 2t^3$  provides  $C^1$  continuity at both transition boundaries, yielding a visually and numerically smooth aperture edge.

```
.get_angle_from_nxy(tot, nx, ny, Lambda, nr)
```

### Method.

Computes the propagation angle  $\alpha$  between the wave-vector  $\mathbf{k} = (k_x, k_y, k_z)$  associated with a Fourier mode  $(n_x, n_y)$  and the optical  $z$ -axis.

### Parameters

- `tot` – If `True`, use the total grid (`.res_tot`, `.axis_tot`); if `False`, use the field-of-view grid.
- `nx`, `ny` – Integer mode numbers in  $x$  and  $y$ .
- `Lambda` – Vacuum wavelength  $\lambda$  (metres).
- `nr` – Real part of the refractive index  $n_r$ .

### Computation

$$k_x = \frac{2\pi n_x}{L}, \quad k_y = \frac{2\pi n_y}{L}, \quad k_0 = \frac{2\pi}{\Lambda},$$

where  $L$  is the nominal side length of the chosen grid (total or FOV). The longitudinal component is

$$k_z = \sqrt{(n_r k_0)^2 - k_x^2 - k_y^2} \quad k_z \leftarrow 0 \text{ if the radicand is negative.}$$

Finally,

$$\alpha = \arccos\left(\frac{k_z}{k_0}\right),$$

which lies in the range  $0 \leq \alpha \leq \pi/2$ .

### Return value

The angle  $\alpha$  (radians) between  $\mathbf{k}$  and the  $z$ -axis.

### `.get_sorted_mode_numbers(fov_only, n_max, return_all_col)`

#### Method.

Returns all integer mode-number pairs  $(n_x, n_y)$  on the chosen grid, ordered by increasing radial index  $n_x^2 + n_y^2$  (equivalently, by increasing polar angle of the corresponding  $k$ -vector with respect to the optical  $z$ -axis).

#### Parameters

- `fov_only` – If `True`, use the field-of-view resolution (`.res_fov`); otherwise use the total resolution (`.res_tot`).
- `n_max` – Optional cut-off. If `n_max > 0`, only modes satisfying  $\sqrt{n_x^2 + n_y^2} \leq n_{\text{max}}$  are returned.
- `return_all_col` – If `True`, the third column  $n_x^2 + n_y^2$  is included in the output; if `False`, only the  $(n_x, n_y)$  pairs are returned.

#### Return value

- If `return_all_col=False`: an  $N_{\text{modes}} \times 2$  integer array. Column 1 gives  $n_x$ , column 2 gives  $n_y$ .
- If `return_all_col=True`: an  $N_{\text{modes}} \times 3$  integer array with the third column containing  $n_x^2 + n_y^2$ .

The list is sorted primarily by  $n_x^2 + n_y^2$  (radial order) and, for degenerate radii, by  $n_x$  and then  $n_y$ .

### `.mode_numbers fov`

#### Read-only property.

Returns an integer array containing every  $(n_x, n_y)$  mode-number pair that can occur on the field-of-view (FOV) grid, sorted by increasing radial index  $n_x^2 + n_y^2$  (see `.get_sorted_mode_numbers(...)`). The first few rows look like

$$[[0, 0], [-1, 0], [0, -1], [0, 1], \dots].$$

The array is built lazily: on the first access the list is generated via the method `.get_sorted_mode_numbers(...)` and cached; subsequent accesses return the cached result.

### `.mode_numbers tot`

#### Read-only property.

Returns an integer array containing every  $(n_x, n_y)$  mode-number pair that can occur on the *total* grid, sorted by increasing radial index  $n_x^2 + n_y^2$  (see `.get_sorted_mode_numbers(...)`). The beginning of the list is identical in form to the FOV version, e.g.

$$[[0, 0], [-1, 0], [0, -1], [0, 1], \dots],$$

The array is built lazily: on the first access the list is generated via the method `.get_sorted_mode_numbers(...)` and cached, subsequent accesses return the cached result.

### `.mode_indices_fov`

#### **Read-only property.**

Returns an integer array that maps each mode-number pair  $(n_x, n_y)$  in `.mode_numbers_fov` to the corresponding *row* and *column* indices of a *k*-space array in FOV resolution.

For every mode the indices are computed once and then cached:

$$\begin{aligned} i &= C - n_y, \\ j &= C - n_x, \end{aligned} \quad C = \begin{cases} \text{res\_fov}/2, & \text{even res\_fov}, \\ (\text{res\_fov} - 1)/2, & \text{odd res\_fov}. \end{cases}$$

Thus each row of the returned array has the form  $[i, j]$  so that `k_space_array[i, j]` yields the Fourier coefficient associated with  $(n_x, n_y)$ . Like the mode-number list, the index array is built lazily on first access and then reused for all subsequent queries.

### `.mode_indices_tot`

#### **Read-only property.**

Returns an integer array that maps each mode-number pair  $(n_x, n_y)$  in `.mode_numbers_tot` to the corresponding *row* and *column* indices of a *k*-space array in total resolution.

For every mode the indices are computed once and then cached:

$$\begin{aligned} i &= C - n_y, \\ j &= C - n_x, \end{aligned} \quad C = \begin{cases} \text{res\_tot}/2, & \text{even res\_tot}, \\ (\text{res\_tot} - 1)/2, & \text{odd res\_tot}. \end{cases}$$

Thus each row of the returned array has the form  $[i, j]$  so that `k_space_array[i, j]` yields the Fourier coefficient associated with  $(n_x, n_y)$ . Like the mode-number list, the index array is built lazily on first access and then reused for all subsequent queries.

### `.get_row_col_idx_from_nx_ny(tot, nx, ny)`

#### **Method.**

Returns the *row* and *column* indices in a *k*-space array corresponding to a given mode-number pair  $(n_x, n_y)$ .

#### **Parameters**

- `tot` – If `True`, the indices for a *k*-space array in total resolution (`.res_tot`) are returned; otherwise for field-of-view resolution (`.res_fov`).
- `nx, ny` – Integer mode numbers.

#### **Return value**

A tuple `(row, col)` with the indices corresponding to  $(n_x, n_y)$ .

**Index formula**

The indices are computed as:

$$\begin{aligned} i &= C - n_y, \\ j &= C - n_x, \end{aligned} \quad C = \begin{cases} \text{res}/2, & \text{even res}, \\ (\text{res} - 1)/2, & \text{odd res}. \end{cases}$$

This matches the internal convention used in `.mode_indices_fov` and `.mode_indices_tot`. `k_space_array[i, j]` will then yield the Fourier coefficient for  $(n_x, n_y)$ .

**`.get_center_index(fov_only)`****Method.**

Returns the array index corresponding to the *zero* position (center coordinate) along one axis of the grid.

**Parameters**

- `fov_only` — If `True`, return the center index for the field-of-view resolution (`.res_fov`); otherwise return it for the total resolution (`.res_tot`).

**Returns**

An integer index `i_center` such that:

$$i_{\text{center}} = \left\lfloor \frac{N}{2} \right\rfloor.$$

**Usage**

For a 2D light-field array `X` in position space of size  $N \times N$ , the center index can be used to access the pixel corresponding to the physical center:

```
i_center = grid.get_center_index(fov_only)
center_value = X[i_center, i_center]
```

This is useful for inspecting or modifying the spatial center in position space.

**`.limit_mode_numbers(X, mode_limit, k_space_in, k_space_out)`****Method.**

Applies a band-limit in Fourier space, retaining only those modes whose radial index satisfies  $\sqrt{n_x^2 + n_y^2} \leq \text{mode\_limit}$ . The routine automatically converts to and from  $k$ -space as needed, and it works with either FOV or total resolution.

**Parameters**

- `X` – 2-D NumPy array holding the light field, either in position or  $k$ -space.
- `mode_limit` – Maximum radial mode number  $\sqrt{n_x^2 + n_y^2}$  to keep.
- `k_space_in` – `True` if `X` is already in  $k$ -space, `False` if it is in position space.

- `k_space_out` – `True` to return the result in  $k$ -space, `False` to return it in position space.

### Algorithm

1. Determine whether `X` is FOV or total size and select the corresponding mode-number axis (`.n_axis_fov` or `.n_axis_tot`).
2. If `k_space_in = False`, transform `X` to  $k$ -space with `fft2_phys_spatial(...)`.
3. Build a circular aperture mask

$$A(n_x, n_y) = \begin{cases} 1, & \sqrt{n_x^2 + n_y^2} \leq \text{mode\_limit}, \\ 0, & \text{otherwise.} \end{cases}$$

Multiply `X` by this mask.

4. If `k_space_out = False`, transform the result back to position space with `.ifft2_phys_spatial(...)`.

### Return value

A 2-D NumPy array of the same shape as `X`, containing only the modes that satisfy the specified radial limit.



# Appendix B

## Light Fields

### B.1 Light-Field Classes

#### Fourier-optics background

In Fourier optics, a polarized light field at a given longitudinal position  $z$  (the main propagation direction) can be expressed either in *position space* (real space)

$$U(x, y; z)$$

(representing the complex amplitude in the transverse  $(x, y)$  plane), or – after a two-dimensional Fourier transform – in *k-space*

$$A(k_x, k_y; z),$$

where  $(k_x, k_y)$  are the transverse components of the spatial wave vector. For numerical work both representations must be sampled on a finite, square grid. Grid generation, critical sampling, and axis bookkeeping are handled by `clsGrid`.

To accurately simulate light propagation, it is often necessary to embed a smaller *field-of-view* (FOV) region of interest into a larger *total* grid. This ensures that diffraction effects and higher spatial frequencies are correctly captured without introducing artifacts. The size of the total grid must satisfy the *critical-sampling condition* for a given FOV size, wavelength, and propagation distance (see `clsGrid`). The class `clsGrid` provides methods to compute and manage such grids.

Every `clsLightField` instance is linked to an associated `clsGrid` object at construction time, and all conversions between position space, *k*-space, FOV resolution, and total resolution are delegated to that grid.

The class `clsLightField` is the central container class for optical fields in the library. Internally it stores a 2D NumPy array that may reside in field-of-view (FOV) or total resolution and in either position space or *k*-space; the current representation is hidden from the user. The class exposes a uniform interface so the field can be retrieved in any desired representation, manipulated (shift, stretch, add, clone, apply transmission/reflection matrices), plotted, and analysed (energy integrals, peak intensities, etc.).

### Object hierarchy and wavelength access

Every `clsLightField` instance is permanently linked to a `clsGrid` instance (via its `.grid` property). In turn, the associated `clsGrid` instance is linked to a `clsCavity` instance (i.e. `clsCavity1path` or `clsCavity2path`) via `.cavity`. This object chain:

$$\text{clsLightField} \longrightarrow \text{clsGrid} \longrightarrow \text{clsCavity}$$

allows the light-field object to indirectly access cavity-related parameters, such as the current wavelength `.Lambda`. Many internal methods rely on this linkage to correctly model wavelength-dependent effects.

### Subclasses

Several subclasses inherit all functionality of `clsLightField` and can generate specific types of optical fields:

- `clsGaussBeam` – Generates a fundamental Gaussian beam with user-defined waist, curvature, and phase.
- `clsSpeckleField` – Produces a fully developed speckle pattern by superimposing random Fourier modes of equal amplitude and random phase.
- `clsTestImage` – Creates simple test patterns (slits, checkerboards, rings, ...) useful for algorithm verification.
- `clsPlaneWaveMixField` – Allows manual construction of a field by specifying an arbitrary set of plane-wave components (i.e. selected Fourier modes) with chosen amplitudes and phases.

These subclasses provide convenient starting points for simulations while retaining the full suite of manipulation and analysis tools inherited from `clsLightField`.

## B.2 Base Class `clsLightField`

### B.2.1 Initialization

`clsLightField(grid)`

#### Constructor.

Constructs a new instance of `clsLightField` and initializes an empty field. The internal field is initialized as empty (“black”).

#### Parameter

- `grid` – Instance of `clsGrid`. Defines the spatial grid and all resolution and axis conversions used by this light field.

Every `clsLightField`-based class is permanently linked to a `clsGrid` instance, provided at construction time. While the `clsGrid` instance cannot be replaced, its parameters (such as resolution) can be modified through its methods.

### B.2.2 Setter and Getter Methods

#### `.set_field(field, k_space)`

Sets the internal field data to the given array. This method can be used to import externally computed fields or to transfer data between different `clsLightField` instances.

##### Parameters

- `field` – 2D NumPy array containing the light field, either in position space or  $k$ -space. The array must match either field-of-view resolution or total resolution as defined by the associated `clsGrid` instance; the resolution is detected automatically.
- `k_space` – Boolean. If `True`, the input array is interpreted as  $k$ -space data; if `False`, as position-space data.

##### Error handling

If the resolution of the input array cannot be determined, an error message is printed and the field is not modified.

#### `.set_field_vec(vec)`

##### Method.

Sets the internal field data based on a vector of Fourier coefficients.

In this library, a light field in  $k$ -space (Fourier domain) can be represented either as a 2D array or as a 1D *vector* containing the Fourier coefficients ordered according to `clsGrid.mode_numbers_fov` or `clsGrid.mode_numbers_tot`. Such vectors are produced by `clsGrid.arr_to_vec(...)` and can be used for linear operations such as applying transmission or reflection matrices.

##### Parameter

- `vec` – NumPy array representing the vector of Fourier coefficients. May be a 1D array of shape  $(N,)$  (line vector) or a 2D array of shape  $(N, 1)$  (column vector). The vector must match either field-of-view resolution or total resolution as defined by the associated `clsGrid` instance; the resolution is detected automatically.

##### Behavior

The method converts the input vector back into a 2D  $k$ -space array using `clsGrid.vec_to_arr(...)` and stores it internally. The internal field is always stored in  $k$ -space after this operation. The resolution (FOV or total) is determined automatically.

##### Error handling

If the resolution of the input vector cannot be determined, an error message is printed and the field is not modified.

```
.get_field_tot(k_space_out, process=0)
```

### Method.

Returns the current field as a 2D NumPy array in *total resolution*.

The output can be requested either in position space or in  $k$ -space, and an optional `process` parameter allows extracting different representations of the field.

### Parameters

- `k_space_out` – Boolean. If `True`, returns the field in  $k$ -space; if `False`, returns the field in position space.
- `process` – Integer flag (default 0) specifying which form of the field to return:
  - 0 — Complex field (default)
  - 1 — Real part
  - 2 — Imaginary part
  - 3 — Absolute value (`abs`)
  - 4 — Phase
  - 5 — Intensity ( $\text{abs}^2$ )

### Return value

2D NumPy array of shape(`.grid.res_tot, .grid.res_tot`) representing the field in total resolution. The requested form is produced by internal processing.

### Remarks

Internally this method uses `clsGrid.convert(...)` to convert between position space and  $k$ -space if needed.

```
.get_field_fov(k_space_out, process=0)
```

### Method.

Returns the current field as a 2D NumPy array in *field-of-view (FOV) resolution*.

The output can be requested either in position space or in  $k$ -space, and an optional `process` parameter allows extracting different representations of the field.

### Parameters

- `k_space_out` — Boolean. If `True`, returns the field in  $k$ -space; if `False`, returns the field in position space.
- `process` — Integer flag (default 0) specifying which form of the field to return:
  - 0 - Complex field (default)
  - 1 - Real part
  - 2 - Imaginary part
  - 3 - Absolute value (`abs`)
  - 4 - Phase
  - 5 - Intensity ( $\text{abs}^2$ )

**Return value**

2D NumPy array of shape `(.grid.res_fov, .grid.res_fov)` representing the field in field-of-view resolution. The requested form is produced by internal processing.

**Remarks**

Internally this method uses `clsGrid.convert(...)` to convert between position space and  $k$ -space if needed.

`.get_field_tot_vec(column_vec=True)`

**Method.**

Returns the current field in *total resolution* as a vector of Fourier coefficients, ordered according to `.grid.mode_numbers_tot`. The returned vector corresponds to the  $k$ -space representation of the field, flattened into a 1D structure suitable for matrix operations or further processing.

**Parameters**

- `column_vec` – Boolean (default `True`). If `True`, the result is returned as a numpy array of shape `(N, 1)` (column vector). If `False`, as a numpy array of shape `(N, )` (line vector).

**Return value**

NumPy array containing the Fourier coefficients of the field in total resolution, ordered according to `.grid.mode_numbers_tot`.

`.get_field_fov_vec(column_vec=True)`

**Method.**

Returns the current field in *field-of-view (FOV) resolution* as a vector of Fourier coefficients, ordered according to `.grid.mode_numbers_fov`. The returned vector corresponds to the  $k$ -space representation of the field, flattened into a 1D structure suitable for matrix operations or further processing.

**Parameters**

- `column_vec` – Boolean (default `True`). If `True`, the result is returned as a numpy array of shape `(N, 1)` (column vector). If `False`, as a numpy array of shape `(N, )` (line vector).

**Return value**

NumPy array containing the Fourier coefficients of the field in field-of-view resolution, ordered according to `.grid.mode_numbers_fov`.

`.clone()`

**Method.**

Creates and returns a *deep copy* of the current `clsLightField`.

**Return value**

A new `clsLightField` instance that contains:

- the same `clsGrid` reference (the grid object itself is *not* duplicated);
- a duplicated copy of the internal field array (so subsequent modifications to one field do not affect the other);
- the same internal flags (`.k_space` and `.fov_only`);
- the same `.name` string.

### B.2.3 Manipulating the Light Field

#### `.apply_TR_mat(TR)`

##### Method.

Applies a transmission or reflection matrix to the current field and returns the resulting field as a new `clsLightField` instance.

##### Background

In Fourier optics, transmission and reflection matrices are used to model how an optical element (such as a mirror, a lens, or the entire cavity) modifies the complex amplitudes of a light field. These matrices act on the vector of Fourier coefficients representing the field.

If the underlying light field is represented on a grid of  $N \times N$  pixels, the corresponding Fourier representation contains  $N^2$  modes. As a result, the associated transmission or reflection matrix must be of size  $N^2 \times N^2$ . Based on this size, the method can automatically determine whether the matrix applies to field-of-view resolution or total resolution.

##### Parameters

- `TR` – 2D NumPy array representing a transmission or reflection matrix. Its size must match the squared resolution of either FOV or total resolution.

##### Return value

A new `clsLightField` instance containing the transformed field. The output field will always be stored internally in  $k$ -space.

##### Remarks

If the size of `TR` does not match either valid resolution, an error is printed and `None` is returned.

#### `.add(other_field)`

##### Method.

Adds a second `clsLightField` to the current field. Because both fields are represented by arrays of complex numbers, the operation is a *coherent* superposition: constructive or destructive interference can occur depending on their relative phases.

##### Parameter

- `other_field` – Another `clsLightField` instance whose field will be superimposed on the field stored in `self`.

## Behaviour

- If `other_field.empty` is True, nothing is changed.
- If the current field is empty, it is replaced by a copy of `other_field`, inheriting its `k_space` and `fov_only` flags.
- If both fields are present:
  - When both share the same resolution (both FOV or both total), the arrays are added directly.
  - When one field is FOV and the other total, the FOV field is first up-sampled to total resolution via `.get_field_tot(...)`; the sum is then taken on the larger grid.
- The addition is carried out in whatever domain the current field is stored (`k_space` or position space).

## Return value

None. The method modifies the object in place.

### `.stretch_x(factor, center_pos=0)`

#### Method.

Horizontally stretches or compresses the field around a chosen transverse position `center_pos`. The operation rescales the field along the *x*-axis while leaving the *y*-axis unchanged; energy is conserved through amplitude normalisation by invoking `clsGrid.stretch_x(...)`.

#### Parameters

- `factor` – Stretch factor. Values  $> 1$  broaden the field, values  $< 1$  compress it , and 1 leaves the field unchanged.
- `center_pos` – Physical *x*-coordinate (in meters) about which the stretch is performed. The default is 0 (the optical axis).

## Behaviour

- If `factor` is exactly 1, or if the internal field is empty, the call is a no-op.
- Otherwise the method retrieves the field in *total* resolution, applies `clsGrid.stretch_x(...)` (position-space domain), and stores the stretched field back in position space.

## Return value

None. The current object is modified in place.

### `.stretch_y(factor, center_pos=0)`

#### Method.

Vertically stretches or compresses the field around a chosen transverse position `center_pos`. The operation rescales the field along the *y*-axis while leaving the *x*-axis unchanged; energy is conserved through amplitude normalisation by invoking `clsGrid.stretch_y(...)`.

### Parameters

- **factor** – Stretch factor. Values  $> 1$  broaden the field, values  $< 1$  compress it, and 1 leaves the field unchanged.
- **center\_pos** – Physical  $y$ -coordinate (in metres) about which the stretch is performed. The default is 0 (the optical axis).

### Behaviour

- If **factor** is exactly 1, or if the internal field is empty, the call is a no-op.
- Otherwise the method retrieves the field in *total* resolution, applies `clsGrid.stretch_y(...)` (position-space domain), and stores the stretched field back in position space.

### Return value

None. The current object is modified in place.

`.shift(x_shift, y_shift)`

### Method.

Translates the field horizontally and/or vertically by a given physical distance. The shift is performed in position space on the *total* grid; regions that move outside the array are discarded and vacated pixels are filled with zeros.

### Parameters

- **x\_shift** – Signed displacement in the  $x$ -direction (meters). Positive values shift the field to the right.
- **y\_shift** – Signed displacement in the  $y$ -direction (meters). Positive values shift the field upward.

### Return value

A tuple `(x_pix, y_pix)` giving the signed shift in *pixels* after conversion with `clsGrid.dist_to_pixels(...)`. The method returns these values even if the shift in meters was so small that no actual shift (in pixels) was applied.

### Behaviour

1. The physical displacements are converted to integer pixel offsets using `.grid.dist_to_pixels(...)`.
2. If the internal field is empty, or if both pixel offsets are zero, the call is a no-op.
3. If the field is currently stored in  $k$ -space or in FOV resolution, it is first converted to *total* resolution in position space so that the shift can be applied consistently.
4. Horizontal and vertical shifts are realised by rolling the array and padding the vacated pixels with zeros. If the requested displacement exceeds the array size in either direction, the field is cleared.

### B.2.4 Plotting and Analysis

`.intensity_integral_fov(aperture=0)`

#### Method.

Computes the *intensity integral* of the current field over the field-of-view (FOV) grid, interpreted as a physical integral over position space  $\iint |U(x, y)|^2 dx dy$ .

#### Parameters

- `aperture` – Optional (default 0). If greater than zero, the integral is limited to pixels within a circular aperture of the given physical diameter (meters). Only pixels where  $R \leq \text{aperture}/2$  contribute to the integral.

#### Return value

- A single scalar (float), representing the physical integral  $\iint |U(x, y)|^2 dx dy$  over the FOV grid (or over the circular aperture region if specified).

#### Remarks

If the field is empty, returns 0. Internally, the field is retrieved via `.get_field_fov(...)` with `process=5`.

`.intensity_integral_tot(aperture=0)`

#### Method.

Computes the *intensity integral* of the current field over the *total* grid, interpreted as a physical integral over position space  $\iint |U(x, y)|^2 dx dy$ .

#### Parameters

- `aperture` – Optional (default 0). If greater than zero, the integral is limited to pixels within a circular aperture of the given physical diameter (meters). Only pixels where  $R \leq \text{aperture}/2$  contribute to the integral.

#### Return value

A single scalar (float), representing the physical integral  $\iint |U(x, y)|^2 dx dy$  over the total grid (or over the circular aperture region if specified).

#### Remarks

If the field is empty, returns 0. Internally, the field is retrieved via `.get_field_tot(...)` with `process=5`.

`.plot_field(what_to_plot, fov_only=True, save_path=None, c_map='hot', vmax_limit=None, norm=None, vmax=None)`

#### Method.

Displays (or saves) a false-color image of the current field. The method offers flexible choices for *what* quantity to plot, whether to use field-of-view or total resolution, how to scale the colormap, and whether to write the figure to disk.

## Parameters

- `what_to_plot` – Integer selector 1 real part, 2 imaginary part, 3 magnitude ( $|U|$ ), 4 phase, 5 intensity ( $|U|^2$ ). See Figure B.1 for example output with different plot settings.
- `fov_only` – If `True` (default) the FOV grid is shown; otherwise the total grid is shown.
- `save_path` – Optional file path. If supplied, the figure is saved instead of displayed.
- `c_map` – Matplotlib colormap name (default `hot`). Specify '`custom`' to load the bespoke colormap distributed with the package.
- `vmax_limit` – Soft upper clip for the color scale. If the maximum data value exceeds this limit the scale is auto-ranged; otherwise it is clipped at `vmax_limit`.
- `norm` – Maximum value for a linear normalization; useful for plotting only the lower half of a hot colormap.
- `vmax` – Hard upper limit for the color scale. If supplied, it overrides `vmax_limit` and forces the scale to saturate exactly at this value.

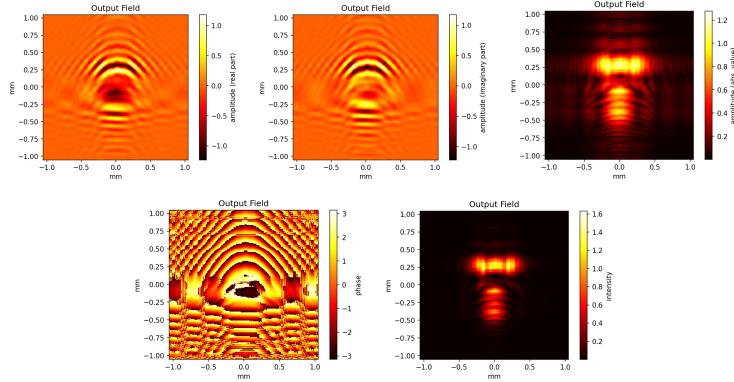


Figure B.1: Example output of `plot_field(...)`: First row: `what_to_plot = 1` (real part), 2 (imaginary part), 3 (absolute value). Second row: 4 (phase), 5 (intensity).

## Behaviour

1. The requested quantity is obtained by invoking the methods `.get_field_fov(...)` or `.get_field_tot(...)` (always in position space).
2. Physical axis limits and units are obtained from `.grid.get_ax_plot_info()`.
3. Color-scale logic
  - If `vmax` is supplied it is used directly.
  - Else if `vmax_limit` is supplied and  $\max(\text{field}) \leq \text{vmax\_limit}$ , the scale is clipped at `vmax_limit`.
  - Otherwise the scale is auto-ranged.

If `norm` is supplied a custom `Normalize` object is built so that the colormap saturates at `norm`.

4. The image is drawn with `origin='lower'` so that the first row appears at the bottom, matching the physical coordinate system.

5. Axis ticks are placed with `AutoLocator` and labeled in the chosen length unit.
6. A color-bar with an appropriate label (*amplitude, phase, intensity, ...*) is added.
7. If `.name` is non-empty it is used as the figure title.
8. Finally the figure is either shown or saved, and the Matplotlib figure is closed to free memory.

**Return value**

None. The method is for visualization / file output and does not modify the field.

### B.2.5 Properties

**`.grid`****Read-only property.**

Returns the `clsGrid` instance associated with this `clsLightField` object. This grid defines the spatial and Fourier-space resolution and coordinate systems used for all field representations.

**Return value**

- The `clsGrid` object linked to this light field.

**`.name`****Read/write property.**

A descriptive name for the `clsLightField` instance.

If set, this name is displayed as the plot title when using `.plot_field(...)`. It can also be used by the user for bookkeeping or identification of the light field object in a larger simulation.

**`.empty`****Read-only property.**

Returns `True` if the internal field of the `clsLightField` instance is currently undefined (`None`); returns `False` otherwise.

Note that this property does not check whether the light field content is physically zero (i.e. a black field), but only whether the internal field array exists.

**`.fov_only`****Read-only property.**

Indicates whether the internal field is currently stored at *field-of-view* (FOV) resolution (`True`) or at *total* resolution (`False`).

Regardless of this internal state, the field can always be retrieved in either resolution using `.get_field_fov(...)` or `.get_field_tot(...)`.

**.k\_space****Read-only property.**

Indicates whether the internal field is currently stored in *k-space* (`True`) or in *position space* (`False`).

Regardless of this internal state, the field can always be retrieved in either representation using `.get_field_fov(...)` or `.get_field_tot(...)`.

**.intensity\_sum\_fov****Read-only property.**

Returns the total *discrete sum* of pixel intensities in the field-of-view (FOV) representation of the current field.

The intensity is computed as the squared magnitude of the complex field at each pixel, i.e.  $|U(x, y)|^2$ . Internally, the result of `.get_field_fov(...)` with `process=5` is used, and `numpy.sum` is applied to it. If the field is empty, returns 0.

**Returns**

A single scalar (float), the sum of all pixel intensities in the FOV grid.

**.intensity\_sum\_tot****Read-only property.**

Returns the total *discrete sum* of pixel intensities in the total-resolution representation of the current field.

The intensity is computed as the squared magnitude of the complex field at each pixel, i.e.  $|U(x, y)|^2$ . Internally, the result of `.get_field_tot(...)` with `process=5` is used, and `numpy.sum` is applied to it. If the field is empty, returns 0.

**Return value**

A single scalar (float), the sum of all pixel intensities in the total-resolution grid.

## B.3 Class `clsGaussBeam`

The class `clsGaussBeam` is a subclass of `clsLightField`. It inherits all methods and properties of `clsLightField`, and thus supports full access to field manipulation, coordinate conversion, plotting, and other utilities.

This class provides functionality to generate a fundamental *Gaussian beam* and store it in the field array.

The electric field of the fundamental  $\text{TEM}_{00}$  Gaussian beam at an axial distance  $z$  from the waist is

$$E(x, y, z) = E_0 \frac{w_0}{w(z)} \exp\left(-\frac{x^2+y^2}{w(z)^2}\right) \exp\left[-i k z - i k \frac{x^2+y^2}{2R(z)} + i \zeta(z)\right],$$

where

- $E_0$  is the peak amplitude at the waist,
- $w_0$  is the waist radius,
- $k = 2\pi/\lambda$  is the wave number,
- $w(z) = w_0\sqrt{1 + (z\lambda/(\pi w_0^2))^2}$  is the beam radius at  $z$ ,
- $R(z) = z[1 + (\pi w_0^2/(z\lambda))^2]$  is the wave-front radius of curvature,
- $\zeta(z) = \arctan(z\lambda/(\pi w_0^2))$  is the Gouy phase.

The generated beam can be positioned and oriented flexibly:

- The beam waist (spot size) can be specified.
- The beam can be centered or offset from the optical axis via appropriate coordinate shifts.
- The beam can propagate along the optical ( $z$ -)axis or under a chosen angle (tilted beam).

### B.3.1 Initialization

`clsGaussBeam(grid)`

#### Constructor.

Creates a new instance of `clsGaussBeam`, derived from `clsLightField`. The constructor links the object to a `clsGrid` instance, as required for all light-field classes in this library.

#### Parameters

- `grid` – A `clsGrid` instance that defines the spatial and  $k$ -space sampling of the field.

#### Behavior

The object is initialized as an empty light field.

#### Remarks

After creation, a Gaussian beam can be generated using the method `.create_beam(...)`.

### B.3.2 Methods

`.create_beam(waist, x_offset=0, y_offset=0,  
x_angle_deg=0, y_angle_deg=0, z=0)`

#### Method.

Generates a fundamental Gaussian beam and stores it as the current field in the `clsGaussBeam` instance. The beam is created in *position space*, using the total grid resolution.

## Parameters

- `waist` – Beam waist  $w_0$  at  $z = 0$  (meters).
- `x_offset`, `y_offset` – Optional. Lateral shift of the beam center in meters (default: centered on the optical axis).
- `x_angle_deg`, `y_angle_deg` – Optional. Propagation tilt angles *in degrees* (not radians!) relative to the  $z$ -axis (horizontal and vertical tilt, respectively). A value of 0 produces a beam aligned with the  $z$ -axis.
- `z` – Optional. Axial position  $z$  (meters) along the beam propagation direction. Allows generation of the beam at a virtual propagation distance.

## Beam model

The generated field represents a fundamental Gaussian beam with waist  $w(z)$ , radius of curvature  $R(z)$ , and Gouy phase  $\zeta(z)$  corresponding to the specified distance  $z$ , as described in the introduction.

If tilt angles are specified, appropriate plane-wave phase factors are applied to the beam to achieve the desired tilt.

## Internal state

- The internal field is set to the generated beam in position space.
- The field is stored in total resolution (`.fov_only` will be `False`).

## Remarks

If called multiple times, the previously stored field will be replaced.

## Important limitation

If a tilt is specified via `x_angle_deg` or `y_angle_deg`, the beam tilt is implemented by applying a phase factor to the beam field. This introduces the correct angular tilt in the wavefront, but does not apply the corresponding geometrical compression of the beam cross-section in the projection onto the  $xy$ -plane. As a result, the beam remains circular on the  $xy$ -plane even at large tilt angles, which is strictly an approximation. For small angles this is usually acceptable, but at larger tilt angles the user may wish to manually apply an appropriate scaling to the beam profile using the methods `clsGrid.stretch_x(...)` and/or `clsGrid.stretch_y(...)`. This limitation will be addressed in a future version of the library.

## `.beam_radius(z)`

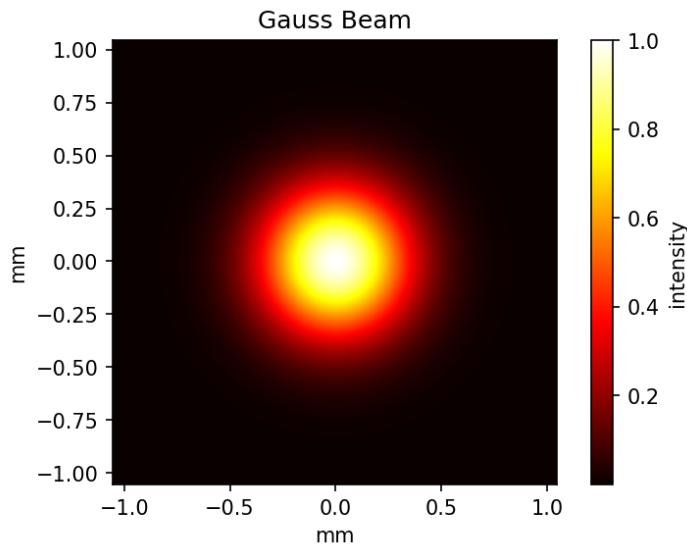
### Method.

Computes the beam radius  $w(z)$  at axial distance  $z$  from the beam waist.

$$w(z) = w_0 \sqrt{1 + \left( \frac{z\lambda}{\pi w_0^2} \right)^2}$$

### Parameter

- `z` – Axial distance (in meters) from the beam waist at which to evaluate the beam radius.



**Figure B.2:** Example Gaussian beam generated by `create_beam(0.0005)`: Centered beam with 0.5 mm waist.

### Return value

- A single float: the beam radius  $w(z)$  at distance  $z$ , in meters.

### Remarks

This is the  $1/e$  amplitude radius of the Gaussian beam envelope at distance  $z$ :  $w(z)$  gives the radius at which the field amplitude drops to  $1/e$  of its on-axis value. The value depends on the beam waist  $w_0$ , the wavelength  $\lambda$ , and the propagation distance  $z$ .

### `.radius_of_curvature(z)`

#### Method.

Computes the *radius of curvature*  $R(z)$  of the beam's phase front at a given longitudinal position  $z$ .

The Gaussian beam is not strictly planar: at each transverse plane  $z$ , the phase fronts are curved surfaces. The *radius of curvature*  $R(z)$  describes the distance from the plane  $z$  to the virtual point where the phase front would converge if extended backward or forward. Mathematically:

$$R(z) = z \left[ 1 + \left( \frac{\pi w_0^2}{z\lambda} \right)^2 \right]$$

### Parameters

- $z$  – Distance from the beam waist (meters).

**Return value**

- A scalar (float)  $R(z)$  in meters. For  $z = 0$ , returns  $+\infty$ , representing a planar phase front at the beam waist.

**Remarks**

Phase fronts of a Gaussian beam transition from planar ( $z = 0$ ) to spherical at large distances. This quantity is used internally when constructing the complex phase of the Gaussian beam (see `.create_beam(...)`).

**`.gouy_phase(z)`****Method.**

Computes the *Gouy phase shift*  $\zeta(z)$  of the Gaussian beam at longitudinal position  $z$ .

The Gouy phase is a characteristic additional phase shift experienced by a focused beam as it propagates through its focus. Unlike a plane wave, a Gaussian beam accumulates an extra phase shift of  $\pi$  radians across its entire Rayleigh range.

Mathematically:

$$\zeta(z) = \arctan\left(\frac{z\lambda}{\pi w_0^2}\right)$$

**Parameters**

- $z$  – Distance from the beam waist (meters).

**Return value**

- A scalar (float), the Gouy phase  $\zeta(z)$  in radians.

**Remarks**

- At the beam waist ( $z = 0$ ):  $\zeta(0) = 0$ .
- As  $z \rightarrow \pm\infty$ :  $\zeta(z) \rightarrow \pm\pi/2$ .
- The full accumulated phase shift across the beam focus is  $\pi$  radians.

This phase shift is included in the construction of the beam in `.create_beam(...)`.

## B.4 Class `clsSpeckleField`

The class `clsSpeckleField` derives from `clsLightField` and therefore inherits the full interface for resolution conversion, plotting, arithmetic operations, and all other utilities already documented for light-field objects. Its specific purpose is to synthesize *random speckle fields*: optical fields whose complex amplitudes are obtained by adding a large number of Fourier modes with equal magnitude and statistically independent random phases. Such speckle patterns occur, for example, after coherent light scatters from a rough surface or passes through a disordered medium.

## Key features

- **Random-mode generation.** Two convenience methods – `.create_field(...)` and `.create_field_eq_distr(...)` – generate speckle patterns by drawing a prescribed number of  $k$ -space modes (plane waves) from the set of all grid points that satisfy  $\sqrt{n_x^2 + n_y^2} \leq n_{\max}$ . Each integer pair  $(n_x, n_y)$  corresponds to a plane wave propagating at a well-defined tilt angle with respect to the optical ( $z$ -) axis.
  - `.create_field(...)` picks modes uniformly from that set of grid points. Because the number of available grid points grows with radius, this produces a bias toward *larger* tilt angles.
  - `.create_field_eq_distr(...)` adjusts the sampling so the resulting plane-wave tilt angles are approximately *uniformly distributed*, giving a more isotropic speckle pattern.
- **Circular aperture in position space.** An optional `aperture` parameter multiplies the field by a circular mask so that the speckle pattern is confined to a finite illumination spot. After masking, the field is passed through `.grid.limit_mode_numbers(...)` to remove spatial-frequency components that exceed the current mode limit ( $\sqrt{n_x^2 + n_y^2} \leq n_{\max}$ ). As a consequence, the aperture edge may appear slightly blurred – consistent with the limited bandwidth – while guaranteeing that no “forbidden” high-order modes are introduced.
- **Angle-aware statistics.** Helper functions such as `.get_max_angle(...)`, `.get_req_emebed_factor(...)`, and `.plot_angle_distribution(...)` allow users to analyse the angular spectrum of the generated field and to estimate how large an embedding grid is required to preserve diffraction features after free propagation.

Because `clsSpeckleField` extends `clsLightField`, the resulting speckle pattern can be manipulated (added, shifted, stretched), converted to  $k$ -space, or plotted exactly like any other light field in the library.

### B.4.1 Initialization

`clsSpeckleField(grid)`

#### Constructor.

Constructs an empty `clsSpeckleField` object and links it to the supplied `clsGrid` instance.

#### Parameter

- `grid` – The `clsGrid` on which the speckle field will be defined. As with all light-field classes, this link is permanent.

After construction, call `.create_field(...)` or `.create_field_eq_distr(...)` to generate an actual speckle pattern.

### B.4.2 Speckle-Generation Methods

```
.create_field(no_of_modes, aperture, seed, fov_equiv=False,
    consider_pos_offset=True)
```

#### Method.

Generates a random speckle pattern by selecting `no_of_modes` Fourier modes with equal amplitude and independent random phases.

#### Parameters

- `no_of_modes` – Number of distinct Fourier *indices* ( $n_x, n_y$ ) that will be populated. In practice we select grid points in the 2-D “index space” ( $n_x, n_y$ ), but each such integer pair maps uniquely to a physical spatial-frequency vector  $(k_x, k_y) = \frac{2\pi}{L}(n_x, n_y)$ . Hence choosing `no_of_modes` indices is equivalent to injecting the same number of plane-wave components in  $k$ -space.
- `aperture` – Physical diameter (meters) of an optional circular mask applied in position space. A value 0 disables masking. After masking, the field is passed once more through `.limit_mode_numbers(...)` so that no spatial-frequency components exceeding the current mode limit are re-introduced. The aperture edge therefore appears slightly blurred – consistent with the finite bandwidth – but the  $k$ -space content remains strictly limited.
- `seed` – Integer seed for the NumPy RNG. If negative, the RNG is left in its current state.
- `fov_equiv` – When `True`, the requested `no_of_modes` is interpreted with respect to *FOV* resolution and then scaled by `.grid.factor`<sup>2</sup> so that the speckle texture (average speckle size) remains visually similar regardless of the padding factor of the total grid. When `False` the mode count refers directly to the total grid.
- `consider_pos_offset` – If `True` (default) the circular aperture mask is centred at the physical FOV position specified by `.grid.pos_offset_x` / `.grid.pos_offset_y`. Otherwise the mask is centered on the total-grid origin.

#### Mode-selection strategy

All integer pairs  $(n_x, n_y)$  that satisfy  $\sqrt{n_x^2 + n_y^2} \leq n_{\max}$  are eligible. The method `.create_field(...)` draws grid points uniformly at random from this set, so modes at larger radii – and therefore larger tilt angles – occur more often simply because there are more of them (see Figure B.4).

#### Effect of `fov_equiv`

Because the number of available grid points scales with the square of the padding factor, moving from FOV to a larger total embedding grid would change the apparent speckle size (fewer, larger speckles). Setting `fov_equiv=True` compensates for this by scaling `no_of_modes` so that the spatial-frequency bandwidth – and hence the speckle texture – matches what would be obtained on the FOV grid.

#### Return value

None. The generated speckle pattern is stored in position space (total resolution).

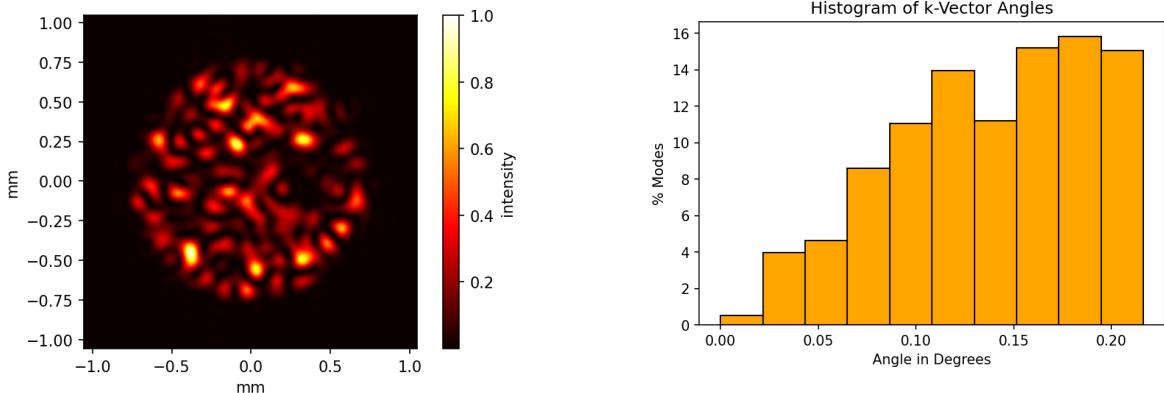


Figure B.3: Speckle pattern generated with `.create_field(500, 0.00015, 0)` on a grid with  $200 \times 200$  pixels.

Figure B.4: Angle distribution of populated  $k$ -space modes: larger tilt angles dominate because the sampling is uniform over the  $(n_x, n_y)$  grid.

```
.create_field_eq_distr(no_of_modes, n_max, aperture, seed,
Lambda=0, consider_pos_offset=True)
```

### Method.

Generates a speckle pattern whose populated plane-wave angles are approximately evenly distributed between 0 and a maximum tilt determined by `n_max`. The method first groups all admissible index pairs  $(n_x, n_y)$  by their radial order  $n_x^2 + n_y^2$ ; it then chooses indices so that each radial shell receives, on average, the same number of modes. Consequently the resulting histogram of propagation angles is nearly flat, in contrast to `.create_field(...)` which naturally favors larger angles.

### Parameters

- `no_of_modes` – Number of modes (plane waves) to draw.
- `n_max` – Maximum radial index  $\sqrt{n_x^2 + n_y^2}$ . If non-positive, the method defaults to the largest index present on the total grid.
- `aperture` – Diameter (meters) of an optional circular mask in position space. Zero disables masking. As with `.create_field(...)`, mode-limiting is applied again after masking to suppress “forbidden” high-order components.
- `seed` – RNG seed ( $< 0 \rightarrow$  keep current RNG state).
- `Lambda` – Wavelength (meters) used to convert  $(n_x, n_y)$  index pairs into physical propagation angles. If 0 (default), the wavelength is automatically taken from the cavity associated with this instance: `.grid.cavity.Lambda`.
- `consider_pos_offset` – Whether the circular mask should be centred at the FOV offset (`True`, default) or at the origin of the total grid (`False`).

### Return value

None. The generated field is stored in position space (total resolution) and normalised so that its peak amplitude is 1.

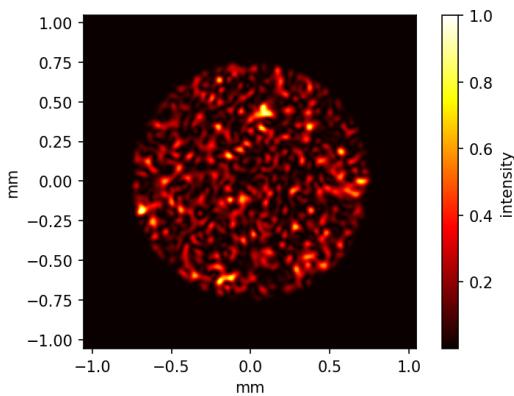


Figure B.5: Speckle pattern generated with `create_field_eq_distr(5000, 30 0.00015, 0)`; total grid  $200 \times 200$ .

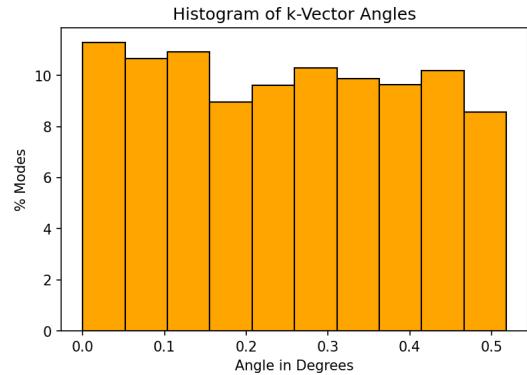


Figure B.6: Angle distribution for the same field: the approximately uniform bar heights confirm the equal-angle sampling strategy.

### B.4.3 Plotting and Analysis

```
.plot_angle_distribution(fov_only, no_of_modes=-1, save_path=None,
Lambda=-1)
```

#### Method.

Plots the angular distribution of the Fourier modes currently present in the speckle field. The method builds a histogram of the tilt angles  $\alpha$  (with respect to the optical  $z$ -axis) corresponding to the  $(n_x, n_y)$  modes currently populated in the field.

#### Parameters

- **fov\_only** – If `True`, the distribution is computed based on the FOV-resolution field; otherwise, the full-resolution field is used.
- **no\_of\_modes** – Optional scaling factor for the histogram counts:
  - If `-1` (default), counts are scaled to represent percentage of modes.
  - If `0`, counts reflect the actual number of populated modes.
  - If positive, counts are scaled to correspond to the specified number of modes.
- **save\_path** – Optional path to save the plot as an image file. If `None` (default), the plot is displayed on-screen.
- **Lambda** – Wavelength (meters) used to convert mode indices to angles. If `-1` (default), the wavelength is automatically taken from the cavity associated with this instance: `.grid.cavity.Lambda`.

#### Plot details

- The tilt angle  $\alpha$  is computed from each populated  $(n_x, n_y)$  mode using `.get_angle_from_nxy(...)`.
- A histogram of the resulting angles (in degrees) is produced, showing the relative population of plane waves with different tilt angles.
- Due to the structure of the  $(n_x, n_y)$  grid, higher angles typically appear more

frequently if the speckle field was generated with `.create_field(...)` (see Figure B.4).

#### Return value

- `hist, bin_edges, intensities`: the histogram data, bin edges, and raw mode intensities used to build the plot.

#### Example Output

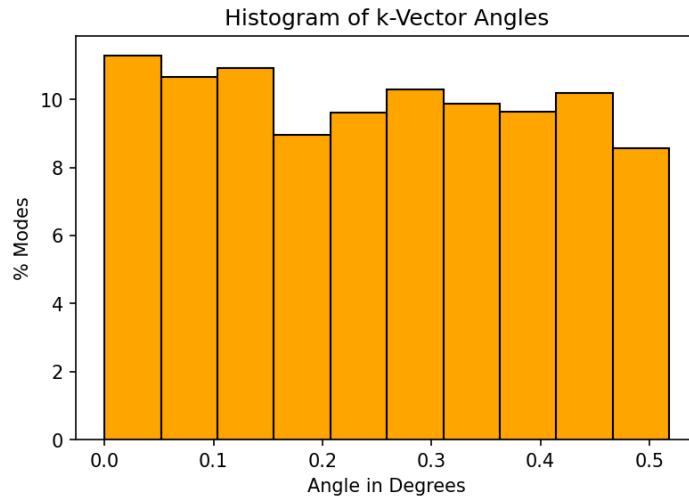


Figure B.7: Example output of `plot_angle_distribution(...)` for a speckle field generated with `create_field_eq_distr(5000, 30, 0.0015, 0)` on a  $200 \times 200$  grid. The histogram shows the relative population of tilt angles  $\alpha$  of the Fourier modes.

### B.4.4 Diagnostic and Helper Methods

#### `.get_aperture()`

##### Method.

Returns the physical aperture diameter (meters) used when generating the current speckle field.

#### Return value

- `float` – Aperture diameter in meters. If no aperture was used (`aperture` parameter set to 0), returns 0.

#### `.get_n_max()`

##### Method.

Returns the current value of  $n_{\max}$ , i.e. the largest admissible integer radius  $\sqrt{n_x^2 + n_y^2}$  in  $k$ -space for the present speckle field.

### How is $n_{\max}$ set?

- `.create_field(...)` computes it automatically from the requested number of modes as  $n_{\max} = \sqrt{\text{no\_of\_modes}/\pi}$ . If `fov_equiv=True`, the mode count is first scaled by `.grid.factor`<sup>2</sup> before the same formula is applied.
- `.create_field_eq_distr(...)` lets the caller supply an explicit `n_max` argument. Passing a non-positive value selects the maximum index supported by the current grid.

### Return value

- `float` – The value of  $n_{\max}$  (dimensionless).

## `.get_no_of_modes()`

### Method.

Returns the number of Fourier modes (`no_of_modes`) used to generate the current speckle field.

### How is `no_of_modes` set?

- When calling `.create_field(...)`, this parameter is passed explicitly by the caller. If `fov_equiv=True`, the internally used mode count is scaled by `.grid.factor`<sup>2</sup>.
- When calling `.create_field_eq_distr(...)`, this parameter is also passed explicitly by the caller and used as-is.

### Return value

- `int` – Number of modes used in creating the current speckle field.

## `.get_max_angle(Lambda, nr)`

### Method.

Returns the maximum  $k$ -vector angle  $\alpha_{\max}$  (in radians) present in the current speckle field.

### Parameters

- `Lambda` – Wavelength in meters.
- `nr` – Real part of the refractive index.

### How is $\alpha_{\max}$ computed?

The method evaluates the maximum possible tilt angle corresponding to the current value of `.get_n_max()`:

$$\alpha_{\max} = \arccos \left( \frac{\sqrt{(nr \cdot k_{\text{tot}})^2 - k_{xy,\max}^2}}{k_{\text{tot}}} \right)$$

where

$$k_{xy,\max} = \text{get\_n\_max} \cdot \frac{2\pi}{L} \quad \text{and} \quad k_{\text{tot}} = \frac{2\pi}{\text{Lambda}}$$

Here,  $L$  is the nominal length of the grid axis in position space.

#### Return value

- `float` – Maximum tilt angle  $\alpha_{\max}$  in radians.

`.get_max_angle_deg(Lambda, nr)`

#### Method.

Returns the maximum  $k$ -vector angle  $\alpha_{\max}$  of the current speckle field, expressed in degrees.

#### Parameters

- `Lambda` – Wavelength in meters.
- `nr` – Real part of the refractive index.

#### Return value

- `float` – Maximum tilt angle  $\alpha_{\max}$  in degrees.

#### Note

Works like `.get_max_angle(...)`, but returns the value in degrees.

`.get_req_emebed_factor(dist, Lambda, nr, alpha_deg=-1)`

#### Method.

Returns the *embedding factor* – that is, how many times larger the *total* grid side length must be than the current field-of-view (FOV) side length `.grid.length_fov` so that, after free-space propagation over a distance `dist`, even the most highly tilted  $k$ -vectors of the speckle field remain inside the simulation domain.

#### Parameters

- `dist` – Propagation distance in meters.
- `Lambda` – Wavelength in meters.
- `nr` – Real part of the refractive index.
- `alpha_deg` (optional, default `-1`) – Maximum tilt angle in degrees to consider. If `-1`, the method uses `.get_max_angle(...)`.

#### Return value

- `float` – Embedding factor  $\frac{\text{total grid side length}}{\text{FOV side length}}$ .

#### Usage

The returned value corresponds to the `factor` argument of the initialization method `.grid.set_opt_res_based_on_sidelength(...)` to create a total grid that safely accommodates the field after propagation.

## B.5 Class `clsTestImage`

The class `clsTestImage` derives from `clsLightField` and therefore inherits the full interface for resolution conversion, plotting, arithmetic operations, and all other utilities already documented for light-field objects. Its specific purpose is to generate simple predefined test images, such as various "T"-shaped patterns or square beams, directly in the optical field. These test patterns are useful for debugging, alignment checks, and testing the behavior of light propagation through optical systems.

### B.5.1 Initialization

```
clsTestImage(grid)
```

#### Constructor.

Constructs an empty `clsTestImage` object and links it to the supplied `clsGrid` instance.

#### Parameters

- `grid` – The `clsGrid` on which the test image will be defined. As with all light-field classes, this link is permanent.

After construction, call `.create_test_image(...)` to generate an actual test pattern.

### B.5.2 Test Image Creation

```
.create_test_image(image, flip_horizontal=False,
flip_vertical=False)
```

#### Method.

Generates a predefined test pattern on the FOV grid.

#### Parameters

- `image` – Integer code selecting the test pattern to generate:
  - 1: Small centered "T"
  - 2: Large centered "T"
  - 3: Large centered "T" with phase shifts
  - 4: Large off-center "T"
  - 5: Large off-center "T" with phase shifts
  - 6: Square beam with side length equal to 1/5 of the total grid width
- `flip_horizontal` (default `False`) – If `True`, flips the test image horizontally.
- `flip_vertical` (default `False`) – If `True`, flips the test image vertically.

#### Return value

- None. The generated test pattern is stored in position space on the FOV grid.

#### Notes

The test patterns can be used for visual inspection, debugging, or to verify the perfor-

mance of propagation routines and cavity configurations. Patterns with phase shifts (`image = 3` and `5`) contain spatially varying phase profiles, suitable for testing phase-sensitive behavior.

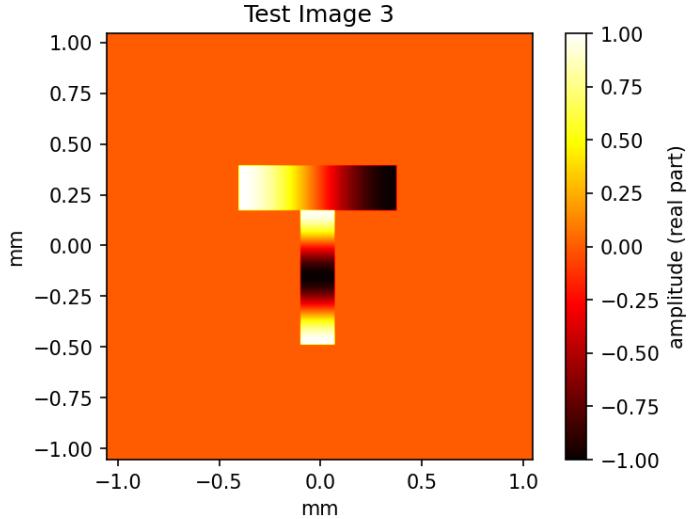


Figure B.8: Test image generated with `.create_test_image(...)` and parameter `image = 3`: – A T-shape with phase shifts.

## B.6 Class `clsPlaneWaveMixField`

The class `clsPlaneWaveMixField` derives from `clsLightField` and therefore inherits the full interface for resolution conversion, plotting, arithmetic operations, and all other utilities already documented for light-field objects. Its specific purpose is to synthesise optical fields composed of one or more user-specified plane waves, each defined by a discrete Fourier index  $(n_x, n_y)$  and complex amplitude. Such fields are useful for constructing test cases with known angular components, for building customised plane-wave superpositions, or for preparing incident fields with controlled  $k$ -vector content.

### B.6.1 Initialization

`clsPlaneWaveMixField(grid)`

#### Constructor.

Constructs an empty `clsPlaneWaveMixField` object and links it to the supplied `clsGrid` instance.

#### Parameters

- `grid` – The `clsGrid` on which the plane-wave field will be defined. As with all light-field classes, this link is permanent.

After construction, the field is initialised to zero. Call `.add_fourier_basis_func(...)` one or more times to add plane-wave components.

### B.6.2 Add Plane Wave

```
.add_fourier_basis_func(nx, ny, amplitude)
```

#### Method.

Adds a single plane-wave component with Fourier index  $(n_x, n_y)$  and complex amplitude to the current field.

#### Parameters

- `nx` – Integer Fourier index  $n_x$ .
- `ny` – Integer Fourier index  $n_y$ .
- `amplitude` – Complex amplitude to assign to this mode.

#### How it works

The method uses `.grid.fourier_basis_func(...)` to generate a normalised Fourier basis function for the given  $(n_x, n_y)$  index pair. This basis function is transformed to  $k$ -space and added (with the specified amplitude) to the current field.

#### Return value

- None. The current field is updated in place.

## Appendix C

# Optical Components: Main Class

Optical components are the *building blocks* of every simulation in this library. Each component – whether it is a lens, mirror, an aperture, or simply a segment of free-space propagation – exposes a compact interface that allows it to be chained with others. Figure C.1 summarises the full class hierarchy.

### Two-port components and `clsCavity1path`

The majority of familiar elements (lenses, mirrors, gratings, soft apertures, etc.) can be modelled as *two-port* components: they possess a **left** side and a **right** side. A linear arrangement of such elements is built on top of `clsCavity1path`. This class acts like an optical “breadboard”: you may append components in the order they appear along a single beam path using `.add_component(component)`. In this way one can model a single resonator, a chain of coupled cavities separated by partially reflective mirrors, or any other one-dimensional optical sequence.

### Four-port components and `clsCavity2path`

Beamsplitters and similar devices require a richer abstraction: a *four-port* component has two physical faces, each of which couples *two* distinct beam paths. The class `clsCavity2path` allows you to stack such four-port components – e.g. multiple beamsplitter mirrors – to form interferometers or coupled resonator networks.

Crucially, `clsCavity2path` can also host ordinary two-port elements using the class `clsOptComponentAdapter`: a lens or mirror can e.g. be placed in one (or both) arms of an interferometer without breaking the four-port formalism.

In summary, `clsOptComponent` and its subclasses provide a modular toolkit for constructing complex optical systems. Two-port components live in a single spatial path and chain together in `clsCavity1path`, while four-port components couple two paths and are assembled in `clsCavity2path`. The adapter class bridges both worlds, ensuring that any combination of elements can be modelled seamlessly.



Figure C.1: Class hierarchy of optical components. The root `clsOptComponent` splits into two branches: 2-port and 4-port components with their subclasses.

## C.1 Abstract Base Class `clsOptComponent`

The class `clsOptComponent` is the abstract base class for all optical components in the library, serving as the root of the component class hierarchy (see Figure C.1). It defines the common interface and basic functionality required for both two-port and four-port optical components.

Being an *abstract* class, `clsOptComponent` is *not intended to be instantiated directly*. Instead,

it provides a foundation for all subclasses by implementing core mechanisms such as:

- standardized storage and access to the grid and cavity context,
- unified management of the working wavelength,
- control of memory and file caching for the component's transfer matrix representation,
- a common interface to declare input/output space preferences (position space vs.  $k$ -space).

The actual physical behaviour of a component – how it modifies the optical field – is expressed through its **transfer and reflection matrices**, whose structure and calculation is the responsibility of each concrete subclass.

`clsOptComponent` also defines the caching and loading mechanisms for these matrices, and provides properties to assist in the efficient chaining of components in a cavity simulation.

All two-port and four-port components derive from this base class and inherit its interface.

### C.1.1 Initialization

`clsOptComponent(name, cavity)`

#### Constructor.

Initialises a new optical component instance.

#### Parameters

- `name` – A descriptive name for the component (string). Used for display and file naming.
- `cavity` – The `clsCavity` instance to which this component belongs. The component is automatically connected to the cavity's grid and wavelength context if this argument is provided.

#### How is the component connected to the cavity?

If a `cavity` instance is provided at construction time, the component is immediately linked to the cavity by calling its internal `_connect_to_cavity(...)` method. This ensures that:

- the component's internal `grid` reference points to the cavity's grid,
- the component's wavelength `Lambda` is set to match the cavity's wavelength,
- the component is aware of the cavity instance to which it belongs.

Later, when the component is formally added to a cavity via the cavity method `clsCavity.add_component(component)`, the cavity will again call `_connect_to_cavity(...)` to assign a unique index (`idx`) to the component. This index ensures that each component can be uniquely identified inside the cavity and is used, together with the `name`, to create unique temporary file names (e.g. for transfer matrix caching).

#### Typical use case

`clsOptComponent` is not instantiated directly, but its constructor is invoked by all

subclasses. Passing a `cavity` at this stage is optional: if omitted, the component will be connected later when added to a cavity via `clsCavity.add_component(component)`.

### `.connect_to_cavity(cavity, grid, idx)`

#### Helper method.

Links the component to a given cavity and grid context, and assigns its index within the cavity.

#### Parameters

- `cavity` – The `clsCavity` instance to which this component will be attached.
- `grid` – The grid instance to be used for this component. Typically, this will be the cavity’s grid.
- `idx` – The unique index to assign to the component within the cavity. If the component’s `idx` is still `-1`, it will be updated to this value.

#### Behavior

This method is called automatically when a component is:

- instantiated with a `cavity` argument, and/or
- formally added to a cavity via `clsCavity.add_component(component)`.

The method stores references to the cavity and grid, synchronises the component’s wavelength `Lambda` with the cavity, and assigns the component index if needed. If the wavelength in the cavity changes, the cavity will automatically update the wavelength `Lambda` in all connected optical components.

#### Typical use case

This method is not intended to be called manually in user code. It forms part of the internal mechanism that ensures each component in a cavity is properly linked and indexed.

## C.1.2 Properties

### `.name`

#### Read/write property.

A descriptive name for the `clsOptComponent` instance.

#### Behavior

The `name` is a user-defined string that serves both as a descriptive label for the component and as an identifier used in filenames for transfer matrix caching. It can be set at component creation time (via the constructor argument `name`) and changed at any later point using this property.

#### Typical use case

Assign meaningful names to components to facilitate debugging, interpretation of simulation output, and the tracking of progress during long simulations. The `name` is also used when constructing temporary filenames for transfer matrix caching.

### [.idx](#)

#### **Read/write property.**

Unique index of the component within its containing `clsCavity` instance.

#### **Behavior**

`idx` is an integer value that uniquely identifies the component within its cavity. It is initially set to `-1` upon component construction. When a two-port-component is added to a cavity via `clsCavity.add_component(component)` or a four-port-component via `clsCavity2path.add_4port_component(component)`, the cavity assigns it a unique index (typically in ascending order of components). This index is used, for example, in constructing unique filenames for caching the component's transfer matrix and its inverse.

#### **Typical use case**

You normally do not set or modify `idx` manually; it is managed automatically by the cavity infrastructure. You may read this property to determine the order in which the component was added to the cavity or to inspect the internal organisation of a simulation.

### [.Lambda](#)

#### **Read/write property.**

Wavelength used by the component, in meters.

#### **Behavior**

`Lambda` defines the working wavelength of the component, expressed in meters. Changing this property automatically updates the corresponding `.Lambda_nm` value and triggers a call to `.clear_mem_cache()`.

This ensures that all cached matrices (transfer matrix, inverse transfer matrix, and – in subclasses – any other component-specific matrices such as transmission or reflection matrices) are invalidated and deleted from memory. This is necessary because any such matrices computed at a previous wavelength are no longer valid after the wavelength changes.

#### **Typical use case**

Normally, you do not set `Lambda` manually on individual components. The cavity instance manages the wavelength centrally and automatically updates `Lambda` for all linked components when the cavity's wavelength changes. The automatic clearing of cached matrices ensures consistency across the entire optical system.

### [.Lambda\\_nm](#)

#### **Read/write property.**

Wavelength used by the component, in nanometers.

#### **Behavior**

`.Lambda_nm` defines the working wavelength of the component, expressed in nanometers. Changing this property automatically updates the corresponding `.Lambda` property (in

meters) and triggers a call to `.clear_mem_cache()`.

Conversely, if you change `.Lambda`, the value of `.Lambda_nm` is automatically kept consistent. Thus, both `.Lambda` and `.Lambda_nm` always represent the same wavelength, simply in different units.

As with `.Lambda`, calling `.clear_mem_cache()` ensures that all cached matrices are invalidated and recomputed when necessary.

#### Typical use case

Normally, you do not set `.Lambda_nm` manually on individual components. The `clsCavity` instance manages the wavelength centrally and automatically updates both `.Lambda` and `.Lambda_nm` for all linked components when the cavity's wavelength changes.

### `.grid`

#### Read-only property.

Reference to the `clsGrid` instance associated with this component.

#### Behavior

`.grid` returns the grid instance that is linked to this component. The grid is set automatically when the component is connected to a `clsCavity` via `._connect_to_cavity(...)`, either at construction time or when the component is added to a cavity.

#### Typical use case

Normally, the grid is accessed through the `.grid` property of the cavity, and then mostly for setting grid parameters via the corresponding methods of `clsGrid`.

### `.cavity`

#### Read-only property.

Reference to the `clsCavity` instance to which this component is connected.

#### Behavior

`.cavity` returns the cavity instance that this component is linked to. The cavity reference is set automatically when the component is connected via `._connect_to_cavity(...)`, either at construction time or when the component is added to a cavity (via `clsCavity.add_component(component)` or with the method `clsCavity2path.add_4port_component(component)`).

### C.1.3 Helper Properties for Efficient Chaining

When a cavity chains together multiple optical components to compute the combined effect on a light field, one method is to simply pass the output of one component as input to the next component. For example, when simulating left-to-right propagation, the cavity must feed the right-side output of one component into the left-side input of the following component.

At each step, the chaining algorithm must decide whether to transfer the light field in *position space* or in *k-space*. Both representations are always usable, as each component can internally convert between them via Fourier transform. However, in many cases, a component natu-

rally receives or produces its field in a particular representation – either because the internal computation is based in that space or because it avoids an unnecessary transform.

To assist the chaining algorithm, every optical component exposes four helper properties:

- `.k_space_in_prefer` – whether  $k$ -space input is preferred;
- `.k_space_in_dont_care` – whether the component has no input preference;
- `.k_space_out_prefer` – whether  $k$ -space output is preferred;
- `.k_space_out_dont_care` – whether the component has no output preference.

This information allows the cavity to minimise unnecessary Fourier transforms during chaining, which improves simulation performance and reduces numerical artifacts. The exact meaning of each property is explained below.

### `.k_space_in_prefer`

#### **Read-only property.**

Indicates whether the component prefers to receive input in  $k$ -space.

#### **Behavior**

`.k_space_in_prefer` returns `True` if it is more natural for the component to process its input field in  $k$ -space, and `False` if position-space input is preferred.

This does not mean that  $k$ -space input is required – both  $k$ -space and position-space input are always supported – but knowing this preference allows the cavity chaining algorithm to select the representation that avoids unnecessary Fourier transforms.

#### **Important**

If `.k_space_in_dont_care` is `True`, this flag is irrelevant and ignored by the chaining algorithm: in that case, the component declares that it can equally well process input in either representation.

#### **Typical use case**

Used internally by the cavity chaining mechanism to optimize the choice of representation when passing the field between components.

### `.k_space_in_dont_care`

#### **Read-only property.**

Indicates whether the component has no preference regarding the representation of its input field.

#### **Behavior**

`.k_space_in_dont_care` returns `True` if the component can equally well process its input field in either  $k$ -space or position space. In this case, the chaining algorithm is free to choose the most convenient representation, typically based on the representation used by the preceding component.

If `.k_space_in_dont_care` returns `True`, the value of `.k_space_in_prefer` can be ignored. If `.k_space_in_dont_care` returns `False`, then `.k_space_in_prefer` indicates

the preferred input representation.

#### Typical use case

Used internally by the cavity chaining mechanism to optimize the choice of representation when passing the field between components.

### `.k_space_out_prefer`

#### Read-only property.

Indicates whether the component prefers to provide its output in *k*-space.

#### Behavior

`.k_space_out_prefer` returns `True` if it is more natural for the component to produce its output field in *k*-space, and `False` if position-space output is preferred.

This does not mean that *k*-space output is required – both *k*-space and position-space output are always supported – but knowing this preference allows the cavity chaining algorithm to select the representation that avoids unnecessary Fourier transforms.

#### Important

If `.k_space_out_dont_care` is `True`, this flag is irrelevant and ignored by the chaining algorithm: in that case, the component declares that it can equally well provide output in either representation.

#### Typical use case

Used internally by the cavity chaining mechanism to optimize the choice of representation when passing the field between components.

### `.k_space_out_dont_care`

#### Read-only property.

Indicates whether the component has no preference regarding the representation of its output field.

#### Behavior

`.k_space_out_dont_care` returns `True` if the component can equally well provide its output field in either *k*-space or position space. In this case, the chaining algorithm is free to choose the most convenient representation, typically based on the representation used by the preceding component.

If `.k_space_out_dont_care` returns `True`, the value of `.k_space_out_prefer` can be ignored. If `.k_space_out_dont_care` returns `False`, then `.k_space_out_prefer` indicates the preferred output representation.

#### Typical use case

Used internally by the cavity chaining mechanism to optimize the choice of representation when passing the field between components.

### C.1.4 Memory and Temporary File Management

#### `.mem_cache_M_bmat`

##### **Read/write property.**

Controls whether the component's transfer matrix (*M-matrix*) is cached in memory.

##### **Behavior**

`.mem_cache_M_bmat` determines whether the component should cache its transfer matrix (*M-matrix*) and its inverse in memory after they are computed.

If this property is set to `True`, then:

- The next time the *M-matrix* is computed (e.g. when accessing `clsOptComponent2port.M_bmat_tot` or `clsOptComponent4port.M_bmat_tot`), it will be stored in memory.
- The next time the inverse *M-matrix* is computed, it will also be stored in memory.
- Subsequent accesses to either matrix will retrieve it from memory instead of recomputing it.

If `.mem_cache_M_bmat` is set to `True`, then `.file_cache_M_bmat` is automatically set to `False`. The two caching modes are mutually exclusive.

If this property is set to `True`, then:

- The next time the *M-matrix* is computed (e.g. when accessing `.M_bmat_tot` or `.M_bmat_tot`), it will be stored in memory.
- Subsequent accesses to the *M-matrix* will retrieve it from memory instead of recomputing it.

If `.mem_cache_M_bmat` is set to `True`, then `.file_cache_M_bmat` is automatically set to `False`. The two caching modes are mutually exclusive.

##### **Purpose of the M-matrix**

The *M-matrix* (transfer matrix) is a block matrix that represents the full effect of the component on the optical field. Its computation can be costly, as it involves combining all transmission and reflection matrices through various matrix operations. Caching the *M-matrix* in memory saves time when the matrix is needed repeatedly.

##### **Typical use case**

Set `.mem_cache_M_bmat` to `True` when running simulations where the same component is used repeatedly and memory is sufficient to hold the cached matrix. This avoids recomputation and improves simulation speed.

#### `.file_cache_M_bmat`

##### **Read/write property.**

Controls whether the component's transfer matrix (*M-matrix*) is cached to disk.

##### **Behavior**

`.file_cache_M_bmat` determines whether the component should save its transfer matrix (*M-matrix*) and its inverse to disk after they are computed.

If this property is set to `True`, then:

- The next time the *M-matrix* is computed (e.g. when accessing `clsOptComponent2port.M_bmat_tot` or `clsOptComponent4port.M_bmat_tot`), it will be saved to a temporary file.
- The next time the inverse *M-matrix* is computed, it will also be saved to a file.
- Subsequent accesses will first check for the file and load it from disk if available, avoiding recomputation.

If `.file_cache_M_bmat` is set to `True`, then `.mem_cache_M_bmat` is automatically set to `False`. The two caching modes are mutually exclusive.

### Purpose of the M-matrix

The *M-matrix* (transfer matrix) is a block matrix that represents the full effect of the component on the optical field. Its computation can be costly, as it involves combining all transmission and reflection matrices through various matrix operations. Caching the matrix to file avoids recomputation and saves memory, especially for large systems.

### Typical use case

Set `.file_cache_M_bmat` to `True` when working with large components where memory usage is a concern. This ensures efficient reuse of previously computed matrices without consuming RAM.

#### `.clear_mem_cache()`

This method deletes all cached matrices from memory, if they are currently cached. This includes the component's transfer matrix (*M-matrix*) and its inverse; and derived sub-classes will additionally delete any other component-specific cached matrices (e.g. lens masks, reflection and transmission matrices, etc.) from memory, if currently cached.

After calling this method, `.mem_cache_M_bmat` remains unchanged, but any cached matrix content is cleared. Subsequent accesses will trigger a recomputation of the corresponding matrix.

### Typical use case

`.clear_mem_cache()` is called automatically when the wavelength of the component changes (see `.Lambda` and `.Lambda_nm`). It can also be called manually to force recalculation of the component's matrices if needed.

#### `.load_M_bmat_tot()`

### Method.

This method loads the component's transfer matrix (*M-matrix*) from a temporary file on disk and stores it in memory.

It is normally invoked automatically if `.file_cache_M_bmat` is `True` and the *M-matrix* is accessed (e.g. via `clsOptComponent2port.M_bmat_tot` or `clsOptComponent4port.M_bmat_tot`). If a corresponding file exists, it is loaded using the filename returned by `clsCavity.full_file_name(...)`.

**Typical use case**

Called automatically by the caching mechanism. Manual calls are rarely needed.

**.load\_inv\_M\_bmat\_tot()****Method.**

This method loads the inverse of the component's transfer matrix (*M-matrix*) from a temporary file on disk and stores it in memory.

It is normally invoked automatically if `.file_cache_M_bmat` is `True` and the inverse *M-matrix* is accessed (e.g. via `.inv_M_bmat_tot`). If a corresponding file exists, it is loaded using the filename returned by `clsCavity._full_file_name(...)`.

**Typical use case**

Called automatically by the caching mechanism. Manual calls are rarely needed.

**.save\_M\_bmat\_tot()****Method.**

This method saves the component's transfer matrix (*M-matrix*) to a temporary file on disk.

It is normally invoked automatically if `.file_cache_M_bmat` is `True` and the *M-matrix* is computed. The filename is determined by `clsCavity._full_file_name(...)`.

**Typical use case**

Called automatically by the caching mechanism. Manual calls are rarely needed.

**.save\_inv\_M\_bmat\_tot()****Method.**

This method saves the inverse of the component's transfer matrix (*M-matrix*) to a temporary file on disk. It is normally invoked automatically if `.file_cache_M_bmat` is `True` and the inverse *M-matrix* is computed. The filename is determined by `clsCavity._full_file_name(...)`.

**Typical use case**

Called automatically by the caching mechanism. Manual calls are rarely needed.

**.delete\_M\_bmat\_tot()****Method.**

This method deletes the temporary files containing the component's transfer matrix (*M-matrix*) and its inverse, if such files exist. It is normally invoked by the `clsCavity` instance when the temporary files of this component are no longer needed. The actual deletion is performed via the method `clsCavity.delete_mat(...)` using the filename provided by `clsCavity._full_file_name(...)`.

**Typical use case**

Called automatically by the cavity when cleaning up temporary files.



## Appendix D

# 2-Port Optical Components for Linear Cavities

### D.1 Abstract Base Class `clsOptComponent2port`

The abstract base class `clsOptComponent2port` represents all *2-port optical components*. These are components with one optical interface on the left and one on the right side. Examples include free-space propagation, lenses, and mirrors. All such components can be chained together in a one-dimensional sequence, as done in a `clsCavity1path` cavity.

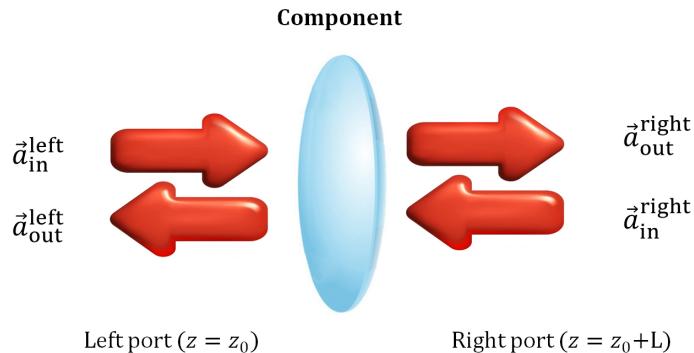


Figure D.1: Conceptual view of a 2-port component.  $\mathbf{a}_{\text{in}}^{\text{left}}$ ,  $\mathbf{a}_{\text{out}}^{\text{left}}$ ,  $\mathbf{a}_{\text{in}}^{\text{right}}$ , and  $\mathbf{a}_{\text{out}}^{\text{right}}$  are all vectors, representing the in- and outgoing light fields by means of Fourier coefficients.

Class `clsOptComponent2port` derives from `clsOptComponent` and remains abstract: it cannot be instantiated directly. Instead, it defines a standard interface that all 2-port components must implement.

Specifically, it provides:

- A `.prop(...)` method to propagate a light field either left-to-right (LTR) or right-to-left (RTL), supporting both position space and  $k$ -space representations.
- Abstract properties to access the key interface matrices:

- `.T_LTR_mat_tot` – left-to-right transmission matrix.
- `.T_RTL_mat_tot` – right-to-left transmission matrix.
- `.R_L_mat_tot` – left reflection matrix.
- `.R_R_mat_tot` – right reflection matrix.

These matrices represent the linear response of the component at its left and right ports. They operate on the total-resolution grid.

- From these four matrices, the class constructs two composite  $2 \times 2$  *block matrices*:
  - The *scattering matrix* (S-matrix), accessed via `.S_bmat_tot`, maps incoming fields to outgoing fields at both ports.
  - The *transfer matrix* (M-matrix), accessed via `.M_bmat_tot`, relates the total field amplitudes at one side of the component to the other, and is essential in cavity modeling.

In this library, we have implemented the following convention for any scattering matrix **S**:

$$\begin{pmatrix} \mathbf{a}_{\text{out}}^{\text{left}} \\ \mathbf{a}_{\text{out}}^{\text{right}} \end{pmatrix} = \mathbf{S} \begin{pmatrix} \mathbf{a}_{\text{in}}^{\text{left}} \\ \mathbf{a}_{\text{in}}^{\text{right}} \end{pmatrix} = \begin{pmatrix} \mathbf{R}_L & \mathbf{T}_{RL} \\ \mathbf{T}_{LR} & \mathbf{R}_R \end{pmatrix} \begin{pmatrix} \mathbf{a}_{\text{in}}^{\text{left}} \\ \mathbf{a}_{\text{in}}^{\text{right}} \end{pmatrix}$$

The scattering matrix **S** is a block matrix composed of four sub-matrices:  $\mathbf{R}_L$  and  $\mathbf{R}_R$  are the reflection matrices for light incident from the left and right, respectively, while  $\mathbf{T}_{LR}$  and  $\mathbf{T}_{RL}$  are the transmission matrices for light passing from left-to-right and right-to-left, respectively. For transfer-matrices our software adheres to the following convention:

$$\begin{pmatrix} \mathbf{a}_{\text{out}}^{\text{left}} \\ \mathbf{a}_{\text{in}}^{\text{left}} \end{pmatrix} = \mathbf{M} \begin{pmatrix} \mathbf{a}_{\text{in}}^{\text{right}} \\ \mathbf{a}_{\text{out}}^{\text{right}} \end{pmatrix}$$

### D.1.1 Initialization

```
clsOptComponent2port(name, cavity)
```

#### Constructor.

Class `clsOptComponent2port` derives from `clsOptComponent` and remains abstract: it cannot be instantiated directly. For details we refer to the description of the constructor `clsOptComponent(...)`.

### D.1.2 Light Field Propagation

```
.prop(E_in, k_space_in, k_space_out, direction)
```

#### Abstract method.

Propagates a light field through the component, either left-to-right (LTR) or right-to-left (RTL). The input and output representations can be either in position space or *k*-space.

### Parameters

- `E_in` – Input field. Usually a full-resolution NumPy array, representing the optical field either in position space or in  $k$ -space, depending on `k_space_in`. For some component classes, when `k_space_in` is `True`, `E_in` may also be given as a scalar, which is interpreted as that scalar multiplied by the identity matrix (i.e., a uniform  $k$ -space input field).
- `k_space_in` – If `True`, `E_in` is interpreted as a  $k$ -space array (or scalar); if `False`, as a position-space array.
- `k_space_out` – If `True`, the returned field is in  $k$ -space; if `False`, in position space.
- `direction` – Direction of propagation, specified as `Dir.LTR` (left-to-right) or `Dir.RTL` (right-to-left).

### Behavior

This method implements the core propagation operation of the component. It is responsible for transforming a specific incoming field into the corresponding outgoing field, respecting the selected propagation direction and the requested input/output representations.

The cavity chaining algorithm can use this method internally to propagate a specific light field through sequences of components.

### Typical use case

Used internally by `clsCavity1path` when simulating the effect of the optical path on a specific light field. Not typically called directly by the user.

## D.1.3 Transmission and Reflection Matrices

### `.T_LTR_mat_tot`

#### Read-only property.

Returns the left-to-right transmission matrix at total resolution.

### Behavior

`.T_LTR_mat_tot` returns the component’s transmission matrix for light propagating from the left side to the right side. The matrix describes how an arbitrary input field, decomposed into plane wave basis functions (in  $k$ -space), is transformed during transmission through the component.

While `.prop(...)` computes the effect of the component on a *specific* input field, `.T_LTR_mat_tot` fully characterizes how *any* input field will be transformed: it represents the linear operator corresponding to the component’s transmission behavior. In derived component classes, the transmission matrix is computed on first access and then cached for efficient reuse.

### Matrix dimensions and meaning

An optical field discretized over a grid with  $N \times N$  pixels has  $N^2$  Fourier coefficients in Fourier space (“ $k$ -space”). The transmission matrix is therefore an array of size  $N^2 \times N^2$ , mapping modal input amplitudes to output amplitudes. Each row and column corresponds to a plane wave basis function, ordered according to

`clsGrid.mode_numbers_tot`. The matrix element at position  $(i, j)$  specifies how strongly the  $j$ -th input mode couples into the  $i$ -th output mode.

#### Typical use case

Used internally to construct the component's *scattering matrix* and *transfer matrix*, and in advanced analyses where an explicit representation of the component's transmission operator is required. To explore the effect of this matrix on a specific light field, you can apply it using `clsLightField.apply_TR_mat(...)`.

### `.T_RTL_mat_tot`

#### Read-only property.

Returns the right-to-left transmission matrix at total resolution.

#### Behavior

`.T_RTL_mat_tot` is fully analogous to `.T_LTR_mat_tot`, but describes transmission in the opposite direction: it returns the matrix representing how an arbitrary input field is transformed when propagating from the right side to the left side through the component.

All remarks regarding matrix dimensions, ordering, and use cases given in the documentation of `.T_LTR_mat_tot` apply identically here. To explore the effect of the transmission matrix on a specific light field, you can apply it using `clsLightField.apply_TR_mat(...)`.

### `.R_L_mat_tot`

#### Read-only property.

Returns the left-side reflection matrix at total resolution.

#### Behavior

`.R_L_mat_tot` provides the matrix that describes how an arbitrary input field is reflected when incident from the left side of the component.

Like `.T_LTR_mat_tot`, this matrix acts on modal amplitudes in  $k$ -space and has shape  $N^2 \times N^2$ . Each element quantifies how a given incoming mode on the left couples into a reflected mode, also on the left. The basis functions and ordering follow `clsGrid.mode_numbers_tot`.

#### Typical use case

Used internally for computing the component's *scattering matrix* and *transfer matrix*, and may be accessed directly in advanced simulations involving back-reflection. To explore the effect of this reflection matrix on a specific light field, you can apply it using `clsLightField.apply_TR_mat(...)`.

### `.R_R_mat_tot`

#### Read-only property.

Returns the right-side reflection matrix at total resolution.

### Behavior

`.R_R_mat_tot` provides the matrix that describes how an arbitrary input field is reflected when incident from the right side of the component.

Like `.T_LTR_mat_tot`, this matrix acts on modal amplitudes in  $k$ -space and has shape  $N^2 \times N^2$ . Each element quantifies how a given incoming mode on the right couples into a reflected mode, also on the right. The basis functions and ordering follow `clsGrid.mode_numbers_tot`.

### Typical use case

Used internally for computing the component's *scattering matrix* and *transfer matrix*, and may be accessed directly in advanced simulations involving back-reflection. To explore the effect of this reflection matrix on a specific light field, you can apply it using `clsLightField.apply_TR_mat(...)`.

## D.1.4 Scattering and Transfer Block Matrix

### `.S_bmat_tot`

#### Read-only property.

Returns the component's scattering matrix (*S-matrix*) as a  $2 \times 2$  block matrix.

### Behavior

`.S_bmat_tot` returns the component's full scattering matrix at total resolution. The S-matrix describes how incoming waves from either side of the component are transformed into outgoing waves on both sides.

We use the following convention for the S-matrix:

$$\begin{pmatrix} \mathbf{a}_{\text{out}}^{\text{left}} \\ \mathbf{a}_{\text{out}}^{\text{right}} \end{pmatrix} = \mathbf{S} \begin{pmatrix} \mathbf{a}_{\text{in}}^{\text{left}} \\ \mathbf{a}_{\text{in}}^{\text{right}} \end{pmatrix} = \begin{pmatrix} \mathbf{R}_L & \mathbf{T}_{RL} \\ \mathbf{T}_{LR} & \mathbf{R}_R \end{pmatrix} \begin{pmatrix} \mathbf{a}_{\text{in}}^{\text{left}} \\ \mathbf{a}_{\text{in}}^{\text{right}} \end{pmatrix}$$

where  $\mathbf{R}_L$ ,  $\mathbf{R}_R$  are the left and right reflection matrices, and  $\mathbf{T}_{LR}$ ,  $\mathbf{T}_{RL}$  are the transmission matrices for the two propagation directions.

Internally, `.S_bmat_tot` returns an instance of `clsBlockMatrix`. The deprecated fall-back behavior, controlled by `clsCavity.use_bmatrix_class`, may alternatively return a simple nested list `[[R_L, T_RTL], [T_LTR, R_R]]`, but this is not guaranteed to be supported in future versions.

### Performance notes

On first access, this property triggers the computation of `.R_L_mat_tot`, `.T_RTL_mat_tot`, `.T_LTR_mat_tot`, and `.R_R_mat_tot`, if they are not yet cached. On subsequent accesses, the S-matrix is constructed immediately from the already cached matrices, without re-computation.

### Typical use case

Used internally to construct the component's *transfer matrix* (*M-matrix*). May also be used directly to analyze the complete input-output behavior of an individual component.

**.M\_bmat\_tot****Read-only property.**

Returns the component's transfer matrix (*M-matrix*) as a  $2 \times 2$  block matrix.

**Behavior**

`.M_bmat_tot` returns the component's full transfer matrix at total resolution. The M-matrix provides a convenient way to describe wave propagation through cascaded components and is particularly well suited for linear cavity simulations.

We use the following convention for the M-matrix:

$$\begin{pmatrix} \mathbf{a}_{\text{out}}^{\text{left}} \\ \mathbf{a}_{\text{in}}^{\text{left}} \end{pmatrix} = \mathbf{M} \begin{pmatrix} \mathbf{a}_{\text{in}}^{\text{right}} \\ \mathbf{a}_{\text{out}}^{\text{right}} \end{pmatrix}$$

Here,  $\mathbf{a}_{\text{in}}^{\text{side}}$  and  $\mathbf{a}_{\text{out}}^{\text{side}}$  denote the modal amplitudes of waves entering or leaving the component on the respective side, expressed in the Fourier basis defined by `clsGrid.mode_numbers_tot`.

**Computation**

The transfer matrix is computed from the scattering matrix  $\mathbf{S}$  via the following relation [4]:

$$\mathbf{M}(\mathbf{S}) = \begin{pmatrix} \mathbf{S}_{12} - \mathbf{S}_{11}\mathbf{S}_{21}^{-1}\mathbf{S}_{22} & \mathbf{S}_{11}\mathbf{S}_{21}^{-1} \\ -\mathbf{S}_{21}^{-1}\mathbf{S}_{22} & \mathbf{S}_{21}^{-1} \end{pmatrix}$$

This conversion is performed internally by the helper method `_S_to_M(...)`, using the matrices `.R_L_mat_tot`, `.T_RTL_mat_tot`, `.T_LTR_mat_tot`, and `.R_R_mat_tot`.

**Caching behavior**

On first access, the M-matrix is computed and may be cached in memory and/or on disk, depending on the settings of `.mem_cache_M_bmat` and `.file_cache_M_bmat`. The complete filename and path is determined by the method `clsCavity._full_file_name(...)` with the parameter `name = "M_mat"`. Subsequent accesses return the cached version.

**Typical use case**

The M-matrix is the primary object used in cavity simulations based on matrix chaining. It allows the combined transfer properties of a series of components to be expressed as a single linear operator. The `.M_bmat_tot` property is accessed internally by `clsCavity1path` when assembling the total cavity matrix.

**.inv\_M\_bmat\_tot****Read-only property.**

Returns the inverse of the component's transfer matrix (*M-matrix*) as a  $2 \times 2$  block matrix.

**Behavior**

`.inv_M_bmat_tot` returns the inverse transfer matrix, computing it on first access

(via `.calc_inv_M_bmat_tot()`) and caching it in memory and/or on disk according to `.mem_cache_M_bmat` and `.file_cache_M_bmat`. The complete filename and path is determined by the method `clsCavity._full_file_name(...)` with the parameter `name = "invM_mat"`. Subsequent accesses return the cached version.

#### Typical use case

The inverse M-matrix is used internally by `clsCavity1path` by the method `clsCavity1path.calc_bulk_field_from_left(...)` and also by the method `clsCavity1path.calc_bulk_field_from_right(...)` to reconstruct the field at intermediate points inside a cavity from the known input or output fields.

#### `.calc_inv_M_bmat_tot()`

##### Method.

Explicitly computes and caches the inverse transfer matrix (inverse *M-matrix*) for this component.

##### Behavior

Calling `.calc_inv_M_bmat_tot()` forces the calculation of the inverse  $2 \times 2$  block transfer matrix and stores the result in memory. After this call, a subsequent access to `.inv_M_bmat_tot` will return the cached matrix immediately without recomputation.

#### Typical use case

Use this method when you need to ensure that the inverse transfer matrix is available in cache.

### D.1.5 Other Properties

#### `.dist_phys`

##### Read-only property.

Returns the physical propagation distance through the component, in meters.

##### Behavior

`.dist_phys` provides the physical distance  $d_{\text{phys}}$  that light propagates along the  $z$ -axis when passing through the component.

For many components this value is 0 (e.g. ideal thin lenses, mirrors, apertures). For components representing actual free-space or material propagation, the value corresponds to the physical length of the propagation path.

#### `.dist_opt`

##### Read-only property.

Returns the optical propagation distance through the component, in meters.

##### Behavior

`.dist_opt` provides the optical path length  $d_{\text{opt}}$  that the light experiences when traversing the component. For simple free-space propagation  $d_{\text{opt}} = d_{\text{phys}}$ . For propagation through a medium of refractive index  $n$ ,  $d_{\text{opt}} = n \cdot d_{\text{phys}}$ . For components that do not correspond to a true propagation segment (e.g. thin lenses, mirrors, apertures), the

optical distance is typically 0.

#### Typical use case

Used in cavity simulations when calculating accumulated optical phase shifts or resonance conditions.

#### `.symmetric`

##### Read-only property.

Indicates whether the component is left-right symmetric.

##### Behavior

`.symmetric` returns `True` if the component (with its current parameters) exhibits left-right symmetry; that is, the transmission and reflection behavior is identical when viewed from the left and from the right. If the component is not symmetric, this property returns `False`.

The value may depend on the current configuration of the component. For example, a mirror or a lens may be symmetric when not tilted, but if tilted would no longer be symmetric.

## D.2 Class `clsPropagation`

The class `clsPropagation` models free-space or material propagation over a specified distance. It derives from `clsOptComponent2port`. There are two parameters to configure: The physical distance and the complex-valued refractive index  $n$ . The real part of  $n$  sets the phase velocity (optical path length), while the imaginary part models absorption (exponential decay).

Two transfer-function methods are available:

- **Rayleigh–Sommerfeld transfer function** (physically exact);
- **Fresnel transfer function** (paraxial approximation, valid for small angles)

Although the Rayleigh–Sommerfeld transfer function is formally exact, it can still produce small numerical artefacts on a finite grid, even when that grid satisfies the critical-sampling condition described in `clsGrid`. Methods such as `clsGrid.set_opt_res_based_on_sidelength(...)` compute an “optimal” total grid size so that a desired field-of-view is embedded without aliasing, ensuring the Fresnel transfer function incurs essentially no discretization artefacts. The Rayleigh–Sommerfeld approach, by contrast, may result in minimal but unavoidable artefacts. Depending on the application – accuracy and the specific simulation parameters, either the exact Rayleigh–Sommerfeld method or the paraxial Fresnel approximation may be preferable.

### D.2.1 Initialization

#### `clsPropagation(name, cavity)`

##### Constructor.

Constructs a new `clsPropagation` component and links it to the given cavity.

**Parameters**

- *name* – A descriptive name for this propagation segment.
- *cavity* – The *clsCavity* instance to which this component belongs.

On initialization, the component's refractive index is set to  $n = 1$  and the physical propagation distance to 1 mm by default. After construction, call *.set\_params(...)* or *.set\_dist\_opt(...)* (and optionally *.set\_ni\_based\_on\_T(...)*) to configure the actual propagation distance and medium. For further details we refer to the description of the constructor of the root class *clsOptComponent(...)*.

**D.2.2 Physical Parameter Definition**

**`.set_params(dist_phys, n)`**

**Method.**

Sets the main physical parameters of the propagation component.

**Parameters**

- *dist\_phys* – Physical propagation distance in meters. Defines the distance over which the field is propagated along the *z*-axis. Must be non-negative; if a negative value is provided, it will be clipped to zero with a warning.
- *n* – Complex refractive index of the medium. The real part determines the phase velocity and optical path length; the imaginary part introduces absorption (exponential decay of the field amplitude).

**Behavior**

When this method is called:

- The cached transmission matrix is cleared to ensure correct recomputation with the new parameters.
- If the propagation component is already part of a cavity, the cavity's cached matrices are cleared to reflect the change.

**Typical use case**

Configure the propagation component to represent free-space propagation ( $n = 1$ ) or propagation through an absorbing or dispersive medium, over the specified physical distance.

**`.set_dist_opt(d_opt)`**

**Convenience method.**

Convenience method to set the physical distance based on a desired optical distance.

**Parameters**

- *d\_opt* – Desired optical propagation distance in meters.

**Behavior**

This method adjusts the physical propagation distance such that the resulting optical

path length corresponds to `d_opt`, based on the current real part of the refractive index:

$$\text{dist\_phys} = \frac{\text{d\_opt}}{\text{Re}(n)}$$

When this method is called:

- The cached transmission matrix is cleared to ensure correct recomputation.
- If the component is part of a cavity, the cavity's cached matrices are also cleared.

### Typical use case

Useful when you wish to directly control the optical path length, rather than manually computing the required physical distance for a given refractive index.

`.set_ni_based_on_T(T)`

#### Method.

Sets the imaginary part of the refractive index so that propagation over the current physical distance results in the specified transmittivity  $T$ .

#### Parameters

- $T$  – Desired transmittivity (dimensionless,  $0 < T \leq 1$ ).

#### Behavior

The imaginary part of the refractive index  $n$  controls exponential attenuation of the optical field:

$$T = \exp(-2 \cdot \text{Im}(n) \cdot k \cdot d_{\text{phys}})$$

where  $k = \frac{2\pi}{\lambda}$  is the vacuum wavenumber.

This method computes and sets  $\text{Im}(n)$  such that the above equation yields the specified  $T$ , based on the current physical propagation distance  $d_{\text{phys}}$  and the current wavelength `clsOptComponent.Lambda`.

#### Important notes

- You must call this method again if the propagation distance or `clsOptComponent.Lambda` is changed later, otherwise the attenuation will no longer match  $T$ .
- The real part of  $n$  remains unchanged.
- Any cached transmission matrix is cleared to ensure recomputation.

### Typical use case

Convenient for introducing controlled absorption. Instead of manually setting  $\text{Im}(n)$ , the user specifies the desired transmittivity, and the corresponding material absorption is automatically computed.

### D.2.3 Light Field Propagation

#### `.transfer_function`

##### **Read/write property.**

Selects which transfer function is used to model propagation.

##### **Behavior**

`.transfer_function` controls which mathematical model is used to compute the field propagation in the `.prop(...)` method and when calculating the transmission matrix:

- 0 – Rayleigh–Sommerfeld transfer function (physically accurate).
- 1 – Fresnel transfer function (paraxial approximation, default value).

##### **Choice of method**

The Rayleigh–Sommerfeld method provides the most accurate physical description. However, even if grid size matches the sampling condition discussed in `clsGrid`, the Rayleigh–Sommerfeld method can sometimes yield slight numerical artefacts, whereas the Fresnel transfer function produces no artefacts when the critical sampling condition is met.

##### **Behavior when changed**

Changing `.transfer_function` automatically clears any cached transmission matrix so that future propagation calls will use the selected model.

#### `.prop(E_in, k_space_in, k_space_out, direction)`

##### **Method.**

Propagates an optical field over the configured propagation distance and refractive index, using either the Rayleigh–Sommerfeld or Fresnel transfer function method.

##### **Parameters**

- `E_in` – Input field. Either a NumPy array (position space or  $k$ -space), or a scalar if `k_space_in` is True.
- `k_space_in` – If True, `E_in` is provided in  $k$ -space; otherwise in position space.
- `k_space_out` – If True, output field is returned in  $k$ -space; otherwise in position space.
- `direction` – Direction of propagation `Dir.LTR` or `Dir.RTL`. For this component, propagation is identical in both directions.

##### **Propagation methods and formulas**

Propagation is implemented via a multiplicative transfer function in  $k$ -space, applied to the  $k$ -space representation of the input field:

$$A(k_x, k_y; z) = A(k_x, k_y; 0) \cdot H(k_x, k_y; z)$$

where the transfer function  $H(k_x, k_y; z)$  depends on the selected method [3]:

- **Rayleigh–Sommerfeld transfer function :**

$$H(k_x, k_y; z) = \exp(i k_z z), \quad k_z = \sqrt{(nk_0)^2 - k_x^2 - k_y^2}$$

- **Fresnel transfer function** (paraxial approximation):

$$H(k_x, k_y; z) = \exp \left[ i \left( nk_0 z - \frac{z}{2nk_0} (k_x^2 + k_y^2) \right) \right]$$

Here,  $n$  is the refractive index,  $k_0 = \frac{2\pi}{\lambda}$  is the vacuum wavenumber, and  $z$  is the physical propagation distance.

### Choice of method

The method is selected via the property `.transfer_function`. See its description for guidance on method choice.

### Typical use case

Use this method to explicitly propagate a given input field through the component, optionally converting between position space and  $k$ -space. Internally, this mechanism is also used to construct the component's transmission matrix (`.T_LTR_mat_tot` and `.T_RTL_mat_tot`).

## D.2.4 Transmission and Reflection Matrices

### `.T_LTR_mat_tot`

#### Read-only property.

Returns the left-to-right transmission matrix of the component, at total resolution.

#### Behavior

`.T_LTR_mat_tot` returns a diagonal transmission matrix that represents the effect of propagating an arbitrary input field over the configured distance and refractive index, using the selected propagation method (see `.transfer_function`).

This matrix is constructed as:

$$\mathbf{T} = \text{diag}(H(k_x, k_y; z))$$

where  $H(k_x, k_y; z)$  is the corresponding Rayleigh–Sommerfeld or Fresnel transfer function, as documented in `.prop(...)`.

Because free-space or homogeneous-medium propagation is a shift-invariant operation in  $k$ -space, the transmission matrix is purely diagonal in the  $k$ -space modal basis.

#### Symmetry

For this component, propagation is identical in both directions. Therefore:

$$\text{T\_LTR\_mat\_tot} = \text{T\_RTL\_mat\_tot}$$

#### Further details

For more information, see the documentation of `clsOptComponent2port.T_LTR_mat_tot`.

### `.T_RTL_mat_tot`

#### **Read-only property.**

Returns the right-to-left transmission matrix of the component, at total resolution. For this component, propagation is identical in both directions. Therefore `.T_RTL_mat_tot` is identical to `.T_LTR_mat_tot`.

### `.R_L_mat_tot`

#### **Read-only property.**

Returns the left-side reflection matrix at total resolution.

#### **Behavior**

`.R_L_mat_tot` always returns the scalar 0. This reflects the fact that `clsPropagation` models ideal free-space or material propagation without interfaces or reflective elements. It is a pure transmission component.

#### **Further details**

For general information, see the documentation of `clsOptComponent2port.R_L_mat_tot`.

### `.R_R_mat_tot`

#### **Read-only property.**

Returns the right-side reflection matrix at total resolution.

#### **Behavior**

`.R_R_mat_tot` always returns the scalar 0. This reflects the fact that `clsPropagation` models ideal free-space or material propagation without interfaces or reflective elements. It is a pure transmission component.

#### **Further details**

For general information, see the documentation of `clsOptComponent2port.R_R_mat_tot`.

### `.calc_T_mat_tot()`

#### **Method.**

Generates the transmission matrix **T** at total resolution for the current propagation parameters and stores it in memory.

#### **Behavior**

`.calc_T_mat_tot()` simulates the propagation of all plane wave basis modes simultaneously and stores the result as a diagonal matrix. The diagonal elements represent how each plane wave mode is affected by the propagation distance, refractive index, and selected transfer function (Rayleigh–Sommerfeld or Fresnel).

Since pure propagation does not couple different  $k$ -space modes, the resulting transmission matrix is diagonal.

#### **Usage note**

Normally this method is called automatically when `.T_LTR_mat_tot` or `.T_RTL_mat_tot` is accessed for the first time. Manual calls are rarely necessary.

### D.2.5 Other Properties and Methods

#### `.symmetric`

**Read-only property.**

Indicates whether the component is left-right symmetric.

**Behavior**

Here, `.symmetric` always returns `True`. Propagation through a homogeneous material of uniform thickness is inherently symmetric with respect to direction: the transformation of a wavefront is the same whether it propagates left-to-right or right-to-left through the medium.

#### `.n`

**Read-only property.**

Returns the complex refractive index  $n$  of the propagation medium.

**Behavior**

`.n` returns the current refractive index  $n$  used in calculating propagation effects:

$$n = n_{\text{real}} + i n_{\text{imag}}$$

The real part  $n_{\text{real}}$  determines the phase velocity and optical path length; the imaginary part  $n_{\text{imag}}$  models absorption (exponential attenuation of the field during propagation).

The value of  $n$  is configured via `.set_params(...)` or indirectly via `.set_ni_based_on_T(...)`.

#### `.dist_opt`

**Read-only property.**

Returns the optical propagation distance in meters.

**Behavior**

`.dist_opt` returns the optical path length:

$$d_{\text{opt}} = d_{\text{phys}} \cdot \text{Re}(n)$$

where  $d_{\text{phys}}$  is the physical propagation distance and  $\text{Re}(n)$  is the real part of the refractive index.

**Typical use case**

Access the optical path length of the propagation component, which is relevant for calculating phase accumulation in interferometric simulations or when adjusting propagation distances programmatically.

#### `.clear_mem_cache()`

**Method.**

Clears all cached matrices and masks associated with this component from memory.

**Behavior**

`.clear_mem_cache()` first calls `clsOptComponent.clear_mem_cache()` to clear the cached  $M$ -matrix and inverse  $M$ -matrix, if present. It then additionally clears the cached transmission matrix  $\mathbf{T}$  of this component (property `.T_LTR_mat_tot` and `.T_RTL_mat_tot`).

**Typical use case**

Called automatically when propagation parameters (distance, refractive index, or transfer function) are changed, or manually to force a full recomputation of the component's matrices.

## D.3 Class `clsSplitPropagation`

The class `clsSplitPropagation` models free-space or material propagation over a specified distance, with *different refractive indices in the top and bottom halves* of the optical field (relative to the  $y = 0$  plane). It derives from `clsOptComponent2port` and behaves similarly to `clsPropagation`, but introduces spatial asymmetry in the propagation medium.

Separate complex-valued refractive indices  $n_1$  (for  $y > 0$ ) and  $n_2$  (for  $y < 0$ ) can be defined. This makes the class suitable for simulating situations where only part of the field propagates through an absorber or a different material (e.g. no absorber above the optical axis, absorber below).

As in `clsPropagation`, two transfer-function methods are available:

- **Rayleigh–Sommerfeld transfer function** (physically exact);
- **Fresnel transfer function** (paraxial approximation).

The propagation distance and refractive indices can be controlled independently. For the two field halves, different optical distances and absorption effects can therefore be realized within the same component.

### D.3.1 Initialization

`clsSplitPropagation(name, cavity)`

**Constructor.**

Initializes a new `clsSplitPropagation` component.

**Parameters**

- `name` – A descriptive name for the component (string).
- `cavity` – Reference to the parent `clsCavity` instance to which the component belongs.

On initialization, the component's refractive indices are set to  $n_1 = n_2 = 1$ , and the physical propagation distance is set to 1mm by default. After construction, call `.set_params(...)` or `.set_dist_opt(...)` (and optionally `.set_ni_based_on_T(...)`) to configure the actual propagation distance and media for the upper and lower beam regions.

For further details we refer to the description of the constructor of the root class `clsOptComponent(...)`.

### D.3.2 Physical Parameter Definition

```
.set_params(dist_phys, n1, n2, dist_corr1=0, dist_corr2=0)
```

#### Method.

Sets the main physical parameters of the split propagation component.

#### Parameters

- `dist_phys` – Physical propagation distance in meters. Defines the distance over which the field is propagated along the  $z$ -axis in both regions. Must be non-negative; if a negative value is provided, it will be clipped to zero with a warning.
- `n1` – Complex refractive index for the upper beam region ( $y > 0$ ). The real part determines the phase velocity and optical path length; the imaginary part introduces absorption.
- `n2` – Complex refractive index for the lower beam region ( $y < 0$ ).
- `dist_corr1` – Optional correction term (meters) added to the propagation distance for the upper region. Default: 0.
- `dist_corr2` – Optional correction term (meters) added to the propagation distance for the lower region. Default: 0.

#### Behavior

When this method is called:

- The cached transmission matrix is cleared to ensure correct recomputation with the new parameters.
- If the propagation component is already part of a cavity, the cavity's cached matrices are cleared to reflect the change.

#### Typical use case

Configure the split propagation component to model propagation through a two-region medium where the upper and lower parts of the field experience different optical properties (e.g. an absorber placed in one half of the beam path).

```
.set_dist_opt(d_opt, top_bottom)
```

#### Convenience method.

Convenience method to set the physical distance based on a desired optical distance, separately for the upper or lower region of the field.

#### Parameters

- `d_opt` – Desired optical propagation distance in meters.
- `top_bottom` – Region selector:
  - 1 – Apply to upper beam region ( $y > 0$ ).
  - 2 – Apply to lower beam region ( $y < 0$ ).

### Behavior

This method adjusts the shared physical propagation distance so that, in the selected region, the optical path length corresponds to `d_opt`, based on the current real part of the corresponding refractive index:

$$\text{dist\_phys} = \frac{\text{d\_opt}}{\text{Re}(n_i)}$$

When this method is called:

- The cached transmission matrix is cleared to ensure correct recomputation.
- If the component is part of a cavity, the cavity's cached matrices are also cleared.

### Typical use case

Useful when you wish to directly control the optical path length, rather than manually computing the required physical distance for a given refractive index.

`.set_ni_based_on_T(T, top_bottom)`

### Method.

Sets the imaginary part of the refractive index for the selected beam region so that propagation over the current physical distance results in the specified transmittivity  $T$ .

### Parameters

- `T` – Desired transmittivity (dimensionless,  $0 < T \leq 1$ ).
- `top_bottom` – Region selector:
  - 1 – Apply to upper beam region ( $y > 0$ ).
  - 2 – Apply to lower beam region ( $y < 0$ ).

### Behavior

The imaginary part of the refractive index  $n_i$  controls exponential attenuation of the optical field in the corresponding region:

$$T = \exp(-2 \cdot \text{Im}(n_i) \cdot k \cdot d_{\text{phys}})$$

where  $k = \frac{2\pi}{\lambda}$  is the vacuum wavenumber.

This method computes and sets  $\text{Im}(n_i)$  such that the above equation yields the specified  $T$ , based on the current physical propagation distance  $d_{\text{phys}}$  and the current wavelength `clsOptComponent.Lambda`. The real part of  $n_i$  remains unchanged.

### Important notes

- You must call this method again if the propagation distance or `clsOptComponent.Lambda` is changed later, otherwise the attenuation will no longer match  $T$ .
- The change only affects the selected beam region.
- Any cached transmission matrix is cleared to ensure recomputation.

**Typical use case**

Convenient for introducing controlled absorption. Instead of manually setting  $\text{Im}(n)$ , the user specifies the desired transmittivity, and the corresponding material absorption is automatically computed.

**D.3.3 Light Field Propagation****.transfer\_function****Read/write property.**

Selects which transfer function is used to model propagation.

**Behavior**

.`transfer_function` controls which mathematical model is used to compute the field propagation in the .`prop(...)` method and when calculating the transmission matrix:

- 0 – Rayleigh–Sommerfeld transfer function (physically accurate).
- 1 – Fresnel transfer function (paraxial approximation, default value).

**Choice of method**

The Rayleigh–Sommerfeld method provides the most accurate physical description. However, even if grid size matches the sampling condition discussed in `clsGrid`, the Rayleigh– Sommerfeld method can sometimes yield slight numerical artefacts, whereas the Fresnel transfer function produces no artefacts when the critical sampling condition is met.

**Behavior when changed**

Changing .`transfer_function` automatically clears any cached transmission matrix so that future propagation calls will use the selected model.

**.prop(E\_in, k\_space\_in, k\_space\_out, direction)****Method.**

Propagates an optical field through the component, using the configured refractive indices for the upper and lower halves of the field.

**Parameters**

- `E_in` – Input field. Either a NumPy array (position space or  $k$ -space), or a scalar if `k_space_in` is `True`.
- `k_space_in` – If `True`, `E_in` is provided in  $k$ -space; otherwise in position space.
- `k_space_out` – If `True`, output field is returned in  $k$ -space; otherwise in position space.
- `direction` – Direction of propagation `Dir.LTR` or `Dir.RTL`. For this component, propagation is identical in both directions.

**Behavior and split propagation**

Propagation is carried out separately for the top and bottom halves of the input field (split along the  $y = 0$  line):

- For  $y > 0$  (upper half), the field is propagated with refractive index `.n1`.
- For  $y < 0$  (lower half), the field is propagated with refractive index `.n2`.

Each half is propagated with the same transfer-function formulas used in `clsPropagation.prop(...)` (Rayleigh–Sommerfeld or Fresnel, depending on `.transfer_function`). The two partial results are then re-assembled into a single output field: the upper rows come from the  $n_1$  propagation, the lower rows from the  $n_2$  propagation.

### Simplification

This approach ignores any coupling or diffraction that would occur at the interface between the two media; it is therefore most accurate when the interface is optically thin and extends far beyond the beam's footprint so that cross-coupling can be neglected.

### Typical use case

Use this method to model asymmetric setups – for example, an absorber placed in only one half of the beam path. Internally, the same mechanism is employed to build the transmission matrices `.T_LTR_mat_tot` and `.T RTL mat_tot`.

## D.3.4 Transmission and Reflection Matrices

### `.T_LTR_mat_tot`

#### Read-only property.

Returns the left-to-right transmission matrix of the split-propagation component, at total resolution.

#### Behavior

The matrix is constructed by propagating *each* plane-wave basis function with `.prop(...)`. This calculation is parallelised with `joblib`; the number of worker processes is taken from `clsCavity.mp_pool_processes` (default: 10). The resulting matrix is generally *not* diagonal, because the upper and lower regions experience different phase and attenuation.

#### Symmetry

Propagation is identical in both directions, hence

$$\text{T\_LTR\_mat\_tot} = \text{T\_RTL\_mat\_tot}.$$

#### Further details

For the transfer-function formulas applied to each mode, see `clsPropagation.prop(...)`. For a general discussion of transmission matrices, see `clsOptComponent2port.T_LTR_mat_tot`.

### `.T_RTL_mat_tot`

#### Read-only property.

Returns the right-to-left transmission matrix of the component, at total resolution.

**Behavior**

For this component, propagation is implemented symmetrically. Therefore:

$$\text{T\_RTL\_mat\_tot} = \text{T\_LTR\_mat\_tot}.$$

The same matrix as returned by `.T_LTR_mat_tot` is used.

**`.R_L_mat_tot`****Read-only property.**

Returns the left-side reflection matrix at total resolution.

**Behavior**

`.R_L_mat_tot` always returns the scalar 0. This reflects the fact that `clsSplitPropagation` models pure propagation through a split medium, without interfaces or reflective elements. It is a purely transmitting component.

**Further details**

For general information, see the documentation of `clsOptComponent2port.R_L_mat_tot`.

**`.R_R_mat_tot`****Read-only property.**

Returns the right-side reflection matrix at total resolution.

**Behavior**

`.R_R_mat_tot` always returns the scalar 0. This reflects the fact that `clsSplitPropagation` models pure propagation through a split medium, without interfaces or reflective elements. It is a purely transmitting component.

**Further details**

For general information, see the documentation of `clsOptComponent2port.R_R_mat_tot`.

**`.calc_T_mat_tot()`****Method.**

Generates the transmission matrix **T** at total resolution for the current propagation parameters and stores it in memory.

**Behavior**

`.calc_T_mat_tot()` simulates the propagation of all plane wave basis modes individually and stores the result as a dense matrix. Unlike `clsPropagation`, this component models propagation with different parameters in the upper and lower halves of the field. Consequently, the transmission matrix is not diagonal, since the split breaks the shift-invariance of the system.

To compute the full transmission matrix, each plane wave basis mode is propagated individually using the `.prop(...)` method. This computation is parallelized across multiple processes; the number of processes is controlled by `clsCavity.mp_pool_processes` (default: 10).

**Usage note**

Normally this method is called automatically when the property `.T_LTR_mat_tot` or `.T_RTL_mat_tot` is accessed for the first time. Manual calls are rarely necessary.

### D.3.5 Other Properties and Methods

**`.symmetric`****Read-only property.**

Indicates whether the component is left-right symmetric.

**Behavior**

Here, `.symmetric` always returns `True`. Although the component allows different propagation parameters for the upper and lower halves of the field, the overall structure and effect of the component are symmetric with respect to the propagation direction: the same transformation is applied whether light travels left-to-right or right-to-left.

**`.n1`****Read-only property.**

Returns the complex refractive index  $n_1$  used for propagating the upper half of the field ( $y > 0$ ).

**Behavior**

`.n1` returns the current complex refractive index  $n_1 = n_{\text{real}} + i n_{\text{imag}}$ , controlling both phase shift and absorption for the upper half.

The value of  $n_1$  is configured via `.set_params(...)` or indirectly with the method `.set_ni_based_on_T(...)`.

**`.n2`****Read-only property.**

Returns the complex refractive index  $n_2$  used for propagating the lower half of the field ( $y < 0$ ).

**Behavior**

`.n2` returns the current complex refractive index  $n_2 = n_{\text{real}} + i n_{\text{imag}}$ , controlling both phase shift and absorption for the lower half.

The value of  $n_2$  is configured via `.set_params(...)` or indirectly with the method `.set_ni_based_on_T(...)`.

**`.dist_opt1`****Read-only property.**

Returns the optical propagation distance in meters for the upper half of the field ( $y > 0$ ).

**Behavior**

.`dist_opt1` returns the optical path length for the upper half:

$$d_{\text{opt1}} = (d_{\text{phys}} + \text{dist\_corr1}) \cdot \text{Re}(n_1)$$

**.`dist_opt2`****Read-only property.**

Returns the optical propagation distance in meters for the lower half of the field ( $y < 0$ ).

**Behavior**

.`dist_opt2` returns the optical path length for the lower half:

$$d_{\text{opt2}} = (d_{\text{phys}} + \text{dist\_corr2}) \cdot \text{Re}(n_2)$$

**.`dist_opt`****Read-only property.**

Returns the overall optical propagation distance in meters.

**Behavior**

.`dist_opt` returns the larger of .`dist_opt1` and .`dist_opt2`:

**.`clear_mem_cache()`****Method.**

Clears all cached matrices associated with this component from memory.

**Behavior**

.`clear_mem_cache()` first calls `clsOptComponent.clear_mem_cache()` to clear the cached *M-matrix* and inverse *M-matrix*, if present. It then additionally clears the cached transmission matrix **T** of this component (properties .`T_LTR_mat_tot` and .`T_RTL_mat_tot`).

**Typical use case**

Called automatically when propagation parameters (`dist_phys`, `n1`, `n2`, or `transfer_function`) are changed, or manually to force a full recomputation of the component's matrices.

## D.4 Class `clsThinLens`

`clsThinLens` derives from `clsOptComponent2port` and models an idealised *thin lens* that may optionally include a circular pupil (aperture) and residual facet reflections. Its action is applied in a single numerical step by multiplying the input field with a complex *lens mask* that combines:

- a phase term representing either a spherical thin-lens or a “perfect” aberration-free lens profile, determined by the focal length  $f$ ;

- an optional soft or hard aperture that limits the beam diameter and can include anti-aliasing;
- an optional complex amplitude factor accounting for residual reflections on the two lens facets. A small “black-level” attenuation can be applied at the pupil edge so that these residual reflections are numerically absorbed instead of persisting on the FFT grid, mimicking the fact that, in reality, facet reflections diverge and leave the optical path.

Key features:

- **Optical parameters** – focal length in metres or millimetres, numerical-aperture-based pupil sizing, and controllable edge attenuation for residual reflections.
- **Lens type** - switch between a spherical thin-lens approximation and a perfect aspherical profile.
- **Residual reflections** – specify facet reflectivity and an aperture that smoothly fades the reflection to zero outside the clear pupil; reflection and transmission coefficients obey energy conservation.
- **Propagation interface** – exposes the standard `prop` method plus transmission and reflection matrices, so it chains seamlessly with other components inside a `clsCavity1path` or `clsCavity2path` cavity.

All masks and matrices are cached for efficiency and automatically cleared when any optical parameter changes.

#### D.4.1 Initialization

`clsThinLens(name, cavity)`

##### Constructor.

Initializes a `clsThinLens` instance and registers it with the specified cavity.

##### Parameters

- `name` – String identifier for this lens instance.
- `cavity` – Reference to the `clsCavity` object to which this component belongs.

##### Behavior

On creation, the lens is configured with the following defaults:

- **Focal length:** `.f = 0.05 m` (equivalent to `.f_mm = 50 mm`).
- **Lens type:** spherical (`.lens_type_spherical = True`).
- **Aperture:** no aperture, see `.aperture`.
- **Residual reflections:** disabled (`.R_residual = 0`).

All optical parameters can be adjusted after construction via the corresponding properties and methods. For further details we refer to the description of the constructor of the root class `clsOptComponent(...)`.

### D.4.2 Optical Parameters

#### [.f](#)

##### **Read/write property.**

Specifies the focal length of the lens, in meters.

##### **Behavior**

The property `.f` holds the focal length of the lens.

When `.f` is changed:

- The corresponding property `.f_mm` (focal length in millimeters) is automatically updated.
- Any cached transmission or reflection matrices and phase masks are cleared to reflect the change.
- If the component belongs to a cavity, the cavity's cached matrices are also cleared.

##### **Typical use case**

Adjust the lens' focusing power by setting `.f` or alternatively `.f_mm`.

#### [.f\\_mm](#)

##### **Read/write property.**

Specifies the focal length of the lens, in millimeters.

##### **Behavior**

The convenience property `.f_mm` holds the focal length of the lens in millimeters.

When `.f_mm` is changed:

- The corresponding property `.f` (focal length in meters) is automatically updated.
- Any cached transmission or reflection matrices and phase masks are cleared to reflect the change.
- If the component belongs to a cavity, the cavity's cached matrices are also cleared.

##### **Typical use case**

Adjust the lens' focusing power by setting `.f_mm` or alternatively `.f`.

#### [.aperture](#)

##### **Read/write property.**

Specifies the lens aperture diameter in meters.

##### **Behavior**

The property `.aperture` defines the diameter of the lens aperture in meters.

When `.aperture` is changed:

- The corresponding property `.aperture_mm` (aperture in millimeters) is automati-

ically updated.

- The cached transmission and reflection phase masks are cleared to reflect the change.
- If the component belongs to a cavity, the cavity's cached matrices are also cleared.

### **Usage note**

A value of 0 disables the aperture (infinite aperture).

#### **.aperture\_mm**

##### **Read/write property.**

Specifies the lens aperture diameter in millimeters.

##### **Behavior**

The convenience property `.aperture_mm` defines the diameter of the lens aperture in millimeters.

When `.aperture_mm` is changed:

- The corresponding property `.aperture` (aperture in meters) is automatically updated.
- The cached transmission and reflection phase masks are cleared to reflect the change.
- If the component belongs to a cavity, the cavity's cached matrices are also cleared.

A value of 0 disables the aperture (infinite aperture).

#### **.set\_aperture\_based\_on\_NA(NA)**

##### **Method.**

Sets the lens aperture diameter based on a specified numerical aperture (NA).

##### **Parameters**

- NA – Numerical aperture (dimensionless,  $0 < \text{NA} \leq 1$ ).

##### **Behavior**

This convenience method computes the aperture diameter corresponding to the given numerical aperture `NA` and the current focal length `.f`, using the relation:

$$\text{aperture} = 2 \cdot f \cdot \tan(\arcsin(\text{NA}))$$

When this method is called:

- The properties `.aperture` and `.aperture_mm` are updated accordingly.
- Cached transmission and reflection phase masks are cleared.
- If the component belongs to a cavity, the cavity's cached matrices are also cleared.

**.aperture\_anti\_alias****Read/write property.**

Controls whether anti-aliasing is applied to the aperture edge.

**Behavior**

`.aperture_anti_alias` determines how the aperture mask is computed when an aperture is applied (`.aperture > 0`):

- If `True` (default), a soft aperture mask with edge smoothing (anti-aliasing) is used.
- If `False`, a hard-edged binary aperture mask is used.

Changing this property clears cached transmission and reflection phase masks and any cached matrices.

**.aperture\_black\_value****Read/write property.**

Sets the transmission value at the outer edge of the aperture mask.

**Behavior**

When an aperture is applied (`.aperture > 0`), `.aperture_black_value` specifies the transmission value at and beyond the aperture radius:

- `0` → perfectly opaque edge (default behavior for a strict aperture).
- Value  $> 0$  → partially transmitting "gray" edge.

**Reason for softening the aperture edge**

If the aperture is set to perfectly black (0), the transmission and reflection matrices for the component can become *ill-conditioned*: certain input modes would be completely blocked, leading to nearly singular matrices that are difficult to invert numerically. This can severely degrade the accuracy and stability of cavity simulations that rely on the scattering or transfer matrix formalism.

By setting `.aperture_black_value` to a small nonzero value (e.g. 0.01), one ensures that all modes retain at least minimal coupling, making the matrices well-conditioned and numerically robust.

**Behavior when changed**

Changing this property clears cached phase masks and matrices so they are recomputed accordingly.

### D.4.3 Lens Type

**.lens\_type\_spherical****Read/write property.**

Specifies whether the lens is modeled as a spherical thin lens.

**Behavior**

`.lens_type_spherical` returns or sets a boolean flag:

- `True` → the lens is modeled as a spherical thin lens. The phase mask applied in `.prop(...)` follows the approximation:

$$\exp\left(-i\frac{k_0}{2f}(x^2 + y^2)\right)$$

- `False` → the lens is modeled as a "perfect" lens (see `.lens_type_perfect`).

Setting `.lens_type_spherical` to `True` automatically sets `.lens_type_perfect` to `False`, and vice versa.

**Behavior when changed**

Changing this property clears cached phase masks and matrices so they are recomputed accordingly.

**`.lens_type_perfect`****Read/write property.**

Specifies whether the lens is modeled as a "perfect" thin lens.

**Behavior**

`.lens_type_perfect` returns or sets a boolean flag:

- `True` → the lens is modeled as an ideal perfect lens. The phase mask applied in `.prop(...)` corresponds to the ideal optical path length that ensures perfect imaging without spherical aberrations:

$$\exp\left(-ik_0\left(\sqrt{f^2 + x^2 + y^2} - f\right)\right)$$

- `False` → the lens is modeled as a spherical thin lens (see `.lens_type_spherical`).

Setting `.lens_type_perfect` to `True` automatically sets `.lens_type_spherical` to `False`, and vice versa.

**Behavior when changed**

Changing this property clears cached phase masks and matrices so they are recomputed accordingly.

#### D.4.4 Light Field Propagation

**`.prop(E_in, k_space_in, k_space_out, direction)`****Method.**

Propagates an optical field through the thin lens by applying its phase mask (and optional aperture mask), optionally converting between position space and  $k$ -space.

**Parameters**

- `E_in` – Input field. Either a NumPy array (position space or  $k$ -space), or a scalar if `k_space_in` is `True`.
- `k_space_in` – If `True`, `E_in` is provided in  $k$ -space; otherwise in position space.
- `k_space_out` – If `True`, output field is returned in  $k$ -space; otherwise in position space.
- `direction` – Direction of propagation `Dir.LTR` or `Dir.RTL`. For this component, propagation is identical in both directions.

### Behavior

The method implements a simple model of a thin lens:

- If `k_space_in` is `True`, the input field is first transformed to position space.
- The lens phase mask (`.lens_mask`) is applied multiplicatively in position space. This mask implements the phase profile of either a spherical lens or a perfect lens, as selected by `.lens_type_spherical` and `.lens_type_perfect`. An optional aperture mask may additionally clip or attenuate the field.
- If `k_space_out` is `True`, the result is transformed back to  $k$ -space.

### Simplifications

- The lens is modeled as an ideal phase object with no physical thickness. No actual propagation step is performed.
- Propagation is strictly local in position space: no mode coupling occurs between different spatial frequencies.
- The optional aperture and residual transmission/reflection are implemented via amplitude scaling.

### Typical use case

Use this method to explicitly compute how a specific input field is transformed by the thin lens.

## `.T_LTR_mat_tot`

### Read-only property.

Returns the left-to-right transmission matrix of the component, at total resolution.

### Behavior

`.T_LTR_mat_tot` returns the transmission matrix  $\mathbf{T}$  that represents how an arbitrary input field in  $k$ -space is transformed by the lens when propagating left-to-right.

This matrix is constructed as follows:

- Each plane-wave basis function (Fourier mode) is transformed to position space.
- The lens phase mask (`.lens_mask`) is applied multiplicatively in position space.
- The result is transformed back to  $k$ -space.
- The resulting output mode is stored as a column of the transmission matrix.

The process is performed for all plane-wave basis functions in parallel (number of processes controlled by `clsCavity.mp_pool_processes`).

### Symmetry

For this component, propagation is identical in both directions. Therefore:

$$\mathbf{T}_{\text{LTR}} \cdot \mathbf{mat} \cdot \mathbf{tot} = \mathbf{T}_{\text{RTL}} \cdot \mathbf{mat} \cdot \mathbf{tot}$$

### Further details

For more information, see the documentation of `clsOptComponent2port.T_LTR_mat_tot`. See also `.prop(...)`, which applies the same transformation to a specific input field.

#### `.T_RTL_mat_tot`

##### Read-only property.

Returns the right-to-left transmission matrix of the component, at total resolution.

##### Behavior

For this component, the left-to-right and right-to-left transmission matrices are identical:

$$\mathbf{T}_{\text{RTL}} \cdot \mathbf{mat} \cdot \mathbf{tot} = \mathbf{T}_{\text{LTR}} \cdot \mathbf{mat} \cdot \mathbf{tot}$$

The same computation and lens phase mask (`.lens_mask`) are used in both directions. For further details, see `.T_LTR_mat_tot`.

#### `.calc_T_mat_tot()`

Generates the transmission matrix  $\mathbf{T}$  at total resolution for the current lens parameters and stores it in memory.

##### Behavior

`.calc_T_mat_tot()` constructs the transmission matrix by simulating how each plane-wave basis mode is transformed by the lens phase mask (property `.lens_mask`):

- Each mode is transformed using `.prop(...)`.
- The resulting field is converted back to modal coefficients.
- The full matrix is assembled by repeating this process for all basis modes.

The calculation can run in parallel if `.par_RT_calc` is `True` (default). The number of parallel processes is determined by `clsCavity.mp_pool_processes`.

##### Usage note

Normally this method is called automatically when `.T_LTR_mat_tot` or `.T_RTL_mat_tot` is accessed for the first time. Manual calls are rarely necessary.

#### `.lens_mask`

##### Read-only property.

Returns the current lens phase mask as a 2D NumPy array in position space.

### Behavior

.lens\_mask returns the phase mask that represents the optical effect of the lens in position space. The mask combines:

- The phase profile of the lens:
  - Spherical lens: phase profile of a thin spherical lens surface.
  - Perfect lens: exact optical path length correction for an ideal aberration-free lens.
- A residual transmission factor .t\_residual (if residual reflection is defined via .set\_residual\_reflection(...)).
- A soft aperture (if .aperture > 0), modulating the mask amplitude toward the edges.

The lens mask is generated lazily when first accessed and cached in memory for efficient reuse. It is cleared automatically when lens parameters change.

### Usage in propagation

The mask is applied in position space during propagation in .prop(...). The transmission matrix .T\_LTR\_mat\_tot is also constructed by applying this mask to each basis function.

## D.4.5 Residual Reflection

```
.set_residual_reflection(R, reflection_aperture, epsilon,
                         black_value)
```

### Method.

Configures the lens' residual reflection behavior and corresponding soft aperture.

### Parameters

- **R** – Desired residual reflectivity (dimensionless,  $0 \leq R \leq 1$ ).
- **reflection\_aperture** – Diameter of the soft aperture applied to the residual reflection, in meters. If 0, no soft aperture is used.
- **epsilon** – Width of the soft transition zone at the aperture edge, in meters.
- **black\_value** – Value of the reflection mask outside the aperture (between 0 and 1). Usually set to a small value ( $\sim 10^{-3}$ ) to allow residual reflections to leave the grid in cavity simulations.

### Behavior

Calling .set\_residual\_reflection(...) sets the desired residual reflectivity  $R$ , and constructs a corresponding .reflection\_mask with a soft-edged aperture.

This mask ensures that reflection coefficients are smoothly reduced to **black\_value** at the grid edges. **Reason:** In FFT-based cavity simulations, residual reflections outside the physical aperture would otherwise remain trapped on the simulation grid indefinitely (an unphysical artifact). The soft edge allows such components to decay naturally.

The exact phase behavior of the reflection is controlled by the parameter **sym\_phase**

of the method `.set_phys_behavior(...)`.

### Behavior when called

- The reflection coefficient  $R$  and corresponding transmission coefficient  $T = 1 - R$  are updated.
- The reflection mask parameters are stored.
- Any cached reflection and transmission matrices and masks are cleared.

### `.set_phys_behavior(sym_phase)`

#### Method.

Selects the phase convention used for the residual facet reflections and, by extension, for the lens scattering matrix  $\mathbf{S}$ .

#### Parameters

- `sym_phase` – Boolean flag `True`: “symmetric-phase” convention (default) `False`: “text-book” real-valued convention

#### Why two conventions?

For energy to be conserved the scattering matrix must be *unitary*, i.e.  $\mathbf{SS}^\dagger = \mathbb{1}$ . With only a single spatial mode one may write

$$\mathbf{S} = \begin{pmatrix} r & t \\ t & r \end{pmatrix}, \quad r, t \in \mathbb{C},$$

where  $R = |r|^2$  is the desired reflectivity.

#### Symmetric-phase convention (`sym_phase = True`)

Setting

$$r = -R - i\sqrt{R}\sqrt{1-R}, \quad t = 1 + r,$$

guarantees unitarity for any  $0 \leq R \leq 1$  and assigns *the same phase shift* to reflections from both sides of the lens [7, p. 55]. This is physically intuitive but introduces a small phase term into  $t$ ; in resonator calculations this phase slightly shifts the resonance condition away from the simple “text-book” cavity length.

#### Real-valued (“text-book”) convention (`sym_phase = False`)

A widely used alternative is to force the coefficients to be real [8, p. 406]:

$$\mathbf{S} = \begin{pmatrix} r & t \\ t & -r \end{pmatrix}, \quad r, t \in [0, 1],$$

which ensures  $\mathbf{S}$  to be unitary and keeps  $t$  real and positive; however, the reflection phases differ for the left and right facets ( $r_R = -r_L$ ).

#### Generalisation to multi-mode case

In this library the single-mode coefficients become diagonal matrices, so in block-matrix notation

$$\mathbf{S} = \begin{pmatrix} \mathbf{R}_L & \mathbf{T}_{RL} \\ \mathbf{T}_{LR} & \mathbf{R}_R \end{pmatrix} = \begin{pmatrix} \mathbf{R} & \mathbf{T} \\ \mathbf{T} - \mathbf{R} & \mathbf{R} \end{pmatrix}, \quad \mathbf{R}, \mathbf{T} \in \mathbb{R}^{N^2 \times N^2}.$$

**Behavior when called**

- Updates the internal residual reflection ( $r,t$ ) coefficients according to the chosen convention.
- Clears cached reflection masks and matrices so that they are rebuilt with the new phase behaviour.

**.R\_residual****Read-only property.**

Returns the residual reflectivity  $R$  currently configured for the lens.

**Behavior**

.R\_residual returns the real-valued residual reflectivity:

$$R \in [0, 1]$$

This value controls the power reflection of the lens surfaces on the left and the right side of the lens. The corresponding reflection and transmission coefficients used in the component's scattering matrix are computed based on this value (see .set\_residual\_reflection(...) and .set\_phys\_behavior(...)).

A value of  $R = 0$  models an ideal anti-reflection-coated lens (no residual reflections). A value  $R > 0$  introduces partial reflection as configured.

**Usage note**

This property is read-only. To update  $R$ , use the method .set\_residual\_reflection(...).

**.r\_residual****Read-only property.**

Returns the current complex reflection coefficient  $r$  used in the component's scattering matrix.

**Behavior**

.r\_residual provides the complex reflection coefficient  $r$  that corresponds to the configured power reflectivity .R\_residual:

$$R = |r|^2$$

The exact value and phase of  $r$  depend on whether .set\_phys\_behavior(...) is set to `sym_phase = True` or `False` (see that method for detailed explanation of the two conventions). In both cases, the value of  $r$  is used to build the component's reflection matrices and scattering matrix.

**Usage note**

This property is read-only. It is automatically updated when .set\_residual\_reflection(...) or .set\_phys\_behavior(...) is called.

**.t\_residual****Read-only property.**

Returns the current complex transmission coefficient  $t$  used in the component's scattering matrix.

**Behavior**

`.t_residual` provides the complex transmission coefficient  $t$  that corresponds to the configured power reflectivity `.R_residual`, ensuring that the component's scattering matrix remains energy-conserving (unitary):

$$|t|^2 + |r|^2 = 1$$

The exact value and phase of  $t$  depend on whether `.set_phys_behavior(...)` is set to `sym_phase = True` or `False` (see that method for detailed explanation of the two conventions).

**Usage note**

This property is read-only. It is automatically updated when `.set_residual_reflection(...)` or `.set_phys_behavior(...)` is called.

**.reflection\_aperture****Read-only property.**

Returns the current reflection aperture radius in meters.

**Behavior**

`.reflection_aperture` returns the radius of the soft aperture applied to the reflection mask of the lens.

The purpose of this aperture is to suppress unphysical reflections at the edge of the grid. In reality, reflections from curved lens surfaces spread outward and leave the optical system. In a finite FFT simulation, such reflections would otherwise remain trapped on the grid indefinitely. The aperture provides a soft spatial window that gradually reduces the reflection amplitude towards the edges, helping to absorb these spurious components.

The aperture radius is set via `.set_residual_reflection(...)`. A value of 0 disables the reflection aperture (no soft window applied).

**.reflection\_aperture\_black****Read-only property.**

Returns the black level used at the edge of the reflection aperture.

**Behavior**

`.reflection_aperture_black` returns the minimum amplitude factor used at and beyond the reflection aperture radius (see `.reflection_aperture`) when constructing the reflection mask.

The reflection mask modulates the residual reflection of the lens according to `.R_residual`.

Inside the aperture, this reflection is applied fully (according to `.R_residual`), while near the edge it is gradually reduced to the value given by `.reflection_aperture_black`.

The main purpose of this “black level” is to prevent residual reflected light from propagating indefinitely within the finite simulation grid: in reality, reflections from a convex lens surface would propagate outward and be lost; in the finite FFT grid used here, they would otherwise circulate unphysically “forever.” To avoid this, the reflection is smoothly faded out near the aperture edge.

A typical value is around 0.01; setting the black level slightly above zero also helps to avoid numerical issues in scattering matrix calculations.

The value is set when calling `.set_residual_reflection(...)`.

### `.reflection_aperture_epsilon`

**Read-only property.** Returns the width of the transition zone (“epsilon”) used when constructing the reflection aperture.

#### Behavior

`.reflection_aperture_epsilon` defines how gradually the residual-reflection mask fades from its full value to the black level near the aperture edge. Using the notation

$$a = \text{reflection\_aperture}/2, \quad \epsilon = \text{reflection\_aperture\_epsilon},$$

the mask equals the full reflection value for radii  $r \leq a - \epsilon$ , equals the black level for  $r \geq a + \epsilon$ , and is smoothly interpolated in the band  $a - \epsilon < r < a + \epsilon$ .

The implementation delegates to `clsGrid.get_soft_aperture_mask(...)`, which uses a cubic smooth-step profile to ensure a  $C^1$ -continuous transition.

A larger `.reflection_aperture_epsilon` produces a softer edge; a smaller value yields a sharper cutoff. The parameter is set when calling `.set_residual_reflection(...)`.

### `.R_L_mat_tot`

**Read-only property.**

Returns the left-side reflection matrix at total resolution.

#### Behavior

If the residual reflectivity `.R_residual` is set to 0, `.R_L_mat_tot` returns the scalar 0.

If `.R_residual` is greater than 0, the reflection matrix is computed (if not already cached) by applying the component’s reflection mask (`.reflection_mask`) to each plane wave basis function. The matrix thus describes how the configured residual reflection affects each mode.

#### Further details

See `clsOptComponent2port.R_L_mat_tot` for general information on reflection matrices.

To understand the construction of the reflection mask, see `.reflection_mask`.

**.R\_R\_mat\_tot****Read-only property.**

Returns the right-side reflection matrix at total resolution.

**Behavior**

If the residual reflectivity `.R_residual` is set to 0, `.R_R_mat_tot` returns the scalar 0.

If `.R_residual` is greater than 0, the reflection matrix is computed (if not already cached) by applying the component's reflection mask (`.reflection_mask`) to each plane-wave basis function, exactly as for `.R_L_mat_tot`. The resulting matrix describes how the configured residual reflection affects each mode on the right side of the lens.

The phase behaviour of the right-side reflection depends on the setting chosen via `.set_phys_behavior(...)`:

- If `set_phys_behavior(sym_phase=True)` was chosen, the left- and right-side reflection matrices are identical:

$$\text{R\_R\_mat\_tot} = \text{R\_L\_mat\_tot}.$$

- If `set_phys_behavior(sym_phase=False)` was chosen, the standard real-valued convention is used and the right-side reflection acquires a sign flip:

$$\text{R\_R\_mat\_tot} = -\text{R\_L\_mat\_tot}.$$

**Further details**

See `clsOptComponent2port.R_R_mat_tot` for general information on reflection matrices, and `.set_phys_behavior(...)` for a full explanation of the two phase conventions.

**.calc\_R\_mat\_tot()**

Generates the reflection matrix **R** at total resolution for the current residual-reflection settings and stores it internally.

**Behavior**

`.calc_R_mat_tot()` proceeds as follows:

1. Accesses the `.reflection_mask` property. This mask incorporates the residual reflection coefficient `.R_residual`, the phase convention chosen via `.set_phys_behavior(...)`, and the soft aperture defined by `.reflection_aperture`, `.reflection_aperture_epsilon`, and `.reflection_aperture_black`.
2. For each plane-wave basis mode on the total grid in position space, multiplies the mode by the reflection mask and converts the result back to modal coefficients.
3. Assembles the resulting column vectors into the full reflection matrix and stores it internally as block (0, 0) of the block matrix container.

If `.par_RT_calc` is `True` (default), the per-mode computations are executed in parallel using `joblib`; the number of worker processes is taken from `clsCavity.mp_pool_processes`.

### Symmetry note

`.calc_R_mat_tot()` always computes and caches *one* reflection matrix instance (representing reflection from the left side).

- If `.set_phys_behavior(...)` was last called with `sym_phase=True` (default), this same matrix is returned unchanged by both `.R_L_mat_tot` and `.R_R_mat_tot`.
- If `sym_phase=False`, the cached matrix is used for `.R_L_mat_tot`, while `.R_R_mat_tot` returns the *negated* version of the cached matrix.

### Usage note

Normally this method is called automatically when `.R_L_mat_tot` or `.R_R_mat_tot` is accessed for the first time and no cached matrix is available. Manual invocation is needed only when you wish to pre-compute the reflection matrix.

## `.reflection_mask1`

### Read-only property.

Returns the complex reflection mask corresponding to the outer facet of the lens.

### Behavior

`.reflection_mask1` provides the phase mask corresponding to residual reflection on the *outer* surface of the lens. This mask is computed on first access and then cached. It incorporates the optical phase shift induced by reflection from the curved facet. The overall reflection mask `.reflection_mask` combines this with `.reflection_mask2` to model multiple reflections between the lens facets.

### Usage note

Mainly used internally by `.reflection_mask` and during the calculation of the reflection matrix. Can be inspected manually for advanced analysis.

## `.reflection_mask2`

### Read-only property.

Returns the complex reflection mask corresponding to the inner facet of the lens.

### Behavior

`.reflection_mask2` provides the phase mask corresponding to residual reflection on the *inner* surface of the lens. This mask is computed on first access and then cached. The total reflection behavior of the lens, as captured in `.reflection_mask`, is based on a combination of this mask and `.reflection_mask1`, scaled by the residual reflection coefficient.

### Usage note

Mainly used internally by `.reflection_mask` and during the calculation of the reflection matrix. Can be inspected manually for advanced analysis.

**.reflection\_mask****Read-only property.**

Returns the complete reflection mask applied when constructing the reflection matrix.

**Behavior**

`.reflection_mask` combines the per-facet reflection masks `.reflection_mask1` and `.reflection_mask2` into a single complex-valued mask:

$$\text{reflection\_mask} = r_{\text{residual}} \cdot \frac{\text{reflection\_mask1} + \text{reflection\_mask2}}{\sqrt{2}}$$

If a residual reflection aperture is defined via `.set_residual_reflection(...)`, a soft-edged aperture mask is additionally applied:

$$\text{reflection\_mask} \leftarrow \text{reflection\_mask} \cdot \text{soft\_aperture\_mask}$$

where the soft aperture is generated via `clsGrid.get_soft_aperture_mask(...)` using the parameters `.reflection_aperture`, `.reflection_aperture_epsilon`, and `.reflection_aperture_black`.

The final mask models the spatially dependent amplitude and phase modulation applied to any field reflecting off the lens.

**Usage note**

`.reflection_mask` is used internally when constructing the reflection matrix via `.calc_R_mat_tot()`, but can also be inspected directly to visualize the combined reflection behavior of the lens.

#### D.4.6 Symmetry and Space-Preferences

**.symmetric****Read-only property.**

Indicates whether the component is left-right symmetric.

**Behavior**

`.symmetric` always returns True.

**.k\_space\_in\_dont\_care****Read-only property.**

Indicates whether the component is indifferent to the input-space representation.

**Behavior**

`.k_space_in_dont_care` always returns False. This means the component is not indifferent to the input representation. It **prefers input in position space**, as indicated by `.k_space_in_prefer`.

**Further details**

For general background on this property, see `clsOptComponent.k_space_in_dont_care`.

**.k\_space\_in\_prefer****Read-only property.**

Indicates whether *k*-space input is preferred.

**Behavior**

`.k_space_in_prefer` always returns `False`, meaning that the component prefers input in position space.

**Further details**

For general background on this property, see `clsOptComponent.k_space_in_prefer`.

**.k\_space\_out\_dont\_care****Read-only property.**

Indicates whether the component is indifferent to the output-space representation.

**Behavior**

`.k_space_out_dont_care` always returns `False`. This means the component is not indifferent to the output representation. It **prefers output in position space**, as indicated by `.k_space_out_prefer`.

**Further details**

For general background on this property, see `clsOptComponent.k_space_out_dont_care`.

**.k\_space\_out\_prefer****Read-only property.**

Indicates whether *k*-space output is preferred.

**Behavior**

`.k_space_out_prefer` always returns `False`, meaning that the component prefers output in position space.

**Further details**

For general background on this property, see `clsOptComponent.k_space_out_prefer`.

#### D.4.7 Memory Management and Other Settings

**.clear\_mem\_cache()****Method.**

Clears all cached matrices and masks associated with this component from memory.

**Behavior**

`.clear_mem_cache()` first calls `clsOptComponent.clear_mem_cache()` to clear the cached *M-matrix* and inverse *M-matrix*, if present.

In addition, the following component-specific cached items are cleared:

- The lens phase mask `.lens_mask`.
- The reflection mask `.reflection_mask`.

- The auxiliary reflection masks `.reflection_mask1` and `.reflection_mask2`.
- The cached transmission matrix  $\mathbf{T}$ .
- The cached reflection matrix  $\mathbf{R}$ .

#### Typical use case

`.clear_mem_cache()` is called automatically whenever a parameter that affects propagation or reflection is changed (e.g. `.f`, `.aperture`, `.set_residual_reflection(...)`, `.set_phys_behavior(...)`, etc.). Manual invocation is required only to explicitly force full recomputation of all matrices and masks.

#### `.par_RT_calc`

##### Read/write property.

Controls whether the reflection and transmission matrices are calculated in parallel.

##### Behavior

If `.par_RT_calc` is set to `True` (default), the methods `.calc_T_mat_tot()` and `.calc_R_mat_tot()` process the individual plane wave modes in parallel using `joblib`. The number of parallel worker processes is taken from `clsCavity.mp_pool_processes`.

If set to `False`, both matrix calculations are performed in a single-threaded, serial manner. This is useful for debugging or in special environments where parallel processing is undesirable.

## D.5 Class `clsGrating`

The class `clsGrating` models a *thin, spatially periodic grating* and derives from `clsOptComponent2port`. The component multiplies the incident field in position space by a *grating mask* – either an **absorption grating** (periodic amplitude modulation) or a **phase grating** (periodic phase modulation):

$$U_{\text{out}}(x, y) = U_{\text{in}}(x, y) \times \text{grate\_mask}(x, y).$$

#### Key features

- **1-D or 2-D cosine grating.** Periods `dx` and `dy` and finite half-widths `x_max`, `y_max` are set with `.set_cos_grating(...)`. The parameter `opt_dim` optionally “snaps” each requested period to the nearest grid spacing so the sampled cosine fits the FFT grid exactly, avoiding aliasing artefacts.
- **Absorption vs. phase grating.** Select with the Boolean properties `.absorbtion_grating` and `.phase_grating`.
  - *Absorption grating:* The mask values oscillate between `min_val` and `max_val`, modelling a periodic attenuation profile.
  - *Phase grating:* First the same cosine profile is generated in the range  $[\text{min\_val}, \text{max\_val}]$ ; this real-valued pattern is then interpreted as a phase  $\phi(x, y)$  and converted to a complex phase factor  $\exp(i\phi(x, y))$ . Hence `min_val` and `max_val` correspond to the minimum and maximum phase shift imparted by the grating.

- **Transmission matrix only.** Because the grating is a purely multiplicative element in position space, its reflection matrices are identically zero; the transmission matrix is built by applying the mask to each modal basis function (see `.calc_T_mat_tot()`).

### D.5.1 Initialization

`clsGrating(name, cavity)`

#### Constructor.

Initializes a new instance of `clsGrating`, a thin optical component that applies either an absorption or a phase modulation to the incident field.

#### Behavior

On construction, the grating parameters are initialized as follows:

- The grating is set to an **absorption grating** by default (`.absorbtion_grating = True, .phase_grating = False`).
- The initial grating periods `.dx` and `.dy` are set to `math.inf`, meaning no periodic modulation is present until configured.
- The grating extends across the full computational domain by default (`.x_max, .y_max = math.inf`).
- The grating mask values are initialized to vary between `.min_val = 0` and `.max_val = 1`.

#### Typical use case

After creating a `clsGrating` instance, use `.set_cos_grating(...)` to configure the desired grating periods, size, and value range. The grating type (absorption or phase) can be selected via `.absorbtion_grating` or `.phase_grating`.

For further details, we refer to the description of the constructor of the root class `clsOptComponent(...)`.

### D.5.2 Grating Parameters

`.set_cos_grating(dx, x_max, dy, y_max, opt_dim, min_val, max_val)`

#### Method.

Configures the grating pattern as a two-dimensional cosine grating with adjustable periods, spatial extent, and value range.

#### Parameters

- `dx` – Grating period along the  $x$ -axis (in meters). Use `math.inf` for no modulation along  $x$ .
- `x_max` – Maximum  $x$ -extent of the grating in positive and negative  $x$  direction. Outside this range, the grating is zero.
- `dy` – Grating period along the  $y$ -axis (in meters). Use `math.inf` for no modulation along  $y$ .
- `y_max` – Maximum  $y$ -extent of the grating in positive and negative  $y$  direction.

- `opt_dim` – If `True`, optimize the specified periods to match the grid sampling for artifact-free simulation.
- `min_val` – Minimum value of the grating profile.
- `max_val` – Maximum value of the grating profile.

### Behavior

Calling `.set_cos_grating(...)` defines a grating mask of the form:

$$g(x, y) = \text{min\_val} + (\text{max\_val} - \text{min\_val}) \cdot \frac{1 - \cos\left(\frac{2\pi x}{\text{dx}}\right)}{2} \cdot \frac{1 - \cos\left(\frac{2\pi y}{\text{dy}}\right)}{2}$$

with optional suppression along either  $x$  or  $y$ , if `dx` or `dy` are set to `math.inf`.

The grating mask has a **minimum at**  $x = 0, y = 0$  by construction.

### Phase grating vs. absorption grating

- If `.phase_grating = True`, the mask is interpreted as a phase modulation:

$$M(x, y) = \exp(i \cdot g(x, y))$$

where  $g(x, y)$  corresponds to the computed grating profile ranging between `min_val` and `max_val`, representing the minimum and maximum phase shift.

- If `.absorbtion_grating = True`, the mask directly modulates the field amplitude according to the computed profile.

### Grid optimization

If `opt_dim` is `True`, the specified grating periods are automatically adapted to better fit the current grid resolution, reducing aliasing artifacts.

### Return value

Returns the (possibly optimized grid-fitting) values `dx`, `x_max`, `dy`, `y_max` as a tuple.

## `.absorbtion_grating`

**Read/write property.**

Specifies whether the grating acts as an **absorption grating**.

### Behavior

If `.absorbtion_grating` is set to `True`, the grating modulates the **field amplitude**:

$$E_{\text{out}}(x, y) = E_{\text{in}}(x, y) \cdot g(x, y)$$

where  $g(x, y)$  is the grating mask computed by `.set_cos_grating(...)` and stored in `.grate_mask`.

The grating mask in this mode has values between `min_val` and `max_val` as defined in `.set_cos_grating(...)`, and directly represents absorption: lower values attenuate the field more strongly.

Setting this property automatically sets `.phase_grating` to the logical opposite value, ensuring that the grating behaves either as an **absorption grating** or as a **phase grating**, but not both simultaneously.

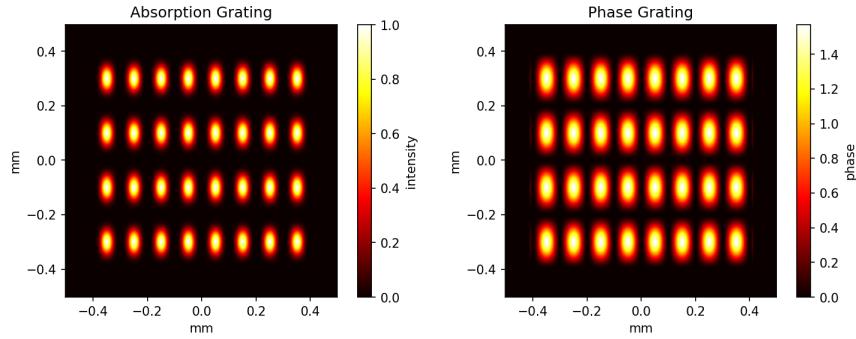


Figure D.2: Plane-wave illumination transmitted through a thin grating. **Left:** absorption grating with mask values varying periodically between 0 (dark) and 1 (bright), producing corresponding amplitude modulations in the field. **Right:** phase grating with constant amplitude (unity) but a spatially varying phase that oscillates between 0 and  $\pi$ . In both cases, the grating period is twice as large in the  $x$ -direction as in the  $y$ -direction, yielding the elongated stripe pattern visible in each panel.

### `.phase_grating`

**Read/write property.**

Specifies whether the grating acts as a **phase grating**.

**Behavior**

If `.phase_grating` is set to `True`, the grating modulates the **field phase**:

$$E_{\text{out}}(x, y) = E_{\text{in}}(x, y) \cdot \exp(i \cdot g(x, y))$$

where  $g(x, y)$  is the grating profile computed by `.set_cos_grating(...)`, with values between `min_val` and `max_val`, interpreted here as phase shifts in radians.

Setting this property automatically sets `.absorbtion_grating` to the logical opposite value, ensuring that the grating behaves either as a **phase grating** or as an **absorption grating**, but not both simultaneously.

### `.grate_mask`

**Read-only property.**

Returns the grating mask that is currently applied to the optical field.

**Behavior**

`.grate_mask` returns a  $N \times N$  array representing the grating, where  $N$  is the total grid size `.grid.res_tot`. The mask is computed using the parameters defined via `.set_cos_grating(...)`, with the following interpretation:

- In **absorption grating** mode (`.absorbtion_grating = True`), the mask contains real values between `min_val` and `max_val` and directly modulates the amplitude of the field.
- In **phase grating** mode (`.phase_grating = True`), the same cosine-based mask

is exponentiated as

$$\exp(i \cdot g(x, y)),$$

where  $g(x, y)$  is the real-valued profile from `.set_cos_grating(...)`. The phase shift at each point thus lies between `min_val` and `max_val` radians.

### Typical use case

`.grate_mask` is applied internally in `.prop(...)` and when computing the transmission matrix `.T_LTR_mat_tot`. You may also access it directly to visualize or analyze the current grating pattern.

## `.dx`

### Read-only property.

Returns the current grating period in the  $x$ -direction, in meters.

#### Behavior

`.dx` reflects the parameter `dx` that was set via the most recent call to `.set_cos_grating(...)`. If `opt_dim` was set to `True` during that call, `dx` may have been adjusted internally to better fit the numerical grid (`clsGrid`). The value returned by `.dx` is thus the actual period used in the mask calculation, which may differ slightly from the `dx` originally passed to `.set_cos_grating(...)`.

If `dx` was set to `math.inf`, there is no grating in the  $x$ -direction.

## `.dy`

### Read-only property.

Returns the current grating period in the  $y$ -direction, in meters.

#### Behavior

The property `.dy` reflects the parameter `dy` that was set via the most recent call to `.set_cos_grating(...)`. If `opt_dim` was set to `True` during that call, `dy` may have been adjusted internally to better fit the numerical grid (`clsGrid`). The value returned by `.dy` is thus the actual period used in the mask calculation, which may differ slightly from the `dy` originally passed to `.set_cos_grating(...)`.

If `dy` was set to `math.inf`, there is no grating in the  $y$ -direction.

## `.x_max`

### Read-only property.

Returns the half-width of the grating region in the  $x$ -direction, in metres.

#### Behaviour

`.x_max` gives the current value of `x_max` used when the grating mask is generated. The cosine modulation is applied only inside the interval

$$x \in [-x_{\text{max}}, x_{\text{max}}].$$

For  $|x| > x_{\text{max}}$  the mask takes its baseline value:

- **Absorption grating:** the mask equals `min_val` (no additional phase factor).
- **Phase grating:** the mask equals  $\exp(i \text{min\_val})$ , i.e. the minimum phase shift.

If `opt_dim` was set to `True` when calling `.set_cos_grating(...)`, the period is snapped to the grid and `x_max` is rescaled accordingly; the value returned by `.x_max` is therefore the *effective* half-width used in the mask.

### `.y_max`

#### **Read-only property.**

Returns the half-width of the grating region in the *y*-direction, in metres.

#### **Behaviour**

`.y_max` yields the current value of `y_max` that was fixed by the most recent call to `.set_cos_grating(...)`. The cosine modulation is applied only inside the interval

$$y \in [-\text{y\_max}, \text{y\_max}].$$

For  $|y| > \text{y\_max}$  the mask reverts to its baseline value:

- **Absorption grating** – the mask equals `min_val`.
- **Phase grating** – the mask equals  $\exp(i \text{min\_val})$ , i.e. the minimum phase shift.

If `opt_dim` was `True` when calling `.set_cos_grating(...)`, the period is snapped to the grid and `y_max` is rescaled accordingly; the value returned by `.y_max` is therefore the *effective* half-width used in the mask.

### `.min_val`

#### **Read-only property.**

Returns `min_val`, the lower bound of the grating profile specified in the most recent call to `.set_cos_grating(...)`.

#### **Behaviour**

- **Absorption grating** (`.absorbtion_grating = True`): `min_val` is the *minimum amplitude transmission* applied at the troughs of the cosine pattern.
- **Phase grating** (`.phase_grating = True`): `min_val` is interpreted as the *minimum phase shift* (in radians) imparted by the grating; the mask value at a trough is  $\exp(i \text{min\_val})$ .

Together with `.max_val`, this property fully defines the modulation depth of the grating.

### `.max_val`

#### **Read-only property.**

Returns `max_val`, the upper bound of the grating profile specified in the most recent call to `.set_cos_grating(...)`.

#### **Behaviour**

- **Absorption grating** (`.absorbtion_grating = True`): `max_val` is the *maximum amplitude transmission* attained at the peaks of the cosine pattern.
- **Phase grating** (`.phase_grating = True`): `max_val` is interpreted as the *maximum phase shift* (in radians) imparted by the grating; the mask value at a peak is  $\exp(i \max\_val)$ .

Together with `.min_val`, this property defines the full modulation depth of the cosine grating.

### D.5.3 Light Field Propagation

`.prop(E_in, k_space_in, k_space_out, direction)`

#### Method.

Applies the grating mask to the input field *in position space* and returns the resulting field, optionally converting between position space and *k*-space.

#### Parameters

- `E_in` – Input field. A NumPy array representing the field either in position space or *k*-space, as specified by `k_space_in`.
- `k_space_in` – `True` if `E_in` is given in *k*-space, `False` if it is in position space.
- `k_space_out` – `True` if the returned field should be in *k*-space, `False` for position space.
- `direction` – Propagation direction (`Dir.LTR` or `Dir.RTL`). For a thin grating the result is identical in both directions.

#### Behaviour

1. If the grating mask has not yet been generated, it is obtained via `.grate_mask`.
2. If `k_space_in = True`, the field is transformed to position space with an inverse FFT so that the mask can be applied.
3. The field in position space is multiplied by the mask:

$$U_{\text{out}}(x, y) = U_{\text{in}}(x, y) \times \text{grate\_mask}(x, y).$$

The mask represents either an amplitude modulation (`absorption_grating`) or a phase modulation (`phase_grating`), according to the current mode.

4. If `k_space_out = True`, the modified field is converted back to *k*-space using a forward FFT.

#### Typical use case

Called internally when constructing the transmission matrix (`.T_LTR_mat_tot`) and in cavity propagation chains where the grating is part of the optical path.

`.T_LTR_mat_tot`

#### Read-only property.

Returns the left-to-right transmission matrix of the component, at total resolution.

### Behavior

`.T_LTR_mat_tot` returns a full transmission matrix that represents the effect of applying the grating mask to an arbitrary input field in  $k$ -space modal basis.

The matrix is constructed by:

1. Propagating each plane wave basis function through the component using `.prop(...)`, with the grating mask applied in position space.
2. Converting the resulting field back to  $k$ -space modal coefficients.
3. Assembling these output vectors into the columns of the transmission matrix.

Since the grating can couple different  $k$ -space modes (due to its spatial periodicity), the transmission matrix is in general *non-diagonal*. Mode mixing is possible both in absorption and phase-grating modes.

### Symmetry

For this component, propagation is identical in both directions. Therefore:

$$\mathbf{T\_LTR\_mat\_tot} = \mathbf{T\_RTL\_mat\_tot}.$$

### Further details

For more information, see the documentation of `clsOptComponent2port.T_LTR_mat_tot`.

### `.T_RTL_mat_tot`

#### Read-only property.

Returns the right-to-left transmission matrix of the component, at total resolution.

### Behavior

`.T_RTL_mat_tot` is identical to `.T_LTR_mat_tot`. Since the grating is a purely thin component applied symmetrically in position space, the transformation is the same in both directions:

$$\mathbf{T\_LTR\_mat\_tot} = \mathbf{T\_RTL\_mat\_tot}.$$

### Further details

For more information, see the documentation of `clsOptComponent2port.T_RTL_mat_tot`.

### `.calc_T_mat_tot()`

### Method

Generates the transmission matrix  $\mathbf{T}$  at total resolution for the current grating parameters and stores it in memory.

### Behavior

`.calc_T_mat_tot()` proceeds as follows:

1. Retrieves the grating mask via `.grate_mask`; this property automatically triggers a recalculation if the mask is not yet cached.

2. For each plane-wave basis mode on the total grid in position space, multiplies the mode by the grating mask and converts the result back to modal coefficients.
3. Assembles the resulting column vectors into the full transmission matrix  $\mathbf{T}$ .

### Triggering

This method is automatically called when:

- `.T_LTR_mat_tot` or `.T_RTL_mat_tot` is accessed for the first time and no cached transmission matrix is available.
- `.set_cos_grating(...)` is called to update the grating parameters.

### Implementation note

The matrix assembly is parallelized using `joblib`. The number of worker processes is set by `clsCavity.mp_pool_processes`.

## D.5.4 Reflection Matrices

### `.R_L_mat_tot`

#### Read-only property.

Returns the left-side reflection matrix at total resolution.

#### Behavior

`.R_L_mat_tot` always returns the scalar 0. The `clsGrating` component is modeled as an ideal thin grating without any reflective behavior; only transmission is implemented.

#### Further details

See `clsOptComponent2port.R_L_mat_tot` for general information on reflection matrices.

### `.R_R_mat_tot`

#### Read-only property.

Returns the right-side reflection matrix at total resolution.

#### Behavior

`.R_R_mat_tot` always returns the scalar 0. The `clsGrating` component is modeled as an ideal thin grating without any reflective behavior; only transmission is implemented.

#### Further details

See `clsOptComponent2port.R_R_mat_tot` for general information on reflection matrices.

## D.5.5 Other Properties and Methods

### `.symmetric`

#### Read-only property.

Indicates whether the component is left-right symmetric.

#### Behavior

`.symmetric` always returns `True`. An ideal thin grating, as modeled by `clsGrating`, transforms the optical field identically when traversed from left to right or from right to left.

### `.k_space_in_dont_care`

#### **Read-only property.**

Indicates whether the component is indifferent to the input-space representation.

#### **Behavior**

`.k_space_in_dont_care` always returns `False`. This means the component is not indifferent to the input representation. It **prefers input in position space**, as indicated by `.k_space_in_prefer`.

#### **Further details**

For general background on this property, see `clsOptComponent.k_space_in_dont_care`.

### `.k_space_in_prefer`

#### **Read-only property.**

Indicates whether *k*-space input is preferred.

#### **Behavior**

`.k_space_in_prefer` always returns `False`, meaning that the component prefers input in position space.

#### **Further details**

For general background on this property, see `clsOptComponent.k_space_in_prefer`.

### `.k_space_out_dont_care`

#### **Read-only property.**

Indicates whether the component is indifferent to the output-space representation.

#### **Behavior**

`.k_space_out_dont_care` always returns `False`. This means the component is not indifferent to the output representation. It **prefers output in position space**, as indicated by `.k_space_out_prefer`.

#### **Further details**

For general background on this property, see `clsOptComponent.k_space_out_dont_care`.

### `.k_space_out_prefer`

#### **Read-only property.**

Indicates whether *k*-space output is preferred.

#### **Behavior**

`.k_space_out_prefer` always returns `False`, meaning that the component prefers output in position space.

**Further details**

For general background on this property, see `clsOptComponent.k_space_out_prefer`.

**.clear\_mem\_cache()****Method**

Clears all cached matrices and masks associated with this component from memory.

**Behavior**

`.clear_mem_cache()` first calls `clsOptComponent.clear_mem_cache()` to clear the cached *M-matrix* and inverse *M-matrix*, if present.

It then additionally clears:

- The cached grating mask `.grate_mask`.
- The cached transmission matrix `T` used by `.T_LTR_mat_tot` and `.T_RTL_mat_tot`.

**Typical use case**

Called automatically when grating parameters are changed (e.g., via `.set_cos_grating(...)`) or manually to force recomputation of the component's matrices.

## D.6 Class `clsTransmissionTilt`

The class `clsTransmissionTilt` represents an *ideal, loss-free phase element* that imposes a user-defined angular tilt on a light field by multiplying the field with separable phase masks in the *x*- and/or *y*-direction. It derives from `clsOptComponent2port` and therefore behaves like a two-port optical component with transparent transmission ( $R = 0$ ) and direction-selective phase transfer matrices.

**Key features:**

- Independent tilt angles in degrees (`.x_angle_deg`, `.y_angle_deg`) or in radians (`.x_angle`, `.y_angle`).
- Optional *zero lines* (`.x_zero_line`, `.y_zero_line`) that define where the phase ramp crosses zero.
- A `.direction` property that determines whether the tilt is applied for left-to-right (LTR) or right-to-left (RTL) propagation.

**Important limitation**

As with `clsGaussBeam`, the tilt is implemented purely as a phase factor. While this correctly changes the wave-front angle, it *does not* apply the geometrical compression that a field cross-section undergoes when projected onto the *xy*-plane at an oblique angle. Consequently, if a beam with a circular footprint enters this component, its footprint remains circular in the simulation grid even after large tilt angles. For small angles this approximation is usually adequate; for larger angles the user may compensate manually by scaling the field with `clsGrid.stretch_x(...)` and/or `clsGrid.stretch_y(...)`. A future release of the library will provide a built-in remedy for this shortcoming.

### D.6.1 Initialization

`clsTransmissionTilt(name, cavity)`

#### Constructor

Initializes a new instance of `clsTransmissionTilt`, an ideal thin optical component that applies a tilt to the wavefront of the incident light field via separable phase masks in  $x$ - and/or  $y$ -direction.

#### Behavior

On construction, the tilt parameters are initialized as follows:

- The tilt angles `.x_angle` and `.y_angle` are set to 0 degree (no tilt).
- The zero lines `.x_zero_line` and `.y_zero_line` are initialized to 0.
- The tilt is applied in the `.direction = Dir.LTR` (left-to-right) direction by default.

#### Typical use case

After creating a `clsTransmissionTilt` instance, configure the desired tilt angles via either `.x_angle_deg/.y_angle_deg` or `.x_angle/.y_angle`. If needed, adjust the zero lines to control where the phase ramps cross zero. Finally, insert the component into the desired position within the optical cavity path.

For further details, we refer to the description of the constructor of the root class `clsOptComponent(...)`.

### D.6.2 Tilt Parameters

#### `.x_angle`

##### Read/write property.

Specifies the tilt angle around the  $y$ -axis (tilt in the  $x$ -direction) in *radians*.

#### Behavior

When `.x_angle` is changed:

- The corresponding property `.x_angle_deg` (angle in degrees) is updated accordingly.
- The cached  $x$ -direction phase mask and the total transmission matrix are cleared so they can be regenerated with the new angle.
- If the component belongs to a cavity, the cavity's cached matrices are also cleared.

#### Usage notes

- Setting either `.x_angle` or `.x_angle_deg` is fully interchangeable; both properties update each other accordingly. The update mechanism ensures that no rounding errors are introduced for the angle in the unit that was used for setting.
- A positive `.x_angle` causes the wavefront to tilt such that, after further propagation, the light field appears shifted towards positive  $x$  (to the right); a negative angle results in a shift towards negative  $x$  (to the left). This behavior applies in

both propagation directions (`Dir.LTR` and `Dir.RTL`).

- The default value is 0, corresponding to no tilt.

### `.x_angle_deg`

#### **Read/write property (convenience).**

Specifies the same tilt angle as `.x_angle`, but in *degrees*.

#### **Behavior**

When `.x_angle_deg` is changed:

- The corresponding property `.x_angle` (angle in radians) is updated accordingly.
- The cached *x*-direction phase mask and the total transmission matrix are cleared so they can be regenerated with the new angle.
- If the component belongs to a cavity, the cavity's cached matrices are also cleared.

#### **Usage notes**

- Setting either `.x_angle` or `.x_angle_deg` is fully interchangeable; both properties update each other accordingly. The update mechanism ensures that no rounding errors are introduced for the angle in the unit that was used for setting.
- A positive `.x_angle_deg` causes the wavefront to tilt such that, after further propagation, the light field appears shifted towards positive *x* (to the right); a negative angle results in a shift towards negative *x* (to the left). This behavior applies in both propagation directions (`Dir.LTR` and `Dir.RTL`).
- The default value is 0, corresponding to no tilt.

### `.y_angle`

#### **Read/write property.**

Specifies the tilt angle around the *x*-axis (tilt in the *y*-direction) in *radians*.

#### **Behavior**

When `.y_angle` is changed:

- The corresponding property `.y_angle_deg` (angle in degrees) is updated accordingly.
- The cached *y*-direction phase mask and the total transmission matrix are cleared so they can be regenerated with the new angle.
- If the component belongs to a cavity, the cavity's cached matrices are also cleared.

#### **Usage notes**

- Setting either `.y_angle` or `.y_angle_deg` is fully interchangeable; both properties update each other accordingly. The update mechanism ensures that no rounding errors are introduced for the angle in the unit that was used for setting.

- A positive `.y_angle` causes the wavefront to tilt such that, after further propagation, the light field appears shifted towards positive  $y$  (upwards); a negative angle results in a shift towards negative  $y$  (downwards). This behavior applies in both propagation directions (`Dir.LTR` and `Dir.RTL`).
- The default value is 0, corresponding to no tilt.

### `.y_angle_deg`

#### **Read/write property (convenience).**

Specifies the same tilt angle as `.y_angle`, but in *degrees*.

#### **Behavior**

When `.y_angle_deg` is changed:

- The corresponding property `.y_angle` (angle in radians) is updated accordingly.
- The cached  $y$ -direction phase mask and the total transmission matrix are cleared so they can be regenerated with the new angle.
- If the component belongs to a cavity, the cavity's cached matrices are also cleared.

#### **Usage notes**

- Setting either `.y_angle` or `.y_angle_deg` is fully interchangeable; both properties update each other accordingly. The update mechanism ensures that no rounding errors are introduced for the angle in the unit that was used for setting.
- A positive `.y_angle_deg` causes the wavefront to tilt such that, after further propagation, the light field appears shifted towards positive  $y$  (upwards); a negative angle results in a shift towards negative  $y$  (downwards). This behavior applies in both propagation directions (`Dir.LTR` and `Dir.RTL`).
- The default value is 0, corresponding to no tilt.

### `.x_zero_line`

#### **Read/write property.**

Specifies the  $x$ -coordinate (in meters) at which the phase ramp in  $x$ -direction crosses zero.

#### **Behavior**

When `.x_zero_line` is changed:

- The cached  $y$ -direction phase mask is cleared.
- The cached total transmission matrix is cleared.
- If the component belongs to a cavity, the cavity's cached matrices are also cleared.

#### **Usage notes**

- The default value is 0, corresponding to a phase ramp centered at  $x = 0$ .
- The property affects only the  $x$ -direction phase mask.

**.y\_zero\_line****Read/write property.**

Specifies the  $y$ -coordinate (in meters) at which the phase ramp in  $y$ -direction crosses zero.

**Behavior**

When `.y_zero_line` is changed:

- The cached  $x$ -direction phase mask is cleared.
- The cached total transmission matrix is cleared.
- If the component belongs to a cavity, the cavity's cached matrices are also cleared.

**Usage notes**

- The default value is 0, corresponding to a phase ramp centered at  $y = 0$ .
- The property affects only the  $y$ -direction phase mask.

**.direction****Read/write property.**

Specifies the propagation direction (`Dir`) in which the tilt is applied.

**Behavior**

When `.direction` is changed:

- The cached total transmission matrix is cleared.
- If the component belongs to a cavity, the cavity's cached matrices are also cleared.

**Usage notes**

- The direction can be set to `Dir.LTR` (left-to-right) or `Dir.RTL` (right-to-left).
- The tilt is only applied when the field propagates in the specified direction; in the opposite direction, the component acts as a transparent element (no shifts).
- The default direction is `Dir.LTR`.

### D.6.3 Light Field Propagation

**.prop(E\_in, k\_space\_in, k\_space\_out, direction)****Method.**

Propagates an input light field `E_in` through the component in the specified direction, applying the tilt phase masks if applicable.

**Parameters**

- `E_in` – Input field. A NumPy array representing the field either in position space or  $k$ -space, as specified by `k_space_in`.

- `k_space_in` – True if `E_in` is given in  $k$ -space, False if it is in position space.
- `k_space_out` – True if the returned field should be in  $k$ -space, False for position space.
- `direction` – Propagation direction (`Dir.LTR` or `Dir.RTL`).

### Behavior

- If `k_space_in` is True, the input field is transformed to position space.
- If the parameter `direction` matches the setting of the property `.direction`, the tilt phase masks are applied to the field:
  - The  $x$ -direction mask `.x_mask` is applied if the tilt angle is non-zero.
  - The  $y$ -direction mask `.y_mask` is applied if the tilt angle is non-zero.
- If the parameter `direction` does not match the setting of the property `.direction`, the field passes through unchanged.
- If `k_space_out` is True, the resulting field is transformed to  $k$ -space before being returned.

### Returns

The propagated output field, either in position space or  $k$ -space, as specified by `k_space_out`.

## `.T_LTR_mat_tot`

**Read-only property.** Returns the left-to-right transmission matrix of the component (total resolution).

### Behavior

- If `.direction` is set to `Dir.LTR`, this property returns the transmission matrix representing the effect of the component in left-to-right propagation:
  - If both tilt angles `.x_angle_deg` and `.y_angle_deg` are zero, the matrix is simply the scalar 1 (no phase change).
  - Otherwise, the full transmission matrix is calculated (if not already cached) by calling `.calc_T_mat_tot()`, and then returned.
- If `.direction` is set to `Dir.RTL`, this property returns 1, i.e. the component acts as a transparent element in left-to-right propagation.

### Further details

For more information, see the documentation of `clsOptComponent2port.T_LTR_mat_tot`.

## `.T_RTL_mat_tot`

**Read-only property.**

Returns the right-to-left transmission matrix of the component (total resolution).

### Behavior

- If `.direction` is set to `Dir.RTL`, this property returns the transmission matrix representing the effect of the component in right-to-left propagation:

- If both tilt angles `.x_angle_deg` and `.y_angle_deg` are zero, the matrix is simply the scalar 1 (no phase change).
- Otherwise, the full transmission matrix is calculated (if not already cached) by calling `.calc_T_mat_tot()`, and then returned.
- If `.direction` is set to `Dir.LTR`, this property returns 1, i.e. the component acts as a transparent element in right-to-left propagation.

### Further details

For more information, see the documentation of `clsOptComponent2port.T_RTL_mat_tot`.

#### `.calc_T_mat_tot()`

##### Method.

Generates the transmission matrix **T** at total resolution for the current tilt parameters and stores it in memory.

##### Behavior

`.calc_T_mat_tot()` proceeds as follows:

1. Retrieves the tilt masks `.x_mask` and `.y_mask`; each mask automatically triggers a recalculation if it is not yet cached.
2. For each plane-wave basis mode on the total grid in position space, multiplies the mode by the tilt masks and converts the result back to modal coefficients.
3. Assembles the resulting column vectors into the full transmission matrix **T**.

##### Triggering

This method is automatically called when:

- `.T_LTR_mat_tot` or `.T_RTL_mat_tot` is accessed for the first time and no cached transmission matrix is available.

##### Implementation note

The matrix assembly is parallelized using `joblib`. The number of worker processes is set by `clsCavity.mp_pool_processes`.

#### `.x_mask`

##### Read-only property.

Returns the current tilt phase mask in *x*-direction, as a 2D NumPy array in position space.

##### Behavior

- If `.x_mask` is accessed and no cached mask is available, it is automatically calculated based on the current tilt angle `.x_angle` (or `.x_angle_deg`) and zero line `.x_zero_line`.
- The mask has the form  $\exp(ik_0x \tan(\alpha_x))$ , where  $k_0$  is the vacuum wavenumber and  $\alpha_x$  the tilt angle.
- If `.x_angle` is zero, the mask is simply the scalar 1.

**Usage notes**

- The property provides direct access to the phase mask applied in position space.
- To force recomputation of the mask, call `.clear_mem_cache()`.

**`.y_mask`****Read-only property.**

Returns the current tilt phase mask in  $y$ -direction, as a 2D NumPy array in position space.

**Behavior**

- If `.y_mask` is accessed and no cached mask is available, it is automatically calculated based on the current tilt angle `.y_angle` (or `.y_angle_deg`) and zero line `.y_zero_line`.
- The mask has the form  $\exp(ik_0 y \tan(\alpha_y))$ , where  $k_0$  is the vacuum wavenumber and  $\alpha_y$  the tilt angle.
- If `.y_angle` is zero, the mask is simply the scalar 1.

**Usage notes**

- The property provides direct access to the phase mask applied in position space.
- To force recomputation of the mask, call `.clear_mem_cache()`.

**D.6.4 Reflection Matrices****`.R_L_mat_tot`****Read-only property.**

Returns the left reflection matrix of the component (total resolution).

**Behavior**

- `.R_L_mat_tot` always returns the scalar 0.
- The component is an ideal loss-free phase element and introduces no reflection in either propagation direction.

**Usage notes**

- For further details, see the documentation of `clsOptComponent2port.R_L_mat_tot`.

**`.R_R_mat_tot`****Read-only property.**

Returns the right reflection matrix of the component (total resolution).

**Behavior**

- `.R_R_mat_tot` always returns the scalar 0.

- The component is an ideal loss-free phase element and introduces no reflection in either propagation direction.

#### Usage notes

- For further details, see the documentation of `clsOptComponent2port.R_R_mat_tot`.

### D.6.5 Other Properties and Methods

#### `.symmetric`

##### Read-only property.

Indicates whether the component is symmetric under inversion of the propagation direction.

##### Behavior

- `.symmetric` always returns `False`.
- The component applies a tilt phase only in the propagation direction specified by `.direction`; in the opposite direction it acts as a transparent element.
- Therefore, the component is not symmetric under direction reversal.

#### Usage notes

- For further details, see the documentation of `clsOptComponent2port.symmetric`.

#### `.k_space_in_dont_care`

##### Read-only property.

Indicates whether the component is indifferent to the input-space representation.

##### Behavior

`.k_space_in_dont_care` always returns `False`. This means the component is not indifferent to the input representation. It prefers **input in position space**, as indicated by `.k_space_in_prefer`.

##### Further details

For general background on this property, see `clsOptComponent.k_space_in_dont_care`.

#### `.k_space_in_prefer`

##### Read-only property.

Indicates whether *k*-space input is preferred.

##### Behavior

`.k_space_in_prefer` always returns `False`, meaning that the component prefers input in position space.

**Further details**

For general background on this property, see `clsOptComponent.k_space_in_prefer`.

**.k\_space\_out\_dont\_care****Read-only property.**

Indicates whether the component is indifferent to the output-space representation.

**Behavior**

`.k_space_out_dont_care` always returns `False`. This means the component is not indifferent to the output representation. It **prefers output in position space**, as indicated by `.k_space_out_prefer`.

**Further details**

For general background on this property, see `clsOptComponent.k_space_out_dont_care`.

**.k\_space\_out\_prefer****Read-only property.** Indicates whether *k*-space output is preferred.**Behavior**

`.k_space_out_prefer` always returns `False`, meaning that the component prefers output in position space.

**Further details**

For general background on this property, see `clsOptComponent.k_space_out_prefer`.

**.clear\_mem\_cache()****Method.**

Clears all cached phase masks and transmission matrices of the component.

**Behavior**

When `.clear_mem_cache()` is called:

- The cached *x*-direction phase mask `.x_mask` is deleted.
- The cached *y*-direction phase mask `.y_mask` is deleted.
- The cached transmission matrix `T` is deleted.
- The base implementation `clsOptComponent.clear_mem_cache()` is also called.

**Usage notes**

- Call this method manually if you want to force recomputation of masks and transmission matrix.
- This method is automatically called when any relevant property of the component is changed (e.g. `.x_angle`, `.x_zero_line`, or `.direction`).

## D.7 Class `clsSoftAperture`

The class `clsSoftAperture` models an *ideal, thin amplitude element* that limits a light field by a circular aperture with a configurable soft transition at its rim. Unlike a hard-edged aperture, the soft aperture implements a smooth radial transmission profile.

### Key features:

- Adjustable aperture diameter in meters (`.aperture`) or millimeters (`.aperture_mm`); both properties keep each other in sync.
- A *transition width* (`.transition_width`) that defines the  $1/e$ -width of the edge roll-off.
- A *black value* (`.black_value`) that sets the residual transmission in the fully blocked region (0 means fully opaque, 1 means fully transparent).

Typical workflow: create a `clsSoftAperture` instance, set the desired aperture diameter and edge softness, and insert the component at the required plane in the optical path.

### D.7.1 Initialization

`clsSoftAperture(name, cavity)`

#### Constructor.

Initializes a new instance of `clsSoftAperture`, a thin optical element that multiplies the incident field with a circular, radially graded transmission mask.

#### Behavior

On construction, the aperture parameters are initialized as follows:

- `.aperture` = 0 m and `.aperture_mm` = 0 mm (no aperture, fully transparent).
- `.transition_width` = 0 m (hard-edged aperture; no radial roll-off).
- `.black_value` = 0 (fully opaque outside the aperture).

#### Typical use case

After creating a `clsSoftAperture` instance, set the desired aperture diameter, edge softness, and residual transmission either individually or with the convenience method `.set_params(...)`. Insert the component at the required plane in the optical path; its effect is symmetric with respect to propagation direction.

For further details, see the constructor of the root class `clsOptComponent(...)`.

### D.7.2 Aperture Parameters

`.set_params(aperture, transition_width, black_value)`

#### Method.

Configures the soft aperture in a single step by setting the clear aperture diameter, the edge roll-off width, and the residual transmission in the blocked region.

#### Parameters

- **aperture** – Clear aperture diameter in meters (.aperture). Values  $\leq 0$  disable the aperture (fully transparent).
- **transition\_width** –  $1/e$ -width of the radial width of the soft transition region (in meters) . Values  $\leq 0$  create a hard-edged aperture.
- **black\_value** – Residual transmission in the fully blocked region (.black\_value). Clipped to the range [0, 1].

### Behavior

Calling .set\_params(...):

1. Clips each argument to its valid range and assigns it to the corresponding property. The aperture diameter in millimetres (.aperture\_mm) is updated automatically.
2. Clears any cached aperture mask and transmission matrix via .clear\_mem\_cache().
3. If the component belongs to a cavity, all cavity transfer matrices are invalidated so the new aperture takes effect in the next simulation step.

### Return value

None.

#### **.aperture**

##### Read/write property.

Specifies the clear-aperture diameter in meters.

### Behavior

The property .aperture defines the diameter of the soft aperture in meters.

When .aperture is changed:

- The corresponding property .aperture\_mm (diameter in millimetres) is automatically updated.
- The cached aperture mask .aperture\_mask and the cached transmission matrix are cleared.
- If the component belongs to a cavity, the cavity's cached matrices are also cleared.

### Usage note

A value of 0 disables the aperture (fully transparent element).

#### **.aperture\_mm**

##### Read/write property (convenience).

Specifies the clear-aperture diameter in millimetres.

### Behavior

The property .aperture\_mm defines the aperture diameter in millimetres.

When .aperture\_mm is changed:

- The corresponding property .aperture (diameter in meters) is automatically updated.

- The cached aperture mask `.aperture_mask` and the cached transmission matrix are cleared.
- If the component belongs to a cavity, the cavity's cached matrices are also cleared.

### Usage note

A value of 0 disables the aperture (fully transparent element).

#### `.transition_width`

##### Read/write property.

Specifies the width (in meters) of the soft transition region at the edge of the aperture.

##### Behavior

- A value of 0 creates a hard-edged aperture. The mask is generated by `clsGrid.get_aperture_mask(...)`.
- A positive value defines the  $1/e$ -width of a smooth radial transition from full transmission to the `.black_value`. The mask is generated by `clsGrid.get_soft_aperture_mask(...)`, which is called with the half-width parameter `transition_width/2`.

When `.transition_width` is changed:

- The cached aperture mask `.aperture_mask` and the cached transmission matrix are cleared.
- If the component belongs to a cavity, the cavity's cached matrices are also cleared.

### Usage note

Negative values are clipped to 0 (hard edge).

#### `.black_value`

##### Read/write property.

Sets the residual transmission value in the fully blocked region of the soft-aperture mask.

##### Behavior

When an aperture is applied (`.aperture > 0`), `.black_value` specifies the transmission level at and beyond the aperture radius:

- 0 → perfectly opaque edge (strict aperture).
- Value  $> 0$  → partially transmitting “gray” edge.

### Why use a non-zero black value?

A perfectly black edge (0) forces the transmission matrix to contain rows with all-zero entries, which makes the matrix (near-)singular and difficult to invert. This can degrade the numerical stability of simulations that rely on scattering or transfer-matrix methods. Choosing a small positive `.black_value` (e.g. 0.01) keeps every mode weakly

coupled, producing well-conditioned matrices while still strongly attenuating light outside the aperture.

### Behavior when changed

Changing `.black_value`:

- Clears the cached aperture mask `.aperture_mask` and the cached transmission matrix.
- Invalidates the cavity's cached matrices (if the component belongs to a cavity).

## D.7.3 Light Field Propagation

`.prop(E_in, k_space_in, k_space_out, direction)`

### Method.

Propagates an input light field `E_in` through the soft aperture, applying the transmission mask and performing optional space– $k$ -space conversions.

### Parameters

- `E_in` – Input field. A NumPy array representing the field either in position space or  $k$ -space, as specified by `k_space_in`.
- `k_space_in` – True if `E_in` is provided in  $k$ -space, `False` if it is in position space.
- `k_space_out` – True if the returned field should be in  $k$ -space, `False` for position space.
- `direction` – Propagation direction (`Dir.LTR` or `Dir.RTL`). For this symmetric component the result is independent of the chosen direction.

### Behavior

1. If `k_space_in` is `True`, the input field is converted to position space.
2. The field is multiplied by the aperture mask `.aperture_mask`. Accessing this property automatically generates or retrieves the cached mask.
3. If `k_space_out` is `True`, the resulting field is converted back to  $k$ -space.

### Return value

The propagated output field, either in position space or  $k$ -space, as specified by `k_space_out`.

`.T_LTR_mat_tot`

### Read-only property.

Returns the left-to-right transmission matrix of the component at total grid resolution.

### Behavior

- If the cached matrix is not yet available, it is generated by calling `.calc_T_mat_tot()` and then stored.
- Because `.symmetric` is `True`, this property returns the same matrix as `.T_RTL_mat_tot`.

- When the aperture is disabled (`.aperture = 0`), and the transmission matrix reduces to the scalar value 1 (no amplitude change).

### `.T_RTL_mat_tot`

#### **Read-only property.**

Returns the right-to-left transmission matrix of the component at total grid resolution.

#### **Behavior**

- If the cached matrix is not yet available, it is generated by calling `.calc_T_mat_tot()` and then stored.
- Because `.symmetric` is `True`, this property returns the same matrix as `.T_LTR_mat_tot`.
- When the aperture is disabled (`.aperture = 0`), the transmission matrix reduces to the scalar value 1 (no amplitude change).

### `.calc_T_mat_tot()`

#### **Method.**

Generates the transmission matrix  $\mathbf{T}$  used by `.T_LTR_mat_tot` and `.T_RTL_mat_tot` at total resolution for the current soft aperture parameters and stores it in memory.

**Behavior** `.calc_T_mat_tot()` proceeds as follows:

1. Retrieves the aperture mask via `.aperture_mask`; this property automatically triggers a recalculation if the mask is not yet cached.
2. For each plane-wave basis mode on the total grid in position space, multiplies the mode by the aperture mask and converts the result back to modal coefficients.
3. Assembles the resulting column vectors into the full transmission matrix  $\mathbf{T}$ .

#### **Triggering**

This method is automatically called when:

- `.T_LTR_mat_tot` or `.T_RTL_mat_tot` is accessed for the first time and no cached transmission matrix is available.
- The aperture parameters (`.aperture`, `.transition_width`, or `.black_value`) are changed.
- `.set_params(...)` is called to update the aperture parameters.

#### **Implementation note**

The matrix assembly is parallelized using `joblib`. The number of worker processes is set by `clsCavity.mp_pool_processes`.

### `.aperture_mask`

#### **Read-only property.**

Returns the current aperture mask as a 2D NumPy array in position space.

**Behavior**

- If the cached mask is not yet available, it is automatically calculated based on the current aperture parameters:
  - If `.aperture` = 0, the mask equals the scalar value 1.
  - If `.transition_width` = 0, a hard-edged circular mask is generated by `clsGrid.get_aperture_mask(...)`.
  - If `.transition_width` > 0, a soft-edged circular mask is generated by `clsGrid.get_soft_aperture_mask(...)`, using half the transition width as parameter.
- The generated mask is cached until any relevant aperture parameter is changed or until `.clear_mem_cache()` is called.

**D.7.4 Reflection Matrices****`.R_L_mat_tot`****Read-only property.**

Returns the left reflection matrix of the component at total resolution.

**Behavior**

`.R_L_mat_tot` always returns the scalar value 0, because the component introduces no reflection in either propagation direction.

**Further details**

For more background on this property, see `clsOptComponent2port.R_L_mat_tot`.

**`.R_R_mat_tot`****Read-only property.**

Returns the right reflection matrix of the component at total resolution.

**Behavior**

`.R_R_mat_tot` always returns the scalar value 0, because the component introduces no reflection in either propagation direction.

**Further details**

For more background on this property, see `clsOptComponent2port.R_R_mat_tot`.

**D.7.5 Other Properties and Methods****`.symmetric`****Read-only property.**

Indicates whether the component is symmetric under inversion of the propagation direction.

**Behavior**

- `.symmetric` always returns True.

- The component applies the same transmission mask in both propagation directions; its behavior is identical for left-to-right and right-to-left propagation.

#### Further details

For general background on this property, see `clsOptComponent2port.symmetric`.

### `.k_space_in_dont_care`

#### Read-only property.

Indicates whether the component is indifferent to the input-space representation.

#### Behavior

`.k_space_in_dont_care` always returns `False`. This means the component is not indifferent to the input representation. It **prefers input in position space**, as indicated by `.k_space_in_prefer`.

#### Further details

For general background on this property, see `clsOptComponent.k_space_in_dont_care`.

### `.k_space_in_prefer`

#### Read-only property.

Indicates whether *k*-space input is preferred.

#### Behavior

`.k_space_in_prefer` always returns `False`, meaning that the component prefers input in position space.

#### Further details

For general background on this property, see `clsOptComponent.k_space_in_prefer`.

### `.k_space_out_dont_care`

#### Read-only property.

Indicates whether the component is indifferent to the output-space representation.

#### Behavior

`.k_space_out_dont_care` always returns `False`. This means the component is not indifferent to the output representation. It **prefers output in position space**, as indicated by `.k_space_out_prefer`.

#### Further details

For general background on this property, see `clsOptComponent.k_space_out_dont_care`.

### `.k_space_out_prefer`

#### Read-only property.

Indicates whether *k*-space output is preferred.

#### Behavior

`.k_space_out_prefer` always returns `False`, meaning that the component prefers output in position space.

#### Further details

For general background on this property, see `clsOptComponent.k_space_out_prefer`.

#### `.clear_mem_cache()`

##### Method.

Clears all cached data of the component.

**Behavior** When `.clear_mem_cache()` is called:

- The cached aperture mask `.aperture_mask` is deleted.
- The cached transmission matrix `T` is deleted.
- The base implementation `clsOptComponent.clear_mem_cache()` is also called.

##### Usage notes

- Call this method manually if you want to force recomputation of the aperture mask and transmission matrix.
- This method is automatically called when any relevant aperture parameter is changed (e.g. `.aperture`, `.transition_width`, or `.black_value`).

## D.8 Class `clsAmplitudeScaling`

The class `clsAmplitudeScaling` models an ideal thin optical element that uniformly scales the amplitude of the light field passing through it.

Key features:

- Scaling factor set by the property `.amplitude_scale_factor`.
- Identical behavior in both propagation directions (`.symmetric = True`).
- No reflection (`.R_L_mat_tot`, `.R_R_mat_tot = 0`).
- No optical or physical propagation distance (`clsOptComponent.dist_phys = 0`, and `clsOptComponent.dist_opt = 0`).

Typical use case: Insert the component at an arbitrary plane in the optical path and adjust `.amplitude_scale_factor` to achieve the desired attenuation or amplification. Note that `.amplitude_scale_factor` can be a *complex* value; this allows the component to apply a global phase shift in addition to amplitude scaling.

### D.8.1 Initialization

```
clsAmplitudeScaling(name, cavity)
```

#### Constructor.

Initializes a new instance of `clsAmplitudeScaling`, an ideal thin optical component that uniformly scales the amplitude of the light field.

#### Behavior

On construction, the component parameters are initialized as follows:

- `.amplitude_scale_factor = 1` (no scaling).

#### Typical use case

After creating a `clsAmplitudeScaling` instance, adjust the desired scaling factor via `.amplitude_scale_factor`. The scaling factor can be a *complex* value, allowing simultaneous amplitude scaling and global phase shifting.

For further details, see the constructor of the root class `clsOptComponent(...)`.

### D.8.2 Scaling Parameter

```
.amplitude_scale_factor
```

#### Read/write property.

Specifies the complex scaling factor applied to the amplitude of the light field.

#### Usage notes

- The scaling factor can be a *complex* value. The magnitude controls amplitude scaling, and the argument (phase) controls the global phase shift applied to the light field.
- The same factor is applied in both propagation directions.
- The default value is 1, corresponding to no scaling.

### D.8.3 Light Field Propagation

```
.prop(E_in, k_space_in, k_space_out, direction)
```

#### Method.

Propagates an input light field `E_in` through the component, applying uniform amplitude scaling and optional space– $k$ -space conversions.

#### Parameters

- `E_in` – Input field. A NumPy array representing the field either in position space or  $k$ -space, as specified by `k_space_in`.
- `k_space_in` – `True` if `E_in` is provided in  $k$ -space, `False` if it is in position space.
- `k_space_out` – `True` if the returned field should be in  $k$ -space, `False` for position space.
- `direction` – Propagation direction (`Dir.LTR` or `Dir.RTL`). For this symmetric

component, the result is independent of the chosen direction.

#### Behavior

1. Multiplies the input field `E_in` by the scalar factor `.amplitude_scale_factor`.
2. If `k_space_in`  $\neq$  `k_space_out`, converts the field between position space and  $k$ -space using `clsGrid.convert(...)`.

#### Return value

The propagated output field, either in position space or  $k$ -space, as specified by `k_space_out`.

### `.T_LTR_mat_tot`

#### Read-only property.

Returns the left-to-right transmission matrix of the component.

#### Behavior

- `.T_LTR_mat_tot` always returns the scalar value `.amplitude_scale_factor`.
- The component applies uniform scaling to all modes; therefore, the transmission matrix reduces to a scalar factor.

#### Usage notes

The same value is also returned by `.T_RTL_mat_tot`, since the component is symmetric.

#### Further details

For background on this property, see `clsOptComponent2port.T_LTR_mat_tot`.

### `.T_RTL_mat_tot`

#### Read-only property.

Returns the right-to-left transmission matrix of the component.

#### Behavior

- `.T_RTL_mat_tot` always returns the scalar value `.amplitude_scale_factor`.
- The component applies uniform scaling to all modes; therefore, the transmission matrix reduces to a scalar factor.

#### Usage notes

The same value is also returned by `.T_LTR_mat_tot`, since the component is symmetric.

#### Further details

For background on this property, see `clsOptComponent2port.T_RTL_mat_tot`.

### D.8.4 Light Field Reflection

### `.reflect(E_in, k_space_in, k_space_out, side)`

#### Method.

Computes the reflected field for a given incident field `E_in`. The method `.reflect(...)`

serves as the conceptual counterpart to `.prop(...)`; however, unlike `.prop(...)`, it is *not* part of the standard interface imposed by the root class `clsOptComponent2port`, and not all two-port components implement it.

#### Parameters

- `E_in` – Input field. A NumPy array representing the field either in position space or  $k$ -space, as specified by `k_space_in`.
- `k_space_in` – `True` if `E_in` is provided in  $k$ -space, `False` if it is in position space.
- `k_space_out` – `True` if the returned field should be in  $k$ -space, `False` for position space.
- `side` – Reflection side (`Side.L` or `Side.R`).

#### Behavior

`.reflect(...)` always returns the scalar value 0. The component is an ideal thin amplitude element and introduces no reflection on either side.

### `.R_L_mat_tot`

#### Read-only property.

Returns the left reflection matrix of the component.

#### Behavior.

The property `.R_L_mat_tot` always returns the scalar value 0, because the component is an ideal thin amplitude element and introduces no reflection in either propagation direction.

#### Further details

For more background on this property, see `clsOptComponent2port.R_L_mat_tot`.

### `.R_R_mat_tot`

#### Read-only property.

Returns the right reflection matrix of the component.

#### Behavior.

The property `.R_R_mat_tot` always returns the scalar value 0, because the component is an ideal thin amplitude element and introduces no reflection in either propagation direction.

#### Further details

For more background on this property, see `clsOptComponent2port.R_L_mat_tot`.

## D.8.5 Other Properties and Methods

### `.symmetric`

#### Read-only property.

Indicates whether the component is symmetric under inversion of the propagation direction.

**Behavior**

`.symmetric` always returns `True`, because the component applies the same amplitude scaling factor in both propagation directions.

**`.k_space_in_dont_care`****Read-only property.**

Indicates whether the component is indifferent to the input-space representation.

**Behavior**

`.k_space_in_dont_care` always returns `True`. This means the component is indifferent to the input representation. It applies the same scaling factor regardless of whether the field is represented in position space or  $k$ -space.

**`.k_space_in_prefer`****Read-only property.**

Indicates whether  $k$ -space input is preferred.

**Behavior**

`.k_space_in_prefer` always returns `False`. Since `.k_space_in_dont_care = True`, this property is irrelevant and can be ignored.

**`.k_space_out_dont_care`****Read-only property.**

Indicates whether the component is indifferent to the output-space representation.

**Behavior**

`.k_space_out_dont_care` always returns `True`. This means the component is indifferent to the output representation. It applies the same scaling factor regardless of whether the field is represented in position space or  $k$ -space.

**`.k_space_out_prefer`****Read-only property.**

Indicates whether  $k$ -space output is preferred.

**Behavior**

`.k_space_out_prefer` always returns `False`. Since `.k_space_out_dont_care = True`, this property is irrelevant and can be ignored.

## D.9 Abstract Base Class `clsMirrorBase2port`

The class `clsMirrorBase2port` is an *abstract* two-port component derived from `clsOptComponent2port`. It provides the common functionality for its concrete subclasses `clsMirror` (thin, flat mirror) and `clsCurvedMirror` (thin, curved mirror).

### Key responsibilities:

- **Power coefficients.** Stores and couples the power reflectivity `.R` and transmissivity `.T`; maintains the corresponding complex field coefficients (`.r_L`, `.r_R`, `.t_LTR`, `.t RTL`) in either *symmetric-phase* or *sign-convention* mode (`.sym_phase`).
- **Mirror tilt.** The class supports mirror tilts around the  $x$ - and  $y$ -axes (via `.rot_around_x`, `.rot_around_y` and their degree-based variants). It also encapsulates the functionality to compute the corresponding phase masks that simulate the geometric tilt of the mirror surface (`.tilt_mask_x_L`, `.tilt_mask_y_R`, etc.).
- **Projection onto tilted mirror surfaces.** The class encapsulates the geometric calculations required to project an incident light field onto the mirror surface, considering both the mirror tilt (around  $x$ ) and a possible angle of incidence in  $+y$  or  $-y$  direction (via `.incident_angle_y` / `.incident_angle_y_deg`). This is essential to correctly simulate astigmatism effects – for example, in bow-tie cavities with tilted curved mirrors.
- **Side relevance flags.** The boolean properties `.left_side_relevant` and `.right_side_relevant` allow you to disable unnecessary computations for a given side. This is particularly useful when reflection behavior on one side is irrelevant for the optical model and would otherwise incur additional cost – for example, due to mirror tilt or astigmatism corrections.

*Example:* In a linear cavity with a partially reflective input mirror on the left and a nearly perfect mirror on the right, only the reflection from the left side is of interest. Setting `.right_side_relevant` to `False` will skip phase mask and projection calculations for the right outer side, saving computation time and memory.

- **“Transmission behaves like reflection” mode.** The class supports a special mode in which the *transmission* behavior in one direction (LTR or RTL) mimics the *reflection* from either the left or right mirror surface (via `.LTR_transm_behaves_like_refl_left` etc.). In this case, the transmission in the opposite direction behaves *neutrally*, meaning the light field passes through the component unchanged. This allows simulating the full optical path of unidirectional cavities (e.g. ring or bow-tie cavities), where the light never exactly retraces its steps, but still forms a closed path.

For example, consider a bow-tie cavity in which the beam meets mirrors  $R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow R_4 \rightarrow R_5$  before returning to  $R_1$ , with free-space propagation segments  $P$  between them. You can model this loop with five mirror components  $C_1, \dots, C_5$  arranged linearly as

$$C_1 \ C_5 \ P \ C_2 \ C_4 \ P \ C_3.$$

Here

- $C_1$  and  $C_2$  emulate the reflections at  $R_1$  and  $R_2$  by setting their LTR *transmission* to behave like reflection from the left or right surface, respectively;
- $C_3$  models the real reflection at  $R_3$ ;
- $C_4$  and  $C_5$  emulate the reflections at  $R_4$  and  $R_5$  by setting their RTL transmission to behave like reflection.

In this way the unidirectional five-mirror loop is captured by a simple linear two-port chain, eliminating the need for a full four-port formalism.

Concrete subclasses inherit all of these mechanisms and add only the geometry-specific phase terms (flat vs. curved). As a result, most mirror-related behaviour can be tuned at the abstract level and reused unchanged across different mirror types.

### D.9.1 Initialization

`clsMirrorBase2port(name, cavity)`

#### Constructor.

Initializes a new instance of `clsMirrorBase2port`.

#### Arguments

- `name` – A string identifier for the component.
- `cavity` – The optical cavity to which this component belongs.

#### Details

The constructor sets default values for the reflection and transmission parameters:

- Power reflection  $R = 1$ , power transmission  $T = 0$
- Complex reflection coefficient  $r = -1$ , complex transmission coefficient  $t = 0$
- Symmetric phase behavior is enabled by default (`.sym_phase = True`)
- Mirror tilt is initially set to zero (no rotation)

For further details, see the constructor of the root class `clsOptComponent(...)`.

### D.9.2 Reflectivity and Transmissivity Parameters

`.R`

#### Read/write property.

Specifies the power reflectivity  $R \in [0, 1]$  of the mirror on either side.

#### Behavior

- Changing `.R` automatically updates the power transmissivity `.T` to  $T = 1 - R$ .
- The corresponding complex reflection and transmission coefficients (`.r_L`, `.r_R`, `.t_LTR`, `.t RTL`) are recalculated accordingly.
- If `.sym_phase = True`, then the complex reflection coefficient includes a physical phase; otherwise, it is real-valued.

#### Value constraints

- Values greater than 1 are clamped to 1 with a warning.
- Values less than 0 are clamped to 0 with a warning.

#### Usage notes

Setting this property invalidates the cavity's cached matrices (if the component belongs to a cavity).

**.T****Read/write property.**

Specifies the power transmissivity  $T \in [0, 1]$  of the mirror on either side.

**Behavior**

- Changing `.T` automatically updates the power reflectivity `.R` to  $R = 1 - T$ .
- The corresponding complex reflection and transmission coefficients (`.r_L`, `.r_R`, `.t_LTR`, `.t_RTL`) are recalculated accordingly.
- If `.sym_phase = True`, then the complex reflection coefficient includes a physical phase; otherwise, it is real-valued.

**Value constraints**

- Values greater than 1 are clamped to 1 with a warning.
- Values less than 0 are clamped to 0 with a warning.

**Usage notes**

Setting this property invalidates the cavity's cached matrices (if the component belongs to a cavity), to ensure consistency.

**`.set_T_non_energy_conserving(T)`****Method.**

Assigns an *unphysical* power transmissivity  $T$  while leaving the power reflectivity `.R` unchanged.

**Why is this needed?**

For a perfectly reflecting mirror (`.R = 1`) the automatically coupled transmissivity is  $T = 0$ . A strictly zero transmissivity makes the transfer matrix singular and causes numerical problems calculating the scattering matrix. Calling `.set_T_non_energy_conserving(...)` lets you inject a small, fictitious  $T > 0$  (e.g.  $10^{-6}$ ), which keeps the transfer matrix well-conditioned.

*Example.* Consider a linear cavity in which the left mirror is the input coupler ( $R < 1$ ) and the right mirror is intended to be perfectly reflective ( $R = 1$ ). Numerically, setting  $R = 1$  forces  $T = 0$  and makes the transfer matrix singular, which in turn prevents the scattering matrix from being computed. By calling `.set_T_non_energy_conserving(...)` with a small fictitious value, say  $T = 10^{-6}$ , you keep the mirror optically lossless for all practical purposes, yet the transfer matrix stays invertible. Because no diagnostics or useful signal is taken from the right-hand output, the tiny "leakage" introduced by this artificial  $T$  has no impact on the simulation results while it stabilises the numerical procedure.

**Behavior**

- Stores the supplied  $T$  directly in `.T` *without* adjusting `.R`.
- Recalculates the complex coefficients (`.r_L`, `.r_R`, `.t_LTR`, `.t_RTL`).
- Invalidates the cavity's cached matrices (if the component belongs to a cavity) to ensure consistency.

### Typical use case

Set `.R = 1` for a perfect reflector and then call `.set_T_non_energy_conserving(1e-6)` to avoid singular transfer matrices while keeping the mirror effectively lossless.

`.set_phys_behavior(sym_phase)`

### Method.

Selects the phase convention used for the mirror's complex reflection and transmission coefficients and, by extension, for the mirror scattering matrix  $\mathbf{S}$ .

### Parameters

- `sym_phase` – Boolean flag `True`: “symmetric-phase” convention (default) `False`: “text-book” real-valued convention

### Why two conventions?

For energy to be conserved the scattering matrix must be *unitary*, i.e.  $\mathbf{SS}^\dagger = \mathbb{1}$ . With only a single spatial mode one may write

$$\mathbf{S} = \begin{pmatrix} r & t \\ t & r \end{pmatrix}, \quad r, t \in \mathbb{C},$$

where  $R = |r|^2$  is the desired reflectivity.

#### Symmetric-phase convention (`sym_phase = True`)

Setting

$$r = -R - i\sqrt{R}\sqrt{1-R}, \quad t = 1 + r,$$

guarantees unitarity for any  $0 \leq R \leq 1$  and assigns *the same phase shift* to reflections from both sides of the mirror [7, p. 55]. This is physically intuitive but introduces a small phase term into  $t$ ; in resonator calculations this phase slightly shifts the resonance condition away from the simple “text-book” cavity length.

#### Real-valued (“text-book”) convention (`sym_phase = False`)

A widely used alternative is to force the coefficients to be real [8, p. 406]:

$$\mathbf{S} = \begin{pmatrix} r & t \\ t & -r \end{pmatrix}, \quad r, t \in [0, 1],$$

which is also unitary and keeps  $t$  real and positive; however, the reflection phases differ for the left and right facets ( $r_R = -r_L$ ).

### Generalisation to the multi-mode case

In this library the single-mode coefficients become diagonal matrices, so in block-matrix notation

$$\mathbf{S} = \begin{pmatrix} \mathbf{R}_L & \mathbf{T}_{RL} \\ \mathbf{T}_{LR} & \mathbf{R}_R \end{pmatrix} = \begin{pmatrix} \mathbf{R} & \mathbf{T} \\ \mathbf{T} - \mathbf{R} & \mathbf{R} \end{pmatrix}, \quad \mathbf{R}, \mathbf{T} \in \mathbb{R}^{N^2 \times N^2}.$$

### Behavior when called

- Updates the internal reflection and transmission coefficients (`.r_L`, `.t_LTR`, etc.) according to the chosen convention.

- Clears cached reflection masks and matrices so that they are rebuilt with the new phase behaviour.

### `.sym_phase`

#### **Read-only property.**

Reports the current phase convention used for the mirror's complex reflection and transmission coefficients.

#### **Return values**

- `True` – *Symmetric-phase* convention (identical reflection phase on both sides; complex  $t$ ).
- `False` – *Real-valued* (“text-book”) convention (real  $r, t$ ; opposite signs for the two reflection phases).

#### **Changing the convention**

This property is read-only; use `.set_phys_behavior(...)` to switch between the two conventions. That method recalculates `.r_L`, `.r_R`, `.t_LTR`, and `.t RTL` and invalidates any cached matrices so the new phase behaviour takes effect.

### `.r_L`

#### **Read-only property.**

Returns the complex reflection coefficient  $r_L$  for light incident from the left side of the mirror.

#### **Behavior**

- Under normal conditions `.r_L` equals the internally stored complex coefficient computed from `.R` and the current phase convention (see `.set_phys_behavior(...)` and `.sym_phase`).
- If `.no_reflection_phase_shift` is `True`, the reflection phase is suppressed:
  - In symmetric-phase mode (`.sym_phase = True`) the value is replaced by  $\sqrt{R}$  (purely real and positive).
  - In text-book mode (`sym_phase = False`) the stored coefficient is already real, so the original value is returned.

#### **Usage notes**

- The magnitude satisfies  $|r_L| = \sqrt{R}$ .
- Any change to `.R`, `.T`, or the phase convention automatically updates this value.

### `.r_R`

#### **Read-only property.**

Returns the complex reflection coefficient  $r_R$  for light incident from the right side of the mirror.

### Behavior

- In symmetric-phase mode (`.sym_phase = True`) the coefficient is identical to `.r_L`:  $r_R = r_L$ .
- In text-book (real-valued) mode (`sym_phase = False`) the right-side coefficient has the opposite sign:  $r_R = -r_L$ .
- If `.no_reflection_phase_shift` is `True`, the phase of the coefficient is removed:
  - In symmetric-phase mode the property returns  $+\sqrt{R}$ .
  - In text-book mode the stored value is already real, so it is returned unchanged (with sign according to the rule above).

### Usage notes

- The magnitude always satisfies  $|r_R| = \sqrt{R}$ .
- Any change to `.R`, `.T`, or the phase convention automatically updates this value.

## `.t_LTR`

### Read-only property.

Returns the complex transmission coefficient  $t_{LTR}$  for light propagating from left to right.

### Behavior

- The coefficient is computed together with `.t_RTL` by the internal routine that enforces unitarity (see `.set_phys_behavior(...)`).
- *Symmetric-phase mode* (`.sym_phase = True`):  $t_{LTR} = 1 + r$  with  $r = .r_L$ . The term carries a small phase whose magnitude depends on  $\sqrt{R}(1 - R)$ .
- *Real-valued (“text-book”) mode* (`sym_phase = False`):  $t_{LTR} = +\sqrt{T}$  – purely real and positive.

### Usage notes

- The magnitude satisfies  $|t_{LTR}| = \sqrt{T}$  in either convention.
- Any change to `.R`, `.T`, or the phase convention automatically updates this value.

## `.t_RTL`

### Read-only property.

Returns the complex transmission coefficient  $t_{RTL}$  for light propagating from right to left.

### Behavior

- As with `.t_LTR`, the coefficient is generated by the internal routine that enforces unitarity (see `.set_phys_behavior(...)`).
- *Symmetric-phase mode* (`.sym_phase = True`):  $t_{RTL} = 1 + r$  with  $r = .r_R$ . Because  $r_R = r_L$  in this mode, the two transmission coefficients are identical:  $t_{RTL} = t_{LTR}$ .
- *Real-valued (“text-book”) mode* (`sym_phase = False`):  $t_{RTL} = +\sqrt{T}$ , purely real and positive – just like  $t_{LTR}$ .

**Usage notes**

- The magnitude satisfies  $|t_{RTL}| = \sqrt{T}$  in either convention.
- Any change to `.R`, `.T`, or the phase convention automatically updates this value.

**`.no_reflection_phase_shift`****Read/write property.**

When set to `True`, suppresses any phase shift upon reflection so that both `.r_L` and `.r_R` return the *same* real-valued coefficient  $+\sqrt{R}$ .

**Behavior**

- If `True`, the phase of the reflection coefficients is removed, independent of the underlying phase convention (`.sym_phase`). Both sides return  $+\sqrt{R}$ .
- If `False`, the reflection coefficients follow the selected phase convention (symmetric-phase or real-valued with sign difference).
- The transmission coefficients (`.t_LTR`, `.t RTL`) are unaffected.
- Changing this flag invalidates the cavity's cached matrices (if the component belongs to a cavity) to ensure consistency.

The default is `False`, retaining the standard phase behaviour chosen via the method `.set_phys_behavior(...)`.

**`.left_side_relevant`****Read/write property.**

Boolean flag that states whether accurate calculations are required for the *left* mirror surface (reflection masks, projection factors, tilt-induced astigmatism, ...).

**Behavior**

- `True` (default): the left-hand reflection is evaluated with full accuracy, including tilt masks and projection corrections if enabled.
- `False`: the code may skip all calculations specific to the left surface, saving time and memory when that port is known to be unused. (For example, a perfect end mirror on the left side of a linear cavity may render the left-hand reflection irrelevant.)
- Changing this flag invalidates the cavity's cached matrices (if the component belongs to a cavity) to ensure consistency.
- If you later activate a feature that *requires* the left reflection – e.g. set `.LTR_transm_behaves_like_refl_left` or `.RTL_transm_behaves_like_refl_left` to `True` – the class will automatically reset `.left_side_relevant` to `True` and emit a console message.

**Usage notes**

- Use this flag (together with `.right_side_relevant`) to avoid costly tilt-mask and projection calculations for mirror surfaces that do not contribute to the simulated signals.

**.right\_side\_relevant****Read/write property.**

Boolean flag indicating whether the *right* mirror surface must be treated with full numerical accuracy (reflection masks, projection factors, tilt-induced astigmatism, . . . ).

**Behavior**

- **True** (default): the right-hand reflection is evaluated in detail, including any enabled tilt or projection corrections.
- **False**: mirror-surface calculations specific to the right side may be skipped, reducing runtime and memory usage when that port plays no role in the simulation. (For example, a perfect end mirror on the right side of a linear cavity may render the right-hand reflection irrelevant.)
- Changing this flag invalidates the cavity's cached matrices (if the component belongs to a cavity) to keep all data consistent.
- If a feature is activated that *requires* the right reflection (e.g. setting the property `.LTR_transm_behaves_like_refl_right` or the property `.RTL_transm_behaves_like_refl_right` to `True`) the class automatically resets `.right_side_relevant` to `True` and prints a console message.

**Usage notes**

- Use this flag with (together with `.left_side_relevant`) to avoid costly tilt-mask and projection calculations for mirror surfaces that do not contribute to the simulated signals.

### D.9.3 Mirror Tilt

**.rot\_around\_x****Read/write property.**

Specifies the mirror tilt angle about the *x*-axis in *radians*.

**Behavior**

- Setting `.rot_around_x` automatically updates the convenience property `.rot_around_x_deg` (angle in degrees).
- The new tilt invalidates cached tilt masks (`.tilt_mask_y_L`, `.tilt_mask_y_R`) and any transfer matrices that depend on them; the cavity's cached matrices are cleared as well.
- Allowed range:  $|\alpha| < \pi/4$ . Values outside this range are rejected with an error message.

**Sign convention**

A *positive* angle tilts the mirror such that the left-side reflection is deflected toward  $+y$  (*up*) and the right-side reflection toward  $-y$  (*down*).

**Usage notes**

- For small tilts the resulting lateral walk-off is modelled by the phase masks; larger

tilts can be combined with `.project_according_to_angles` and `.consider_tilt_astigmatism` to include projection and astigmatism effects.

- Use the degree-based setter `.rot_around_x_deg` if working in degrees is more convenient.

### `.rot_around_x_deg`

#### **Read/write property.**

Specifies the mirror tilt angle about the x-axis in *degrees*.

#### **Behavior**

- Setting `.rot_around_x_deg` automatically updates the radian-based property `.rot_around_x`.
- The new tilt invalidates cached tilt masks (`.tilt_mask_y_L`, `.tilt_mask_y_R`) and any transfer matrices that depend on them; the cavity's cached matrices are cleared as well.
- Allowed range:  $|\alpha| < 45^\circ$ . Values outside this range are rejected with an error message.

#### **Sign convention**

A *positive* angle tilts the mirror such that the left-side reflection is deflected toward  $+y$  (*up*) and the right-side reflection toward  $-y$  (*down*).

#### **Usage notes**

- This is a convenience property for users working in degrees. Use the radian-based version `.rot_around_x` if needed for computation.
- Setting either `.rot_around_x` or `.rot_around_x_deg` is fully interchangeable; both properties update each other accordingly. The update mechanism ensures that no rounding errors are introduced for the angle in the unit that was used for setting.

### `.rot_around_y`

#### **Read/write property.**

Specifies the mirror tilt angle about the y-axis in *radians*.

#### **Behavior**

- Setting `.rot_around_y` automatically updates the degree-based property `.rot_around_y_deg`.
- The new tilt invalidates cached tilt masks (`.tilt_mask_x_L`, `.tilt_mask_x_R`) and any transfer matrices that depend on them; the cavity's cached matrices are cleared as well.
- Allowed range:  $|\alpha| < \pi/4$ . Values outside this range are rejected with an error message.

### Sign convention

A *positive* angle tilts the mirror such that the left-side reflection is deflected toward  $-x$  (*left* when viewed from the right), and the right-side reflection is deflected toward  $+x$  (*right* when viewed from the right).

### Usage notes

- For small tilts the resulting lateral walk-off is modelled by the phase masks; larger tilts can be combined with `.project_according_to_angles` and `.consider_tilt_astigmatism` to include projection and astigmatism effects.
- Use the degree-based setter `.rot_around_y_deg` if working in degrees is more convenient.

## `.rot_around_y_deg`

### Read/write property.

Specifies the mirror tilt angle about the y-axis in *degrees*.

### Behavior

- Setting `.rot_around_y_deg` automatically updates the radian-based property `.rot_around_y`.
- The new tilt invalidates cached tilt masks (`.tilt_mask_x_L`, `.tilt_mask_x_R`) and any transfer matrices that depend on them; the cavity's cached matrices are cleared as well.
- Allowed range:  $|\alpha| < 45^\circ$ . Values outside this range are rejected with an error message.

### Sign convention

A *positive* angle tilts the mirror such that the left-side reflection is deflected toward  $-x$  (*left* when viewed from the right) and the right-side reflection toward  $+x$  (*right* when viewed from the right).

### Usage notes

- This is a convenience property for users working in degrees. Use the radian-based version `.rot_around_y` if needed for computation.
- Setting either `.rot_around_y` or `.rot_around_y_deg` is fully interchangeable; both properties update each other accordingly. The update mechanism ensures that no rounding errors are introduced for the angle in the unit that was used for setting.

## `.mirror_tilted`

### Read-only property.

Returns `True` if the mirror is tilted around either the x- or y-axis, i.e. if either `.rot_around_x` or `.rot_around_y` is nonzero.

### Usage notes

- This is a convenience property that simplifies conditional logic when e.g. deciding

whether projection effects or tilt masks should be considered.

- It is purely evaluative and does not trigger any side effects or calculations.

### `.tilt_mask_x_L`

#### **Read-only property.**

Returns the spatial phase mask applied to the *left-hand* reflection when the mirror is tilted about the y-axis.

#### **Definition of the mask**

For a mirror rotation angle  $\alpha_y = .rot\_around\_y$  (in radians) the left-side mask is

$$\tilde{M}_{x,L}(x, y) = \exp[i k_0 x \tan(-2\alpha_y)], \quad k_0 = \frac{2\pi}{\lambda},$$

with  $x$  taken from the total grid axis `grid.axis_tot`. The factor  $-2\alpha_y$  stems from geometric optics: the incident ray strikes the surface at an angle  $\alpha_y$  and is reflected through an additional  $\alpha_y$ , giving a total angular deflection of  $2\alpha_y$ ; the minus sign accounts for the fact that a positive  $\alpha_y$  deflects the left-hand reflection toward  $-x$ .

#### **Behavior**

- If  $\alpha_y = 0$ , or if `.apply_tilt_masks = False`, or if `.left_side_relevant = False`, the property returns the scalar value 1.
- Otherwise the mask is generated lazily on first access by `.calc_tilt_masks_x()` and cached for reuse.

#### **Usage notes**

- Multiplying a field by  $\tilde{M}_{x,L}$  in position space imparts the correct lateral phase ramp that steers the reflected beam according to the mirror tilt.
- The corresponding mask for the right-hand reflection is `.tilt_mask_x_R`.

### `.tilt_mask_x_R`

#### **Read-only property.**

Returns the spatial phase mask applied to the *right-hand* reflection when the mirror is tilted about the y-axis.

#### **Definition of the mask**

For a mirror rotation angle  $\alpha_y = .rot\_around\_y$  (in radians) the right-side mask is

$$\tilde{M}_{x,R}(x, y) = \exp[i k_0 x \tan(+2\alpha_y)], \quad k_0 = \frac{2\pi}{\lambda},$$

with  $x$  taken from the total grid axis `grid.axis_tot`. A positive  $\alpha_y$  deflects the right-hand reflection toward  $+x$ , hence the phase ramp carries  $+2\alpha_y$ .

#### **Behavior**

- If  $\alpha_y = 0$ , or if `.apply_tilt_masks = False`, or if `.right_side_relevant = False`, the property returns the scalar value 1.

- Otherwise the mask is generated lazily on first access by `.calc_tilt_masks_x()` and cached for reuse.

### Usage notes

- Multiplying a field by  $\tilde{M}_{x,R}$  in position space imparts the correct lateral phase ramp that steers the reflected beam according to the mirror tilt.
- The corresponding mask for the left-hand reflection is `.tilt_mask_x_L`.

### `.tilt_mask_y_L`

#### Read-only property.

Returns the spatial phase mask applied to the *left-hand* reflection when the mirror is tilted about the x-axis.

#### Definition of the mask

For a mirror rotation angle  $\alpha_x = .rot_around_x$  (in radians) the left-side mask is

$$\tilde{M}_{y,L}(x, y) = \exp[i k_0 y \tan(+2\alpha_x)], \quad k_0 = \frac{2\pi}{\lambda},$$

with  $y$  taken from the total grid axis `grid.axis_tot`. A positive  $\alpha_x$  deflects the left-hand reflection toward  $+y$ , hence the phase ramp carries  $+2\alpha_x$ .

#### Behavior

- If  $\alpha_x = 0$ , or if `.apply_tilt_masks = False`, or if `.left_side_relevant = False`, the property returns the scalar value 1.
- Otherwise the mask is generated lazily on first access by `.calc_tilt_masks_y()` and cached for reuse.

### Usage notes

- Multiplying a field by  $\tilde{M}_{y,L}$  in position space imparts the correct vertical phase ramp that steers the reflected beam according to the mirror tilt.
- The corresponding mask for the right-hand reflection is `.tilt_mask_y_R`.

### `.tilt_mask_y_R`

#### Read-only property.

Returns the spatial phase mask applied to the *right-hand* reflection when the mirror is tilted about the x-axis.

#### Definition of the mask

For a mirror rotation angle  $\alpha_x = .rot_around_x$  (in radians) the right-side mask is

$$\tilde{M}_{y,R}(x, y) = \exp[i k_0 y \tan(-2\alpha_x)], \quad k_0 = \frac{2\pi}{\lambda},$$

with  $y$  taken from the total grid axis `grid.axis_tot`. A positive  $\alpha_x$  deflects the right-hand reflection toward  $-y$ , hence the phase ramp carries  $-2\alpha_x$ .

#### Behavior

- If  $\alpha_x = 0$ , or if `.apply_tilt_masks = False`, or if `.right_side_relevant = False`, the property returns the scalar value 1.
- Otherwise the mask is generated lazily on first access by `.calc_tilt_masks_y()` and cached for reuse.

### Usage notes

- Multiplying a field by  $\tilde{M}_{y,R}$  in position space imparts the correct vertical phase ramp that steers the reflected beam according to the mirror tilt.
- The corresponding mask for the left-hand reflection is `.tilt_mask_y_L`.

## `.calc_tilt_masks_x()`

### Method.

Computes and stores the horizontal (x-direction) phase masks `.tilt_mask_x_L` and `.tilt_mask_x_R` used to simulate reflections from a mirror that is tilted about the y-axis.

### Definition

Let  $\alpha_y = .rot_around_y$  be the mirror tilt in radians. Then:

- For the *left-hand* reflection:

$$\tilde{M}_{x,L}(x, y) = \exp[i k_0 x \tan(-2\alpha_y)]$$

- For the *right-hand* reflection:

$$\tilde{M}_{x,R}(x, y) = \exp[i k_0 x \tan(+2\alpha_y)]$$

where  $k_0 = \frac{2\pi}{\lambda}$ , and  $x$  is taken from the simulation grid `grid.axis_tot`.

### Behavior

- If  $\alpha_y = 0$ , or if `.apply_tilt_masks = False`, both masks are set to the scalar 1.
- If `.left_side_relevant = False`, the left-hand mask is not computed.
- If `.right_side_relevant = False`, the right-hand mask is not computed.

### Usage notes

- This method is called automatically on first access to the respective mask properties. Manual calls are only needed to force re-computation.

## `.calc_tilt_masks_y()`

### Method.

Computes and stores the vertical (y-direction) phase masks `.tilt_mask_y_L` and `.tilt_mask_y_R`, which simulate reflections from a mirror that is tilted about the x-axis.

### Definition

Let  $\alpha_x = .rot_around_x$  be the mirror tilt in radians. Then:

- For the *left-hand* reflection:

$$\tilde{M}_{y,L}(x, y) = \exp[i k_0 y \tan(+2\alpha_x)]$$

- For the *right-hand* reflection:

$$\tilde{M}_{y,R}(x, y) = \exp[i k_0 y \tan(-2\alpha_x)]$$

where  $k_0 = \frac{2\pi}{\lambda}$ , and  $y$  is taken from the simulation grid `grid.axis_tot`.

### Behavior

- If  $\alpha_x = 0$ , or if `.apply_tilt_masks = False`, both masks are set to the scalar 1.
- If `.left_side_relevant = False`, the left-hand mask is not computed.
- If `.right_side_relevant = False`, the right-hand mask is not computed.

### Usage notes

- This method is called automatically on first access to the respective mask properties. Manual calls are only needed to force re-computation.

#### `.apply_tilt_masks`

##### Read/write property.

Determines whether tilt-dependent phase masks (`.tilt_mask_x_L`, `.tilt_mask_x_R`, `.tilt_mask_y_L`, `.tilt_mask_y_R`) are applied to simulate mirror tilts.

### Behavior

- When `True` (default): The mirror tilt angles (`.rot_around_x`, `.rot_around_y`) are interpreted as physically relevant and lead to phase masks being applied.
- When `False`: All tilt masks are deleted, and – when called – the methods `.calc_tilt_masks_x()` and `.calc_tilt_masks_y()` assign the scalar value 1 to the respective tilt masks instead of computing spatial phase masks. Hence the mirror behaves as if perfectly perpendicular to the beam axis.
- Disabling the masks immediately clears any cached tilt mask arrays (to free memory).
- Any change clears the cavity’s cached matrices, if the mirror belongs to a cavity.

## D.9.4 Projection and Astigmatism Parameters

Any optical field can be described as a superposition of many plane-wave modes, each with its own propagation direction. *Nevertheless, most beams possess a dominant or “principal” direction*, which is expressed by  $\alpha_y = .incident_angle_y$ . This angle specifies the macroscopic incidence angle between the beam axis and the `xz`-plane, i.e. the angle that determines how much the beam is displaced upward ( $+y$  for  $\alpha_y > 0$ ) or downward ( $-y$  for  $\alpha_y < 0$ ) as it propagates toward the mirror. The settings collected in this subsection govern how the optical field footprint is *projected* onto a (possibly tilted) mirror surface and how that footprint stretches or compresses as a result of incidence angle and mirror tilt, effects that can introduce astigmatism in tilted *curved* mirrors. Currently the library supports only a single

macroscopic incidence angle `.incident_angle_y`, which describes tilt in the  $y-z$  plane; an  $x-z$  counterpart is not yet implemented.

### Geometric projection

- **Incident field in the  $xy$ -plane.** While the following discussion applies to arbitrary light fields, we illustrate it using the example of a circular Gaussian beam. For  $\alpha_y \neq 0$ , such a beam appears as an ellipse in the laboratory  $xy$ -plane, *stretched* in the  $y$ -direction by the factor  $1/\cos \alpha_y$ . This stretch is intrinsic to the incident field and is present before the beam reaches the mirror.
- **Untilted mirror.** If the mirror surface is perpendicular to the  $z$ -axis (`.rot_around_x = 0`), the beam footprint is projected onto the mirror without further stretching; consequently `.get_projection_factor_y1(...)` returns 1.
- **Tilted mirror ( $\beta_x \neq 0$ ).** When the mirror is rotated about the  $x$ -axis by  $\beta_x = .rot_around_x$ , the  $xy$ -projected incident field is mapped onto the slanted mirror surface. Depending on the relative signs and magnitudes of  $\alpha_y$  and  $\beta_x$ , the elliptical footprint can stretch further, contract, or even return to a circle. The factor by which the  $y$ -extent of the  $xy$ -projected incident field is stretched or compressed is given by `.get_projection_factor_y1(...)`. If `.consider_tilt_astigmatism` is set to `True`, this calculation forms the basis for modelling astigmatic effects in curved mirrors.
- **Reflected footprint.** After specular reflection the field is re-projected onto the reference  $xy$ -plane; the final footprint is governed by the same two angles. The stretch/-compression in  $y$ -direction between incident field and reflected field is quantified by `.get_projection_factor_y2(...)`.

### `.incident_angle_y`

#### Read/write property.

Specifies the macroscopic incidence angle  $\alpha_y$  (in *radians*) between the beam axis and the  $xz$ -plane, i.e. the angle that determines how much the beam is displaced upward ( $+y$  for  $\alpha_y > 0$ ) or downward ( $-y$  for  $\alpha_y < 0$ ) as it propagates toward the mirror.

#### Behavior

- Setting `.incident_angle_y` automatically updates `.incident_angle_y_deg`.
- Allowed range:  $|\alpha_y| < \pi/2$ ; larger values are rejected.
- Changing the angle clears cached projection factors and any cavity matrices that depend on them.

#### Sign convention

A *positive*  $\alpha_y$  denotes a beam that arrives from below, tilted toward  $+y$ .

#### Usage notes

- Use `.incident_angle_y_deg` for degree-based input.
- Setting  $\alpha_y = 0$  corresponds to normal incidence.

### `.incident_angle_y_deg`

**Read/write property.**

Specifies the macroscopic incidence angle  $\alpha_y$  in *degrees*, measured between the beam axis and the  $xz$ -plane.

**Behavior**

- Setting `.incident_angle_y` automatically updates `.incident_angle_y_deg`.
- Allowed range:  $|\alpha_y| < 90^\circ$ ; larger values are rejected.
- Changing the angle clears cached projection factors and any cavity matrices that depend on them.

**Sign convention**

A *positive* value corresponds to a beam arriving from below, tilted toward  $+y$ .

**Usage notes**

Setting either `.incident_angle_y` or `.incident_angle_y_deg` is fully interchangeable; both properties update each other accordingly. The update mechanism ensures that no rounding errors are introduced for the angle in the unit that was used for setting.

**`.get_projection_factor_y1(side)`****Method.**

Returns the factor by which the *incident* field is stretched or compressed in the  $y$ -direction when it is projected from the laboratory  $xy$ -plane onto the (possibly tilted) mirror surface.

**Parameters**

- `side` – Which mirror surface is hit (`Side.LEFT` or `Side.RIGHT`).

**Formula**

Let

$$\alpha_y = .\text{incident\_angle\_y}, \quad \beta_x = .\text{rot\_around\_x},$$

and define

$$\beta_x^{(\text{side})} = \begin{cases} \beta_x & \text{for side = LEFT,} \\ -\beta_x & \text{for side = RIGHT.} \end{cases}$$

Then

$$\text{factor} = \frac{\cos \alpha_y}{\cos(\beta_x^{(\text{side})} + \alpha_y)}.$$

- `factor > 1`: footprint is stretched in  $y$ .
- `factor < 1`: footprint is compressed in  $y$ .
- `factor = 1`: no change (e.g. untilted mirror with  $\beta_x = 0$ ).

**Usage notes**

- The value is used internally when `.project_according_to_angles` is `True`.
- When `.consider_tilt_astigmatism` is `True`, the same factor is applied to the mirror's phase masks so that astigmatic effects are modelled consistently.

**`.get_projection_factor_y2(side)`****Method.**

Returns the factor by which the *reflected* field is stretched or compressed in the  $y$ -direction when it is projected back from the mirror surface to the laboratory  $xy$ -plane.

**Parameters**

- `side` – Reflection side (`Side.LEFT` or `Side.RIGHT`).

**Formula**

With

$$\alpha_y = .\text{incident\_angle\_y}, \quad \beta_x = .\text{rot\_around\_x},$$

define

$$\beta_x^{(\text{side})} = \begin{cases} \beta_x & \text{for side = LEFT,} \\ -\beta_x & \text{for side = RIGHT.} \end{cases}$$

The projection factor for the reflected field is then

$$\text{factor} = \frac{\cos \alpha_y}{\cos(2\beta_x^{(\text{side})} + \alpha_y)}.$$

- factor  $> 1$ : reflected footprint is stretched in  $y$ .
- factor  $< 1$ : reflected footprint is compressed in  $y$ .
- factor  $= 1$ : no change (e.g. mirror untilted or special angle combination).

**Usage notes**

- Applied automatically when `.project_according_to_angles` is `True`; the reflected field array is resampled accordingly.
- Together with `.get_projection_factor_y1(...)`, this factor enables consistent modelling of astigmatism in tilted curved mirrors when `.consider_tilt_astigmatism` is `True`.

**`.project_according_to_angles`****Read/write property.**

Controls whether the *reflected output field* is stretched or compressed in the  $y$ -direction according to the geometric projection factors dictated by the incidence angle  $\alpha_y = .\text{incident\_angle\_y}$  and the mirror tilt  $\beta_x = .\text{rot\_around\_x}$ .

**Behavior**

- `False` (default): The output field retains its original extent in  $y$ ; no resampling is performed.
- `True`: Before the field is returned to the simulation grid it is resampled in  $y$  by the factor provided by `.get_projection_factor_y2(...)`.

### Implementation details

- When disabled, `.left_refl_size_adjust` and `.right_refl_size_adjust` both return `False`.
- Toggling this flag clears cached transfer matrices and any cavity caches so that the new behaviour is taken into account.

#### `.left_refl_size_adjust`

##### Read-only property.

Indicates whether the *reflected* field exiting the mirror on the `left` side will be resampled (stretched or compressed) in the *y*-direction.

##### Behavior

- Returns `True` if both `.project_according_to_angles` is `True` and `.get_projection_factor_y2(...)`  $\neq 1$  when called with `Side.LEFT`.
- Otherwise returns `False`.

##### Implications

When `True`, the library enlarges or shrinks the left-hand reflected field array by the projection factor in *y*-direction.

#### `.right_refl_size_adjust`

##### Read-only property.

Indicates whether the *reflected* field exiting the mirror on the `right` side will be resampled (stretched or compressed) in the *y*-direction.

##### Behavior

- Returns `True` if both `.project_according_to_angles` is `True` and `.get_projection_factor_y2(...)`  $\neq 1$  when called with `Side.RIGHT`.
- Otherwise returns `False`.

##### Implications

When `True`, the library enlarges or shrinks the right-hand reflected field array by the projection factor in *y*-direction.

#### `.consider_tilt_astigmatism`

##### Read/write property.

Enables or disables compensation for *astigmatic stretching* of the mirror's phase masks that arises when a tilted mirror is illuminated at a finite incidence angle.

##### Behavior

- **Default:** `False`. The reflection-phase masks are used "as is"; no stretch/compression is applied, so the mirror behaves as if its curvature were identical in both transverse directions.
- `True`. Before the masks are applied, their *y*-extent is multiplied by the pro-

jection factor returned by `.get_projection_factor_y1(...)` (`Side.LEFT` or `Side.RIGHT`, depending on the port). Consequently, `.left_mask_adjust` or `.right_mask_adjust` returns `True` whenever that factor differs from unity.

### Purpose

Activating this flag allows curved mirrors that are simultaneously tilted and hit at an angle to reproduce tilt-induced astigmatism (different effective curvature in the two orthogonal planes). When the flag is off, the mirror's curvature is treated as isotropic regardless of projection geometry.

#### `.left_mask_adjust`

##### **Read-only property.**

Indicates whether the phase mask that acts on the `left`-hand reflection (or on LTR transmission when the mirror operates in “transmission behaves like reflection” mode) must be stretched or compressed in the *y*-direction to account for tilt–astigmatism projection.

##### **Behavior**

- Returns `True` precisely when
  1. `.consider_tilt_astigmatism` is `True`, and
  2. `.get_projection_factor_y1(...)` (`Side.LEFT`)  $\neq 1$ .
- Otherwise returns `False`.

##### **Implications**

When `True`, the library rescales the left-hand reflection mask by the corresponding projection factor before it is applied, ensuring that tilt-induced astigmatism of a curved mirror is modelled correctly.

#### `.right_mask_adjust`

##### **Read-only property.**

Indicates whether the phase mask that acts on the `right`-hand reflection (or on RTL transmission when the mirror operates in “transmission behaves like reflection” mode) must be stretched or compressed in the *y*-direction to account for tilt–astigmatism projection.

##### **Behavior**

- Returns `True` precisely when
  1. `.consider_tilt_astigmatism` is `True`, and
  2. `.get_projection_factor_y1(...)` (`Side.RIGHT`)  $\neq 1$ .
- Otherwise returns `False`.

##### **Implications**

When `True`, the library rescales the right-hand reflection mask by the corresponding projection factor before it is applied, ensuring that tilt-induced astigmatism of a curved mirror is modelled correctly.

### D.9.5 Transmission-Behaves-Like-Reflection Mode

The class supports a special mode in which the *transmission* in one propagation direction (`Dir.LTR` or `Dir.RTL`) mimics the *reflection* from either the left or right mirror surface (controlled via `.LTR_transm_behaves_like_refl_left`, `.LTR_transm_behaves_like_refl_right`, `.RTL_transm_behaves_like_refl_left`, and `.RTL_transm_behaves_like_refl_right`). In the opposite propagation direction the transmission remains *neutral*, i.e. the field passes unchanged. This mechanism lets a linear two-port chain emulate the optical path of a unidirectional cavity, such as a ring or bow-tie resonator.

#### Example (five-mirror bow-tie loop).

A beam meets the mirrors  $R_1 \rightarrow R_2 \rightarrow R_3 \rightarrow R_4 \rightarrow R_5$  before returning to  $R_1$ , with propagation segments  $P$  in between. We can model this loop with five mirror components  $C_1, \dots, C_5$  placed linearly:

$$C_1 \ C_5 \ P \ C_2 \ C_4 \ P \ C_3.$$

- $C_1$  and  $C_2$  emulate reflections at  $R_1$  and  $R_2$  by setting their LTR transmission to behave like reflection from the appropriate surface.
- $C_3$  provides the real reflection of  $R_3$ .
- $C_4$  and  $C_5$  emulate reflections at  $R_4$  and  $R_5$  by setting their RTL transmission to behave like reflection from the appropriate surface.

Thus the unidirectional five-mirror path is captured by a straight two-port chain – avoiding the complexity of a full four-port formalism while preserving the correct sequence of reflections.

#### `.LTR_transm_behaves_like_refl_left`

##### Read/write property.

When `True`, the **Left–To–Right** transmission of the mirror is replaced by the mirror's own *left-hand reflection* behaviour, while the genuine left reflection is suppressed ( $r_L = 0$ ).

##### Behavior

- Setting this flag to `True` ...
  1. forces `.left_side_relevant` to `True` (if it was `False`),
  2. sets the property `.LTR_transm_behaves_like_refl_right` and the property `.LTR_transm_behaves_neutral` to `False`,
  3. clears the cavity cache so the new scattering behaviour is reflected in subsequent calculations.
- Switching the flag from `True` to `False` restores normal LTR transmission.

#### `.LTR_transm_behaves_like_refl_right`

##### Read/write property.

When `True`, the **Left–To–Right** transmission of the mirror is replaced by the mirror's own *right-hand reflection* behaviour, while the genuine left reflection is suppressed ( $r_L = 0$ ).

### Behavior

- Setting this flag to True...
  1. forces `.right_side_relevant` to True (if it was False),
  2. sets the property `.LTR_transm_behaves_like_refl_left` and the property `.LTR_transm_behaves_neutral` to False,
  3. clears the cavity cache so the new scattering behaviour is reflected in subsequent calculations.
- Switching the flag from True to False restores normal LTR transmission.

#### `.LTR_transm_behaves_neutral`

##### Read/write property.

When True, the Left–To–Right transmission becomes *neutral*, i.e. the light field passes through the mirror unchanged. In this case, the mirror’s genuine left-hand reflection is suppressed ( $r_L = 0$ ).

### Behavior

- Setting this flag to True automatically sets `.LTR_transm_behaves_like_refl_left` and `.LTR_transm_behaves_like_refl_right` to False.
- The cavity cache is cleared so the updated transmission behavior takes effect in subsequent calculations.

#### `.RTL_transm_behaves_like_refl_left`

##### Read/write property.

When True, the Right–To–Left transmission of the mirror is replaced by the mirror’s own *left-hand reflection* behaviour, while the genuine right reflection is suppressed ( $r_R = 0$ ).

### Behavior

- Setting this flag to True...
  1. forces `.left_side_relevant` to True (if it was False),
  2. sets the property `.RTL_transm_behaves_like_refl_right` and the property `.RTL_transm_behaves_neutral` to False,
  3. clears the cavity cache so the new scattering behaviour is reflected in subsequent calculations.
- Switching the flag from True to False restores normal RTL transmission.

#### `.RTL_transm_behaves_like_refl_right`

##### Read/write property.

When True, the Right–To–Left transmission of the mirror is replaced by the mirror’s own *right-hand reflection* behaviour, while the genuine right reflection is suppressed ( $r_R = 0$ ).

**Behavior**

- Setting this flag to True ...
  1. forces `.right_side_relevant` to True (if it was False),
  2. sets the property `.RTL_transm_behaves_like_refl_left` and the property `.RTL_transm_behaves_neutral` to False,
  3. clears the cavity cache so the new scattering behaviour is reflected in subsequent calculations.
- Switching the flag from True to False restores normal RTL transmission.

**`.RTL_transm_behaves_neutral`****Read/write property.**

When True, the Right–To–Left transmission becomes *neutral*, i.e. the light field passes through the mirror unchanged. In this case, the mirror’s genuine right-hand reflection is suppressed ( $r_R = 0$ ).

**Behavior**

- Setting this flag to True automatically sets `.RTL_transm_behaves_like_refl_left` and `.RTL_transm_behaves_like_refl_right` to False.
- The cavity cache is cleared so the updated transmission behavior takes effect in subsequent calculations.

## D.9.6 Memory Management

**`.clear_mem_cache()`****Method.**

Clears all internal memory-cached data related to this mirror component, including inherited caches and any precomputed tilt phase masks.

**Behavior**

- Calls the parent method `clsOptComponent.clear_mem_cache()` to clear the transfer matrix and inverse transfer matrix cache.
- Deletes the following tilt phase masks (if present):
  - `.tilt_mask_x_L`, `.tilt_mask_x_R`
  - `.tilt_mask_y_L`, `.tilt_mask_y_R`

**Usage notes**

- This method is typically called internally when a property is changed that affects the mirror’s scattering behavior.
- May be called manually if the user wants to free memory or force re-computation of the tilt masks.

## D.10 Class `clsMirror`

The class `clsMirror` represents an *ideal, flat, thin* mirror with arbitrary power reflectivity and transmissivity. It derives from `clsMirrorBase2port`, which in turn derives from `clsOptComponent2port`. Consequently, `clsMirror` inherits every method and property documented for those base classes – most notably the full set of reflection/transmission coefficients (`.R`, `.T`, `.r_L`, `.t_LTR`, *etc.*) and the rich machinery for mirror tilt, incidence angle, projection factors, and “transmission-behaves-like-reflection” modes.

The present subclass adds only what is specific to a *flat* mirror:

- Efficient helpers to build the left and right reflection matrices that include optional tilt masks and projection/stretch factors.
- Fast-path symmetry and  $k$ -space preferences that bypass heavy calculations when the mirror is untilted.
- Cache management tuned for the flat-mirror case.

All higher-level behaviour – tilt-induced astigmatism, side relevance flags, or mapping unidirectional cavities onto a linear two-port sequence – operates exactly as described for the base class and therefore needs no further repetition here.

### D.10.1 Initialization

`clsMirror(name, cavity)`

#### Constructor.

Initialises a new instance of `clsMirror`, a flat, thin partially-reflective mirror.

#### Behaviour on construction

- Calls the parent constructor of `clsMirrorBase2port`, which sets up all generic two-port mirror parameters (reflection and transmission coefficients, tilt angles, etc.).
- Creates the private caches which will hold the left- and right-hand reflection matrices when tilt or projection effects are requested.

#### Typical use

Construct the mirror, set its power reflectivity (`.R`) or transmissivity (`.T`), adjust any desired tilt or incidence angles, and insert it into a cavity chain. For further details see the constructor descriptions of the root classes `clsMirrorBase2port` and `clsOptComponent2port`.

### D.10.2 Reflection

`.reflect(E_in, k_space_in, k_space_out, side)`

#### Method.

Computes the reflected field for a given, specific incident field `E_in`. The method `.reflect` serves as the *conceptual counterpart* to `.prop(...)`; however, unlike

.prop(...), it is *not* part of the standard interface imposed by the root class `clsOptComponent2port`, and not all two-port components implement it.

### Parameters

- `E_in`  
Input field. A NumPy array representing the field either in position space or  $k$ -space, as specified by `k_space_in`.
- `k_space_in`  
`True` if `E_in` is provided in  $k$ -space, `False` if it is in position space.
- `k_space_out`  
`True` if the returned field should be in  $k$ -space, `False` for position space.
- `side`  
Reflection side (`Side.LEFT` or `Side.RIGHT`).

### Behavior

- *Transmission-Behaves-Like-Reflection mode* (see Subsection D.9.5). The method returns the scalar 0 whenever reflection on the chosen side is *suppressed* because the corresponding transmission is configured to imitate a reflection:
  - `side = Side.LEFT` ⇒ zero is returned if
 

```
.LTR_transm_behaves_like_refl_left == True    or
LTR_transm_behaves_like_refl_right == True    or
LTR_transm_behaves_neutral == True.
```
  - `side = Side.RIGHT` ⇒ zero is returned if
 

```
.RTL_transm_behaves_like_refl_left == True    or
RTL_transm_behaves_like_refl_right == True    or
RTL_transm_behaves_neutral == True.
```
- *Normal reflection*. If none of the above flags are active for the chosen side, the method:
  - multiplies the field by `.r_L` or `.r_R`;
  - applies tilt masks in  $x$  and/or  $y$  if the mirror is rotated;
  - stretches or compresses the field in  $y$  according to the projection factor returned by `.get_projection_factor_y2(...)` when the property `.project_according_to_angles` is `True`;
  - converts between  $k$ -space and position space as required by `k_space_out`.

### `.R_L_mat_tot`

#### Read-only property.

Returns the total-resolution reflection matrix for an arbitrary light field incident from the `left` side of the mirror.

### Behavior

- *Transmission-Behaves-Like-Reflection mode* (see Subsection D.9.5.) If any of the following flags are `True`
  - `.LTR_transm_behaves_like_refl_left`,

- `.LTR_transm_behaves_like_refl_right,`
- `.LTR_transm_behaves_neutral`

then genuine left reflection is disabled and the property returns the scalar 0.

- Otherwise, if the mirror is untilted and no projection stretch is required (`.mirror_tilted = False` and `.left_refl_size_adjust = False`), the matrix collapses to the scalar reflection coefficient `.r_L`.
- Otherwise (if the mirror is tilt or projection is active), the full matrix is returned:
  1. If not already cached, it is built by `.calc_R_L_mat_tot()`, which
    - multiplies each Fourier-basis mode by the reflection coefficient `r_L`;
    - applies any required *x*- and *y*-tilt masks;
    - stretches or compresses the mode in *y*-direction by the projection factor `.get_projection_factor_y2(...)` when the property `.project_according_to_angles` is `True`;
    - converts the result back to modal coefficients and assembles the columns in parallel using `joblib`.
  2. The completed matrix is cached for subsequent calls.

### Usage notes

- Clearing any tilt, incidence angle, or side-relevance flag invalidates the cache so that the matrix is rebuilt the next time it is requested.
- For mirrors where left reflection is irrelevant (`.left_side_relevant = False`), this property simply returns `.r_L`.

### `.R_R_mat_tot`

#### Read-only property.

Returns the total-resolution reflection matrix for an arbitrary light field incident from the right side of the mirror.

#### Behavior

- *Transmission-Behaves-Like-Reflection mode* (see Subsection D.9.5.) If any of the following flags are `True`
  - `.RTL_transm_behaves_like_refl_left,`
  - `.RTL_transm_behaves_like_refl_right,`
  - `.RTL_transm_behaves_neutral`

then genuine right reflection is disabled and the property returns the scalar 0.

- Otherwise, if the mirror is untilted and no projection stretch is required (`.mirror_tilted = False` and `.right_refl_size_adjust = False`), the matrix collapses to the scalar reflection coefficient `.r_R`.
- Otherwise (if the mirror is tilt or projection is active), the full matrix is returned:
  1. If not already cached, it is built by `.calc_R_R_mat_tot()`, which
    - multiplies each Fourier-basis mode by the reflection coefficient `r_R`;
    - applies any required *x*- and *y*-tilt masks;

- stretches or compresses the mode in  $y$ -direction by the projection factor `.get_projection_factor_y2(...)` when the property `.project_according_to_angles` is `True`;
- converts the result back to modal coefficients and assembles the columns in parallel using `joblib`.

2. The completed matrix is cached for subsequent calls.

### Usage notes

- Clearing any tilt, incidence angle, or side-relevance flag invalidates the cache so that the matrix is rebuilt the next time it is requested.
- For mirrors where right reflection is irrelevant (`.right_side_relevant = False`), this property simply returns `.r_R`.

### `.calc_R_L_mat_tot()`

#### Method.

Explicitly (re)computes the left-side reflection matrix `.R_L_mat_tot`, accounting for mirror tilt and projection if applicable. This method is automatically invoked when the matrix is first requested and not already cached.

#### Behavior

- If `.left_side_relevant` is `False`, the method sets `R_L_mat_tot` to the scalar `.r_L` and returns.
- If the mirror is untilted and no projection adjustment is required (`.mirror_tilted = False`, `.left_refl_size_adjust = False`), the matrix collapses to the scalar `.r_L`.
- Otherwise:
  1. If tilt is active, the method ensures the required phase masks (`tilt_mask_x_L`, `tilt_mask_y_L`) are computed.
  2. If projection is active (`.project_according_to_angles = True` and `.left_refl_size_adjust = True`), it determines the vertical projection factor via `.get_projection_factor_y2(...)`.
  3. Each Fourier mode is:
    - initialized via `grid.fourier_basis_func()`,
    - multiplied by the reflection factor `r_L`,
    - multiplied by tilt masks (if applicable),
    - stretched/compressed in  $y$  (if applicable),
    - transformed back into modal coefficients.
  4. All matrix columns are computed in parallel using `joblib`.
  5. The full matrix is stored as an internal cache.
- If `.LTR_transm_behaves_like_refl_left` or related transmission-reflection flags are set, a note is printed explaining that this matrix governs the transmission behavior.

### Usage notes

- Calling this method directly is typically unnecessary; it is triggered automatically if the cached result is absent or invalidated.
- The cache is cleared whenever relevant parameters are changed.

#### `.calc_R_R_mat_tot()`

##### Method.

Explicitly (re)computes the right-side reflection matrix `.R_R_mat_tot`, accounting for mirror tilt and projection if applicable. This method is automatically invoked when the matrix is first requested and not already cached.

##### Behavior

- If `.right_side_relevant` is `False`, the method sets `R_R_mat_tot` to the scalar `.r_R` and returns.
- If the mirror is untilted and no projection adjustment is required (`.mirror_tilted = False`, `.right_refl_size_adjust = False`), the matrix collapses to the scalar `.r_R`.
- Otherwise:
  1. If tilt is active, the method ensures the required phase masks (`tilt_mask_x_R`, `tilt_mask_y_R`) are computed.
  2. If projection is active (`.project_according_to_angles = True` and `.right_refl_size_adjust = True`), it determines the vertical projection factor via `.get_projection_factor_y2(...)`.
  3. Each Fourier mode is:
    - initialized via `grid.fourier_basis_func()`,
    - multiplied by the reflection factor `r_R`,
    - multiplied by tilt masks (if applicable),
    - stretched/compressed in `y` (if applicable),
    - transformed back into modal coefficients.
  4. All matrix columns are computed in parallel using `joblib`.
  5. The full matrix is stored as an internal cache.
- If `.RTL_transm_behaves_like_refl_right` or related transmission-reflection flags are set, a note is printed explaining that this matrix governs the transmission behavior.

### Usage notes

- Calling this method directly is typically unnecessary; it is triggered automatically if the cached result is absent or invalidated.
- The cache is cleared whenever relevant parameters are changed.

### D.10.3 Transmission

```
.prop(E_in, k_space_in, k_space_out, direction)
```

#### Method.

Propagates a specific input light field `E_in` in the specified direction (`Dir.LTR` or `Dir.RTL`) through the mirror. This method is part of the standard interface of all two-port components (`clsOptComponent2port`).

#### Parameters

- `E_in` - Input field. A NumPy array representing the field either in position space or  $k$ -space, as specified by `k_space_in`.
- `k_space_in` - `True` if `E_in` is provided in  $k$ -space, `False` if it is in position space.
- `k_space_out` - `True` if the returned field should be in  $k$ -space, `False` for position space.
- `direction` - Propagation direction (`Dir.LTR` or `Dir.RTL`).

#### Behavior

- If one of the *Transmission-Behaves-Like-Reflection Mode* flags (see Subsec. D.9.5) is active for the given direction – specifically, if any of the following are `True`:
  - `.LTR_transm_behaves_like_refl_left`
  - `.LTR_transm_behaves_like_refl_right`
  - `.LTR_transm_behaves_neutral`
for `direction = LTR`, or
  - `.RTL_transm_behaves_like_refl_left`
  - `.RTL_transm_behaves_like_refl_right`
  - `.RTL_transm_behaves_neutral`
for `direction = RTL`, then propagation is interpreted as emulated reflection:
  - If reflection is emulated from the left or right, the corresponding reflection factor is applied.
  - If the mirror is tilted, the respective tilt masks are applied (in position space).
  - If `.project_according_to_angles` is `True`, the field is stretched or compressed in the  $y$ -direction accordingly.
- If the neutral mode is active, the input field is returned unchanged.
- If no such mode is active, the field is multiplied by the usual transmission factor: `.t_LTR` or `.t_RTL`.
- The field is converted to the desired output representation ( $k$ -space or position space) as needed.

#### Usage notes

- For most mirrors, this method is the standard way to describe their effect in a cavity propagation chain.
- The field is internally converted to position space if mirror tilt or projection adjustment is required.

**.T\_LTR\_mat\_tot****Read-only property.**

Returns the total-resolution transmission matrix for an arbitrary light field propagating from left to right (LTR) through the mirror.

**Behavior**

- If one of the following *Transmission-Behaves-Like-Reflection Mode* flags (see Sub-section D.9.5) is set:
  - `.LTR_transm_behaves_like_refl_left`
  - `.LTR_transm_behaves_like_refl_right`

the method returns the corresponding reflection matrix.

- If `.LTR_transm_behaves_neutral` is set, the matrix is the scalar 1, indicating that the field passes through unchanged.
- Otherwise, the transmission is described by the standard scalar transmission coefficient `.t_LTR`.

**.T\_RTL\_mat\_tot****Read-only property.**

Returns the total-resolution transmission matrix for an arbitrary light field propagating from right to left (RTL) through the mirror.

**Behavior**

- If one of the following *Transmission-Behaves-Like-Reflection Mode* flags (see Sub-section D.9.5) is set:
  - `.RTL_transm_behaves_like_refl_left`
  - `.RTL_transm_behaves_like_refl_right`

the method returns the corresponding reflection matrix.

- If `.RTL_transm_behaves_neutral` is set, the matrix is the scalar 1, indicating that the field passes through unchanged.
- Otherwise, the transmission is described by the standard scalar transmission coefficient `.t_RTL`.

#### D.10.4 Symmetry and Space Preferences

**.symmetric****Read-only property.**

Returns `True` if the mirror behaves identically for light incident from either side, i.e. when no asymmetry arises from geometric or numerical effects.

**Behavior**

- Returns `False` if any of the following are active:
  - `.mirror_tilted`

- `.left_refl_size_adjust`
- `.right_refl_size_adjust`
- Returns `True` only if all of the above are `False`.

### `.k_space_in_dont_care`

**Read-only property.**

Indicates whether the component is indifferent to the representation (position space or  $k$ -space) of the *input* field.

**Behavior**

This property returns `True` if the mirror is untilted, i.e. `.rot_around_x = 0` and `.rot_around_y = 0`. Otherwise, it returns `False` because the mirror tilt is implemented in position space, requiring conversion if the input is in  $k$ -space.

### `.k_space_in_prefer`

**Read-only property.**

Indicates whether the component *prefers* to receive input fields in  $k$ -space rather than in position space.

**Behavior**

This property always returns `False`, because certain operations (e.g. mirror tilt) require position-space representation internally.

### `.k_space_out_dont_care`

**Read-only property.**

Indicates whether the component is indifferent to the representation (position space or  $k$ -space) of the *output* field.

**Behavior**

This property returns `True` if the mirror is untilted, i.e. `.rot_around_x = 0` and `.rot_around_y = 0`. Otherwise, it returns `False` because the mirror tilt is implemented in position space, requiring conversion if the output is desired in  $k$ -space.

### `.k_space_out_prefer`

**Read-only property.**

Indicates whether the component *prefers* to return output fields in  $k$ -space rather than in position space.

**Behavior**

This property always returns `False`, because certain operations (e.g. mirror tilt) are implemented in position space and thus the natural representation for output is position space.

### D.10.5 Memory Management

`.clear_mem_cache()`

#### Method.

Frees all cached data specific to this flat mirror, ensuring that future computations use up-to-date parameters.

#### Behavior

Invokes the base implementation `.clear_mem_cache()` to delete *tilt masks* and other generic caches. Additionally clears the private reflection-matrix caches.

#### Usage notes

The method is called automatically whenever a change to mirror tilt, incidence angle, side-relevance flags, or reflection/transmission coefficients invalidates the cached matrices.

## D.11 Class `clsCurvedMirror`

The class `clsCurvedMirror` represents an ideal thin *curved* mirror (partially transparent or fully reflective) and derives from `clsMirrorBase2port`, itself a child of `clsOptComponent2port`. It therefore inherits all generic two-port functionality (reflectivity/transmissivity coefficients, mirror-tilt handling, “Transmission-Behaves-Like-Reflection” flags, projection/astigmatism logic, ...) and adds the following curved-mirror-specific features:

- **Independent curvature on each facet.** Separate focal lengths `.f_R_L` / `.f_R_R` (or their millimetre variants) allow convex, concave, or flat surfaces on either side. Helper methods `.is_convex(...)` and `.is_concave(...)` classify each facet.
- **Spherical vs. “perfect” (paraxially exact) phase profiles.** The shape selector `.mirror_type_spherical` / `.mirror_type_perfect` chooses between a quadratic phase mask (spherical mirror approximation) and the exact hyperbolic phase appropriate for a perfect focusing surface.
- **Lens action for transmitted light.** A partially transparent curved mirror behaves like a thin lens with finite absorption. When both facets participate in transmission, their curvatures together with the substrate refractive index `.n` determine an effective focal length `.f_T`, and the corresponding quadratic (or perfect) *lens mask* is applied to the passing field.
- **Consistent integration with mirror tilt/projection/astigmatism framework.** All curved-surface masks, tilt masks, and projection factors are combined in the correct sequence for both direct reflection and “reflection-via-transmission” emulation modes.

### D.11.1 Initialization

`clsCurvedMirror(name, cavity)`

#### Constructor.

Initialises a new `clsCurvedMirror` instance, i.e. a thin curved mirror component with independent curvatures on its two facets.

#### Default state

- Left-hand facet: `.f_R_L = 0.05 m` (concave, focusing).
- Right-hand facet: `.f_R_R = ∞` (optically flat).
- Substrate refractive index `.n = 1.5`.
- Mirror type: `.mirror_type_spherical = True` (spherical mirror).
- All phase-mask and scattering-matrix caches are initialised empty, so they will be created on demand.

#### Typical use

After construction, adjust the facet focal lengths (`.f_R_L`, `.f_R_R`), select the desired mirror type (`.mirror_type_perfect` or `.mirror_type_spherical`), and – if it is a partially reflective mirror and the transmission behavior matters in your simulation – optionally set a custom refractive index via `.n`. For further details see the constructor documentation of the parent classes `clsMirrorBase2port` and `clsOptComponent2port`.

### D.11.2 Curvature and Focal-Length Parameters

`.f_R_L`

#### Read/write property.

Focal length (in meters) of the mirror surface on the *left* facet.

#### Sign convention

Positive values denote *concave* mirrors (focusing), negative values denote *convex* mirrors (defocusing). A value of `math.inf` corresponds to a flat surface.

#### Behavior

Internally synchronized with `.f_R_L_mm`. Setting this property deletes all relevant cached matrices and phase masks. If the mirror is part of a cavity, the cavity's cache is also cleared via `cavity.clear()`.

**See also** `.f_R_L_mm`, `.radius_L`, `.is_concave(...)`, `.is_convex(...)`

`.f_R_L_mm`

#### Read/write property.

This convenience property sets and holds the focal length focal length (in millimeters) of the mirror surface on the *left* facet.

#### Sign convention

Positive values denote *concave* mirrors (focusing), negative values denote *convex* mir-

rors (defocusing). A value of `math.inf` corresponds to a flat surface.

### Behavior

Internally synchronized with `.f_R_L`. Setting this property deletes all relevant cached matrices and phase masks. If the mirror is part of a cavity, the cavity's cache is also cleared via `cavity.clear()`.

**See also** `.f_R_L`, `.radius_L`, `.is_concave(...)`, `.is_convex(...)`

## `.f_R_R`

### Read/write property.

Focal length (in meters) of the mirror surface on the *right* facet.

### Sign convention

Positive values denote *concave* mirrors (focusing), negative values denote *convex* mirrors (defocusing). A value of `math.inf` corresponds to a flat surface.

### Behavior

Internally synchronized with `.f_R_R_mm`. Setting this property deletes all relevant cached matrices and phase masks. If the mirror is part of a cavity, the cavity's cache is also cleared via `cavity.clear()`.

**See also** `.f_R_R_mm`, `.radius_R`, `.is_concave(...)`, `.is_convex(...)`

## `.f_R_R_mm`

### Read/write property.

This convenience property sets and holds the focal length (in millimeters) of the mirror surface on the *right* facet.

### Sign convention

Positive values denote *concave* mirrors (focusing), negative values denote *convex* mirrors (defocusing). A value of `math.inf` corresponds to a flat surface.

### Behavior

Internally synchronized with `.f_R_R`. Setting this property deletes all relevant cached matrices and phase masks. If the mirror is part of a cavity, the cavity's cache is also cleared via `cavity.clear()`.

**See also** `.f_R_R`, `.radius_R`, `.is_concave(...)`, `.is_convex(...)`

## `.radius_L`

### Read-only property.

Radius of curvature (in meters) of the mirror surface on the *left* facet.

### Sign convention

A positive radius denotes a *convex* surface (defocusing), a negative radius denotes a *concave* surface (focusing). A value of `math.inf` corresponds to a flat surface.

### Relation to focal length

Defined as `radius_L = -2 · f_R_L`.

**See also** `.f_R_L`, `.f_R_L_mm`, `.is_concave(...)`, `.is_convex(...)`

### `.radius_R`

**Read-only property.**

Radius of curvature (in meters) of the mirror surface on the *right* facet.

**Sign convention**

A positive radius denotes a *concave* surface (focusing), a negative radius denotes a *convex* surface (defocusing). A value of `math.inf` corresponds to a flat surface.

**Relation to focal length**

Defined as `radius_R = 2 · f_R_R`.

**See also** `.f_R_R`, `.f_R_R_mm`, `.is_concave(...)`, `.is_convex(...)`

### `.is_convex(side)`

**Method.**

Returns `True` if the specified mirror facet is convex, i.e. if its focal length is negative.

**Parameter**

`side` – Specifies the side to check: `Side.LEFT` or `Side.RIGHT`.

**Behavior**

- Returns `True` if the focal length is negative.
- Returns `False` otherwise (including for flat facets with `math.inf` focal length).

**See also** `.is_concave(...)`, `.f_R_L`, `.f_R_R`

### `.is_concave(side)`

**Method.**

Returns `True` if the specified mirror facet is concave, i.e. if its focal length is positive.

**Parameter**

`side` – Specifies the side to check: `Side.LEFT` or `Side.RIGHT`.

**Behavior**

- Returns `True` if the focal length is positive.
- Returns `False` otherwise (including for flat facets with `math.inf` focal length).

**See also** `.is_convex(...)`, `.f_R_L`, `.f_R_R`

### `.f_T`

**Read-only property.**

A semi-reflective, curved mirror behaves like a thin lens (with some absorption) for

transmitted light. This property returns the effective focal length  $f_T$  (in meters) of this lens, as determined by the curvature of both mirror facets and the refractive index  $n$ .

### Formula

$$f_T = \frac{1}{(1 - n) \left( \frac{1}{R_L} - \frac{1}{R_R} \right)}$$

where  $R_L$  and  $R_R$  are the radii of curvature of the left and right mirror facets, respectively, and  $n$  is the refractive index inside the component (see `.n`).

### See also

`.f_R_L`, `.f_R_R`, `.radius_L`, `.radius_R`, `.n`

## D.11.3 Mirror-Shape Selection

### `.mirror_type_spherical`

#### Read/write property.

Selects whether the mirror facets are treated as *spherical surfaces* or – alternatively – as *perfect (aspherical) surfaces*.

#### Behavior

`.mirror_type_spherical` holds a Boolean flag:

- **True** – Both facets are modeled as *spherical* surfaces; the reflection and transmission phase masks are calculated from the quadratic optical-path formula appropriate for a spherical mirror. See `.mirror_mask_L`, `.mirror_mask_R`, and `.lens_mask` for the exact expressions.
- **False** – Facets are modeled as *perfect* (aberration-free) surfaces; the complementary flag `.mirror_type_perfect` is set to **True**.

Setting this property automatically toggles `.mirror_type_perfect` to the opposite value.

#### When changed

All cached mirror masks, lens masks, and dependent matrices are discarded; if the mirror belongs to a cavity, the cavity cache is cleared as well.

See also `.mirror_type_perfect`, `.mirror_mask_L`, `.mirror_mask_R`, `.lens_mask`

### `.mirror_type_perfect`

#### Read/write property.

Selects whether the mirror facets are treated as *perfect aspherical surfaces* or – alternatively – as *spherical* surfaces.

#### Behavior

`.mirror_type_perfect` holds a Boolean flag:

- **True** – Both facets are modeled as *perfect mirrors* surfaces, i.e. the reflection and

transmission phase masks are computed from the exact optical path length of a wavefront originating from the focal point. See `.mirror_mask_L`, `.mirror_mask_R`, and `.lens_mask` for details.

- `False` – The facets are modeled as *spherical* surfaces; the complementary flag `.mirror_type_spherical` is set to `True`.

Setting this property automatically toggles `.mirror_type_spherical` to the opposite value.

#### When changed

All cached mirror masks, lens masks, and dependent matrices are discarded. If the mirror belongs to a cavity, the cavity cache is also cleared.

See also `.mirror_type_spherical`, `.mirror_mask_L`, `.mirror_mask_R`, `.lens_mask`

### D.11.4 Reflection

```
.reflect(E_in, k_space_in, k_space_out, side)
```

#### Method.

Computes the field reflected from one facet of the curved mirror component. This method is the conceptual counterpart to `.prop(...)`, but not all components implement it.

#### Parameters

- `E_in` – Input field. A NumPy array representing the incoming optical field, in either position or  $k$ -space.
- `k_space_in` – Boolean flag indicating whether `E_in` is provided in  $k$ -space (`True`) or in position space (`False`).
- `k_space_out` – Boolean flag indicating whether the output field should be returned in  $k$ -space (`True`) or in position space (`False`).
- `side` – Indicates which mirror facet the field is incident on; must be `Side.LEFT` or `Side.RIGHT`.

#### Behavior

- If the mirror is part of a cavity and any of the *Transmission-Behaves-Like-Reflection Mode* flags are active (see Subsection D.9.5), and the transmission in the current propagation direction is mimicking reflection from this side, the method returns zero.
- Otherwise, the field is reflected with:
  1. The correct reflection coefficient (`.r_L` or `.r_R`),
  2. A spatial phase modulation derived from the mirror's curvature and selected shape model (`.mirror_type_spherical` or `.mirror_type_perfect`),
  3. Optional tilt masks and projection factors (if configured in the base class),
  4. The result is transformed into the desired output representation.

**Usage notes**

- For efficient execution, internal matrix caches are used.
- The reflection behavior depends both on the geometric configuration (e.g. curvature, tilt) and on settings inherited from `clsMirrorBase2port(...)`.

**`.R_L_mat_tot`****Read-only property.**

Returns the total-resolution reflection matrix for an arbitrary field incident from the `left` facet of the `clsCurvedMirror`.

**Behavior**

- *Transmission-Behaves-Like-Reflection mode* (see Subsection D.9.5). If any of the flags
  - `.LTR_transm_behaves_like_refl_left`,
  - `.LTR_transm_behaves_like_refl_right`,
  - `.LTR_transm_behaves_neutral`
is `True`, genuine left reflection is disabled and the property returns the scalar 0.
- If `.left_side_relevant = False` the facet is declared irrelevant; the property simply returns the scalar coefficient `.r_L` (flat, untilted mirror assumption).
- If the facet is flat *and* no tilt or projection stretch is required (`.f_R_L = math.inf`, `.mirror_tilted = False`, `.left_refl_size_adjust = False`), the matrix again collapses to `.r_L`.
- Otherwise the full matrix is returned as calculated by `.calc_R_L_mat_tot()`. The finished matrix is cached for future use.

**Usage notes**

Altering curvature, tilt, projection settings, or side-relevance flags invalidates the cache; the matrix is regenerated on the next access. See `clsOptComponent2port.R_L_mat_tot` for general information on reflection matrices.

To understand the construction of the reflection mask, see `.mirror_mask_R`.

**`.R_R_mat_tot`****Read-only property.**

Returns the total-resolution reflection matrix for an arbitrary field incident from the `right` facet of `clsCurvedMirror`.

**Behavior**

- *Transmission-Behaves-Like-Reflection mode* (see Subsection D.9.5). If any of the flags
  - `.RTL_transm_behaves_like_refl_left`,
  - `.RTL_transm_behaves_like_refl_right`,
  - `.RTL_transm_behaves_neutral`

is `True`, genuine right-hand reflection is disabled and the property returns the scalar 0.

- If `.right_side_relevant = False`, the facet is declared irrelevant; the property simply returns the scalar coefficient `.r_R` (flat, untilted mirror assumption).
- If the facet is flat *and* no tilt or projection stretch is required (`.f_R_R = math.inf`, `.mirror_tilted = False`, `.right_refl_size_adjust = False`), the matrix again collapses to `.r_R`.
- Otherwise the full matrix is returned as calculated by `.calc_R_R_mat_tot()`. The finished matrix is cached for subsequent calls.

### Usage notes

Changing curvature, tilt, projection settings, or side-relevance flags invalidates the cache; the matrix is rebuilt upon the next access.

See `clsOptComponent2port.R_R_mat_tot` for general information on reflection matrices. To understand the construction of the reflection mask, see `.mirror_mask_R`.

### `.calc_R_L_mat_tot()`

#### Method.

(Re)computes the left-side total-resolution reflection matrix `.R_L_mat_tot` whenever it is not already cached or has been invalidated by parameter changes.

#### Behavior

1. *Trivial cases.*
  - If `.left_side_relevant = False` the facet is ignored and the matrix is set to the scalar reflection coefficient `.r_L`.
  - If the facet is flat and no mirror tilt or projection stretch is active (`f_R_L = math.inf`, `.mirror_tilted = False`, `.left_refl_size_adjust = False`), the matrix likewise reduces to `.r_L`.
2. *Full matrix computation.* Required whenever the surface is curved, the mirror is tilted, or projection stretch is enabled.
  - (a) If the surface is curved (`.f_R_L ≠ math.inf`), generate the curved-mirror phase mask via `.mirror_mask_L`.
  - (b) If the mirror is tilted, ensure the tilt masks `.tilt_mask_x_L` and `.tilt_mask_y_L` are available.
  - (c) For each Fourier basis mode  $(n_x, n_y)$ :
    - multiply by the scalar reflection coefficient `.r_L`;
    - multiply by the curved-mirror mask (if any);
    - apply the  $x$ - and  $y$ -tilt masks (if the corresponding tilt is non-zero);
    - **stretch in  $y$ -direction** when `.left_refl_size_adjust = True`, using the projection factor obtained from `.get_projection_factor_y2(...)` with parameter (`Side.LEFT`).
    - convert the result back to modal coefficients.
  - (d) Assemble all columns in parallel with `joblib` and cache the completed matrix

for future access.

If `.par_RT_calc` is `True` (default), the per-mode computations are executed in parallel using `joblib`; the number of worker processes is taken from `clsCavity.mp_pool_processes`.

### Cache invalidation

Changing curvature, tilt angles, incidence angle, projection settings, or side-relevance flags deletes the cached matrix; it is rebuilt on the next call.

## `.calc_RR_mat_tot()`

### Method.

(Re)computes the *right*-side total-resolution reflection matrix `.RR_mat_tot` whenever no valid cached version is available.

### Behavior

#### 1. Trivial cases.

- If `.right_side_relevant = False`, the facet is ignored and the matrix is set to the scalar reflection coefficient `.r_R`.
- If the facet is flat and no mirror tilt or projection stretch is active (`f_R_R = math.inf`, `.mirror_tilted = False`, `.right_refl_size_adjust = False`), the matrix likewise reduces to `.r_R`.

#### 2. Full matrix computation.

Executed when the surface is curved, the mirror is tilted, or projection stretch is required.

- (a) If the surface is curved ( $\text{f}_\text{R}_\text{R} \neq \text{math.inf}$ ), create the curved-mirror phase mask with `.mirror_mask_R`.
- (b) If the mirror is tilted, ensure the tilt masks `.tilt_mask_x_R` and `.tilt_mask_y_R` are available.
- (c) For each Fourier basis mode  $(n_x, n_y)$ :
  - multiply by the scalar reflection coefficient `.r_R`;
  - multiply by the curved-mirror mask (if present);
  - apply the  $x$ - and  $y$ -tilt masks (whenever the respective tilt is non-zero);
  - **stretch in  $y$ -direction** when `.right_refl_size_adjust = True`, using the projection factor from `.get_projection_factor_y2(...)` with parameter `(Side.RIGHT)`;
  - convert the result back to modal coefficients.
- (d) Assemble all columns in parallel via `joblib` and store the finished matrix in the internal cache.

If `.par_RT_calc` is `True` (default), the per-mode computations are executed in parallel using `joblib`; the number of worker processes is taken from `clsCavity.mp_pool_processes`.

### Cache invalidation

Any change to curvature, tilt angles, incidence angle, projection settings, or side-

relevance flags removes the cached matrix so it is recomputed on the next request.

#### **.mirror\_mask\_L**

##### **Read-only property.**

Delivers the complex *surface-phase* mask that is multiplied onto a field during reflection at the **left** facet. The mask accounts for

- the phase delay introduced by the curved mirror surface itself, and
- the astigmatic stretch/compression of the surface profile when the incident field is projected onto the mirror under finite incidence angle and/or mirror tilt (if `.left_mask_adjust = True`).

It *does not* include the subsequent stretch/compression of the *reflected* field nor the separate *x*- and *y*-tilt phase masks (`.tilt_mask_x_L`, `.tilt_mask_y_L`); those are applied later in the reflection pipeline.

##### **Behavior**

- The mask is lazily generated on first access and thereafter cached.
- If `.left_side_relevant = False`, the property immediately returns the scalar 1.
- Otherwise, if `.left_mask_adjust = True`, the *y*-axis of the total grid is scaled by the projection factor from `.get_projection_factor_y1(...)` (Side.LEFT) so that the phase profile is defined on the *projected* coordinates.
- The phase mask itself depends on the selected mirror-shape flag. With  $k_0 = 2\pi/\lambda$ :

##### **Spherical facet**

$$\exp\left[-i \frac{k_0}{2f_{R,L}} (x^2 + y^2)\right]$$

##### **Perfect (ideal) facet**

$$\exp\left[-i k_0 \left(\sqrt{f_{R,L}^2 + x^2 + y^2} - f_{R,L}\right)\right]$$

- Any subsequent request retrieves the cached mask unless it has been invalidated by changes to focal length, mirror-shape flags, projection settings, tilt, wavelength, or grid parameters.

##### **Cache invalidation**

Modifying focal-length properties, the shape flags (`.mirror_type_spherical` or `.mirror_type_perfect`), projection/tilt settings, or side-relevance flags erases the cached mask so it can be re-generated on the next access.

#### **.mirror\_mask\_R**

##### **Read-only property.**

Delivers the complex *surface-phase* mask that is multiplied onto a field during reflection at the **right** facet. The mask accounts for

- the phase delay introduced by the curved mirror surface itself, and

- the astigmatic stretch/compression of the surface profile when the incident field is projected onto the mirror under finite incidence angle and/or mirror tilt (if `.right_mask_adjust = True`).

It *does not* include the subsequent stretch/compression of the *reflected* field nor the separate  $x$ - and  $y$ -tilt phase masks (`.tilt_mask_x_R`, `.tilt_mask_y_R`); those are applied later in the reflection pipeline.

### Behavior

- The mask is lazily generated on first access and thereafter cached.
- If `.right_side_relevant = False`, the property immediately returns the scalar 1.
- Otherwise, if `.right_mask_adjust = True`, the  $y$ -axis of the total grid is scaled by the projection factor from `.get_projection_factor_y1(...)` (`Side.RIGHT`) so that the phase profile is defined on the *projected* coordinates.
- The phase mask itself depends on the selected mirror-shape flag. With  $k_0 = 2\pi/\lambda$ :

#### Spherical facet

$$\exp\left[-i \frac{k_0}{2f_{R,R}} (x^2 + y^2)\right]$$

#### Perfect (ideal) facet

$$\exp\left[-i k_0 \left(\sqrt{f_{R,R}^2 + x^2 + y^2} - f_{R,R}\right)\right]$$

- Any subsequent request retrieves the cached mask unless it has been invalidated by changes to focal length, mirror-shape flags, projection settings, tilt, wavelength, or grid parameters.

### Cache invalidation

Modifying focal-length properties, the shape flags (`.mirror_type_spherical` or `.mirror_type_perfect`), projection/tilt settings, or side-relevance flags erases the cached mask so it can be re-generated on the next access.

## D.11.5 Transmission

### `.n`

#### Read/write property.

A semi-reflective, curved mirror behaves like a thin lens (with some absorption) for transmitted light. The property `.n` stores the *real* refractive index  $n$  of the mirror for *transmitted* light. The default is  $n = 1.5$  (typical glass).

### Behavior

Reading `.n` returns the current value of `.n`. Writing a new value: If the mirror belongs to a cavity, the cavity cache is also cleared.

### Role in optical calculations

The value of `.n` enters the formula for the effective focal length of transmitted light, see `.f_T`.

### Usage notes

Keep  $n$  real; absorption and loss are specified via `.R` and `.T`.

```
.prop(E_in, k_space_in, k_space_out, direction)
```

### Method.

Computes the transmitted field when the input field `E_in` propagates through the mirror, either left-to-right (LTR) or right-to-left (RTL). This method implements the interface `.prop(...)` defined by the base class `clsOptComponent2port`.

### Parameters

- `E_in` – Input field. A NumPy array representing the field, either in position space or  $k$ -space.
- `k_space_in` – `True` if `E_in` is given in  $k$ -space; `False` if in position space.
- `k_space_out` – `True` if the returned field should be in  $k$ -space; `False` for position space.
- `direction` – Direction of propagation: `Dir.LTR` or `Dir.RTL`.

### Behavior

- *Transmission-Behaves-Like-Reflection Mode* (see Subsection D.9.5) – If one of the following flags is `True`, the transmission path imitates mirror reflection:
  - `.LTR_transm_behaves_like_refl_left` (for LTR),
  - `.LTR_transm_behaves_like_refl_right` (for LTR),
  - `.RTL_transm_behaves_like_refl_left` (for RTL),
  - `.RTL_transm_behaves_like_refl_right` (for RTL).

In these cases, the corresponding reflection coefficient is applied (including tilt and projection effects), and the reflection side determines whether left or right tilt masks are used.

- If the corresponding neutral-transmission flag (`.LTR_transm_behaves_neutral` or `.RTL_transm_behaves_neutral`) is `True`, the input field passes through unchanged.
- Otherwise (normal transmission), the `.lens_mask` is applied, which includes the effect of the curved mirror surfaces, and the according transmission coefficient `.t_LTR` or `.t_RTL`. Considerations of astigmatism effects because of mirror tilt or incident angle are currently *not* considered for transmission (to be included in a future version).

```
.T_LTR.mat_tot
```

### Read-only property.

Returns the total-resolution transmission matrix for an arbitrary light field propagating from left to right (LTR) through the curved mirror.

### Behavior

- *Transmission-Behaves-Like-Reflection Mode* (see Subsection D.9.5): If one of the following flags is set:

- `.LTR_transm_behaves_like_refl_left`
- `.LTR_transm_behaves_like_refl_right`

the method returns the corresponding reflection matrix.

- If `.LTR_transm_behaves_neutral` is set, the matrix is the scalar 1, indicating that the field passes through unchanged.
- Otherwise:
  - If the mirror has infinite effective focal length (`.f_T = math.inf`), the mirror does not act as a lens; the matrix collapses to the scalar transmission coefficient `.t_LTR`.
  - If the right side is irrelevant (`.right_side_relevant = False`), the method again returns `.t_LTR`.
  - Otherwise, the full transmission matrix is computed by `.calc_T_mat_tot()` and cached for reuse. This method constructs the transmission matrix by simulating how each plane-wave basis mode is transformed by the lens phase mask (property `.lens_mask`).

### `.T_RTL_mat_tot`

#### Read-only property.

Returns the total-resolution transmission matrix for an arbitrary light field propagating from right to left (RTL) through the curved mirror.

#### Behavior

- *Transmission-Behaves-Like-Reflection Mode* (see Subsection D.9.5): If one of the following flags is set:
  - `.RTL_transm_behaves_like_refl_left`
  - `.RTL_transm_behaves_like_refl_right`

the method returns the corresponding reflection matrix.

- If `.RTL_transm_behaves_neutral` is set, the matrix is the scalar 1, indicating that the field passes through unchanged.
- Otherwise:
  - If the mirror has infinite effective focal length (`.f_T = math.inf`), the mirror does not act as a lens; the matrix collapses to the scalar transmission coefficient `.t_RTL`.
  - If the left side is irrelevant (`.left_side_relevant = False`), the method again returns `.t_RTL`.
  - Otherwise, the full transmission matrix is computed by `.calc_T_mat_tot()` and cached for reuse. This method constructs the transmission matrix by simulating how each plane-wave basis mode is transformed by the lens phase mask (property `.lens_mask`).

### `.calc_T_mat_tot()`

#### Method.

Generates the transmission matrix  $\mathbf{T}$  at total resolution for the current mirror geometry and stores it in memory.

#### Behavior

This method constructs the transmission matrix by simulating how each plane-wave basis mode is transformed by the curved mirror's lens phase mask (property `.lens_mask`):

- Each basis mode is scaled by the transmission coefficient  $.t_{LTR} = .t_{RTL}$ .
- If both sides are relevant, the mode is additionally multiplied by the lens mask.
- The resulting field is converted back to modal coefficients.
- All columns of the matrix are assembled in this way.

If `.par_RT_calc` is `True`, the matrix is computed in parallel. The number of parallel processes is set by `clsCavity.mp_pool_processes`.

#### Usage notes

- This method is automatically invoked the first time `.T_LTR_mat_tot` or `.T RTL_mat_tot` is accessed, if no cached matrix is available.
- Manual calls are typically unnecessary.
- Since transmission is symmetric in this model, the same matrix is calculated for both propagation directions: `T_LTR_mat_tot = T_RTL_mat_tot`.

### `.lens_mask`

#### Read-only property.

Returns the phase mask that implements the lensing action of the mirror on transmitted light. The mask is automatically computed the first time it is accessed.

#### Behavior

- If the effective focal length `.f_T` is infinite, the mirror does not act as a lens, and the mask is simply the scalar 1.
- Otherwise, the method generates a 2D complex phase mask that imparts the correct optical path difference across the aperture.
- The wavenumber is computed as  $k_0 = \frac{2\pi}{\lambda}$ , where  $\lambda$  is the internal wavelength in the component medium.
- The specific form of the mask depends on the mirror shape:
  - **Spherical mirror:**

$$\text{mask}(x, y) = \exp \left( -i \frac{k_0}{2f_T} (x^2 + y^2) \right)$$

- **Perfect aspherical mirror:**

$$\text{mask}(x, y) = \exp \left( -i k_0 \left( \sqrt{f_T^2 + x^2 + y^2} - f_T \right) \right)$$

- If one of the sides is irrelevant (i.e. `.left_side_relevant` or `.right_side_relevant` is `False`), the lensing behavior is ignored and the mask is set to 1.

#### Note

At present, the phase mask does *not* account for mirror tilt. This limitation will be addressed in a future version.

### D.11.6 Symmetry and Space Preferences

#### `.symmetric`

##### **Read-only property.**

Returns `True` if the mirror behaves identically for light incident from either side, i.e. when no asymmetry arises from geometric or numerical effects.

##### **Behavior**

- Returns `False` if any of the following are active:
  - `.mirror_tilted`
  - `.left_refl_size_adjust`
  - `.right_refl_size_adjust`
  - asymmetry in curvature: `.radius_L ≠ .radius_R`
- Returns `True` only if all of the above conditions are false.

#### `.k_space_in_dont_care`

##### **Read-only property.**

Indicates whether the component is indifferent to the representation (position space or *k*-space) of the *input* field.

##### **Behavior**

This property always returns `False`, since internal computations – such as the application of the lens mask for transmitted light – require the input field in position space.

#### `.k_space_in_prefer`

##### **Read-only property.**

Indicates whether the component *prefers* to receive input fields in *k*-space rather than in position space.

##### **Behavior**

This property always returns `False`, because curved mirror operations – such as transmission through the lens mask – are defined in position space.

**.k\_space\_out\_dont\_care****Read-only property.**

Indicates whether the component is indifferent to the representation (position space or  $k$ -space) of the *output* field.

**Behavior**

This property always returns `False`, since the output field is generated via operations in position space and must be converted explicitly if needed in  $k$ -space.

**.k\_space\_out\_prefer****Read-only property.**

Indicates whether the component *prefers* to return output fields in  $k$ -space rather than in position space.

**Behavior**

This property always returns `False`, because the natural output of internal computations – such as those involving lens masks or reflection matrices – is in position space.

### D.11.7 Memory Management and Other Settings

**.clear\_mem\_cache()****Method.**

Clears all cached numerical data from memory that is specific to the curved mirror's optical behavior. This method extends `.clear_mem_cache()` and is automatically called whenever the mirror geometry or any relevant parameter (e.g. curvature or refractive index) is modified.

**Behavior**

In addition to clearing the base-class memory cache, this method:

- Deletes the cached lens phase mask (`.lens_mask`),
- Deletes the mirror surface phase masks (left and right),
- Deletes the cached reflection matrices,
- Deletes the cached transmission matrix.

**Usage note**

This method is automatically invoked when relevant properties are changed, but may also be called manually.

**.par\_RT\_calc****Read/write property.**

Controls whether the reflection and transmission matrices are calculated in parallel.

**Behavior**

If `.par_RT_calc` is set to `True` (default), the methods `.calc_R_R_mat_tot()`,

`.calc_R_L_mat_tot()`, and `.calc_T_mat_tot()` process the individual plane wave modes in parallel using `joblib`. The number of parallel worker processes is taken from `clsCavity.mp_pool_processes`.

If set to `False`, the matrix calculations are performed in a single-threaded, serial manner. This is useful for debugging or in special environments where parallel processing is undesirable.

## D.12 Class `clsSplitMirror`

The class `clsSplitMirror` represents a *flat, thin* mirror whose optical behaviour is **different above and below the horizontal mid-plane** ( $y = 0$ ). The upper half of the aperture is characterised by the power coefficients ( $R_1, T_1$ ) and the lower half by ( $R_2, T_2$ ); the corresponding complex amplitude-coefficients ( $r_1, t_1$ ) and ( $r_2, t_2$ ) are generated automatically according to the chosen phase convention.

### Inheritance.

At present the class derives directly from `clsOptComponent2port` (instead of `clsMirrorBase2port`); unification with the common mirror base class is planned for a future release. Despite this interim lineage, `clsSplitMirror` implements the same matrix interface ( $\mathbf{R}_L, \mathbf{R}_R, \mathbf{T}$ ) and tilt functionality as the other mirror components.

### Key features

- Independent reflectivity / transmissivity in the *upper* ( $y > 0$ ) and *lower* ( $y < 0$ ) half-planes, set via `.R1`, `.T1`, `.R2`, `.T2`.
- Separate complex amplitude coefficients ( $r_1, t_1$ ) / ( $r_2, t_2$ ) obeying either a *symmetric-phase* or *text-book real* convention; switchable with `.set_phys_behavior(...)`.
- Optional rotation about the  $x$ - and  $y$ -axes (`.rot_around_x_deg`, `.rot_around_y_deg`), with automatic phase-ramp masks applied separately to left and right reflections.
- “Side relevance” flags (`.left_side_relevant`, `.right_side_relevant`) to skip the expensive computation of reflection matrices that are not required in a given cavity configuration.
- Full transfer-matrix support: `.R_L_mat_tot`, `.R_R_mat_tot`, and `.T_LTR_mat_tot` = `.TRTL_mat_tot` are all cached for efficient reuse.

### D.12.1 Initialization

`clsSplitMirror(name, cavity)`

#### Constructor.

Creates a new `clsSplitMirror` instance and registers it with a `clsCavity` object.

#### Parameters

- `name` – Human-readable identifier for the component (string).
- `cavity` – Reference to the owning `clsCavity`; may be `None` for stand-alone use.

### Default state

- Symmetric-phase convention enabled (`.sym_phase = True`).
- Upper half:  $R_1 = 1, T_1 = 0$ ; lower half:  $R_2 = 1, T_2 = 0$  (perfect reflection on both halves).
- No mirror tilt (`.rot_around_x_deg, .rot_around_y_deg = 0°`).
- All tilt masks, reflection and transmission matrices are “not yet available”; they will be built lazily on first access.

### Typical use

1. Instantiate the mirror and insert it into the cavity sequence.
2. Set the desired power coefficients with `.R1/.T1` and `.R2/.T2`.
3. Optionally adjust the phase convention via `.set_phys_behavior(...)` and add tilt angles using `.rot_around_x_deg / .rot_around_y_deg`.

For the constructor semantics of the underlying base class, see `clsOptComponent2port`.

## D.12.2 Reflectivity and Transmissivity Parameters

### `.R1`

#### Read/write property.

Sets or queries the *power reflectivity*  $R_1 \in [0, 1]$  of the **upper** half ( $y > 0$ ) of the split mirror.

#### Behavior

- Writing a new value automatically updates the complementary transmissivity  $T_1 = 1 - R_1$ .
- Out-of-range assignments are clamped to the interval  $[0, 1]$  with a warning.
- Updating  $R_1$  recalculates the complex amplitude coefficients `.r1_L`, `.r1_R`, `.t1_LTR`, and `.t1_RTL` according to the current phase convention (see `.set_phys_behavior(...)`), clears all cached reflection / transmission matrices, and invalidates the cavity cache.

#### Usage note

Setting this property invalidates the cavity’s cached matrices (if the component belongs to a cavity), to ensure consistency. Setting  $R_1 = 1$  (perfect reflection) makes  $T_1 = 0$ . To avoid numerical singularities in transfer-matrix calculations, you can inject a small fictitious transmissivity via `.set_T1_non_energy_conserving(T1)`, or choose a high but finite value (e.g.  $R_1 = 0.9999$ ).

### .R2

#### Read/write property.

Sets or queries the *power reflectivity*  $R_2 \in [0, 1]$  of the **lower** half ( $y < 0$ ) of the split mirror.

#### Behavior

- Writing a new value automatically updates the complementary transmissivity  $T_2 = 1 - R_2$ .
- Out-of-range assignments are clamped to the interval  $[0, 1]$  with a warning.
- Updating  $R_2$  recalculates the complex amplitude coefficients `.r2_L`, `.r2_R`, `.t2_LTR`, and `.t2 RTL` according to the current phase convention (see `.set_phys_behavior(...)`), clears all cached reflection / transmission matrices, and invalidates the cavity cache.

#### Usage note

Setting this property invalidates the cavity's cached matrices (if the component belongs to a cavity), to ensure consistency. Setting  $R_2 = 1$  (perfect reflection) makes  $T_2 = 0$ . To avoid numerical singularities in transfer-matrix calculations, you can inject a small fictitious transmissivity via `.set_T2_non_energy_conserving(T2)`, or choose a high but finite value (e.g.  $R_2 = 0.9999$ ).

### .T1

#### Read/write property.

Sets or queries the *power transmissivity*  $T_1 \in [0, 1]$  of the **upper** half ( $y > 0$ ) of the split mirror.

#### Behavior

- Writing a new value automatically updates the complementary reflectivity  $R_1 = 1 - T_1$ .
- Out-of-range assignments are clamped to the interval  $[0, 1]$  with a warning.
- Updating  $T_1$  recalculates the complex amplitude coefficients `.r1_L`, `.r1_R`, `.t1_LTR`, and `.t1 RTL` according to the current phase convention (see `.set_phys_behavior(...)`), clears all cached reflection / transmission matrices, and invalidates the cavity cache.

#### Usage note

Setting this property invalidates the cavity's cached matrices (if the component belongs to a cavity), to ensure consistency.

### .T2

#### Read/write property.

Sets or queries the *power transmissivity*  $T_2 \in [0, 1]$  of the **lower** half ( $y < 0$ ) of the split mirror.

**Behavior**

- Writing a new value automatically updates the complementary reflectivity  $R_2 = 1 - T_2$ .
- Out-of-range assignments are clamped to the interval  $[0, 1]$  with a warning.
- Updating  $T_2$  recalculates the complex amplitude coefficients `.r2_L`, `.r2_R`, `.t2_LTR`, and `.t2 RTL` according to the current phase convention (see `.set_phys_behavior(...)`), clears all cached reflection / transmission matrices, and invalidates the cavity cache.

**Usage note**

Setting this property invalidates the cavity's cached matrices (if the component belongs to a cavity), to ensure consistency.

**`.set_T1_non_energy_conserving(T1)`****Method.**

Overrides the top-half transmissivity  $T_1$  of the mirror, allowing values that break energy conservation (i.e.  $R_1 + T_1 \neq 1$ ). This method can be used to manually set  $T_1$  without automatically updating  $R_1 = 1 - T_1$ .

**Behavior**

- Sets the internal transmissivity value  $T_1$  to the provided value, regardless of whether  $T_1 \in [0, 1]$ .
- Does not adjust  $R_1$  accordingly. The user is responsible for maintaining physical consistency, if desired.
- Updates the corresponding complex amplitudes `.t1_LTR` and `.t1 RTL`.
- Clears all cached reflection / transmission matrices and invalidates the cavity cache.

**Usage note**

This method is useful in edge cases where  $R_1 = 1$  would result in singular transfer matrices. Setting  $T_1 > 0$  artificially avoids such issues.

**`.set_T2_non_energy_conserving(T2)`****Method.**

Overrides the bottom-half transmissivity  $T_2$  of the mirror, allowing values that break energy conservation (i.e.  $R_2 + T_2 \neq 1$ ). This method can be used to manually set  $T_2$  without automatically updating  $R_2 = 1 - T_2$ .

**Behavior**

- Sets the internal transmissivity value  $T_2$  to the provided value, regardless of whether  $T_2 \in [0, 1]$ .
- Does not adjust  $R_2$  accordingly. The user is responsible for maintaining physical consistency, if desired.
- Updates the corresponding complex amplitudes `.t2_LTR` and `.t2 RTL`.

- Clears all cached reflection / transmission matrices and invalidates the cavity cache.

### Usage note

This method is useful in edge cases where  $R_2 = 1$  would result in singular transfer matrices. Setting  $T_2 > 0$  artificially avoids such issues.

## `.set_phys_behavior(sym_phase)`

### Method.

Selects the phase convention used for the complex reflection and transmission coefficients of both halves of the split mirror ( $r_1, t_1, r_2, t_2$ ). The two supported conventions are identical to those described for ordinary mirrors in `clsMirrorBase2port.set_phys_behavior(...)`:

- `sym_phase = True` (default) – “symmetric-phase” convention, same reflection phase on both sides.
- `sym_phase = False` – real-valued (“text-book”) convention, left and right reflections differ by a sign.

Calling the method

- recalculates `.r1_L`, `.r1_R`, `.t1_LTR`, `.t1 RTL`, `.r2_L`, `.r2_R`, `.t2_LTR`, `.t2 RTL`;
- clears all cached reflection / transmission matrices; and
- invalidates the cavity cache to ensure subsequent calculations use the new phase convention.

For the theoretical background of the two conventions, see the detailed explanation in section `clsMirrorBase2port.set_phys_behavior(...)`.

## `.sym_phase`

### Read-only property.

Indicates which phase convention is currently applied to the split mirror’s complex coefficients.

### Returned values

- `True` – *Symmetric-phase* convention (identical reflection phase for left and right incidence; complex  $t_1, t_2$ ).
- `False` – *Real-valued* (“text-book”) convention (real  $r_1, r_2, t_1, t_2$ ; the right-incidence reflection phases have the opposite sign of the left-incidence ones).

### Changing the convention

This property is read-only; to switch conventions call `.set_phys_behavior(...)`. That method updates `.r1_L`, `.r1_R`, `.r2_L`, `.r2_R`, `.t1_LTR`, `.t1 RTL`, `.t2_LTR`, `.t2 RTL` and clears all cached matrices to ensure the new phase behaviour is used.

**.r1.L****Read-only property.**

Returns the complex reflection coefficient  $r_1^{(L)}$  for light reflected from the **upper half** ( $y > 0$ ) of the mirror, when incident from the *left*.

**Behavior**

- This value is computed from the current power reflectivity `.R1` and the active phase convention (see `.set_phys_behavior(...)` and `.sym_phase`).
- In symmetric-phase mode, the reflection phase is the same on both sides.
- In real-valued mode, the returned value is real and positive.

**Usage notes**

- The magnitude satisfies  $|r_1^{(L)}| = \sqrt{R_1}$ .
- Any update to `.R1`, `.T1`, or the phase convention triggers a recalculation.

**.r1.R****Read-only property.**

Returns the complex reflection coefficient  $r_1^{(R)}$  for light reflected from the **upper half** ( $y > 0$ ) of the mirror, when incident from the *right*.

**Behavior**

- This value is computed from the current power reflectivity `.R1` and the active phase convention (see `.set_phys_behavior(...)` and `.sym_phase`).
- In symmetric-phase mode (`sym_phase = True`), the value equals `.r1.L`.
- In real-valued mode (`sym_phase = False`), the value has opposite sign:  $r_1^{(R)} = -r_1^{(L)}$ .

**Usage notes**

- The magnitude satisfies  $|r_1^{(R)}| = \sqrt{R_1}$ .
- Any update to `.R1`, `.T1`, or the phase convention triggers a recalculation.

**.r2.L****Read-only property.**

Returns the complex reflection coefficient  $r_2^{(L)}$  for light reflected from the **lower half** ( $y < 0$ ) of the mirror, when incident from the *left*.

**Behavior**

- This value is computed from the current power reflectivity `.R2` and the active phase convention (see `.set_phys_behavior(...)` and `.sym_phase`).
- In symmetric-phase mode, the reflection phase is the same on both sides.
- In real-valued mode, the returned value is real and positive.

**Usage notes**

- The magnitude satisfies  $|r_2^{(L)}| = \sqrt{R_2}$ .
- Any update to `.R2`, `.T2`, or the phase convention triggers a recalculation.

**`.r2_R`****Read-only property.**

Returns the complex reflection coefficient  $r_2^{(R)}$  for light reflected from the **lower half** ( $y < 0$ ) of the mirror, when incident from the *right*.

**Behavior**

- This value is computed from the current power reflectivity `.R2` and the active phase convention (see `.set_phys_behavior(...)` and `.sym_phase`).
- In symmetric-phase mode (`sym_phase = True`), the value equals `.r2_L`.
- In real-valued mode (`sym_phase = False`), the value has opposite sign:  $r_2^{(R)} = -r_2^{(L)}$ .

**Usage notes**

- The magnitude satisfies  $|r_2^{(R)}| = \sqrt{R_2}$ .
- Any update to `.R2`, `.T2`, or the phase convention triggers a recalculation.

**`.t1_LTR`****Read-only property.**

Returns the complex transmission coefficient  $t_1^{(LTR)}$  for light transmitted through the **upper half** ( $y > 0$ ) of the mirror, from left to right.

**Behavior**

- The coefficient is computed together with `.t1_RTL` by the internal routine that enforces unitarity (see `.set_phys_behavior(...)`).
- *Symmetric-phase mode* (`.sym_phase = True`):  $t_1^{(LTR)} = 1 + r$  with  $r = .r1_L$ . The term carries a small phase whose magnitude depends on  $\sqrt{R_1(1 - R_1)}$ .
- *Real-valued (“text-book”) mode* (`sym_phase = False`):  $t_1^{(LTR)} = +\sqrt{T_1}$  – purely real and positive.

**Usage notes**

- The magnitude satisfies  $|t_1^{(LTR)}| = \sqrt{T_1}$  in either convention.
- Any change to `.R1`, `.T1`, or the phase convention automatically updates this value.

### [.t1\\_RTL](#)

#### Read-only property.

Returns the complex transmission coefficient  $t_1^{(\text{RTL})}$  for light transmitted through the **upper half** ( $y > 0$ ) of the mirror, from right to left.

#### Behavior

- The coefficient is computed together with [.t1\\_LTR](#) by the internal routine that enforces unitarity (see [.set\\_phys\\_behavior\(...\)](#)).
- *Symmetric-phase mode* (`.sym_phase = True`):  $t_1^{(\text{RTL})} = 1 + r$  with  $r = .r1_R$ . The term carries a small phase whose magnitude depends on  $\sqrt{R_1(1 - R_1)}$ .
- *Real-valued (“text-book”) mode* (`.sym_phase = False`):  $t_1^{(\text{RTL})} = +\sqrt{T_1}$  – purely real and positive.

#### Usage notes

- The magnitude satisfies  $|t_1^{(\text{RTL})}| = \sqrt{T_1}$  in either convention.
- Any change to `.R1`, `.T1`, or the phase convention automatically updates this value.

### [.t2\\_LTR](#)

#### Read-only property.

Returns the complex transmission coefficient  $t_2^{(\text{LTR})}$  for light transmitted through the **lower half** ( $y < 0$ ) of the mirror, from left to right.

#### Behavior

- The coefficient is computed together with [.t2\\_RTL](#) by the internal routine that enforces unitarity (see [.set\\_phys\\_behavior\(...\)](#)).
- *Symmetric-phase mode* (`.sym_phase = True`):  $t_2^{(\text{LTR})} = 1 + r$  with  $r = .r2_L$ . The term carries a small phase whose magnitude depends on  $\sqrt{R_2(1 - R_2)}$ .
- *Real-valued (“text-book”) mode* (`.sym_phase = False`):  $t_2^{(\text{LTR})} = +\sqrt{T_2}$  – purely real and positive.

#### Usage notes

- The magnitude satisfies  $|t_2^{(\text{LTR})}| = \sqrt{T_2}$  in either convention.
- Any change to `.R2`, `.T2`, or the phase convention automatically updates this value.

### [.t2\\_RTL](#)

#### Read-only property.

Returns the complex transmission coefficient  $t_2^{(\text{RTL})}$  for light transmitted through the **lower half** ( $y < 0$ ) of the mirror, from right to left.

**Behavior**

- The coefficient is computed together with `.t2_LTR` by the internal routine that enforces unitarity (see `.set_phys_behavior(...)`).
- *Symmetric-phase mode* (`.sym_phase = True`):  $t_2^{(\text{RTL})} = 1 + r$  with  $r = .r2_R$ . The term carries a small phase whose magnitude depends on  $\sqrt{R_2(1 - R_2)}$ .
- *Real-valued ("text-book") mode* (`sym_phase = False`):  $t_2^{(\text{RTL})} = +\sqrt{T_2}$  – purely real and positive.

**Usage notes**

- The magnitude satisfies  $|t_2^{(\text{RTL})}| = \sqrt{T_2}$  in either convention.
- Any change to `.R2`, `.T2`, or the phase convention automatically updates this value.

**`.left_side_relevant`****Read/write property.**

Boolean flag that states whether accurate calculations are required for the *left* mirror surface (reflection masks, tilt-induced phase shifts, ...).

**Behavior**

- **True** (default): the left-hand reflection is evaluated with full accuracy, including tilt masks if applicable.
- **False**: the code may skip all calculations specific to the left surface, saving time and memory when that port is known to be unused (e.g. if you are not interested in the light field on the left side of the cavity).
- Changing this flag invalidates the cavity's cached matrices (if the component belongs to a cavity) to ensure consistency.

**Usage notes**

- Use this flag (together with `.right_side_relevant`) to avoid costly tilt-mask calculations for mirror surfaces that do not contribute to the simulated signals.

**`.right_side_relevant`****Read/write property.**

Boolean flag that states whether accurate calculations are required for the *right* mirror surface (reflection masks, tilt-induced phase shifts, ...).

**Behavior**

- **True** (default): the right-hand reflection is evaluated with full accuracy, including tilt masks if applicable.
- **False**: the code may skip all calculations specific to the right surface, saving time and memory when that port is known to be unused (e.g. if you are not interested in the light field on the right side of the cavity).

- Changing this flag invalidates the cavity's cached matrices (if the component belongs to a cavity) to ensure consistency.

#### Usage notes

- Use this flag (together with `.left_side_relevant`) to avoid costly tilt-mask calculations for mirror surfaces that do not contribute to the simulated signals.

### D.12.3 Mirror Tilt

#### `.rot_around_x_deg`

##### Read/write property.

Specifies the mirror tilt angle about the  $x$ -axis in *degrees*.

##### Behavior

- Setting this property updates the internal representation of the mirror tilt.
- The new tilt invalidates cached tilt masks and any reflection or transmission matrices that depend on them; the cavity's cached matrices are cleared as well.
- Allowed range:  $|\alpha| < 45^\circ$ . Values outside this range are rejected with an error message.

##### Sign convention

A *positive* angle tilts the mirror such that the left-side reflection is deflected toward  $+y$  (*up*) and the right-side reflection toward  $-y$  (*down*).

#### Usage notes

- This class currently provides only a degree-based interface for the tilt angle. Unlike in the mirror base class `clsMirrorBase2port`, there is no corresponding `.rot_around_x` property in radians.

#### `.rot_around_y_deg`

##### Read/write property.

Specifies the mirror tilt angle about the  $y$ -axis in *degrees*.

##### Behavior

- Setting this property updates the internal representation of the mirror tilt.
- The new tilt invalidates cached tilt masks and any reflection or transmission matrices that depend on them; the cavity's cached matrices are cleared as well.
- Allowed range:  $|\alpha| < 45^\circ$ . Values outside this range are rejected with an error message.

##### Sign convention

A *positive* angle tilts the mirror such that the left-side reflection is deflected toward  $+x$  (*right*) and the right-side reflection toward  $-x$  (*left*).

**Usage notes**

- This class currently provides only a degree-based interface for the tilt angle. Unlike in the mirror base class `clsMirrorBase2port`, there is no corresponding `.rot_around_y` property in radians.

**`.tilt_mask_x_L`****Read-only property.**

Returns the spatial phase mask applied to the *left-hand* reflection when the split mirror is tilted about the y-axis.

**Definition of the mask**

For a mirror rotation angle  $\alpha_y$  the mask is

$$\tilde{M}_{x,L}(x, y) = \exp[i k_0 x \tan(-2\alpha_y)], \quad k_0 = \frac{2\pi}{\lambda},$$

with  $x$  taken from the total grid axis `grid.axis_tot`. The factor  $-2\alpha_y$  follows from geometrical optics: a positive rotation about the y-axis steers the left-side reflection toward  $-x$ .

**Behavior**

- If  $\alpha_y = 0$  or `.left_side_relevant = False`, the property returns the scalar value 1.
- Otherwise the mask is generated lazily on first access and then cached.

**Usage notes**

- Multiplying a position-space field by  $\tilde{M}_{x,L}$  imposes the correct lateral phase ramp for the prescribed tilt on the left-hand reflection.
- The corresponding mask for the right-hand reflection is `.tilt_mask_x_R`.

**`.tilt_mask_x_R`****Read-only property.**

Returns the spatial phase mask applied to the *right-hand* reflection when the split mirror is tilted about the y-axis.

**Definition of the mask**

For a mirror rotation angle  $\alpha_y$  the mask is

$$\tilde{M}_{x,R}(x, y) = \exp[i k_0 x \tan(+2\alpha_y)], \quad k_0 = \frac{2\pi}{\lambda},$$

with  $x$  taken from the total grid axis `grid.axis_tot`. A positive rotation about the y-axis steers the right-side reflection toward  $+x$ , hence the sign change in the tangent term.

**Behavior**

- If  $\alpha_y = 0$  or `.right_side_relevant = False`, the property returns the scalar value 1.

- Otherwise the mask is generated lazily on first access and then cached.

### Usage notes

- Multiplying a position-space field by  $\tilde{M}_{x,R}$  imposes the correct lateral phase ramp for the prescribed tilt on the right-hand reflection.
- The corresponding mask for the left-hand side is `.tilt_mask_x_L`.

### `.tilt_mask_y_L`

#### Read-only property.

Returns the spatial phase mask applied to the *left-hand* reflection when the split mirror is tilted about the **x**-axis.

**Definition of the mask** For a mirror rotation angle  $\alpha_x$  the mask is

$$\tilde{M}_{y,L}(x, y) = \exp[i k_0 y \tan(+2\alpha_x)], \quad k_0 = \frac{2\pi}{\lambda},$$

with  $y$  taken from the total grid axis `grid.axis_tot`. A positive rotation about the **x**-axis steers the left-side reflection toward  $+y$ ; hence the  $+2\alpha_x$  in the tangent term.

#### Behavior

- If  $\alpha_x = 0$  or `.left_side_relevant = False`, the property returns the scalar value 1.
- Otherwise the mask is generated lazily on first access and then cached.

### Usage notes

- Multiplying a position-space field by  $\tilde{M}_{y,L}$  imposes the correct vertical phase ramp for the prescribed tilt on the left-hand reflection.
- The corresponding mask for the right-hand reflection is `.tilt_mask_y_R`.

### `.tilt_mask_y_R`

#### Read-only property.

Returns the spatial phase mask applied to the *right-hand* reflection when the split mirror is tilted about the **x**-axis.

**Definition of the mask**

For the same rotation angle  $\alpha_x$  (in radians) the right-side mask is

$$\tilde{M}_{y,R}(x, y) = \exp[i k_0 y \tan(-2\alpha_x)], \quad k_0 = \frac{2\pi}{\lambda},$$

again with  $y$  taken from `grid.axis_tot`. The sign in the tangent is inverted because a positive  $\alpha_x$  deflects the right-side reflection toward  $-y$ .

#### Behavior

- If  $\alpha_x = 0$  or `.right_side_relevant = False`, the property returns the scalar 1.
- Otherwise the mask is generated lazily on first access and cached for reuse.

**Usage notes**

- Multiplying a position-space field by  $\tilde{M}_{y,R}$  imparts the correct vertical phase ramp for the prescribed  $x$ -tilt on the right-hand reflection.
- The complementary mask for the left-hand reflection is `.tilt_mask_y_L`.

**D.12.4 Reflection Matrices****`.R_L_mat_tot`****Read-only property.**

Returns the total-resolution reflection matrix  $\mathbf{R}_L$  for light incident from the **left** side of the *split* mirror.

**Behavior**

- The first call triggers `.calc_R_L_mat_tot()` if the matrix has not yet been cached; subsequent calls simply fetch the cached matrix.
- `.calc_R_L_mat_tot()` proceeds as follows:
  1. For every plane-wave basis mode  $\psi_{n_x,n_y}$  of the total grid it multiplies the *upper* half ( $y > 0$ ) by the coefficient `.r1_L` and the *lower* half ( $y < 0$ ) by `.r2_L`.
  2. If the mirror is tilted (`rot_around_x_deg` or `rot_around_y_deg` non-zero) and `.left_side_relevant` is `True`, the appropriate  $x$ - and/or  $y$ -tilt masks are applied.
  3. The result is converted back to modal coefficients; all columns are assembled – optionally in parallel – into the dense matrix  $\mathbf{R}_L$ , which is then cached.
- If `.left_side_relevant` is `False` the same routine still runs, but any tilt-mask multiplication is skipped, so the matrix effectively reduces to a diagonal operator that merely inserts the two complex factors  $r_1^{(L)}$  and  $r_2^{(L)}$  in their respective halves.

**Usage notes**

- The top-half and bottom-half reflectivities can be tuned independently via `.R1` and `.R2`; the change is immediately reflected here after the cache is invalidated.
- Clearing or altering any tilt angle, side-relevance flag, or phase convention forces the matrix to be rebuilt on the next access.

**`.R_R_mat_tot`****Read-only property.**

Returns the total-resolution reflection matrix  $\mathbf{R}_R$  for light incident from the **right** side of the *split* mirror.

**Behavior**

- The first call triggers `.calc_R_R_mat_tot()` if the matrix has not yet been cached; subsequent calls simply fetch the cached matrix.

- `.calc_R_R_mat_tot()` proceeds as follows:
  1. For every plane-wave basis mode  $\psi_{n_x, n_y}$  of the total grid it multiplies the *upper* half ( $y > 0$ ) by the coefficient `.r1_R` and the *lower* half ( $y < 0$ ) by `.r2_R`.
  2. If the mirror is tilted (`rot_around_x_deg` or `rot_around_y_deg` non-zero) and `.right_side_relevant` is `True`, the appropriate  $x$ - and/or  $y$ -tilt masks are applied.
  3. The result is converted back to modal coefficients; all columns are assembled – optionally in parallel – into the dense matrix  $\mathbf{R}_R$ , which is then cached.
- If `.right_side_relevant` is `False` the same routine still runs, but any tilt-mask multiplication is skipped, so the matrix effectively reduces to a diagonal operator that merely inserts the two complex factors  $r_1^{(R)}$  and  $r_2^{(R)}$  in their respective halves.

### Usage notes

- The top-half and bottom-half reflectivities can be tuned independently via `.R1` and `.R2`; the change is immediately reflected here after the cache is invalidated.
- Clearing or altering any tilt angle, side-relevance flag, or phase convention forces the matrix to be rebuilt on the next access.

## `.calc_R_L_mat_tot()`

### Method.

Explicitly (re)computes the left-hand reflection matrix  $\mathbf{R}_L$  for the current optical parameters of the split mirror and caches it. The routine is normally invoked lazily when `.R_L_mat_tot` is first accessed or after a cache-invalidating change.

### Algorithm

1. *Tilt-mask preparation.* If at least one of the tilt angles `rot_around_x_deg`, `rot_around_y_deg` is non-zero and `.left_side_relevant` is `True`, the required  $x$ - and/or  $y$ -tilt masks are generated (or reused if already present).
2. *Mode-by-mode synthesis.* Let  $N = .grid.res\_tot^2$ . For every modal index  $i \in [0, N - 1]$ :
  - (a) Create the Fourier basis function  $\psi_{n_x, n_y}(\mathbf{r})$  in *position* space.
  - (b) Multiply the *upper* half ( $y > 0$ ) by the complex coefficient `.r1_L` and the *lower* half ( $y < 0$ ) by `.r2_L`.
  - (c) If the mirror is tilted and the left side is relevant, multiply by the pre-computed  $x$ - and/or  $y$ -tilt masks.
  - (d) Convert the modified field back to a coefficient vector with `.grid.arr_to_vec(...)` and store it as column  $i$  of  $\mathbf{R}_L$ .

If `.par_RT_calc` is `True` (default), the per-mode computations run in parallel via `joblib`; the number of worker processes is taken from `clsCavity.mp_pool_processes`.

3. *Finalisation.* After all columns are assembled, the fully-populated matrix is marked as available so subsequent calls can return the cached result instantly.

**Usage notes**

- Setting `.left_side_relevant` to `False` skips the tilt-mask multiplication, reducing the computational cost when the left reflection is not required.
- The cached matrix is deleted automatically whenever you change reflectivities, phase convention, tilt angles, or side-relevance flags.

`.calc_R_R_mat_tot()`

**Method.**

Explicitly (re)computes the right-hand reflection matrix  $\mathbf{R}_R$  for the current optical parameters of the split mirror and caches it. The routine is normally invoked lazily when `.R_R_mat_tot` is first accessed or after a cache-invalidating change.

**Algorithm**

1. *Tilt-mask preparation.* If at least one of the tilt angles `rot_around_x_deg`, `rot_around_y_deg` is non-zero and `.right_side_relevant` is `True`, the required *x*- and/or *y*-tilt masks are generated (or reused if already present).
2. *Mode-by-mode synthesis.* Let  $N = .grid.res\_tot^2$ . For every modal index  $i \in [0, N - 1]$ :
  - (a) Create the Fourier basis function  $\psi_{n_x, n_y}(\mathbf{r})$  in *position* space.
  - (b) Multiply the *upper* half ( $y > 0$ ) by the complex coefficient `.r1_R` and the *lower* half ( $y < 0$ ) by `.r2_R`.
  - (c) If the mirror is tilted and the right side is relevant, multiply by the pre-computed *x*- and/or *y*-tilt masks.
  - (d) Convert the modified field back to a coefficient vector with `.grid.arr_to_vec(...)` and store it as column  $i$  of  $\mathbf{R}_R$ .

If `.par_RT_calc` is `True` (default), the per-mode computations run in parallel via `joblib`; the number of worker processes is taken from `clsCavity.mp_pool_processes`.

3. *Finalisation.* After all columns are assembled, the fully-populated matrix is marked as available so subsequent calls can return the cached result instantly.

**Usage notes**

- Setting `.right_side_relevant` to `False` skips the tilt-mask multiplication, reducing the computational cost when the right reflection is not required.
- The cached matrix is deleted automatically whenever you change reflectivities, phase convention, tilt angles, or side-relevance flags.

### D.12.5 Transmission

```
.prop(E_in, k_space_in, k_space_out, direction)
```

#### Method.

Propagates an input field through the *split* mirror, applying the complex transmission coefficients of the upper and lower halves. The routine fulfils the standard `.prop(...)` interface of `clsOptComponent2port`.

#### Parameters

- `E_in` - Input optical field (NumPy array, position or *k*-space).
- `k_space_in` - `True` if `E_in` is supplied in *k*-space, `False` for position space.
- `k_space_out` - `True` to obtain the result in *k*-space, `False` for position space.
- `direction` - Direction of propagation (`Dir.LTR` or `Dir.RTL`). For a split mirror the forward and reverse paths are identical, so `direction` is ignored.

#### Algorithm

1. If `k_space_in` is `True`, perform an inverse FFT to position space.
2. *Row-wise scaling.* Let  $t_1 = .t1\_LTR (= .t1\_RTL)$  and  $t_2 = .t2\_LTR (= .t2\_RTL)$ . Denote by  $N$  the number of rows of the 2-D field array.
  - Upper half ( $y > 0$ ): multiply rows  $\lceil N/2 \rceil \dots N - 1$  by  $t_1$ .
  - Lower half ( $y < 0$ ): multiply rows  $0 \dots \lfloor N/2 \rfloor - 1$  by  $t_2$ .
  - If  $N$  is odd, the centre row receives the average coefficient  $(t_1 + t_2)/2$ .
3. If `k_space_out` is `True`, transform the modified field back to *k*-space.

```
.T_LTR_mat_tot
```

#### Read-only property.

Returns the total-resolution transmission matrix  $\mathbf{T}_{LTR}$  for a field propagating from *left to right* through the split mirror. Because the component is reciprocal, exactly the same matrix is returned by `.T_RTL_mat_tot`.

#### Behaviour

- On first access (or after the cache has been invalidated) the matrix is generated by `.calc_T_mat_tot()`; subsequent calls simply fetch the cached result.
- Construction principle:
  1. For each Fourier basis mode the rows with  $y > 0$  are multiplied by `.t1_LTR` while the rows with  $y < 0$  receive `.t2_LTR`.
  2. The modified field is recast into modal coefficients and inserted as a column of  $\mathbf{T}_{LTR}$ .
  3. If `.par_RT_calc` is `True` (default), these per-mode operations are executed in parallel using the worker count set in `clsCavity.mp_pool_processes`.
- The algorithm neglects diffraction or coupling at the horizontal interface between the two halves; the element is treated as an *infinitely thin*, piecewise-homogeneous layer that applies its transmission factors row-wise.

**Usage notes**

- Any parameter change that alters the transmission behaviour (e.g. modifying `.T1`, `.T2`, the corresponding reflectivities, the phase convention, or a tilt angle) automatically clears the cached matrix so that it is rebuilt on the next request.
- Since  $\mathbf{T}_{\text{LTR}} = \mathbf{T}_{\text{RTL}}$ , only one matrix needs to be stored, saving memory.

`.T RTL mat tot`

**Read-only property.**

Returns the total-resolution transmission matrix  $\mathbf{T}_{\text{RTL}}$  for a field propagating from *right to left* through the split mirror. Because the component is reciprocal, exactly the same matrix is returned by `.T LTR mat tot`.

**Behaviour**

- On first access (or after the cache has been invalidated) the matrix is generated by `.calc_T_mat_tot()`; subsequent calls simply fetch the cached result.
- Construction principle:
  1. For each Fourier basis mode the rows with  $y > 0$  are multiplied by `.t1 RTL` while the rows with  $y < 0$  receive `.t2 RTL`.
  2. The modified field is recast into modal coefficients and inserted as a column of  $\mathbf{T}_{\text{RTL}}$ .
  3. If `.par_RT_calc` is `True` (default), these per-mode operations are executed in parallel using the worker count set in `clsCavity.mp_pool_processes`.
- The algorithm neglects diffraction or coupling at the horizontal interface between the two halves; the element is treated as an *infinitely thin*, piecewise-homogeneous layer that applies its transmission factors row-wise.

**Usage notes**

- Any parameter change that alters the transmission behaviour (e.g. modifying `.T1`, `.T2`, the corresponding reflectivities, the phase convention, or a tilt angle) automatically clears the cached matrix so that it is rebuilt on the next request.
- Since  $\mathbf{T}_{\text{RTL}} = \mathbf{T}_{\text{LTR}}$ , only one matrix needs to be stored, saving memory.

`.calc_T_mat_tot()`

**Method.**

Explicitly (re)computes the transmission matrix  $\mathbf{T}_{\text{LTR}} = \mathbf{T}_{\text{RTL}}$  for the current optical parameters of the split mirror and caches it. The method is usually invoked lazily when `.T LTR mat tot` or `.T RTL mat tot` is first accessed or after a cache-invalidating change.

### Algorithm

1. Let  $N = \text{.grid.res\_tot}^2$ . For every modal index  $i \in [0, N - 1]$ :
  - (a) Generate the corresponding Fourier basis function  $\psi_{n_x, n_y}(\mathbf{r})$  in position space.
  - (b) Multiply the *upper* rows ( $y > 0$ ) of the field by `.t1_LTR`, the *lower* rows ( $y < 0$ ) by `.t2_LTR`.
  - (c) Convert the modified field back to a coefficient vector using `.grid.arr_to_vec(...)` and insert it as column  $i$  of  $\mathbf{T}_{\text{LTR}}$ .

If `.par_RT_calc` is `True` (default), these operations are distributed across multiple worker processes via `joblib`. The number of processes is set in `clsCavity.mp_pool_processes`.

2. Once all columns are assembled, the matrix is cached for future use.

### Usage notes

- Because the component is reciprocal, this method builds the same matrix for both propagation directions:  $\mathbf{T}_{\text{LTR}} = \mathbf{T}_{\text{RTL}}$ .
- The algorithm neglects diffraction or mode mixing at the horizontal boundary; the mirror is modelled as an infinitely thin element with row-wise transmission coefficients.
- Any change to reflectivities, transmission values, phase convention, or tilt angles automatically deletes the cached matrix, so this routine will run again when needed.

## D.12.6 Symmetry and Space Preferences

### `.symmetric`

#### Read-only property.

Indicates whether the split mirror is *geometrically symmetric*, i.e. not tilted.

#### Behavior

- Returns `True` if both `.rot_around_x_deg` and `.rot_around_y_deg` are equal to zero.
- Returns `False` if the mirror is tilted around either axis.

### `.k_space_in_dont_care`

#### Read-only property.

Indicates whether the component is indifferent to the representation (position space or  $k$ -space) of the *input* field.

#### Behavior

This property always returns `False`, because the internal transmission logic applies spatially varying coefficients (`.t1_LTR`, `.t2_LTR`) in position space.

**.k\_space\_in\_prefer****Read-only property.**

Indicates whether the component *prefers* to receive input fields in *k*-space rather than in position space.

**Behavior**

This property always returns `False`, because transmission through the mirror requires the input field in position space so that the appropriate transmission coefficients can be applied above and below  $y = 0$ .

**.k\_space\_out\_dont\_care****Read-only property.**

Indicates whether the component is indifferent to the representation (position space or *k*-space) of the *output* field.

**Behavior**

This property always returns `False`, since transmission is computed in position space, and any requested *k*-space output requires an explicit Fourier transform.

**.k\_space\_out\_prefer****Read-only property.**

Indicates whether the component *prefers* to return output fields in *k*-space rather than in position space.

**Behavior**

This property always returns `False`, because the natural output of the transmission operation is in position space. A conversion to *k*-space is only performed if explicitly requested.

## D.12.7 Memory Management and Other Settings

**.clear\_mem\_cache()****Method.**

Clears all cached numerical data from memory that is specific to the split mirror's optical behavior. This method is automatically called whenever any relevant parameter (e.g. reflectivities, phase convention, or tilt angles) is modified.

**Behavior**

This method:

- Deletes the cached tilt phase masks for *x*- and *y*-tilts on both the left and right sides,
- Deletes the cached reflection matrices  $\mathbf{R}_L$  and  $\mathbf{R}_R$ ,
- Deletes the cached transmission matrix  $\mathbf{T}$ .

**Usage note**

This method is automatically invoked when relevant properties are changed, but may also be called manually.

**.par\_RT\_calc****Read/write property.**

Controls whether the reflection and transmission matrices are calculated in parallel.

**Behavior**

If `.par_RT_calc` is set to `True` (default), the methods `.calc_R_R_mat_tot()`, `.calc_R_L_mat_tot()`, and `.calc_T_mat_tot()` process the individual plane wave modes in parallel using `joblib`. The number of parallel worker processes is taken from `clsCavity.mp_pool_processes`.

If set to `False`, the matrix calculations are performed in a single-threaded, serial manner. This is useful for debugging or in special environments where parallel processing is undesirable.

## Appendix E

# 4-Port Optical Components for Linear Cavities

### E.1 Abstract Base Class `clsOptComponent4port`

The abstract base class `clsOptComponent4port` represents all *4-port optical components*. Such components have two distinct physical paths (denoted “A” and “B”) that enter from both the left and the right side, making a total of four ports. A canonical example is a non-polarising beamsplitter, where an incident field can either stay within its original path or be cross-coupled into the opposite path. All 4-port components can be chained in the two-path configuration used by `clsCavity2path` cavities.

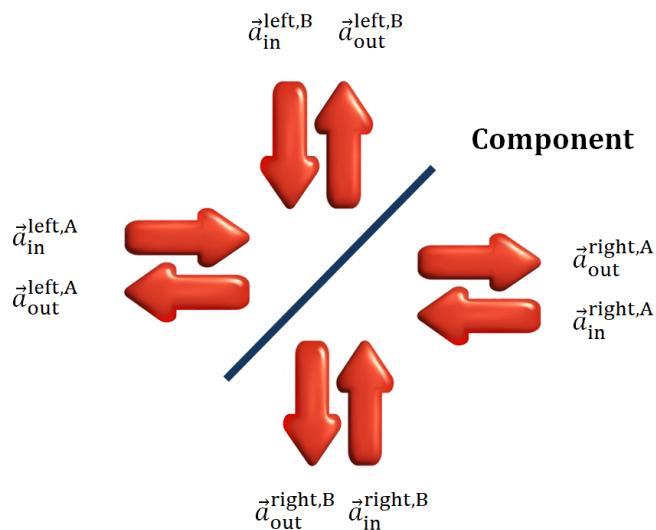


Figure E.1: Conceptual view of a 4-port component. Each side supports two independent spatial paths (A and B). Consequently, four input amplitude vectors ( $\mathbf{a}_{in}^{left,A}$ ,  $\mathbf{a}_{in}^{left,B}$ ,  $\mathbf{a}_{in}^{right,A}$ ,  $\mathbf{a}_{in}^{right,B}$ ) are mapped to four output vectors ( $\mathbf{a}_{out}^{left,A}$ ,  $\mathbf{a}_{out}^{left,B}$ ,  $\mathbf{a}_{out}^{right,A}$ ,  $\mathbf{a}_{out}^{right,B}$ ) by a  $4 \times 4$  block matrix.

Like its 2-port counterpart, `clsOptComponent4port` is *abstract*: it cannot be instantiated directly. Instead, it defines a common interface that every concrete 4-port element must implement. In particular it provides

- A `.prop(...)` method that propagates two incoming light fields (`in_A` and `in_B`) either left-to-right (`Dir.LTR`) or right-to-left (`Dir.RTL`), taking path coupling into account.
- Low-level access to the element's four basic  $2 \times 2$  *block* matrices:
  - `.get_R_bmat_tot(...)` – reflection block matrix for a given side.
  - `.get_T_bmat_tot(...)` – transmission block matrix for a given propagation direction.
  - `.get_R_mat_tot(...)` – reflection matrix for a specific path-to-path channel.
  - `.get_T_mat_tot(...)` – transmission matrix for a specific path-to-path channel.

These matrices operate on the total-resolution spatial grid used throughout the library.

- From the above, two composite  $4 \times 4$  block matrices are constructed:
  - The *scattering matrix* `.S_bmat_tot`, which maps all incoming amplitudes to all outgoing amplitudes.
  - The *transfer matrix* `.M_bmat_tot`, which relates the total field on one side of the component to the other and is central to cavity analysis.

### Block-matrix conventions

For a 2-port component the scattering relation is expressed by Eq. (D.1). In the 4-port case each side carries two paths, so every “vector” in Eq. (D.1.4) becomes a *vector of vectors*. Specifically we expand

$$\mathbf{a}_{\text{in}}^{\text{left}} = \begin{pmatrix} \mathbf{a}_{\text{in}}^{\text{left},A} \\ \mathbf{a}_{\text{in}}^{\text{left},B} \end{pmatrix}, \quad \mathbf{a}_{\text{in}}^{\text{right}} = \begin{pmatrix} \mathbf{a}_{\text{in}}^{\text{right},A} \\ \mathbf{a}_{\text{in}}^{\text{right},B} \end{pmatrix}, \quad \mathbf{a}_{\text{out}}^{\text{left}} = \begin{pmatrix} \mathbf{a}_{\text{out}}^{\text{left},A} \\ \mathbf{a}_{\text{out}}^{\text{left},B} \end{pmatrix}, \quad \mathbf{a}_{\text{out}}^{\text{right}} = \begin{pmatrix} \mathbf{a}_{\text{out}}^{\text{right},A} \\ \mathbf{a}_{\text{out}}^{\text{right},B} \end{pmatrix}. \quad (\text{E.1})$$

With this notation the scattering equation becomes

$$\begin{pmatrix} \mathbf{a}_{\text{out}}^{\text{left}} \\ \mathbf{a}_{\text{out}}^{\text{right}} \end{pmatrix} = \mathbf{S} \begin{pmatrix} \mathbf{a}_{\text{in}}^{\text{left}} \\ \mathbf{a}_{\text{in}}^{\text{right}} \end{pmatrix}, \quad \mathbf{S} = \begin{pmatrix} \mathbf{R}_L & \mathbf{T}_{\text{RTL}} \\ \mathbf{T}_{\text{LTR}} & \mathbf{R}_R \end{pmatrix}, \quad (\text{E.2})$$

where each block (e.g.  $\mathbf{R}_L$ ) is itself a  $2 \times 2$  block matrix that resolves the two paths:

$$\mathbf{R}_L = \begin{pmatrix} \mathbf{R}_L^{A \rightarrow A} & \mathbf{R}_L^{B \rightarrow A} \\ \mathbf{R}_L^{A \rightarrow B} & \mathbf{R}_L^{B \rightarrow B} \end{pmatrix}, \quad \mathbf{T}_{\text{RTL}} = \begin{pmatrix} \mathbf{T}_{\text{RTL}}^{A \rightarrow A} & \mathbf{T}_{\text{RTL}}^{B \rightarrow A} \\ \mathbf{T}_{\text{RTL}}^{A \rightarrow B} & \mathbf{T}_{\text{RTL}}^{B \rightarrow B} \end{pmatrix}, \quad \text{etc.} \quad (\text{E.3})$$

### Sub-matrix naming scheme.

Every sub-matrix in  $\mathbf{S}$  carries (i) a side or direction tag and (ii) a path-to-path superscript, for example:

- $\mathbf{R}_L^{A \rightarrow B}$  – “reflect at the *left* interface, taking light that started in path *A* and sending it back out in path *B*.”
- $\mathbf{T}_{\text{RTL}}^{B \rightarrow A}$  – “transmit *right-to-left*, converting a field that came in on the right in path *B* into a field that exits on the left in path *A*.”

With this notation the full  $4 \times 4$  scattering relation reads

$$\begin{pmatrix} \mathbf{a}_{\text{out}}^{\text{left},A} \\ \mathbf{a}_{\text{out}}^{\text{left},B} \\ \mathbf{a}_{\text{out}}^{\text{right},A} \\ \mathbf{a}_{\text{out}}^{\text{right},B} \end{pmatrix} = \begin{pmatrix} \mathbf{R}_L^{A \rightarrow A} & \mathbf{R}_L^{B \rightarrow A} & \mathbf{T}_{\text{RTL}}^{A \rightarrow A} & \mathbf{T}_{\text{RTL}}^{B \rightarrow A} \\ \mathbf{R}_L^{A \rightarrow B} & \mathbf{R}_L^{B \rightarrow B} & \mathbf{T}_{\text{RTL}}^{A \rightarrow B} & \mathbf{T}_{\text{RTL}}^{B \rightarrow B} \\ \mathbf{T}_{\text{LTR}}^{A \rightarrow A} & \mathbf{T}_{\text{LTR}}^{B \rightarrow A} & \mathbf{R}_R^{A \rightarrow A} & \mathbf{R}_R^{B \rightarrow A} \\ \mathbf{T}_{\text{LTR}}^{A \rightarrow B} & \mathbf{T}_{\text{LTR}}^{B \rightarrow B} & \mathbf{R}_R^{A \rightarrow B} & \mathbf{R}_R^{B \rightarrow B} \end{pmatrix} \begin{pmatrix} \mathbf{a}_{\text{in}}^{\text{left},A} \\ \mathbf{a}_{\text{in}}^{\text{left},B} \\ \mathbf{a}_{\text{in}}^{\text{right},A} \\ \mathbf{a}_{\text{in}}^{\text{right},B} \end{pmatrix}. \quad (\text{E.4})$$

Equation (E.4) makes explicit how every incoming amplitude vector couples to every outgoing one through the scattering block-matrix.

The transfer-matrix convention is generalised analogously:

$$\begin{pmatrix} \mathbf{a}_{\text{out}}^{\text{left}} \\ \mathbf{a}_{\text{in}}^{\text{left}} \end{pmatrix} = \mathbf{M} \begin{pmatrix} \mathbf{a}_{\text{in}}^{\text{right}} \\ \mathbf{a}_{\text{out}}^{\text{right}} \end{pmatrix}. \quad (\text{E.5})$$

The  $4 \times 4$  *transfer* relation is written in the same expanded form:

$$\begin{pmatrix} \mathbf{a}_{\text{out}}^{\text{left},A} \\ \mathbf{a}_{\text{out}}^{\text{left},B} \\ \mathbf{a}_{\text{in}}^{\text{left},A} \\ \mathbf{a}_{\text{in}}^{\text{left},B} \end{pmatrix} = \begin{pmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} & \mathbf{M}_{13} & \mathbf{M}_{14} \\ \mathbf{M}_{21} & \mathbf{M}_{22} & \mathbf{M}_{23} & \mathbf{M}_{24} \\ \mathbf{M}_{31} & \mathbf{M}_{32} & \mathbf{M}_{33} & \mathbf{M}_{34} \\ \mathbf{M}_{41} & \mathbf{M}_{42} & \mathbf{M}_{43} & \mathbf{M}_{44} \end{pmatrix} \begin{pmatrix} \mathbf{a}_{\text{in}}^{\text{right},A} \\ \mathbf{a}_{\text{in}}^{\text{right},B} \\ \mathbf{a}_{\text{out}}^{\text{right},A} \\ \mathbf{a}_{\text{out}}^{\text{right},B} \end{pmatrix}. \quad (\text{E.6})$$

Here each  $\mathbf{M}_{ij}$  is itself a square matrix of size  $n \times n$  that captures the channel-resolved coupling between the corresponding input and output amplitude vectors.

The library provides efficient routines to convert between  $\mathbf{S}$  and  $\mathbf{M}$  and to invert the latter; see `.M_bmat_tot` and `.calc_inv_M_bmat_tot()` for details.

### E.1.1 Initialization

`clsOptComponent4port(name, cavity)`

#### Constructor.

Class `clsOptComponent4port` derives from `clsOptComponent` and remains abstract: it cannot be instantiated directly. For details we refer to the description of the constructor `clsOptComponent(...)`.

### E.1.2 Light Field Propagation

`.prop(in_A, in_B, direction)`

#### Default implementation method.

The method propagates two incident light fields through the component by explicitly applying the channel-resolved transmission matrices. For a total-resolution grid with  $N \times N$  points the full transfer block matrix spans  $(2N^2) \times (2N^2)$ ; multiplying it by the input vector therefore entails four dense matrix–vector products of size  $2N^2$ , which is functional but computationally heavy.

### Parameters

- `in_A` – instance of `clsLightField`, representing the input field in path A on the entrance side determined by `direction`.
- `in_B` – instance of `clsLightField`, representing the input field in path B.
- `direction` – `Dir.LTR` (left-to-right) or `Dir.RTL` (right-to-left).

### Return value

`(out_A, out_B)` – two `clsLightField` objects containing the propagated fields in paths A and B on the exit side.

### Algorithm

*Left-to-right propagation (Dir.LTR):*

$$\begin{pmatrix} \text{out\_A} \\ \text{out\_B} \end{pmatrix} = \underbrace{\begin{pmatrix} \mathbf{T}_{\text{LTR}}^{A \rightarrow A} & \mathbf{T}_{\text{LTR}}^{B \rightarrow A} \\ \mathbf{T}_{\text{LTR}}^{A \rightarrow B} & \mathbf{T}_{\text{LTR}}^{B \rightarrow B} \end{pmatrix}}_{\mathbf{T}_{\text{LTR}}} \begin{pmatrix} \text{in\_A} \\ \text{in\_B} \end{pmatrix}.$$

*Right-to-left propagation (Dir.RTL):*

$$\begin{pmatrix} \text{out\_A} \\ \text{out\_B} \end{pmatrix} = \underbrace{\begin{pmatrix} \mathbf{T}_{\text{RTL}}^{A \rightarrow A} & \mathbf{T}_{\text{RTL}}^{B \rightarrow A} \\ \mathbf{T}_{\text{RTL}}^{A \rightarrow B} & \mathbf{T}_{\text{RTL}}^{B \rightarrow B} \end{pmatrix}}_{\mathbf{T}_{\text{RTL}}} \begin{pmatrix} \text{in\_A} \\ \text{in\_B} \end{pmatrix}.$$

The four  $N^2 \times N^2$  sub-matrices  $\mathbf{T}_{\text{LTR/RTL}}^{X \rightarrow Y}$  are obtained on demand via `.get_T_mat_tot(...)`.

### E.1.3 Full Transmission and Reflection Block Matrix

`.get_T_bmat_tot(direction)`

#### Abstract method.

Returns the  $2 \times 2$  *transmission block matrix* (see `clsBlockMatrix`) for the selected propagation direction.

### Parameters

- `direction` – `Dir.LTR` (left-to-right) or `Dir.RTL` (right-to-left).

### Behavior

For a light field discretised on an  $N \times N$  grid – i.e. with  $N^2$  modal amplitudes per path – the returned operator has dimensions  $(2N^2) \times (2N^2)$ :

$$\mathbf{T}_{\text{dir}} = \begin{pmatrix} \mathbf{T}_{\text{dir}}^{A \rightarrow A} & \mathbf{T}_{\text{dir}}^{B \rightarrow A} \\ \mathbf{T}_{\text{dir}}^{A \rightarrow B} & \mathbf{T}_{\text{dir}}^{B \rightarrow B} \end{pmatrix},$$

where each sub-matrix  $\mathbf{T}_{\text{dir}}^{X \rightarrow Y}$  maps the  $N^2$  input modes of path X to the  $N^2$  output modes of path Y.

This block matrix is used, together with its transmission counterpart from `.get_R_bmat_tot(...)`, to construct the component's scattering matrix `.S_bmat_tot` and transfer matrix `.M_bmat_tot`.

### Implementation notes

The method is declared *abstract*; concrete subclasses supply the actual computation.

`.get_R_bmat_tot(side)`

#### Abstract method.

Returns the  $2 \times 2$  *reflection block matrix* (see `clsBlockMatrix`) for the interface chosen by `side`.

#### Parameters

- `side` – `Side.LEFT` or `Side.RIGHT`; selects the physical interface at which the reflection occurs.

#### Behavior

For a total grid of  $N \times N$  pixels the returned operator has size  $(2N^2) \times (2N^2)$ :

$$\mathbf{R}_{\text{side}} = \begin{pmatrix} \mathbf{R}_{\text{side}}^{A \rightarrow A} & \mathbf{R}_{\text{side}}^{B \rightarrow A} \\ \mathbf{R}_{\text{side}}^{A \rightarrow B} & \mathbf{R}_{\text{side}}^{B \rightarrow B} \end{pmatrix},$$

where each sub-matrix  $\mathbf{R}_{\text{side}}^{X \rightarrow Y}$  maps the  $N^2$  modal amplitudes incident in path  $X$  to the  $N^2$  reflected amplitudes in path  $Y$ .

This block matrix is used, together with its transmission counterpart from `.get_T_bmat_tot(...)`, to construct the component's scattering matrix `.S_bmat_tot` and transfer matrix `.M_bmat_tot`.

### Implementation notes

The method is declared *abstract*; concrete subclasses supply the actual computation.

## E.1.4 Transmission and Reflection Matrix per Path

`.get_T_mat_tot(direction, path1, path2)`

#### Abstract method.

Returns the *channel-resolved transmission matrix* at total resolution.

#### Parameters

- `direction` – `Dir.LTR` (left-to-right) or `Dir.RTL` (right-to-left); selects the propagation direction.
- `path1` – incoming path, `Path.A` or `Path.B`.
- `path2` – outgoing path, `Path.A` or `Path.B`.

#### Behavior

For a light field sampled on an  $N \times N$  grid the method returns an  $N^2 \times N^2$  matrix  $\mathbf{T}_{\text{direction}}^{\text{path1} \rightarrow \text{path2}}$  that maps the modal amplitudes of an incident field (in `path1`) to those of the transmitted field (in `path2`) for the specified propagation direction.

These channel-resolved matrices are the building blocks of the  $2 \times 2$  transmission block matrices produced by `.get_T_bmat_tot(...)` and ultimately feed into the global scattering block matrix `.S_bmat_tot`.

### Implementation notes

The method is declared *abstract*; concrete subclasses supply the actual computation.

`.get_R_mat_tot(side, path1, path2)`

#### Abstract method.

Returns the *channel-resolved reflection matrix* at total resolution.

#### Parameters

- `side` – `Side.LEFT` or `Side.RIGHT`; specifies the interface where the reflection occurs.
- `path1` – incoming path, `Path.A` or `Path.B`.
- `path2` – outgoing path, `Path.A` or `Path.B`.

#### Behavior

For a light field sampled on an  $N \times N$  grid the method returns an  $N^2 \times N^2$  matrix  $\mathbf{R}_{\text{side}}^{\text{path1} \rightarrow \text{path2}}$  that maps the modal amplitudes of the incident field (in `path1`) to those of the reflected field (in `path2`) at the specified side.

These channel-resolved matrices are the building blocks of the  $2 \times 2$  reflection block matrices produced by `.get_R_bmat_tot(...)` and ultimately feed into the global scattering block matrix `.S_bmat_tot`.

### Implementation notes

The method is declared *abstract*; concrete subclasses supply the actual computation.

## E.1.5 Scattering and Transfer Block Matrices

`.S_bmat_tot`

#### Read-only property.

Returns the component's scattering matrix (*S-matrix*) as a  $4 \times 4$  block matrix.

#### Behavior

At total resolution the S-matrix has two equivalent views. First, a compact  $2 \times 2$  block representation

$$\mathbf{S} = \begin{pmatrix} \mathbf{R}_L & \mathbf{T}_{RTL} \\ \mathbf{T}_{LTR} & \mathbf{R}_R \end{pmatrix},$$

where each quadrant is by itself a  $2 \times 2$  block matrix that can be retrieved with `.get_R_bmat_tot(...)` and `.get_T_bmat_tot(...)`. By expanding each quadrant into its path-resolved sub-matrices, we get the full  $4 \times 4$  block matrix structure

$$\mathbf{S} = \begin{pmatrix} \mathbf{R}_L^{A \rightarrow A} & \mathbf{R}_L^{B \rightarrow A} & \mathbf{T}_{RTL}^{A \rightarrow A} & \mathbf{T}_{RTL}^{B \rightarrow A} \\ \mathbf{R}_L^{A \rightarrow B} & \mathbf{R}_L^{B \rightarrow B} & \mathbf{T}_{RTL}^{A \rightarrow B} & \mathbf{T}_{RTL}^{B \rightarrow B} \\ \mathbf{T}_{LTR}^{A \rightarrow A} & \mathbf{T}_{LTR}^{B \rightarrow A} & \mathbf{R}_R^{A \rightarrow A} & \mathbf{R}_R^{B \rightarrow A} \\ \mathbf{T}_{LTR}^{A \rightarrow B} & \mathbf{T}_{LTR}^{B \rightarrow B} & \mathbf{R}_R^{A \rightarrow B} & \mathbf{R}_R^{B \rightarrow B} \end{pmatrix}.$$

Here, all sub-matrices originate from `.get_R_mat_tot(...)` and `.get_T_mat_tot(...)`.

Internally the property returns an instance of `clsBlockMatrix`. If the legacy option `clsCavity.use_bmatrix_class` is disabled a plain nested list may be produced, but this fallback is deprecated.

### Performance notes

On first access, the required channel-resolved matrices are computed (and cached); later calls merely assemble the S-matrix from these cached blocks.

### Typical use case

Provides the input for the transfer matrix `.M_bmat_tot` and allows direct analysis of the component's complete input–output behaviour.

#### `.M_bmat_tot`

##### Read-only property.

Returns the component's transfer matrix (*M-matrix*) as a  $4 \times 4$  block matrix.

##### Behavior

The M-matrix relates the total field on one side of the component to the other. For the 4-port case we use the convention introduced in Eq. (E.6):

$$\begin{pmatrix} \mathbf{a}_{\text{out}}^{\text{left}} \\ \mathbf{a}_{\text{in}}^{\text{left}} \end{pmatrix} = \mathbf{M} \begin{pmatrix} \mathbf{a}_{\text{in}}^{\text{right}} \\ \mathbf{a}_{\text{out}}^{\text{right}} \end{pmatrix}$$

Each block  $\mathbf{M}_{ij}$  is itself a  $2 \times 2$  block matrix (see `clsBlockMatrix`) whose entries are  $N^2 \times N^2$  mode-coupling matrices. Hence, the total transfer matrix is a  $4 \times 4$  block matrix, linking the left and the right side of the component:

$$\begin{pmatrix} \mathbf{a}_{\text{out}}^{\text{left,A}} \\ \mathbf{a}_{\text{out}}^{\text{left,B}} \\ \mathbf{a}_{\text{in}}^{\text{left,A}} \\ \mathbf{a}_{\text{in}}^{\text{left,B}} \end{pmatrix} = \begin{pmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} & \mathbf{M}_{13} & \mathbf{M}_{14} \\ \mathbf{M}_{21} & \mathbf{M}_{22} & \mathbf{M}_{23} & \mathbf{M}_{24} \\ \mathbf{M}_{31} & \mathbf{M}_{32} & \mathbf{M}_{33} & \mathbf{M}_{34} \\ \mathbf{M}_{41} & \mathbf{M}_{42} & \mathbf{M}_{43} & \mathbf{M}_{44} \end{pmatrix} \begin{pmatrix} \mathbf{a}_{\text{in}}^{\text{right,A}} \\ \mathbf{a}_{\text{in}}^{\text{right,B}} \\ \mathbf{a}_{\text{out}}^{\text{right,A}} \\ \mathbf{a}_{\text{out}}^{\text{right,B}} \end{pmatrix}. \quad (\text{E.7})$$

##### Computation

`.M_bmat_tot` converts the scattering matrix  $\mathbf{S}$  obtained from the reflection and transmission operators into  $\mathbf{M}$  via the analytic formula [4]

$$\mathbf{M} = \begin{pmatrix} \mathbf{S}_{12} - \mathbf{S}_{11}\mathbf{S}_{21}^{-1}\mathbf{S}_{22} & \mathbf{S}_{11}\mathbf{S}_{21}^{-1} \\ -\mathbf{S}_{21}^{-1}\mathbf{S}_{22} & \mathbf{S}_{21}^{-1} \end{pmatrix},$$

where each  $\mathbf{S}_{ij}$  is a  $2 \times 2$  block. Matrix inversion and multiplication are carried out by the block-matrix helper routines listed in *Global Functions for Processing Block Matrices*, e.g. `bmat2_inv(...)` for the  $2 \times 2$  inversion.

##### Caching behaviour

On first access the M-matrix is generated and may be cached in memory and/or on disk, depending on `.mem_cache_M_bmat` and `.file_cache_M_bmat`. Subsequent accesses return the cached blocks directly, avoiding recomputation.

**Typical use case**

The M-matrix is the primary object chained in cavity simulations. `clsCavity2path` queries `.M_bmat_tot` for every component when assembling the global cavity transfer operator.

`.inv_M_bmat_tot`

**Read-only property.**

Returns the inverse of the component's transfer matrix (*M-matrix*) as a  $4 \times 4$  block matrix.

**Behavior**

`.inv_M_bmat_tot` returns the inverse transfer matrix, computing it on first access (via `.calc_inv_M_bmat_tot()`) and caching it in memory and/or on disk according to `.mem_cache_M_bmat` and `.file_cache_M_bmat`. Subsequent accesses return the cached version.

**Typical use case**

The inverse M-matrix is used internally by `clsCavity2path` by the method `clsCavity2path.calc_bulk_field_from_left(...)` and also by the method `clsCavity2path.calc_bulk_field_from_right(...)` to reconstruct the field at intermediate points inside a cavity from the known input or output fields.

`.calc_inv_M_bmat_tot()`

**Method.**

Explicitly computes and caches the inverse transfer matrix (inverse *M-matrix*) for this component.

**Behavior**

Calling `.calc_inv_M_bmat_tot()` forces the calculation of the inverse  $4 \times 4$  block transfer matrix and stores the result in memory. After this call, a subsequent access to `.inv_M_bmat_tot` will return the cached matrix immediately without recomputation.

**Typical use case**

Use this method when you need to ensure that the inverse transfer matrix is available in cache.

### E.1.6 Distances

`.get_dist_phys()`

**Abstract method.**

Returns the physical propagation distance through the component, given separately for the two spatial paths.

**Behavior**

The method returns a tuple  $(d_{\text{Path.A}}, d_{\text{Path.B}})$  with both entries expressed in meters. For many elements the distances are  $0\text{m}$  (e.g. ideal thin mirrors or lenses); for components that model actual free-space or material propagation, each value equals the physical

path length encountered by the respective beam.

### `.get_dist_opt()`

#### Abstract method.

Returns the *optical* propagation distance through the component, given separately for the two spatial paths.

#### Behavior

The method returns a tuple  $(d_{\text{opt},\text{Path.A}}, d_{\text{opt},\text{Path.B}})$  with both entries expressed in meters. For pure free-space propagation the optical and physical distances coincide,  $d_{\text{opt}} = d_{\text{phys}}$ . If a path contains material of refractive index  $n$ , the value is scaled accordingly,  $d_{\text{opt}} = n \cdot d_{\text{phys}}$ .

## E.2 Class `clsBeamSplitterMirror`

The concrete class `clsBeamSplitterMirror` models an ideal, non-polarising beam-splitter. It derives from `clsOptComponent4port`, which itself extends the common base `clsOptComponent`; therefore, every method and property defined for those super-classes is available here without modification.

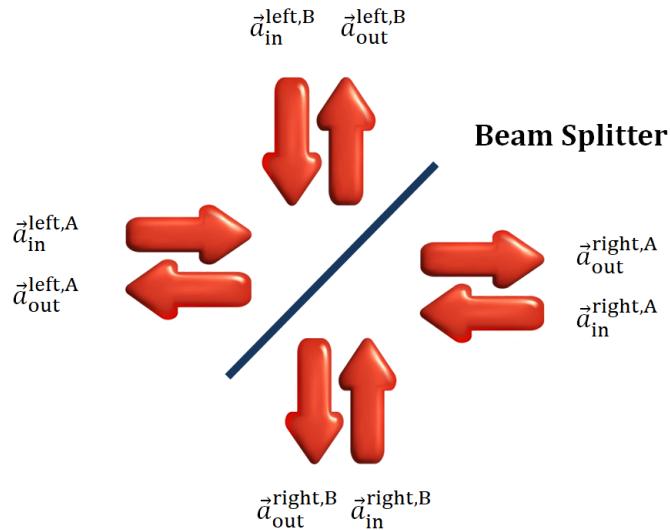


Figure E.2: Conceptual 4-port view of a beam-splitter mirror. Each side supports two spatial paths (A and B), and the element couples them according to its reflection (R) and transmission (T) coefficients.

The class stores and exposes the power reflectivity `.R` and transmissivity `.T`. The underlying complex field coefficients ( $r_L, r_R, t_{LTR}, t_{RTL}$ ) are maintained internally in either *symmetric-phase* or *sign-convention* mode, selectable via `.set_phys_behavior(...)`. These complex coefficients are not directly accessible but are used to build the reflection and transmission blocks returned by the public interface.

With all path mixing limited to simple identity or exchange operations, the class `clsBeamSplitterMirror` provides a computationally lightweight yet fully compliant 4-port component.

Reflection always swaps the two spatial paths  $A$  and  $B$  while transmission never mixes them. Consequently, the full  $4 \times 4$  scattering matrix contains only eight non-zero sub-matrices:

$$\mathbf{S} = \begin{pmatrix} 0 & \mathbf{R}_L^{B \rightarrow A} & \mathbf{T}_{RTL}^{A \rightarrow A} & 0 \\ \mathbf{R}_L^{A \rightarrow B} & 0 & 0 & \mathbf{T}_{RTL}^{B \rightarrow B} \\ \mathbf{T}_{LTR}^{A \rightarrow A} & 0 & 0 & \mathbf{R}_R^{B \rightarrow A} \\ 0 & \mathbf{T}_{LTR}^{B \rightarrow B} & \mathbf{R}_R^{A \rightarrow B} & 0 \end{pmatrix},$$

where a superscript  $A \rightarrow B$  denotes “input path  $A$ , output path  $B$ ” (and analogously for the other combinations).

### E.2.1 Initialization

`clsBeamSplitterMirror(name, cavity)`

#### Constructor.

Creates a new beam-splitter component and registers it with the parent cavity.

#### Parameters

- `name` – Human-readable identifier (string). Appears in log messages and cache files.
- `cavity` – Reference to the `clsCavity2path` object to which this component belongs.

#### Behavior

The component starts with power reflection `.R = 1` and, consequently, power transmissivity `.T = 0`.

### E.2.2 Reflectivity and Transmissivity Parameters

`.R`

#### Read/write property.

Specifies the *power* reflectivity  $R \in [0, 1]$  of the beam-splitter. When  $R$  is set, its value is clamped to the range  $[0, 1]$  and the power transmissivity is updated automatically via `.T = 1 - .R`.

#### Internal field coefficients

The complex amplitude coefficients  $(r_L, r_R, t_{LTR}, t_{RTL})$  are derived from  $R$  according to the phase convention selected with `.set_phys_behavior(...)`:

- *Symmetric-phase mode* (`sym_phase = True`, default):

$$r_L = r_R = r = -R - i\sqrt{R}\sqrt{1-R}, \quad t_{LTR} = t_{RTL} = t = 1 + r.$$

- *Sign-convention mode* (`sym_phase = False`):

$$r_L = +\sqrt{R}, \quad r_R = -\sqrt{R}, \quad t_{LTR} = t_{RTL} = +\sqrt{1-R}.$$

These complex coefficients are stored internally and used to build the reflection and transmission matrices exposed by the public interface.

### `.T`

#### Read/write property.

Specifies the *power* transmissivity  $T \in [0, 1]$  of the beam-splitter. When  $T$  is set, its value is clamped to the range  $[0, 1]$  and the power reflectivity is updated automatically via

$$\mathbf{.R} = 1 - \mathbf{T}.$$

#### Internal field coefficients

Let  $R = 1 - T$ . Depending on the phase convention chosen with `.set_phys_behavior(...)`, the complex amplitude coefficients ( $r_L, r_R, t_{LTR}, t_{RTL}$ ) are derived as follows:

- *Symmetric-phase mode* (`sym_phase = True`, default):

$$r_L = r_R = r = -R - i\sqrt{R}\sqrt{T}, \quad t_{LTR} = t_{RTL} = t = 1 + r.$$

- *Sign-convention mode* (`sym_phase = False`):

$$r_L = +\sqrt{R}, \quad r_R = -\sqrt{R}, \quad t_{LTR} = t_{RTL} = +\sqrt{T}.$$

These internal coefficients are used to build the reflection and transmission matrices returned by the public interface.

### `.set_phys_behavior(sym_phase)`

#### Method.

Selects the phase convention used for the beam-splitter's complex reflection and transmission coefficients and, by extension, for its scattering matrix  $\mathbf{S}$ .

#### Parameters

- `sym_phase` – Boolean flag `True`: “symmetric-phase” convention (default) `False`: “text-book” real-valued convention

#### Why two conventions?

Energy conservation requires the scattering matrix to be *unitary*. In the single-mode limit one may write

$$\mathbf{S} = \begin{pmatrix} r & t \\ t & r \end{pmatrix}, \quad r, t \in \mathbb{C}, \quad R = |r|^2.$$

#### Symmetric-phase convention (`sym_phase = True`)

$$r = -R - i\sqrt{R}\sqrt{1 - R}, \quad t = 1 + r.$$

Both facets impose the same phase on reflected beams, which is often the physically intuitive choice.

**Real-valued (“text-book”) convention (`sym_phase = False`)**

$$\mathbf{S} = \begin{pmatrix} r & t \\ t & -r \end{pmatrix}, \quad r, t \in [0, 1],$$

so that  $r_R = -r_L$  while  $t$  remains real and positive.

### Generalisation to the 4-port case

In block-matrix notation the single-mode coefficients become diagonal  $N^2 \times N^2$  matrices, giving

$$\mathbf{S} = \begin{pmatrix} \mathbf{R}_L & \mathbf{T}_{RL} \\ \mathbf{T}_{LR} & \mathbf{R}_R \end{pmatrix} = \begin{pmatrix} \mathbf{R} & \mathbf{T} \\ \mathbf{T} & -\mathbf{R} \end{pmatrix}, \quad \mathbf{R}, \mathbf{T} \in \mathbb{R}^{N^2 \times N^2}.$$

### Behavior when called

Updates the internal complex coefficients according to the chosen convention and clears any cached block matrices so they are rebuilt with the new phase behaviour.

## E.2.3 Full Transmission and Reflection Block Matrix

`.get_T_bmat_tot(direction)`

### Method.

Returns the  $2 \times 2$  transmission block matrix for the beam-splitter

### Parameter

`direction` - `Dir.LTR` (left-to-right) or `Dir.RTL` (right-to-left);

### Behaviour

Transmission never mixes the spatial paths:

$$\mathbf{T}_{LTR} = \begin{pmatrix} t_{LTR} & 0 \\ 0 & t_{LTR} \end{pmatrix}, \quad \mathbf{T}_{RTL} = \begin{pmatrix} t_{RTL} & 0 \\ 0 & t_{RTL} \end{pmatrix}.$$

### Returns

By default the method returns a `clsBlockMatrix` instance. The legacy nested-list variant is only produced when `clsCavity.use_bmatrix_class` is set to `False`; that fallback is deprecated.

`.get_R_bmat_tot(side)`

**Method.** Returns the  $2 \times 2$  reflection block matrix for the beam-splitter’s `Side.LEFT` or `Side.RIGHT` facet.

### Parameter

`side` - `Side.LEFT` or `Side.RIGHT`.

**Behaviour**

Reflection always swaps the two spatial paths:

$$\mathbf{R}_L = \begin{pmatrix} 0 & r_L \\ r_L & 0 \end{pmatrix}, \quad \mathbf{R}_R = \begin{pmatrix} 0 & r_R \\ r_R & 0 \end{pmatrix},$$

**Returns**

By default the method returns a `clsBlockMatrix` instance. The legacy nested-list variant is only produced when `clsCavity.use_bmatrix_class` is set to `False`; that fallback is deprecated.

**E.2.4 Transmission and Reflection Matrix per Path**

`.get_T_mat_tot(direction, path1, path2)`

**Method.**

Returns the *channel-resolved* transmission coefficient for a specified propagation direction and path combination.

**Parameters**

- `direction` – `Dir.LTR` (left-to-right) or `Dir.RTL` (right-to-left).
- `path1` – incoming path, `Path.A` or `Path.B`.
- `path2` – outgoing path, `Path.A` or `Path.B`.

**Behaviour**

Transmission occurs *only* when the incoming and outgoing paths are identical:

$$\mathbf{T}_{\text{direction}}^{\text{A} \rightarrow \text{B}} = \mathbf{T}_{\text{direction}}^{\text{B} \rightarrow \text{A}} = 0, \quad \mathbf{T}_{\text{LTR}}^{\text{A} \rightarrow \text{A}} = \mathbf{T}_{\text{LTR}}^{\text{B} \rightarrow \text{B}} = t_{\text{LTR}}, \quad \mathbf{T}_{\text{RTL}}^{\text{A} \rightarrow \text{A}} = \mathbf{T}_{\text{RTL}}^{\text{B} \rightarrow \text{B}} = t_{\text{RTL}}.$$

**Return value**

A complex scalar:

$$\begin{cases} 0, & \text{if } \text{path1} \neq \text{path2}, \\ t_{\text{LTR}}, & \text{if } \text{direction} = \text{LTR} \text{ and paths identical,} \\ t_{\text{RTL}}, & \text{if } \text{direction} = \text{RTL} \text{ and paths identical.} \end{cases}$$

The coefficients  $t_{\text{LTR}}$  and  $t_{\text{RTL}}$  are governed by the current power transmissivity `.T` and the phase convention selected via `.set_phys_behavior(...)`.

`.get_R_mat_tot(side, path1, path2)`

**Method.**

Returns the *channel-resolved* reflection coefficient for a specified interface and path combination.

### Parameters

- `side` – `Side.LEFT` or `Side.RIGHT`; chooses the facet on which the reflection occurs.
- `path1` – incoming path, `Path.A` or `Path.B`.
- `path2` – outgoing path, `Path.A` or `Path.B`.

### Behaviour

The beam-splitter reflects *only* between *different* paths; identical path pairs yield zero:

$$\mathbf{R}_{\text{side}}^{\text{A} \rightarrow \text{A}} = \mathbf{R}_{\text{side}}^{\text{B} \rightarrow \text{B}} = 0, \quad \mathbf{R}_L^{\text{B} \rightarrow \text{A}} = \mathbf{R}_L^{\text{A} \rightarrow \text{B}} = r_L, \quad \mathbf{R}_R^{\text{B} \rightarrow \text{A}} = \mathbf{R}_R^{\text{A} \rightarrow \text{B}} = r_R.$$

### Returns

A complex scalar:

$$\begin{cases} 0, & \text{if } \text{path1} = \text{path2}, \\ r_L, & \text{if } \text{side} = \text{LEFT} \text{ and paths differ,} \\ r_R, & \text{if } \text{side} = \text{RIGHT} \text{ and paths differ.} \end{cases}$$

The coefficients  $r_L$  and  $r_R$  are governed by the phase convention chosen via `.set_phys_behavior(...)` and are determined by the current power reflectivity `.R`.

## E.2.5 Distances

### `.get_dist_phys()`

#### Method.

Reports the physical propagation distance through the beam-splitter for each path.

#### Return value

A tuple  $(0, 0)$  (in meters), indicating that the idealized mirror is treated as an optically thin element with no geometrical length in either path.

### `.get_dist_opt()`

#### Method.

Reports the optical propagation distance through the beam-splitter for each path.

#### Returns

A tuple  $(0, 0)$  (in metres). Because the element is modelled as loss-less and infinitesimally thin, the optical path length is identical to the physical length, i.e. zero in both paths.

## E.2.6 I/O Representation Preference Flags

### `.k_space_in_dont_care`

#### Read-only property.

States whether the beam-splitter has a preference regarding the representation (position space or  $k$ -space) of the *input* field.

**Behavior**

Always returns `True`, meaning the component accepts input fields in either representation without preference.

**.k\_space\_in\_prefer****Read-only property.**

Indicates which representation the component would prefer for the *input* field.

**Behavior**

Always returns `False`. Because `.k_space_in_dont_care` is `True`, this preference flag is effectively ignored.

**.k\_space\_out\_dont\_care****Read-only property.**

States whether the beam-splitter has a preference regarding the representation (position space or *k*-space) of the *output* field.

**Behavior**

Always returns `True`, meaning the component can deliver output fields in either representation without preference.

**.k\_space\_out\_prefer****Read-only property.**

Indicates which representation the component would prefer for the *output* field.

**Behavior**

Always returns `False`. Because `.k_space_out_dont_care` is `True`, this preference flag is effectively ignored.

### E.3 Class `clsOptComponentAdapter`

`clsOptComponentAdapter` converts one or two *two-port* elements into a fully fledged *four-port* component by plugging them into the individual spatial paths (A and B) of the 4-port model. The class derives from `clsOptComponent4port` and therefore inherits the complete scattering/transfer-matrix interface.

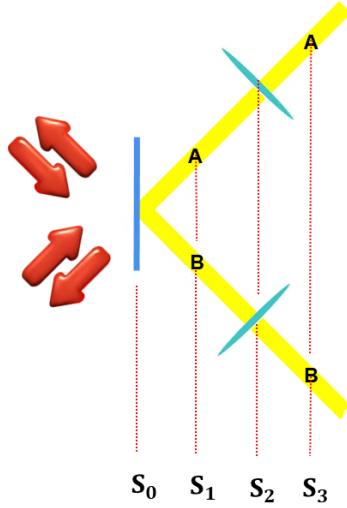


Figure E.3: Illustrative cavity segment. A beam-splitter mirror, represented by `clsBeamSplitterMirror` with  $4 \times 4$  scattering block matrix  $\mathbf{S}_0$  is followed by identical optical sub-chains in paths A and B: free-space propagation, a thin lens, and more propagation. Each pair of these two-port components is wrapped by a `clsOptComponentAdapter` that exposes a  $4 \times 4$  block scattering matrix ( $\mathbf{S}_1, \mathbf{S}_2, \dots$ ).

Using `clsOptComponentAdapter` you can insert arbitrary chains of two-port components between genuine four-port elements such as the beam-splitter ( $\mathbf{S}_0$ ). For every pair (or single) of two-port devices attached to paths A and B, the adapter constructs a block-diagonal  $4 \times 4$  scattering matrix ( $\mathbf{S}_1, \mathbf{S}_2, \dots$ ). Each section of the cavity - regardless of whether it originated as a two-port or four-port element - thus exposes a uniform four-port interface, enabling `clsCavity2path` to treat all segments in exactly the same way.

As a concrete example, consider the first adapter segment (immediately after the beam-splitter in Fig. E.3). With identical `clsPropagation` elements inserted in both paths, the resulting four-port scattering block matrix becomes

$$\mathbf{S}_1 = \begin{pmatrix} 0 & 0 & \mathbf{T}_{RTL}^{A \rightarrow A} & 0 \\ 0 & 0 & 0 & \mathbf{T}_{RTL}^{B \rightarrow B} \\ \mathbf{T}_{LTR}^{A \rightarrow A} & 0 & 0 & 0 \\ 0 & \mathbf{T}_{LTR}^{B \rightarrow B} & 0 & 0 \end{pmatrix}.$$

Only two block diagonals are populated:

$\mathbf{T}_{LTR}^{A \rightarrow A}$  and  $\mathbf{T}_{LTR}^{B \rightarrow B}$  come from the 2-port component method `clsPropagation.T_LTR_mat_tot` and are the left-to-right transmission matrices returned by for the propagation in paths A and B, respectively.

$\mathbf{T}_{RTL}^{A \rightarrow A}$  and  $\mathbf{T}_{RTL}^{B \rightarrow B}$  come from `clsPropagation.T_RTL_mat_tot` and describe the reverse (right-to-left) propagation.

`clsOptComponentAdapter` collects these two-port matrices, places them in the correct quadrants to form  $\mathbf{S}_1$ . Subsequent adapters ( $\mathbf{S}_2, \mathbf{S}_3, \dots$ ) are built in the same way, allowing `clsCavity2path` to treat every inserted segment as a standard four-port component.

After assembling  $\mathbf{S}_1$ , `clsOptComponentAdapter` allows to access the corresponding  $4 \times 4$  transfer block matrix  $\mathbf{M}_1$  via the inherited property `clsOptComponent4port.M_bmat_tot`. These  $\mathbf{M}$ -matrices are what `clsCavity2path` ultimately chains together, so every adapter segment integrates seamlessly with the rest of the four-port components in the cavity model.

### E.3.1 Initialization

`clsOptComponentAdapter(name, cavity)`

#### Constructor.

Constructs an `clsOptComponentAdapter` placeholder that can later host one or two two-port sub-components.

#### Parameters

- `name` – Optional string identifier. If left empty, the adapter builds a composite name from its sub-components (e.g. “Prop in path A, Lens in path B”).
- `cavity` – Reference to the `clsCavity2path` object to which this component belongs

#### Behavior

1. Initializes with no two-port components assigned to paths A or B. In this state the adapter acts as an identity element: zero reflection, unit transmission, and zero distance.
2. Registers itself as a four-port component through the constructor of the base-class `clsOptComponent4port`, which assigns a unique index (`.idx`) that identifies the adapter within its cavity.
3. Whenever a two-port component is later attached via `.connect_component(...)`, it is silently connected to the cavity as an auxiliary element and given its own index (`idx + 1000` for path A, `idx + 2000` for path B).

### E.3.2 Component Management

`.connect_component(component, path)`

#### Method.

Assigns (or removes) a two-port sub-component in the specified path.

#### Parameters

- `component` – Instance of any class derived from `clsOptComponent2port`, or `None` to detach an existing component.
- `path` – `Path.A` or `Path.B`; selects the spatial path that receives the sub-component.

**Behavior**

1. If the adapter already belongs to a cavity, it calls `clsCavity.clear()` to invalidate any cached global matrices.
2. Assigns the sub-component to `Path.A` or `Path.B`, replacing any previous entry.
3. When the two-port `component` is not `None` it is silently connected to the same cavity and grid. A unique auxiliary index is assigned to the two-port component: `idx + 1000` for path A, `idx + 2000` for path B (or `-2/-3` before the adapter itself has an index).

**`.component_A`****Read-only property.**

Returns the two-port component currently connected to path A, or `None` if no component is attached.

**`.component_B`****Read-only property.**

Returns the two-port component currently connected to path B, or `None` if no component is attached.

**`.name`****Read-only property.**

Provides a human-readable identifier for the adapter.

**Behavior**

- If an explicit name was supplied when the adapter was constructed, that string is returned unchanged.
- Otherwise the adapter composes a name from its sub-components:
  - `<compA>.name` in path A if only path A is populated,
  - `<compB>.name` in path B if only path B is populated, or
  - `<compA>.name` in path A, `<compB>.name` in path B when both paths are populated.
- If no sub-components are connected and no explicit name was given, the property returns an empty string.

**E.3.3 Full Transmission and Reflection Block Matrix****`.get_T_bmat_tot(direction)`****Method.**

Returns the  $2 \times 2$  transmission block matrix for the adapter in the requested propagation direction.

**Parameter**

`direction` – `Dir.LTR` (left-to-right) or `Dir.RTL` (right-to-left).

**Behavior**

The adapter simply places the path-local two-port transmission matrices on the diagonal; no cross-coupling occurs between paths:

$$\mathbf{T}_{\text{direction}} = \begin{pmatrix} \mathbf{T}_A & \mathbf{0} \\ \mathbf{0} & \mathbf{T}_B \end{pmatrix},$$

where

$$\mathbf{T}_A = \begin{cases} .T\_LTR\_{\text{mat\_tot}}, & \text{path A component and } \text{direction} = \text{LTR} \\ .T\_RTL\_{\text{mat\_tot}}, & \text{path A component and } \text{direction} = \text{RTL} \\ 1, & \text{no component in path A} \end{cases}$$

$$\mathbf{T}_B = \begin{cases} .T\_LTR\_{\text{mat\_tot}}, & \text{path B component and } \text{direction} = \text{LTR} \\ .T\_RTL\_{\text{mat\_tot}}, & \text{path B component and } \text{direction} = \text{RTL} \\ 1, & \text{no component in path B} \end{cases}$$

If a path is empty the corresponding diagonal element is the identity (1), so the adapter behaves transparently along that path.

**Returns**

By default the method returns a `clsBlockMatrix` instance; the deprecated nested-list fallback `[[T_A, 0], [0, T_B]]` is only produced when `clsCavity.use_bmatrix_class` is set to `False`.

**.get\_R\_bmat\_tot(side)****Method.**

Returns the  $2 \times 2$  reflection block matrix for the adapter at the requested interface.

**Parameter**

`side` – `Side.LEFT` or `Side.RIGHT`.

**Behavior**

The matrix is block-diagonal because the two spatial paths do not couple under reflection:

$$\mathbf{R}_{\text{side}} = \begin{pmatrix} \mathbf{R}_A & \mathbf{0} \\ \mathbf{0} & \mathbf{R}_B \end{pmatrix}.$$

- $\mathbf{R}_A$  is taken from the path-A sub-component if present (`.R_L_mat_tot` or `.R_R_mat_tot`, depending on `side`); otherwise it is  $\mathbf{0}$  (no reflection).
- $\mathbf{R}_B$  is obtained analogously from the path-B sub-component, or set to  $\mathbf{0}$  when no component is attached.

**Returns**

By default the method returns a `clsBlockMatrix` instance containing the two diagonal blocks. If `clsCavity.use_bmatrix_class` is `False` (deprecated), a nested list `[[R_A, 0], [0, R_B]]` is returned instead.

### E.3.4 Transmission and Reflection Matrix per Path

`.get_T_mat_tot(direction, path1, path2)`

#### Method.

Returns the transmission matrix between two paths for a given propagation direction.

#### Parameters

- `direction` – `Dir.LTR` (left-to-right) or `Dir.RTL` (right-to-left)
- `path1, path2` – `Path.A` or `Path.B`

#### Behavior

Only intra-path transmission is allowed (`path1 == path2`). If `path1 ≠ path2`, the method returns 0.

- If `path1 == A` and a sub-component is attached, the transmission matrix is retrieved from `clsOptComponent2port.T_LTR_mat_tot` or `.T_RTL_mat_tot`, depending on `direction`.
- If `path1 == B`, the same logic applies to the path-B sub-component.
- If no component is attached, the default return value is 1, modeling neutral transmission.

`.get_R_mat_tot(side, path1, path2)`

#### Method.

Returns the *channel-resolved* reflection matrix for a specified interface and path combination.

#### Parameters

- `side` – `Side.LEFT` or `Side.RIGHT`; chooses the facet on which the reflection occurs.
- `path1` – incoming path, `Path.A` or `Path.B`.
- `path2` – outgoing path, `Path.A` or `Path.B`.

#### Behavior

Reflection is possible only when the incoming and outgoing paths are identical. If `path1 ≠ path2` the method returns 0. Otherwise the adapter forwards the request to the appropriate two-port sub-component:

$$R = \begin{cases} \text{component\_A.R\_L\_mat\_tot}, & \text{path A, side LEFT} \\ \text{component\_A.R\_R\_mat\_tot}, & \text{path A, side RIGHT} \\ \text{component\_B.R\_L\_mat\_tot}, & \text{path B, side LEFT} \\ \text{component\_B.R\_R\_mat\_tot}, & \text{path B, side RIGHT} \\ 0, & \text{no component in the selected path.} \end{cases}$$

**Returns**

An  $N^2 \times N^2$  matrix when the selected sub-component exists; otherwise the scalar 0 (identity-like transmission behavior with no reflection).

**E.3.5 Distances****`.get_dist_phys()`****Method.**

Returns the physical propagation distance through the component, given separately for the two spatial paths.

**Behavior**

The method returns a tuple  $(d_{\text{Path.A}}, d_{\text{Path.B}})$  with both entries expressed in meters.

- If a 2-port component is attached in path A, its `.dist_phys` is returned as the first tuple element.
- If a 2-port component is attached in path B, its `.dist_phys` is returned as the second tuple element.
- If no component is attached in a given path, the respective distance defaults to 0.

**`.get_dist_opt()`****Method.**

Returns the optical path length experienced by light traversing the component, given separately for the two spatial paths.

**Behavior**

The method returns a tuple  $(d_{\text{opt}}^{\text{Path.A}}, d_{\text{opt}}^{\text{Path.B}})$  with both entries expressed in meters.

- If a 2-port component is attached in path A, its `.dist_opt` is returned as the first tuple element.
- If a 2-port component is attached in path B, its `.dist_opt` is returned as the second tuple element.
- If no component is attached in a given path, the respective value defaults to 0.

**E.3.6 I/O Representation Preference Flags****`.k_space_in_dont_care`****Read-only property.**

Indicates whether the component is indifferent to the representation (position space or *k*-space) of the *input* field.

**Behavior**

Returns `True` if neither a path-A nor a path-B sub-component is connected. Otherwise delegates to the respective sub-component in path A (if present) or path B.

- If both paths are unassigned: returns `True`.
- If only one path is assigned: returns the value of that component's `k_space_in_dont_care` property.
- If both paths are assigned: returns the value from path A.

#### `.k_space_in_prefer`

##### **Read-only property.**

Indicates whether the adapter prefers to receive the input field in *k*-space.

##### **Behavior**

If neither sub-component is connected, the adapter returns `False`, corresponding to position-space input. If only one path contains a component, the return value reflects the value of that component's `k_space_in_prefer` property. If both paths are populated, only the value of path A is considered (for consistency).

#### `.k_space_out_dont_care`

##### **Read-only property.**

Indicates whether the component is indifferent to the representation (position space or *k*-space) of the *output* field.

##### **Behavior**

Returns `True` if neither a path-A nor a path-B sub-component is connected. Otherwise delegates to the respective sub-component in path A (if present) or path B.

- If both paths are unassigned: returns `True`.
- If only one path is assigned: returns the value of that component's `k_space_out_dont_care` property.
- If both paths are assigned: returns the value from path A.

#### `.k_space_out_prefer`

##### **Read-only property.**

Indicates whether the adapter prefers to provide the output field in *k*-space.

##### **Behavior**

If neither sub-component is connected, the adapter returns `False`, corresponding to position-space input. If only one path contains a component, the return value reflects the value of that component's `k_space_out_prefer` property. If both paths are populated, only the value of path A is considered (for consistency).

## E.4 Class `clsTransmissionMixer`

`clsTransmissionMixer` provides a four-port component that *mixes* the two spatial paths without any reflection. It is conceptually the transmission-only counterpart of a beam-splitter:

$$\mathbf{R}_L = \mathbf{R}_R = \mathbf{0}, \quad \mathbf{T}_{LTR} = \mathbf{T}_{RTL} = \begin{pmatrix} t_{\text{same}} & t_{\text{mix}} \\ t_{\text{mix}} & t_{\text{same}} \end{pmatrix}.$$

The power coefficients  $T_{\text{same}}$  ( $A \rightarrow A$ ,  $B \rightarrow B$ ) and  $T_{\text{mix}}$  ( $A \rightarrow B$ ,  $B \rightarrow A$ ) satisfy  $T_{\text{same}} + T_{\text{mix}} = 1$  and can be set interchangeably via the properties `.T_same` and `.T_mix`. From them the class derives the complex amplitude coefficients  $t_{\text{same}}$  and  $t_{\text{mix}}$ .

### Phase conventions.

Two flags control how phases are assigned:

- `.sym_phase` chooses between the *symmetric-phase* prescription (identical phase on both facets) and the real-valued “text-book” convention ( $t \in \mathbb{R}$ ).
- `.refl_behavior_for_path_mixing` decides whether the path-mixing channels behave “like a reflection” (receive the phase of a reflector) or the same-path channels do.

`clsTransmissionMixer` derives from `clsOptComponent4port`, and can be used like any other four-port component. Its scattering block matrix  $\mathbf{S}$  and transfer block matrix  $\mathbf{M}$  are generated on demand through the inherited properties `clsOptComponent4port.S_bmat_tot` and `clsOptComponent4port.M_bmat_tot`.

### Typical use case.

A common application is illustrated in Fig. E.4. In this ring cavity, the blue beam-splitter on the top-left is modeled by `clsBeamSplitterMirror`, which feeds horizontal incident light into path A (top arm), and vertical incident light into path B (left arm). However, the two black  $90^\circ$  turning mirrors on the top-right and bottom-left side merely redirect the beams while keeping their logical path labels unchanged. Physically they behave like reflections, but in the simulation they must appear as *unit-transmission, phase-shifting* elements. Assigning a `clsTransmissionMixer` with `.refl_behavior_for_path_mixing=False` to simulate both of these corners accomplishes exactly that: Same-path channels ( $A \rightarrow A$ ,  $B \rightarrow B$ ) receive the reflection-like phase, while cross-path channels ( $A \rightarrow B$ ,  $B \rightarrow A$ ) remain zero. Note that a `clsBeamSplitterMirror` would mix paths A and B via its off-diagonal reflection blocks, which is not what the corner mirrors do; hence we use `clsTransmissionMixer` instead.

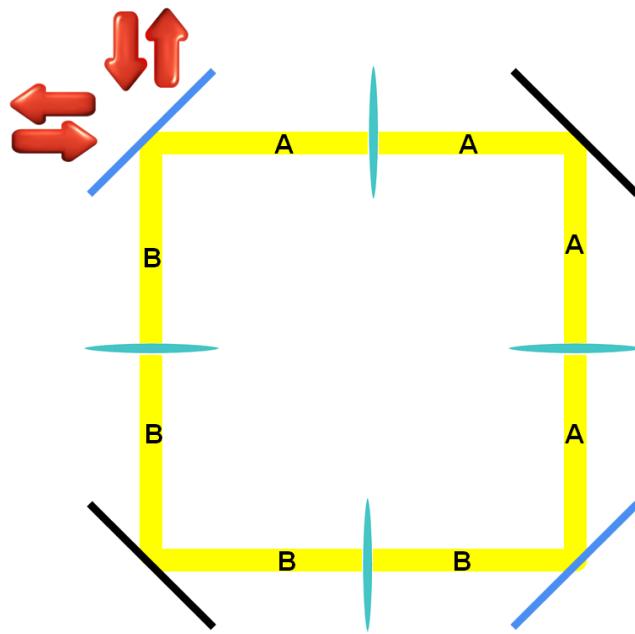


Figure E.4: Ring cavity used to illustrate `clsTransmissionMixer`. The blue element at the upper left is a `clsBeamSplitterMirror` that injects light into logical paths A (top arm) and B (left arm). The two black 90° turning mirrors (top-right and bottom-left) redirect the beams while leaving their path labels unchanged; in the simulation they are modeled by `clsTransmissionMixer` instances with `.refl_behavior_for_path_mixing=False`, providing unit transmission and a reflection-like phase shift in each path.

#### E.4.1 Initialization

```
clsTransmissionMixer(name, cavity)
```

##### Constructor.

Creates a four-port transmission component with customizable mixing and phase behavior between spatial paths A and B.

##### Parameters

- `name` - Human-readable component name.
- `cavity` - Reference to the `clsCavity2path` object to which this component belongs.

##### Behavior

- Initializes as a perfect transmitter: `.T_same` is set to 1, `.T_mix` to 0.
- Amplitude transmission coefficients are derived from these power values based on phase convention settings.
- Phase behavior is set to symmetric mode (`.sym_phase = True`) and configured to assign reflection-like phases to path-mixing channels. (`.refl_behavior_for_path_mixing = True`).

- Registers the component as a `clsOptComponent4port`, including assignment of a unique index `.idx` within the cavity.

#### E.4.2 Transmission Coefficients

##### `.T_same`

###### **Read/write property.**

Specifies the power transmission between *identical* paths: A→A and B→B.

###### **Behavior**

- Automatically clamps values to the range [0, 1]. If an out-of-range value is assigned, a warning is printed and the value is adjusted.
- When set, `.T_mix` is automatically updated to  $1 - T_{\text{same}}$ .
- The corresponding amplitude transmission coefficients are recomputed based on the settings of `.sym_phase` and `.refl_behavior_for_path_mixing`.
- Clears the parent cavity if it is assigned, to ensure matrix cache invalidation.

##### `.T_mix`

###### **Read/write property.**

Specifies the power transmission between *crossed* paths: A→B and B→A.

###### **Behavior**

- Automatically clamps values to the range [0, 1]. If an out-of-range value is assigned, a warning is printed and the value is adjusted.
- When set, `.T_same` is automatically updated to  $1 - T_{\text{mix}}$ .
- The corresponding amplitude transmission coefficients are recomputed based on the settings of `.sym_phase` and `.refl_behavior_for_path_mixing`.
- Clears the parent cavity if it is assigned, to ensure matrix cache invalidation.

#### E.4.3 Phase-Behavior Flags

##### `.sym_phase`

###### **Read/write property.**

Selects the phase convention used for computing amplitude transmission coefficients.

###### **Behavior**

- If set to `True`, applies a physically motivated “symmetric-phase” convention.
- If set to `False`, uses real-valued textbook-style amplitudes.
- See `clsBeamSplitterMirror.set_phys_behavior(...)` for details on these conventions.
- Changing the value clears the parent cavity (if present) and recomputes the amplitude coefficients.

### `.refl_behavior_for_path_mixing`

#### Read/write property.

Chooses which transmission channels receive the *reflection-type* phase and which keep the *transmission* phase. Works together with `.sym_phase`.

#### Behavior

- **True:** cross-path channels ( $A \rightarrow B$ ,  $B \rightarrow A$ ) get the reflection-type phase; same-path channels ( $A \rightarrow A$ ,  $B \rightarrow B$ ) keep the transmission phase.
- **False:** same-path channels get the reflection-type phase; cross-path channels keep the transmission phase.
- Changing either flag immediately recalculates the amplitude coefficients and clears cached matrices in the parent cavity.

#### Amplitude assignment rule

With power coefficients  $T_{\text{same}}$  and  $T_{\text{mix}}$  ( $T_{\text{same}} + T_{\text{mix}} = 1$ ) the complex amplitudes  $t_{\text{same}}$  and  $t_{\text{mix}}$  are

► **Symmetric-phase convention (`.sym_phase = True`)**

- `.refl_behavior_for_path_mixing = True`

$$t_{\text{mix}} = -T_{\text{mix}} - i\sqrt{T_{\text{mix}}}\sqrt{1 - T_{\text{mix}}}, \quad t_{\text{same}} = 1 + t_{\text{mix}}$$

- `.refl_behavior_for_path_mixing = False`

$$t_{\text{same}} = -T_{\text{same}} - i\sqrt{T_{\text{same}}}\sqrt{1 - T_{\text{same}}}, \quad t_{\text{mix}} = 1 + t_{\text{same}}$$

► **Text-book (real) convention (`.sym_phase = False`)**

- `.refl_behavior_for_path_mixing = True`

$$t_{\text{mix}} = \sqrt{T_{\text{mix}}}, \quad t_{\text{same}} = \sqrt{1 - T_{\text{mix}}}$$

- `.refl_behavior_for_path_mixing = False`

$$t_{\text{same}} = \sqrt{1 - T_{\text{same}}}, \quad t_{\text{mix}} = \sqrt{T_{\text{same}}}$$

#### E.4.4 Full Transmission and Reflection Block Matrix

### `.get_T_bmat_tot(direction)`

#### Method.

Returns the full  $2 \times 2$  amplitude transmission block matrix for either direction of propagation.

**Parameter**

- `direction` – Propagation direction: `Dir.LTR` (left-to-right) or `Dir.RTL` (right-to-left).

**Behavior**

- Same-path channels ( $A \rightarrow A$ ,  $B \rightarrow B$ ) use the internal amplitude coefficient `t_same`.
- Cross-path channels ( $A \rightarrow B$ ,  $B \rightarrow A$ ) use the internal coefficient `t_mix`.
- These amplitude values are derived from the power transmission values `.T_same` and `.T_mix` using a unitary scattering matrix construction. The phase convention is selected using `.sym_phase`, while the role of reflection-like vs. transmission-like phase behavior is controlled via `.refl_behavior_for_path_mixing`.
- The result is independent of direction and has the symmetric form:

$$\mathbf{T}_{\text{LTR}} = \mathbf{T}_{\text{RTL}} = \begin{pmatrix} t_{\text{same}} & t_{\text{mix}} \\ t_{\text{mix}} & t_{\text{same}} \end{pmatrix}.$$

- By default, the method returns a `clsBlockMatrix`.
- If `clsCavity.use_bmatrix_class` is set to `False`, a plain nested list is returned instead (deprecated).

`.get_R_bmat_tot(side)`

**Method.**

Returns the  $2 \times 2$  amplitude reflection block matrix for the specified facet.

**Parameter**

- `side` – `Side.LEFT` or `Side.RIGHT`

**Behavior**

- This component does not implement any reflection. All entries of the reflection matrix are zero.
- The return value is by default a `clsBlockMatrix` instance with all-zero blocks.
- If `clsCavity.use_bmatrix_class` is set to `False`, a plain nested list `[[0, 0], [0, 0]]` is returned instead (deprecated).
- The presence or absence of reflection is independent of the `.refl_behavior_for_path_mixing` flag, which only influences the phase of cross-path transmission.

#### E.4.5 Transmission and Reflection Matrix per Path

`.get_T_mat_tot(direction, path1, path2)`

**Method.**

Returns the amplitude transmission coefficient between two specified paths, for either left-to-right or right-to-left propagation.

**Parameters**

- `direction` – Transmission direction; must be `Dir.LTR` or `Dir.RTL`.
- `path1` – Input path: `Path.A` or `Path.B`.
- `path2` – Output path: `Path.A` or `Path.B`.

**Behavior**

- If `path1 == path2`, the returned value is the same-path transmission amplitude `t_same`.
- If `path1` and `path2` differ, the returned value is the path-mixing transmission amplitude `t_mix`.
- The method returns a scalar, not a matrix.

`.get_R_mat_tot(side, path1, path2)`

**Method.**

Returns the amplitude reflection coefficient between two specified paths for incidence from the given side.

**Parameters**

- `side` – Incidence side; must be `Side.LEFT` or `Side.RIGHT`.
- `path1` – Input path: `Path.A` or `Path.B`.
- `path2` – Output path: `Path.A` or `Path.B`.

**Behavior**

- Always returns 0.
- The `clsTransmissionMixer` does not introduce any reflection.

#### E.4.6 Distances

`.get_dist_phys()`

**Method.**

Returns the physical distances associated with the component in spatial paths A and B.

**Behavior**

Always returns (0, 0) since `clsTransmissionMixer` does not introduce any physical length into either path.

**Returns**

A tuple (`dist_A`, `dist_B`) in meters, both entries always 0.

`.get_dist_opt()`

**Method.**

Returns the optical distances associated with the component in spatial paths A and B.

**Behavior**

Always returns (0, 0) since `clsTransmissionMixer` does not introduce any optical length into either path.

**Returns**

A tuple (`dist_A`, `dist_B`) in meters, both entries always 0.

### E.4.7 I/O Representation Preference Flags

**.k\_space\_in\_dont\_care****Read-only property.**

States whether the transmission mixer has a preference regarding the representation (position space or *k*-space) of the *input* field.

**Behavior**

Always returns `True`, meaning the component accepts input fields in either representation without preference.

**.k\_space\_in\_prefer****Read-only property.**

Indicates which representation the component would prefer for the *input* field.

**Behavior**

Always returns `False`. Because `.k_space_in_dont_care` is `True`, this preference flag is effectively ignored.

**.k\_space\_out\_dont\_care****Read-only property.**

States whether the transmission mixer has a preference regarding the representation (position space or *k*-space) of the *output* field.

**Behavior**

Always returns `True`, meaning the component can deliver output fields in either representation without preference.

**.k\_space\_out\_prefer****Read-only property.**

Indicates which representation the component would prefer for the *output* field.

**Behavior**

Always returns `False`. Because `.k_space_out_dont_care` is `True`, this preference flag is effectively ignored.

## E.5 Class `clsReflectionMixer`

`clsReflectionMixer` is the reflection-only counterpart to `clsTransmissionMixer`. Where the latter couples the two spatial paths purely through transmission, `clsReflectionMixer` couples them exclusively through reflection:

$$\mathbf{T}_{\text{LTR}} = \mathbf{T}_{\text{RTL}} = \mathbf{0}, \quad \mathbf{R}_L = \mathbf{R}_R = \begin{pmatrix} r_{\text{same}} & r_{\text{mix}} \\ r_{\text{mix}} & r_{\text{same}} \end{pmatrix}.$$

The power coefficients  $R_{\text{same}}$  ( $A \rightarrow A$ ,  $B \rightarrow B$ ) and  $R_{\text{mix}}$  ( $A \rightarrow B$ ,  $B \rightarrow A$ ) satisfy  $R_{\text{same}} + R_{\text{mix}} = 1$  and are set via `.R_same` and `.R_mix`. From these, the class derives the complex amplitude coefficients  $r_{\text{same}}$  and  $r_{\text{mix}}$ , governed by the phase-convention flags `.sym_phase` and `.refl_behavior_for_path_mixing`.

Like its transmission-only sibling, `clsReflectionMixer` inherits the full four-port interface from `clsOptComponent4port`, providing on-demand scattering and transfer block matrices through the inherited properties `clsOptComponent4port.S_bmat_tot` and `clsOptComponent4port.M_bmat_tot`.

### E.5.1 Initialization

```
clsReflectionMixer(name, cavity)
```

#### Constructor.

Creates a four-port reflection component with customizable mixing and phase behavior between spatial paths A and B.

#### Parameters

- `name` – Human-readable component name.
- `cavity` – Reference to the `clsCavity2path` object to which this component belongs.

#### Behavior

- Initializes as a perfect reflector: `.R_same=1` and `.R_mix=0`.
- Amplitude reflection coefficients are derived from these power values based on phase convention settings.
- Phase behavior is set to symmetric mode (`.sym_phase=True`) and is configured to assign reflection-like phases to path-mixing channels (`.refl_behavior_for_path_mixing=True`).
- Registers the component as a `clsOptComponent4port`, obtaining a unique index `.idx` within the cavity.

### E.5.2 Reflection Coefficients

#### `.R_same`

##### **Read/write property.**

Specifies the *power reflection* for identical paths: A→A and B→B.

##### **Behavior**

- Automatically clamps values to the range [0, 1]. If an out-of-range value is assigned, a warning is printed and the value is adjusted.
- When set, `.R_mix` is automatically updated to  $1 - R_{\text{same}}$ .
- The corresponding amplitude transmission coefficients are recomputed based on the settings of `.sym_phase` and `.refl_behavior_for_path_mixing`.
- Clears the parent cavity if it is assigned, to ensure matrix cache invalidation.

#### `.R_mix`

##### **Read/write property.**

Specifies the power reflection between *crossed* paths: A→B and B→A.

##### **Behavior**

- Automatically clamps values to the range [0, 1]. If an out-of-range value is assigned, a warning is printed and the value is adjusted.
- When set, `.R_same` is automatically updated to  $1 - R_{\text{mix}}$ .
- The corresponding amplitude reflection coefficients are recomputed based on the settings of `.sym_phase` and `.refl_behavior_for_path_mixing`.
- Clears the parent cavity if it is assigned, to ensure matrix cache invalidation.

### E.5.3 Phase-Behavior Flags

#### `.sym_phase`

##### **Read/write property.**

Selects the phase convention used for computing amplitude reflection coefficients.

##### **Behavior**

- If set to `True`, applies a physically motivated “symmetric-phase” convention.
- If set to `False`, uses real-valued textbook-style amplitudes.
- See `clsBeamSplitterMirror.set_phys_behavior(...)` for details on these conventions.
- Changing the value clears the parent cavity (if present) and recomputes the amplitude coefficients.

### `.refl_behavior_for_path_mixing`

#### Read/write property.

Determines which reflection channels receive the *reflection-type* phase and which are treated with a *transmission-type* phase, in combination with the phase convention selected by `.sym_phase`.

#### Behavior

- **True:** *Cross-path* reflections ( $A \rightarrow B$ ,  $B \rightarrow A$ ) get the reflection-type phase ( $r$ ); same-path reflections ( $A \rightarrow A$ ,  $B \rightarrow B$ ) get the transmission-type phase ( $t$ ).
- **False:** Same-path reflections get the reflection-type phase, while cross-path reflections get the transmission-type phase.
- Any change to this flag (or to `.sym_phase`) immediately recalculates the amplitude coefficients and clears cached matrices in the parent cavity.

#### Phase-assignment rule

Let  $R_{\text{same}}$  be the power reflected back into the *same* path ( $A \rightarrow A$ ,  $B \rightarrow B$ ) and  $R_{\text{mix}}$  the power reflected into the *cross* path ( $A \rightarrow B$ ,  $B \rightarrow A$ ) with  $R_{\text{same}} + R_{\text{mix}} = 1$ . The complex amplitudes  $r_{\text{same}}$  and  $r_{\text{mix}}$  are obtained from these power values according to two Boolean flags

- `.sym_phase` – chooses a *symmetric-phase* (complex) or *text-book* (real) convention,
- `.refl_behavior_for_path_mixing` – decides whether the reflection-type phase is assigned to the cross-path or to the same-path channels

#### ► Symmetric-phase convention (`.sym_phase = True`)

- `refl_behavior_for_path_mixing = True`

$$r_{\text{mix}} = -R_{\text{mix}} - i\sqrt{R_{\text{mix}}} \sqrt{1 - R_{\text{mix}}}, \quad r_{\text{same}} = 1 + r_{\text{mix}}$$

- `refl_behavior_for_path_mixing = False`

$$r_{\text{same}} = -R_{\text{same}} - i\sqrt{R_{\text{same}}} \sqrt{1 - R_{\text{same}}}, \quad r_{\text{mix}} = 1 + r_{\text{same}}$$

#### ► Text-book (real) convention (`.sym_phase = False`)

Amplitudes are real and change sign on the *right* facet:

- `refl_behavior_for_path_mixing = True`

$$r_{\text{mix}} = \pm\sqrt{R_{\text{mix}}}, \quad r_{\text{same}} = \pm\sqrt{1 - R_{\text{mix}}}$$

- `refl_behavior_for_path_mixing = False`

$$r_{\text{same}} = \pm\sqrt{R_{\text{same}}}, \quad r_{\text{mix}} = \pm\sqrt{1 - R_{\text{same}}}$$

The plus sign applies to the `Side.LEFT` facet, the minus sign to `Side.RIGHT`.

Any change of these flags recomputes the amplitudes and clears cached matrices in the parent cavity to keep the global model consistent.

### E.5.4 Full Transmission and Reflection Block Matrix

`.get_T_bmat_tot(direction)`

#### Method.

Returns the full  $2 \times 2$  amplitude *transmission* block matrix for either direction of propagation.

#### Parameter

- `direction` – Propagation direction: `Dir.LTR` (left-to-right) or `Dir.RTL` (right-to-left).

#### Behavior

- `clsReflectionMixer` represents a *pure reflector*; no power is transmitted in either path.
- Consequently every entry of the transmission block matrix is zero:

$$\mathbf{T}_{\text{LTR}} = \mathbf{T}_{\text{RTL}} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}.$$

- By default the method returns a zero-filled `clsBlockMatrix`.
- If `clsCavity.use_bmatrix_class` is set to `False`, a plain nested list `[[0, 0], [0, 0]]` is returned instead (deprecated).

`.get_R_bmat_tot(side)`

#### Method.

Returns the  $2 \times 2$  amplitude reflection block matrix for the specified facet.

#### Parameter

- `side` – `Side.LEFT` or `Side.RIGHT`

#### Behavior

- The matrix is *symmetric* in path space:

$$\mathbf{R}_{\text{side}} = \begin{pmatrix} r_{\text{same}} & r_{\text{mix}} \\ r_{\text{mix}} & r_{\text{same}} \end{pmatrix}.$$

- The complex amplitudes  $r_{\text{same}}$  and  $r_{\text{mix}}$  are computed from `.R_same`, `.R_mix`, `.sym_phase`, and `.refl_behavior_for_path_mixing` as described in the phase-assignment section of `.refl_behavior_for_path_mixing`.
- In the *text-book* convention (`.sym_phase = False`) the entire matrix changes sign on the right facet:

$$\mathbf{R}_{\text{RIGHT}} = -\mathbf{R}_{\text{LEFT}}.$$

No sign flip occurs in the symmetric-phase convention.

- By default the method returns a `clsBlockMatrix`.
- If `clsCavity.use_bmatrix_class` is set to `False`, a nested list such as `[[r_same, r_mix], [r_mix, r_same]]` is returned instead (deprecated).

### E.5.5 Transmission and Reflection Matrix per Path

```
.get_T_mat_tot(direction, path1, path2)
```

#### Method.

Returns the amplitude transmission coefficient between two specified paths, for either left-to-right or right-to-left propagation.

#### Parameters

- `direction` – Transmission direction; must be `Dir.LTR` or `Dir.RTL`.
- `path1` – Input path: `Path.A` or `Path.B`.
- `path2` – Output path: `Path.A` or `Path.B`.

#### Behavior

- Always returns 0.
- The `clsReflectionMixer` does not introduce any transmission.

```
.get_R_mat_tot(side, path1, path2)
```

#### Method.

Returns the scalar amplitude reflection coefficient for the specified input/output path pair at the chosen facet.

#### Parameters

- `side` – `Side.LEFT` or `Side.RIGHT`.
- `path1` – Incoming path: `Path.A` or `Path.B`.
- `path2` – Outgoing path: `Path.A` or `Path.B`.

#### Behavior

- If `path1 == path2`, the returned value is the same-path reflection amplitude  $r_{\text{same}}$  defined in `.refl_behavior_for_path_mixing`.
- If `path1` and `path2` differ, the returned value is the cross-path reflection amplitude  $r_{\text{mix}}$  defined in `.refl_behavior_for_path_mixing`.
- In the real “text-book” convention (`.sym_phase = False`) the entire reflection amplitude acquires a minus sign on the `Side.RIGHT` facet. No sign flip occurs in the symmetric-phase convention.
- The method returns a scalar, not a matrix.

### E.5.6 Distances

```
.get_dist_phys()
```

#### Method.

Returns the physical distances associated with the component in spatial paths A and B.

**Behavior**

Always returns (0, 0) since `clsReflectionMixer` does not introduce any physical length into either path.

**Return value**

A tuple (`dist_A`, `dist_B`) in meters, both entries always 0.

**.get\_dist\_opt()****Method.**

Returns the optical distances associated with the component in spatial paths A and B.

**Behavior**

Always returns (0, 0) since `clsReflectionMixer` does not introduce any optical length into either path.

**Return value**

A tuple (`dist_A`, `dist_B`) in meters, both entries always 0.

### E.5.7 I/O Representation Preference Flags

**.k\_space\_in\_dont\_care****Read-only property.**

States whether the reflection mixer has a preference regarding the representation (position space or *k*-space) of the *input* field.

**Behavior**

Always returns `True`, meaning the component accepts input fields in either representation without preference.

**.k\_space\_in\_prefer****Read-only property.**

Indicates which representation the component would prefer for the *input* field.

**Behavior**

Always returns `False`. Because `.k_space_in_dont_care` is `True`, this preference flag is effectively ignored.

**.k\_space\_out\_dont\_care****Read-only property.**

States whether the reflection mixer has a preference regarding the representation (position space or *k*-space) of the *output* field.

**Behavior**

Always returns `True`, meaning the component can deliver output fields in either representation without preference.

**.k\_space\_out\_prefer****Read-only property.**

Indicates which representation the component would prefer for the *output* field.

**Behavior**

Always returns **False**. Because `.k_space_out_dont_care` is **True**, this preference flag is effectively ignored.

# Appendix F

## Cavity Classes

### F.1 Abstract Base Class `clsCavity`

`clsCavity` is the abstract base class for all cavity models. It provides the entire *infrastructure* - component management, wavelength handling, caching, multiprocessing, file I/O, and high-level matrix assembly - on top of which concrete subclasses such as `clsCavity1path` (single optical path) and `clsCavity2path` (dual path) build their specific optical logic.

- **Component container.**

The cavity maintains an *ordered list* `.components` of optical elements, acting as an virtual “breadboard” on which you place components in the exact sequence in which light encounters them. Every entry must derive from `clsOptComponent`:

- For a `clsCavity1path` (linear cavity) the list contains pure `clsOptComponent2port` objects such as mirrors, thin lenses, free-space propagation segments, arranged one after another along the single axis.
- For a `clsCavity2path` (ring or two-path cavity), the list must hold `clsOptComponent4port` objects. These may be genuine four-port devices (for example a `clsBeamSplitterMirror`) or pairs of matched two-port elements wrapped by `clsOptComponentAdapter`, which inserts corresponding 2-port components into paths A and B in the same order.

Methods such as `.get_component(...)`, `.get_last_component()`, and `.component_count` provide structured access to this list.

- **Wavelength and grid parameters.**

The working wavelength is set via `.Lambda` (meters) or `.Lambda_nm` (nanometers). Once  $\lambda$  is fixed, the associated `clsGrid` instance (`.grid`) determines the square sampling grid on the  $xy$ -plane. For a field-of-view of side length  $L_{\text{fov}}$ , propagation distance  $z$ , and total grid side length  $L_{\text{tot}}$ , the critical-sampling condition [3, p. 193]

$$N_{\text{tot}} = \frac{\lambda z}{L_{\text{tot}}}$$

pins down the required number of pixels per side  $N_{\text{tot}}$ . In practice,  $N_{\text{tot}}$  must be an

integer, therefore  $N_{\text{tot}}$  is rounded to the nearest integer. Because typically  $L_{\text{fov}} < L_{\text{tot}}$ , the desired FOV is embedded in a larger  $N_{\text{tot}} \times N_{\text{tot}}$  array; the extra pixels act as numerical padding and prevent numerical artefacts. After having defined the wavelength, the  $xy$ -grid can be initialized with e.g. `.grid.set_opt_res_based_on_sidelength(...)` or `.grid.set_opt_res_tot_based_on_res_fov(...)` so that the sampling criterion is met while the requested FOV is correctly embedded.

- **Transfer block matrix.**

The method `.calc_M_bmat_tot(...)` traverses the ordered component list, retrieves each component's transfer matrix, and chains them by successive matrix multiplication. The result is stored in `.M_bmat_tot`, a global block transfer matrix that represents the cumulative action of the entire cavity (or of a user-selected sub-range of components).

- **Reflection and transmission matrices.**

The class provides cavity-wide reflection matrices  $\mathbf{R}_L, \mathbf{R}_R$  and transmission matrices  $\mathbf{T}_{\text{LTR}}, \mathbf{T}_{\text{RTL}}$  through the properties `.R_L_mat_tot`, `.R_R_mat_tot`, `.T_LTR_mat_tot`, and `.T RTL_mat_tot`. These matrices are calculated on first access based on the transfer matrix

`.M_bmat_tot` by `.calc_R_L_mat_tot()`, `.calc_R_R_mat_tot()`, `.calc_T_LTR_mat_tot()`, and  
`.calc_T_RTL_mat_tot()`.

- **Caching.**

Using the properties and helper routines listed in Section F.1.10 you can activate several caching strategies to keep large matrices in RAM or on disk once they have been computed. This avoids redundant calculations and speeds up simulations.

- **Multiprocessing.**

Many numerically intensive tasks in the library are parallelised with `joblib.Parallel`. The preferred level of parallelism is stored in `.mp_pool_processes`; this integer tells every participating routine how many worker processes (CPU cores) it may launch. You can adjust the value at run time to trade memory footprint against wall-clock speed.

- **File and folder utilities.**

The properties `.folder`, `.tmp_folder`, and `.sep_char` define where output and temporary data are stored and how list entries are separated. Helper routines such as the thread-safe `.write_to_file(...)` and the matrix serializers `.save_mat(...)` / `.load_mat(...)` give a uniform, lock-protected way to save large matrices and other simulation results.

- **Single-Step mode.**

For high-resolution grids and long component chains, building the global transfer matrix in one pass can require many hours of CPU time and substantial RAM. With *Single-Step mode* (methods and properties listed in Section F.1.12) the task is split into a sequence of discrete jobs - typically one per component plus a final accumulation step. Each job runs in its own process, writes its intermediate result to a temporary file, and terminates; the next job resumes from that file, so the calculation can be paused, resumed, or distributed across machines without repeating work. Activate the feature by setting `.single_step_mode = True` and then driving the pipeline with the single-step helper routines.

- **Resonance-wavelength calculation.**

When mirrors use the “symmetric-phase” convention (see `clsMirrorBase2port.set_phys_behavior(...)`), each reflection contributes a small extra phase that shifts the true resonance wavelength away from the text-book value  $\lambda = 2L/q$ . The helper functions `.resonance_data_simple_cavity(...)` (for a two-mirror cavity) and also `.resonance_data_8f_cavity(...)` (for an 8f cavity) return the corrected resonance wavelength, wavenumber, longitudinal mode index, and free-spectral-range so that simulations can be seeded with the physically exact operating point.

Because `clsCavity` is abstract, it cannot be instantiated directly; all concrete cavity classes must implement the abstract matrix builders and resolution-conversion routines declared here.

### F.1.1 Initialization

```
clsCavity(name, full_precision=True)
```

#### Constructor (abstract).

Initialises the common infrastructure shared by all cavity subclasses.

##### Parameters

- `name` – Human-readable identifier used in log messages and file names.
- `full_precision` – If `True` (default) internal arrays use `float64/complex128`; otherwise `float32/complex64` to reduce memory.

##### Behaviour

- Selects the numeric precision and allocates an empty `clsGrid` instance.
- Creates empty containers for reflection, transmission, scattering, and transfer matrices as well as an empty component list.
- Sets sensible defaults for caching, multiprocessing, etc.

##### Typical workflow

1. Instantiate a concrete subclass (`clsCavity1path` or `clsCavity2path`).
2. Specify the working wavelength via `.Lambda` or `.Lambda_nm`.
3. Initialize the sampling grid with `.grid.set_opt_res_based_on_sidelength(...)` or `.grid.set_opt_res_tot_based_on_res_fov(...)` or a related method.
4. Append components in the optical order they are encountered.

Because `clsCavity` is abstract it cannot be instantiated directly; use one of its concrete subclasses instead.

### F.1.2 Core Configuration

```
.name
```

#### Read/write property.

Stores a human-readable identifier for the cavity instance. Modifying `.name` does *not* clear or invalidate any pre-computed matrices; only the identifier string is updated.

**.grid****Read-only property.**

Returns the `clsGrid` instance that defines every spatial and spectral coordinate system used by the cavity.

**Behavior**

- The same `clsGrid` object is shared by all components in the cavity; editing it therefore changes the sampling basis globally.
- Typically this property is used to initialize the sampling grid with `.grid.set_opt_res_based_on_sidelength(...)` or the alternative method `.grid.set_opt_res_tot_based_on_res_fov(...)` or another related method.

**.progress****Read-only property.**

Returns the shared `clsProgressPrinter` instance that produces the formatted progress and timing output for the cavity itself and for every component attached to it.

**.use\_bmatrix\_class****Read/write property.**

Enables or disables use of `clsBlockMatrix` for storing large block matrices.

**Behavior**

- Default value is `True`. All cavity-level reflection, transmission, scattering, and transfer block matrices are then stored in efficient, on-disk-swappable `clsBlockMatrix` instances.
- Setting the flag to `False` forces the code to fall back to store block matrices in nested Python lists. *This mode is deprecated and not recommended:* it increases memory usage. Also, setting the flag to `False` sets `.use_swap_files_in_bmatrix_class` to `False`.
- Changing the flag does not immediately recalculate any matrices, but you should call `.clear()` afterwards to invalidate caches built with the old storage mode.

### F.1.3 Multiprocessing

**.mp\_pool\_processes****Read/write property.**

Specifies the number of parallel worker processes used throughout the simulation for heavy computations.

**Behavior**

- Applies to multiprocessing across the entire library, including matrix operations within optical components.
- Internally used by routines relying on `joblib.Parallel`.

- Values below 1 are clamped to 1.

#### `.close_mp_pool()`

##### **Legacy method.**

This method currently has no functionality and is retained for compatibility with earlier versions of the library, in which `clsCavity` managed an internal multiprocessing pool directly. In the current implementation, parallel processing is handled via `joblib.Parallel`, and pool management is externalized.

### F.1.4 Wavelength Parameters

#### `.Lambda`

##### **Read/write property.**

Working wavelength expressed in *meters*.

##### **Behavior**

- Reading the property returns the current wavelength  $\lambda$  in m.
- Writing a new value
  - stores it internally,
  - updates the convenience property `.Lambda_nm` to  $\lambda \times 10^9$  nm,
  - updates the wavelength in every optical component of the cavity (`component.Lambda =  $\lambda$` ),
  - invokes `.clear()` to invalidate all matrices that depend on the old wavelength.

#### `.Lambda_nm`

##### **Read/write property.**

Working wavelength expressed in *nanometers*.

##### **Behavior**

- Reading the property returns the current wavelength  $\lambda$  in nm.
- Writing a new value
  - stores it internally,
  - updates the property `.Lambda` to  $\lambda/10^9$  m,
  - updates the wavelength in every optical component of the cavity (`component.Lambda_nm =  $\lambda$` ),
  - invokes `.clear()` to invalidate all matrices that depend on the old wavelength.

**.Lambda\_ref****Read/write property.**

Stores a *reference* wavelength in meters. This value is meant for bookkeeping (for example, the central wavelength around which a resonance scan is performed) and does not influence any internal calculations.

**Behavior**

- Reading the property returns the current reference wavelength  $\lambda_{\text{ref}}$  in m.
- Writing a new value updates the convenience property `.Lambda_ref_nm` to  $\lambda_{\text{ref}} \times 10^9$  nm.
- Changing `.Lambda_ref` has *no computational side effects*: no components are updated and no matrices are cleared, because the reference wavelength is informational only.

**.Lambda\_ref\_nm****Read/write property.**

Stores a *reference* wavelength in nanometers. This value is meant for bookkeeping (for example, the central wavelength around which a resonance scan is performed) and does not influence any internal calculations.

**Behavior**

- Reading the property returns the current reference wavelength  $\lambda_{\text{ref}}$  in nm.
- Writing a new value updates the property `.Lambda_ref` to  $\lambda_{\text{ref}}/10^9$  m.
- Changing `.Lambda_ref_nm` has *no computational side effects*: no components are updated and no matrices are cleared, because the reference wavelength is informational only.

### F.1.5 Folder and File Handling

**.folder****Read/write property.**

Absolute or relative path that serves as the root directory for every file created by the cavity.

**Behavior**

- If the string is empty (default), all matrix and log files are written to the current working directory.
- Setting a non-empty path prepends that path to every file name produced by `.full_file_name(...)`, thereby redirecting output from `.save_mat(...)`, `.load_mat(...)`, `.delete_mat(...)`, `.file_exists(...)`, and all component-level caching helpers to the chosen folder.

```
.full_file_name(name, idx=-1, file_extension="pkl",
    start_with_underscore=False)
```

### Helper method.

Constructs a fully qualified file name for any matrix or data object that belongs to the cavity.

#### Parameters

- `name` – Base identifier for the file (for example "M\_bmat" or "R\_L\_mat").
- `idx` – Component index; set to `-1` if the file is not tied to a specific component.
- `file_extension` – File-name extension, default "pkl".
- `start_with_underscore` – If `True`, an underscore is prepended to the generated name (useful for hidden or temporary files).

#### File-name format

- Always begins with the unique cavity identifier `.UID`.
- If `idx ≥ 0`: `{UID}-{idx}-{name}.{ext}` otherwise: `{UID}-{name}.{ext}`.
- If `start_with_underscore = True`, an underscore is prefixed.
- If `.folder` is non-empty, the path is prepended so that the file is placed in that directory.

#### Typical usage

The routine standardizes file names and paths for a family of component-level helpers, including

- `clsOptComponent.load_M_bmat_tot()`
- `clsOptComponent.save_M_bmat_tot()`
- `clsOptComponent.delete_M_bmat_tot()`
- `clsOptComponent.load_inv_M_bmat_tot()`
- `clsOptComponent.save_inv_M_bmat_tot()`

so that every stored matrix can be traced unambiguously to its cavity and, if relevant, to the component that generated it. In addition, this method is used when component transfer matrices are *cached*:

- `clsOptComponent2port.M_bmat_tot`
- `clsOptComponent4port.M_bmat_tot`
- `clsOptComponent2port.inv_M_bmat_tot`

In `clsCavity` itself this filename generator method is used by

- `.save_mat(...)`
- `.load_mat(...)`
- `.delete_mat(...)`
- `.file_exists(...)`

**.sep\_char****Read/write property.**

Specifies the character used to separate list elements when `.write_to_file(...)` appends data rows to a text file.

**Behavior**

- Default value is a comma `" , "`, yielding standard comma-separated (CSV) output.
- The character is inserted between items in the list `data` passed to `.write_to_file(...)`
- Changing `.sep_char` affects all subsequent calls to `write_to_file`. Existing files are not modified.

**F.1.6 Reflection and Transmission Matrices – Total Resolution****.R\_L\_mat\_tot****Read/write property.**

Returns the cavity's *left-incidence reflection matrix* at total resolution  $N \times N$ . When accessed for the first time, the property calls the subclass implementation of `clsCavity1path.calc_R_L_mat_tot()` or `clsCavity2path.calc_R_L_mat_tot()`, and caches the result. Subsequent reads return the stored copy.

**Meaning and dimensions**

- `clsCavity1path` computes an  $N^2 \times N^2$  array  $\mathbf{R}_L$ , the top-left entry of the global  $2 \times 2$  scattering block matrix.
- `clsCavity2path` returns a  $2 \times 2$  block matrix whose blocks are themselves  $N^2 \times N^2$  arrays; this block is the top-left quadrant of the cavity's  $4 \times 4$  scattering block matrix.

**Setter**

Assignments overwrite the cached matrix, which can be useful for injecting a pre-computed result manually.

**.R\_R\_mat\_tot****Read/write property.**

Returns the cavity's *right-incidence reflection matrix* at total resolution  $N \times N$ . When accessed for the first time, the property calls the subclass implementation of `clsCavity1path.calc_R_R_mat_tot()` or `clsCavity2path.calc_R_R_mat_tot()`, and caches the result. Subsequent reads return the stored copy.

**Meaning and dimensions**

- `clsCavity1path` computes an  $N^2 \times N^2$  array  $\mathbf{R}_R$ , the bottom-right entry of the global  $2 \times 2$  scattering block matrix.
- `clsCavity2path` returns a  $2 \times 2$  block matrix whose blocks are themselves  $N^2 \times N^2$  arrays; this block is the bottom-right quadrant of the cavity's  $4 \times 4$  scattering block

matrix.

#### Setter

Assignments overwrite the cached matrix, which can be useful for injecting a pre-computed result manually.

#### `.T_LTR_mat_tot`

##### Read/write property.

Returns the cavity's *left-to-right transmission matrix* at total resolution  $N \times N$ . When accessed for the first time, the property calls the subclass implementation of `clsCavity1path.calc_T_LTR_mat_tot()` or `clsCavity2path.calc_T_LTR_mat_tot()`, and caches the result. Subsequent reads return the stored copy.

##### Meaning and dimensions

- `clsCavity1path` computes an  $N^2 \times N^2$  array  $\mathbf{T}_{\text{LTR}}$ , the bottom-left entry of the global  $2 \times 2$  scattering block matrix.
- `clsCavity2path` returns a  $2 \times 2$  block matrix whose blocks are themselves  $N^2 \times N^2$  arrays; this block is the bottom-left quadrant of the cavity's  $4 \times 4$  scattering block matrix.

#### Setter

Assignments overwrite the cached matrix, which can be useful for injecting a pre-computed result manually.

#### `.T_RTL_mat_tot`

##### Read/write property.

Returns the cavity's *right-to-left transmission matrix* at total resolution  $N \times N$ . When accessed for the first time, the property calls the subclass implementation of `clsCavity1path.calc_T_RTL_mat_tot()` or `clsCavity2path.calc_T_RTL_mat_tot()`, and caches the result. Subsequent reads return the stored copy.

##### Meaning and dimensions

- `clsCavity1path` computes an  $N^2 \times N^2$  array  $\mathbf{T}_{\text{RTL}}$ , the top-right entry of the global  $2 \times 2$  scattering block matrix.
- `clsCavity2path` returns a  $2 \times 2$  block matrix whose blocks are themselves  $N^2 \times N^2$  arrays; this block is the top-right quadrant of the cavity's  $4 \times 4$  scattering block matrix.

#### Setter

Assignments overwrite the cached matrix, which can be useful for injecting a pre-computed result manually.

**.calc\_R\_L\_mat\_tot()****Abstract method.**

Must be implemented by subclasses to calculate the full-resolution reflection matrix for left incidence.

**Meaning**

- In `clsCavity1path` this produces the top-left entry of the cavity's  $2 \times 2$  scattering block matrix.
- In `clsCavity2path` this produces the top-left  $2 \times 2$  block of the cavity's  $4 \times 4$  scattering block matrix.

**Usage**

This method is called automatically on first access to `.R_L_mat_tot`, and its result is cached internally.

**.calc\_R\_R\_mat\_tot()****Abstract method.**

Must be implemented by subclasses to calculate the full-resolution reflection matrix for right incidence.

**Meaning**

- In `clsCavity1path` this produces the bottom-right entry of the cavity's  $2 \times 2$  scattering block matrix.
- In `clsCavity2path` this produces the bottom-right  $2 \times 2$  block of the cavity's  $4 \times 4$  scattering block matrix.

**Usage**

This method is called automatically on first access to `.R_R_mat_tot`, and its result is cached internally.

**.calc\_T\_LTR\_mat\_tot()****Abstract method.**

Must be implemented by subclasses. Calculates the cavity's *left-to-right transmission matrix* at total resolution  $N \times N$  and stores it internally.

**Meaning and dimensions**

- `clsCavity1path` computes an  $N^2 \times N^2$  array  $\mathbf{T}_{\text{LTR}}$ , the bottom-left entry of the global  $2 \times 2$  scattering block matrix.
- `clsCavity2path` returns a  $2 \times 2$  block matrix whose blocks are themselves  $N^2 \times N^2$  arrays; this block is the bottom-left quadrant of the cavity's  $4 \times 4$  scattering block matrix.

**Usage**

This method is called automatically on first access to `.T_LTR_mat_tot`, and its result is cached internally.

**.calc\_T\_RTL\_mat\_tot()****Abstract method.**

Must be implemented by subclasses. Calculates the cavity's *right-to-left transmission matrix* at total resolution  $N \times N$  and stores it internally.

**Meaning and dimensions**

- `clsCavity1path` computes an  $N^2 \times N^2$  array  $\mathbf{T}_{RTL}$ , the top-right entry of the global  $2 \times 2$  scattering block matrix.
- `clsCavity2path` returns a  $2 \times 2$  block matrix whose blocks are themselves  $N^2 \times N^2$  arrays; this block is the top-right quadrant of the cavity's  $4 \times 4$  scattering block matrix.

**Usage**

This method is called automatically on first access to `.T_RTL_mat_tot`, and its result is cached internally.

**F.1.7 Reflection and Transmission – FOV Resolution****.R\_L\_mat\_fov****Read/write property.**

Returns the cavity's left-incidence reflection matrix at *field-of-view* (FOV) resolution. When accessed for the first time, the property calls the subclass implementation of `.convert_R_L_mat_tot_to_fov()`, which accesses `.R_L_mat_tot` and calculates the equivalent reflection matrix for the smaller FOV resolution. Then the result is cached; subsequent reads return the stored copy.

**Setter**

Assignments overwrite the cached FOV-resolution reflection matrix.

**.R\_R\_mat\_fov****Read/write property.**

Returns the cavity's right-incidence reflection matrix at *field-of-view* (FOV) resolution. When accessed for the first time, the property calls the subclass implementation of `.convert_R_R_mat_tot_to_fov()`, which accesses `.R_R_mat_tot` and calculates the equivalent reflection matrix for the smaller FOV resolution. Then the result is cached; subsequent reads return the stored copy.

**Setter**

Assignments overwrite the cached FOV-resolution reflection matrix.

**.T\_LTR\_mat\_fov****Read/write property.**

Returns the cavity's left-to-right transmission matrix at *field-of-view* (FOV) resolution. When accessed for the first time, the property calls the subclass implementation of `.convert_T_LTR_mat_tot_to_fov()`, which accesses `.T_LTR_mat_tot` and calculates the

equivalent transmission matrix for the smaller FOV resolution. Then the result is cached; subsequent reads return the stored copy.

#### **Setter**

Assignments overwrite the cached FOV-resolution transmission matrix.

##### `.T_RTL_mat_fov`

#### **Read/write property.**

Returns the cavity's right-to-left transmission matrix at *field-of-view* (FOV) resolution. When accessed for the first time, the property calls the subclass implementation of `.convert_T_RTL_mat_tot_to_fov()`, which accesses `.T_RTL_mat_tot` and calculates the equivalent transmission matrix for the smaller FOV resolution. Then the result is cached; subsequent reads return the stored copy.

#### **Setter**

Assignments overwrite the cached FOV-resolution transmission matrix.

##### `.convert_R_L_mat_tot_to_fov()`

#### **Abstract method.**

Must be implemented by subclasses. Converts the left-incidence reflection matrix at *total* resolution, `.R_L_mat_tot`, to the corresponding matrix at *field-of-view* (FOV) resolution and stores the result internally.

#### **Usage**

This routine is invoked automatically on first access to `.R_L_mat_fov`; no direct user call is normally required.

##### `.convert_R_R_mat_tot_to_fov()`

#### **Abstract method.**

Must be implemented by subclasses. Converts the right-incidence reflection matrix at *total* resolution, `.R_R_mat_tot`, to the corresponding matrix at *field-of-view* (FOV) resolution and stores the result internally.

#### **Usage**

This routine is invoked automatically on first access to `.R_R_mat_fov`; no direct user call is normally required.

##### `.convert_T_LTR_mat_tot_to_fov()`

#### **Abstract method.**

Must be implemented by subclasses. Converts the left-to-right transmission matrix at *total* resolution, `.T_LTR_mat_tot`, to the corresponding matrix at *field-of-view* (FOV) resolution and stores the result internally.

#### **Usage**

This routine is invoked automatically on first access to `.T_LTR_mat_fov`; no direct user call is normally required.

**`.convert_T_RTL_mat_tot_to_fov()`****Abstract method.**

Must be implemented by subclasses. Converts the right-to-left transmission matrix at *total* resolution, `.T_RTL_mat_tot`, to the corresponding matrix at *field-of-view* (FOV) resolution and stores the result internally.

**Usage**

This routine is invoked automatically on first access to `.T_RTL_mat_fov`; no direct user call is normally required.

**F.1.8 Full Scattering and Transfer Block Matrices****`.S_bmat_tot`****Read/write property.**

Holds the cavity's *global scattering block matrix* at full resolution.

**Meaning and dimensions**

- In `clsCavity1path` the value is expected to be a  $2 \times 2$  block matrix whose blocks are each  $N^2 \times N^2$  arrays.
- In `clsCavity2path` the value is expected to be a  $4 \times 4$  block matrix with the same per-block dimensions.

**Computation**

The base class merely provides the storage slot; it does *not* calculate **S** automatically when the property is read. Each concrete cavity class must implement its own routine (if required) to compute the full scattering block matrix from the full transfer block matrix `.M_bmat_tot`.

**`.M_bmat_tot`****Read/write property.**

Holds the cavity's *global transfer block matrix* at full resolution.

**Meaning and dimensions**

- In `clsCavity1path` the value is a  $2 \times 2$  block matrix whose blocks are each  $N^2 \times N^2$  arrays.
- In `clsCavity2path` the value is a  $4 \times 4$  block matrix with the same per-block dimensions.

**Computation**

This base class does *not* create **M** automatically when the property is accessed. Instead, the full (or partial) transfer matrix can be computed by calling `.calc_M_bmat_tot(...)`, which is implemented in this base class, and chains the component transfer matrices via block-matrix multiplication and stores the result here.

```
.calc_M_bmat_tot(idx_from=0, idx_to=999)
```

### Method.

Calculates the cavity's *global transfer block matrix* `.M_bmat_tot` for the complete optical train or for a contiguous subsequence of components.

### Parameters

- `idx_from` – Index of the first component to include (default 0). The index refers to `.components`.
- `idx_to` – Index of the last component to include, *inclusive*. If the value exceeds the component count, it is clipped to the last component ( $N - 1$ ).

### Behavior

1. Sets `.single_step_mode = False` to ensure continuous execution.
2. Iterates from `idx_from` to `idx_to` and, for each component, obtains its per-component transfer matrix `.M_bmat_tot` while honoring the component's memory/file-caching flags.
3. Chains all component matrices in their respective order by block-matrix multiplication using the helper function `bmat_mul(...)`.
4. If `.use_bmatrix_class` is `True`, all intermediate and final matrices are instances of `clsBlockMatrix`.
5. Progress is reported through `clsProgressPrinter`.
6. Upon completion the fully assembled block matrix is stored in `.M_bmat_tot`.

### Typical use case

- Call with default arguments to build the transfer matrix for the entire cavity: `.calc_M_bmat_tot()`.
- Supply limits to calculate the transfer matrix for a contiguous subsequence of components, e.g. `.calc_M_bmat_tot(idx_from=3, idx_to=7)`.

## F.1.9 Component Management

```
.components
```

### Read-only property.

Returns the internal Python list that stores all optical components currently inserted in the cavity, in the exact order in which they were added.

### Meaning

Each entry is an instance of a subclass of `clsOptComponent` - a two-port element in `clsCavity1path` or a four-port element in `clsCavity2path`.

### Typical use case

Iterate over all components: `for comp in cavity.components: ...`

### Modification

The list should be treated as read-only; insertions/removals are performed

via the dedicated helper methods `clsCavity1path.add_component(component)` and `clsCavity2path.add_4port_component(component)`.

### `.get_component(i)`

#### **Method.**

Returns the optical component of type `clsOptComponent` at index `i` in `.components`.

#### **Parameter**

- `i` – Zero-based position of the desired element.

#### **Behavior**

- If `i` is within `[0, component_count - 1]`, the corresponding component instance is returned.
- For negative or out-of-range indices the method returns `None`, allowing safe probing without exception handling.

### `.get_last_component()`

#### **Method.**

Returns the *last (rightmost)* optical component in `.components`.

#### **Behavior**

- If the component list is non-empty, the method returns the element at index `.component_count - 1`.
- If the cavity contains no components, the method returns `None`.

### `.component_count`

#### **Read-only property.**

Returns the current number of optical components stored in `.components`.

#### **Return value**

An integer  $N \geq 0$ .

## F.1.10 Caching and Memory

Efficient handling of the large transfer block matrices that arise in high-resolution cavity models is critical. The framework offers four caching modes that can be influenced at run time via the properties `.use_swap_files_in_bmatrix_class`, `.allow_temp_mem_caching`, and `.allow_temp_file_caching`, or on a per-component basis, through cavity method `.activate_component_file_cache(...)` or the flags `clsOptComponent.mem_cache_M_bmat` and `clsOptComponent.file_cache_M_bmat` on component level.

### 1. No caching, no block-swapping

- `.use_swap_files_in_bmatrix_class = False`
- `.allow_temp_mem_caching = False`

- `.allow_temp_file_caching = False`
- for all components: `clsOptComponent.mem_cache_M_bmat = False`
- for all components: `clsOptComponent.file_cache_M_bmat = False`

### Behavior

Every time a component's transfer matrix  $\mathbf{M}$  is needed during `.calc_M_bmat_tot(...)`, it is recomputed from scratch and all of its blocks are kept in RAM until that component's contribution to the global chain product is complete.

#### 2. No caching, with block-swapping

- `.use_swap_files_in_bmatrix_class = True`
- `.allow_temp_mem_caching = False`
- `.allow_temp_file_caching = False`
- for all components: `clsOptComponent.mem_cache_M_bmat = False`
- for all components: `clsOptComponent.file_cache_M_bmat = False`

### Behavior

As above, every component transfer matrix is recomputed when required, but `clsBlockMatrix` off-loads blocks not currently in use to swap-files located in `.tmp_folder`. These files are deleted as soon as the respective multiplication is finished. This minimizes RAM usage but is typically slower than *temporary file caching* (configuration 4) while offering the same memory footprint. **Therefore this configuration is not recommended.**

#### 3. Temporary *RAM* caching.

- `.use_swap_files_in_bmatrix_class = False`
- `.allow_temp_mem_caching = True`
- `.allow_temp_file_caching = False`
- for all components: `clsOptComponent.mem_cache_M_bmat = False`
- for all components: `clsOptComponent.file_cache_M_bmat = False`

### Behavior:

During `.calc_M_bmat_tot(...)` the algorithm detects whether a component encountered at position  $i$  in `.components` will re-appear at a later index  $j > i$ . If so, the framework sets `clsOptComponent.mem_cache_M_bmat=True` for that instance, keeping all blocks of its transfer matrix  $\mathbf{M}$  resident in RAM until its final use. After the last occurrence the flag is cleared again. **This mode gives the *fastest* retrieval but large matrices can exhaust available RAM.**

#### 4. Temporary file caching on matrix block-swapping level. (default configuration)

- `.use_swap_files_in_bmatrix_class = True`
- `.allow_temp_mem_caching = True`
- `.allow_temp_file_caching = False`
- for all components: `clsOptComponent.mem_cache_M_bmat = False`
- for all components: `clsOptComponent.file_cache_M_bmat = False`

**Behavior:**

The same algorithm as before enables `clsOptComponent.mem_cache_M_bmat` for components that appear multiple times in `.components`. Because the flag `.use_swap_files_in_bmatrix_class` is `True`, `clsBlockMatrix` swaps matrix blocks not currently needed to temporary files in the folder `.tmp_folder`. This means that although `.allow_temp_mem_caching = True`, effectively only references to these swap files are held in RAM, which equates to file caching. The swap-files are deleted after the corresponding matrix is no longer required. This mode is slower than pure-RAM caching (configuration 3) but still markedly faster than no caching (configuration 1) while keeping the RAM footprint small. **This configuration is the standard configuration.**

**5. Temporary file caching on component level.**

- `.use_swap_files_in_bmatrix_class = False`
- `.allow_temp_mem_caching = False`
- `.allow_temp_file_caching = True`
- *for all components:* `clsOptComponent.mem_cache_M_bmat = False`
- *for all components:* `clsOptComponent.file_cache_M_bmat = False`

**Behavior:**

The same algorithm as before enables `clsOptComponent.file_cache_M_bmat` for components that appear multiple times in `.components`. This means that the transfer matrix **M** will be saved into a file in the folder specified by `.folder` (using the helper method `clsOptComponent.save_M_bmat_tot()`). After the last occurrence the flag is cleared again, and the file is deleted.

**Comparison with configuration 4:**

All matrix blocks of the currently processed **M** matrix reside in RAM temporarily (no block-level swapping), so this mode can exhaust memory on very large grids, but it keeps the cached matrices easily accessible in `.folder` for the lifetime of the computation.

**6. Permanent RAM caching on component level**

- Set the following flag for individual components:  
`clsOptComponent.mem_cache_M_bmat = True`
- The global flag `.allow_temp_mem_caching` is irrelevant for these components.

**Behavior**

When a component's transfer matrix **M** is first created it is retained in memory for the entire run, even after its last use. The matrix is never flushed or re-computed:

- If `.use_swap_files_in_bmatrix_class=False` all matrix blocks stay resident in RAM, giving fastest access at the cost of potentially large memory consumption.
- If `.use_swap_files_in_bmatrix_class=True` only lightweight references stay in RAM while individual blocks are kept in swap files – memory friendly but no longer pure-RAM caching.

**7. Permanent file caching on component level**

- Set the following flag for individual components:  
`clsOptComponent.file_cache_M_bmat = True`

- This flag can be set for multiple successive components with the cavity method `.activate_component_file_cache(...)`
- The global flag `.allow_temp_file_caching` is irrelevant for these components.

### Behavior

When a component's transfer matrix  $M$  is first created it is saved to a file in the folder specified by `.folder` by the helper method `clsOptComponent.save_M_bmat_tot()`. This file remains, even after the program has ended, and will be re-used when the program is re-started.

Choosing the right mode is a trade-off between speed, RAM usage, and disk space. For most cases, the default configuration 4 is recommended.

### `.allow_temp_mem_caching`

#### Read/write property.

Enables or disables *automatic* temporary RAM caching of per-component transfer matrices during `.calc_M_bmat_tot(...)`.

### Behavior

- **True** – While chaining the transfer matrices of the individual cavity components the algorithm detects components that will be reused later and sets their flag `clsOptComponent.mem_cache_M_bmat = True` temporarily.
  - If `.use_swap_files_in_bmatrix_class` is `False` the  $M$ -matrix blocks stay fully resident in RAM (fastest; may consume significant memory).
  - If `.use_swap_files_in_bmatrix_class` is `True` only lightweight references stay in RAM while individual blocks are swapped to temporary files in `.tmp_folder`; memory-friendly but with file I/O overhead.

After the component's last use the flag is cleared again (unless it was `True` beforehand). This corresponds to “Temporary RAM caching” (configuration 3) when block-swapping is disabled, or to a hybrid RAM/file caching mode when block-swapping is enabled (configuration 4).

- **False** – No automatic RAM caching is performed; matrices are discarded as soon as the current multiplication step finishes, unless the component's own `.mem_cache_M_bmat` flag has been set manually (see “Permanent RAM caching”, configuration 6).
- Setting this property to `True` automatically sets `.allow_temp_file_caching = False` to avoid conflicting strategies.

### Note

Components whose `.mem_cache_M_bmat` is already `True` remain cached regardless of the cavity-level setting.

### `.allow_temp_file_caching`

#### Read/write property.

Enables or disables *automatic* temporary **file** caching of per-component transfer ma-

trices during `.calc_M_bmat_tot(...)`.

### Behavior

- **True** – While chaining the transfer matrices of the individual cavity components the algorithm detects components that will be reused later and sets their flag `clsOptComponent.file_cache_M_bmat=True` temporarily.
  - Each component then writes its **M**-matrix to a temporary file in the directory given by `.folder` via `.save_M_bmat_tot()`.
  - During subsequent reuse the matrix is loaded from disk, avoiding recomputation while keeping RAM consumption low.
  - After the component's final occurrence the flag is cleared again (unless it was already **True**) and the cached file is deleted.

This corresponds to “Temporary *file caching* on component level” (configuration 5). It is mutually exclusive with automatic RAM caching.

- **False** – No automatic file caching is performed; matrices are recomputed unless the component's own `.file_cache_M_bmat` flag was set manually (see “Permanent *file caching*”, configuration 4).
- Setting this property to **True** automatically sets `.allow_temp_mem_caching=False` to avoid conflicting strategies.

### Note

Components whose `.file_cache_M_bmat` is already **True** remain file-cached irrespective of the cavity-level switch.

## `.file_cache_min_calc_time`

### Read/write property.

Sets a time threshold (in *seconds*) that determines whether a freshly computed transfer matrix **M** is written to disk when the automatic *file caching* mechanism is active.

### Behavior

- The default value is 3 s. If a component's **M**-matrix takes  $\leq 3$  s to compute, the matrix is considered “cheap” and is *not* stored even when file caching is enabled.
- If the computation time exceeds the threshold, the matrix is saved through `.save_M_bmat_tot()` (either because `.allow_temp_file_caching=True` or because the component's own `clsOptComponent.file_cache_M_bmat` flag is set).
- Writing  $t < 0$  is clipped to 0 so that every matrix qualifies for caching.

## `.activate_component_file_cache(idx_from=0, idx_to=999)`

### Method.

Enables *permanent file caching* of the transfer matrix **M** for a user-selectable subset of cavity components.

## Parameters

- `idx_from` – Index of the first component to affect (0 by default).<sup>1</sup>
- `idx_to` – Index of the last component to affect, *inclusive*. Values  $\geq \text{.component\_count}$  are clipped to the last component.

## Behavior

1. Iterates over the selected slice of `.components`.
2. Sets the per-component flag `clsOptComponent.file_cache_M_bmat = True`. Thereafter each affected component will *always* save its M-matrix to a persistent file in `.folder`, regardless of the global settings `.allow_temp_file_caching` or `.file_cache_min_calc_time`.
3. Cached files remain on disk until explicitly removed via the method `.delete_cached_component_files()` or by calling the component's own `.delete_M_bmat_tot()`.

## Typical use case

When a cavity model reuses the *same* optical component instance (e.g. a lens or beam-splitter) many times, invoking `activate_component_file_cache()` ensures that its costly transfer matrix is computed once and subsequently loaded from disk.

### `.use_swap_files_in_bmatrix_class`

#### Read/write property.

Controls whether every `clsBlockMatrix` created writes all blocks (except blocks that are mere scalar values) to temporary files instead of keeping them in RAM.

This is an abstract property with no implementation in the base class `clsCavity`. The implementation in the two existing sub-classes `clsCavity1path` and `clsCavity2path` have the default value `True`.

## Behavior

- When `.use_bmatrix_class` is `True`, this flag is passed to the `file_caching` parameter of the `clsBlockMatrix(...)` constructor. Each matrix block is then stored on disk inside `.tmp_folder`.
- With file caching enabled, a full  $4 \times 4$  transfer block matrix typically occupies about one-sixteenth of the memory required for an all-in-RAM representation; only lightweight block indices remain in memory.
- If `.use_bmatrix_class` is `False`, the flag is silently forced to `False` as well, because plain NumPy arrays cannot be swapped to disk block-by-block.

### `.tmp_folder`

#### Read/write property.

Path to the directory that stores the temporary files created by `clsBlockMatrix` when on-disk block caching is enabled.

---

<sup>1</sup>The index refers to `.components`; negative values are clipped to 0.

### Behavior

- Relevant only if `.use_bmatrix_class` is `True`. The value of `.tmp_folder` is forwarded to the `tmp_dir` parameter of every `clsBlockMatrix(...)` constructor.
- If the property is an empty string (default), temporary files are created in the current working directory.
- Changing the folder path affects only matrices instantiated *after* the change; existing `clsBlockMatrix` objects keep their original disk locations.

### `.clear_results()`

#### Abstract method.

Implementations in the concrete subclasses purge all simulation output light fields (the left and right output field, as well as bulk fields) that may have been calculated and stored from a previous run, guaranteeing that subsequent calculations start from a clean state.

#### When it is called

The helper `.clear()` invokes `.clear_results()` automatically whenever a parameter (wavelength, grid, component property, ...) changes, ensuring that stale output data cannot be reused inadvertently in the next simulation run.

### `.clear()`

#### Method.

Clears all cached system-level matrices and simulation results. This method is automatically invoked whenever a property that affects the simulation (wavelength, grid, component configuration, ...) is changed.

### Behavior

- Sets the following properties to `None`:
  - `.R_L_mat_tot`
  - `.R_R_mat_tot`
  - `.T_LTR_mat_tot`
  - `.T_RTL_mat_tot`
  - `.R_L_mat_fov`
  - `.R_R_mat_fov`
  - `.T_LTR_mat_fov`
  - `.T_RTL_mat_fov`
  - `.M_bmat_tot`
  - `.S_bmat_tot`
- Calls `.clear_results()` to delete all output light fields.

### Use case

Called internally to ensure consistency whenever input parameters change. Can also be invoked manually to forcibly reset the internal state.

```
.delete_cached_component_files()
```

#### Method.

Deletes all files used for caching the individual component transfer matrices  $\mathbf{M}$ .

#### Behavior

Iterates over all elements of `.components` and invokes the method `clsOptComponent.delete_M_bmat_tot()` on each. This removes any potentially cached component transfer matrix (or inverse transfer matrix) files.

#### Use case

Useful to manually clean up disk storage, especially when working with persistent or long-running simulations that cache matrices across multiple runs.

### F.1.11 File I/O Utilities

```
.write_to_file(file_name, data)
```

#### Method.

Appends a line of structured data to a text file, optionally in the subdirectory defined by `.folder`.

#### Parameters

- `file_name` – Name of the output file (relative or absolute path).
- `data` – A Python list. Each element will be serialized and written as a string, separated by `.sep_char`.

#### Behavior

The method opens (or creates) the specified file in append mode. It locks the file using the `portalocker` library to ensure safe parallel access. The elements of `data` are written as a single line, with entries separated by `.sep_char` and a space. A newline character is appended after each write.

If `.folder` is nonempty, the file path is automatically prefixed with that directory.

#### Use case

Primarily intended for saving simulation results during parameter sweeps or batch processing.

```
.save_mat(name, m, idx=-1, msg="")
```

#### Method.

Saves a matrix `m` to disk in a `.pkl` file under a filename derived from a base `name`, which is completed (including the full path) by `.full_file_name(...)`.

#### Parameters

- `name` – Base name used to construct the filename.
- `m` – The matrix to store (can be a NumPy array or a `clsBlockMatrix` instance).
- `idx` – Optional component index; if  $\geq 0$ , included in the filename. Default is `-1`

(omitted).

- `msg` – Optional status message to show during saving. If omitted, a default message “saving `name`” is printed.

### Behavior

- Constructs a full path using `.full_file_name(...)`.
- If the matrix is a `clsBlockMatrix` instance, its `clsBlockMatrix.keep_tmp_files` flag is set `True` to ensure that any associated swap files are preserved.
- Serializes and stores the matrix using `joblib.dump(...)`.
- Progress is reported via `clsProgressPrinter`.

### Related methods

- `.load_mat(...)` – Loads a matrix saved in this way.
- `.full_file_name(...)` – Controls how the file name and the target path is assembled.

`.load_mat(name, idx=-1, msg="")`

#### Method.

Loads a matrix from a `.pk1` file created previously via `.save_mat(...)` from a filename derived from a base `name`, which is completed (including the full path) by `.full_file_name(...)`.

#### Parameters

- `name` – Base name used to construct the filename.
- `idx` – Optional component index used to disambiguate filenames (default `-1` disables indexing).
- `msg` – Optional progress message; if empty, a default message is printed.

#### Behavior

- Constructs the full filename using `.full_file_name(...)`.
- Loads the file using `joblib.load`.
- If the result is a `clsBlockMatrix`, sets its flag `clsBlockMatrix.keep_tmp_files = True` to prevent premature file deletion.
- Returns the loaded matrix.

#### Related methods

- `.save_mat(...)` – Saves a matrix.
- `.full_file_name(...)` – Controls how the file name and the target path is assembled.

```
.file_exists(name, idx=-1)
```

### Method.

Checks whether a file with a filename derived from a base `name`, which is completed (including the full path) by `._full_file_name(...)`, exists in the target directory.

### Parameters

- `name` – Base name used to construct the filename.
- `idx` – Optional component index to disambiguate filenames (default `-1` disables indexing).

### Behavior

- Constructs the full filename using `._full_file_name(...)`.
- Checks whether the file exists using `os.path.exists`.
- Returns `True` if the file is present, `False` otherwise.

### Typical usage

Can be used to check whether a matrix file previously written by `.save_mat(...)` is still available before attempting to load it via `.load_mat(...)`.

```
.delete_mat(name, idx=-1, msg="")
```

### Method.

Deletes a matrix file from disk. The filename is derived from the base `name`, which is completed (including the full path) by `._full_file_name(...)`.

### Parameters

- `name` – Base name used to construct the filename.
- `idx` – Optional component index to disambiguate filenames (default `-1` disables indexing).
- `msg` – Optional progress message; if empty, a default message is printed.

### Behavior

- Constructs the full filename using `._full_file_name(...)`.
- If the file exists:
  - Prints a progress message using `clsProgressPrinter`.
  - If `.use_bmatrix_class` and `.use_swap_files_in_bmatrix_class` are both `True` and the file contains a `clsBlockMatrix` object:
    - \* Loads the file via `joblib`.
    - \* Disables file retention by setting `.keep_tmp_files = False`.
    - \* Calls `.clear()` to remove swap files.
  - Deletes the matrix file from disk using `os.remove`.
  - Closes the progress context.
- If the file does not exist, nothing is done.

**.save\_M\_bmat()****Method.**

Saves the cavity's global transfer block matrix `.M_bmat_tot` to disk using `.save_mat(...)`.

**Behavior**

Calls `.save_mat(...)` with:

- `name = "M_bmat"`
- `m = .M_bmat_tot`
- `msg = "saving transfer block matrix of cavity"`

**Typical usage**

Call this method after calculating the total transfer matrix using `.calc_M_bmat_tot(...)` to persist it for later use via `.load_M_bmat()`.

**.load\_M\_bmat()****Method.**

Loads the cavity's global transfer block matrix from disk using `.load_mat(...)` and stores the result in `.M_bmat_tot`.

**Behavior**

Calls `.load_mat(...)` with:

- `name = "M_bmat"`
- `msg = "loading transfer block matrix of cavity"`

**Typical usage**

Call this method to restore a previously saved total transfer matrix (see `.save_M_bmat()`) without recomputing it via `.calc_M_bmat_tot(...)`.

**.M\_bmat\_file\_exists()****Method.**

Checks whether a saved version of the cavity's global transfer block matrix exists on disk.

**Behavior**

Calls `.file_exists(...)` with `name = "M_bmat"` and returns `True` if the corresponding file is present, `False` otherwise.

**Typical usage**

Use this method to decide whether `.load_M_bmat()` can be called without triggering a file-not-found error.

### F.1.12 Single-Step Mode

For large, high-resolution systems the construction of the global transfer matrix  $\mathbf{M}_{\text{cav}} = \mathbf{M}_0 \mathbf{M}_1 \mathbf{M}_2 \dots$  can take many hours. *Single-step mode* decomposes this job into discrete, restartable steps that must run in order for a given wavelength, but can run concurrently across different wavelengths – ideal for cluster environments.

Assume a single wavelength. On its first invocation, a single-step program builds only the first component’s transfer matrix  $\mathbf{M}_0$  and writes it to disk as the current partial product. On the next invocation, it reloads that partial product, computes  $\mathbf{M}_1$ , updates the product to  $\mathbf{M}_0 \mathbf{M}_1$ , and stores it again. This proceeds component by component until all  $N = \text{.component\_count}$  elements have been incorporated, yielding the full  $\mathbf{M}_{\text{cav}}$ . One or more post-processing steps then use  $\mathbf{M}_{\text{cav}}$  e.g., to evaluate output fields, and write the final simulation results.

In practice you may want to sweep many wavelengths (e.g., 17 points around resonance). Single-step execution remains *sequential per wavelength*, but *independent across wavelengths*. Thus, while one worker is still building  $\mathbf{M}_0$  at  $\lambda_1$ , another can already build  $\mathbf{M}_0$  at  $\lambda_2$ , and a third might have progressed to  $\mathbf{M}_0 \mathbf{M}_1$  at  $\lambda_3$ . Each wavelength maintains its own on-disk partial products, so multiple instances of the same program can run safely in parallel on a cluster, each advancing a different wavelength.

All of this is orchestrated by the properties and methods described in this section, together with `clsTaskManager`, which assigns the next unfinished  $\langle \text{simulation}, \text{step} \rangle$  pair to each worker process.

#### Number of required steps per simulation

The required number of steps per simulation (i.e. the number of times `.single_step(...)` needs to be called) is defined by three settings:

- `.pre_steps` (default 0) – number of *pre-processing* steps executed *before* the first optical component (e.g. to pre-calculate some heavy component transfer matrices). Work for these indices is carried out by the user-overridable hook `.pre_step(...)`, which `.single_step(...)` invokes automatically whenever `step ≤ .pre_steps`.
- `.component_count` – one step for every optical component in the cavity. When `.single_step(...)` lands in this range it (i) loads (or initialises) the current global matrix  $\mathbf{M}_{\text{cav}}$  as calculated so far, (ii) calculates (or, if cached, loads) the component’s transfer matrix  $\mathbf{M}_i$ , (iii) multiplies it into the running product  $\mathbf{M}_{\text{cav}} \leftarrow \mathbf{M}_{\text{cav}} \mathbf{M}_i$  and (iv) stores the updated product to disk for the next process.
- `.additional_steps` (default 1) – number of *post-processing* steps that use the finished cavity matrix, typically to evaluate output fields and store results. These steps invoke the hook `.additional_step(...)`, which is automatically called by `.single_step(...)` at the last calls.

The read-only property `.total_steps` gives the overall length of the step sequence:

```
.total_steps = .pre_steps + .component_count + .additional_steps.
```

The listing below shows a minimal worker script that can distribute 31 different wavelength simulation across multiple cluster nodes (when started in parallel), utilizing `clsTaskManager`:

```

1 # -----
2 # 1) Custom cavity: sub-classed from clsCavity1path
3 #   to override .additional_step(...)
4 # -----
5 class clsMyCavity(clsCavity1path):
6     def __init__(self, name):
7         super().__init__(name)
8
9     # post-processing step: what you actually want to simulate
10    def additional_step(self, step: int):
11        # insert your code using the final cavity transfer
12        # matrix here (e.g. calculating some cavity output fields
13        # when sending in some interesting input fields)
14        pass
15
16 # -----
17 # 2) Worker process: perform exactly ONE (sim, step) pair
18 # -----
19 folder = "/shared/results"
20 tmp = "/shared/tmp"
21
22 # instantiate cavity (fresh for every worker)
23 myCavity = clsMyCavity("Demo Cavity")
24 myCavity.folder = folder
25 myCavity.tmp_folder = tmp
26 myCavity.additional_steps = 1      # one post-processing step
27
28 # add code initializing the grid and adding components to cavity here
29 # ...
30
31 # wavelengths to sweep
32 points = 31 # number of simulations with different wavelengths
33 lambda_vec = get_sample_vec(points, 2e-9, 633e-9, True)
34
35 # get new task from task manager -----
36 task_manager = clsTaskManager(points, myCavity.total_steps, folder)
37 sim, step = task_manager.get_next_task()
38 if sim >= 0: # (-1,-1) means "nothing to do right now"
39     myCavity.Lambda = lambda_vec[sim] # configure wavelength
40     myCavity.UID = sim # ensure unique filenames per simulation
41     myCavity.single_step(step) # perform exactly ONE step
42     task_manager.end_task(index, step) # mark as finished

```

### How the worker script operates

- **Define cavity sub-class.** Create a thin subclass `clsMyCavity` that only overrides `.additional_step(...)` (cf. lines 1–14). Place the *actual physics* of your study here - for example, create an incident field and evaluate and save the reflected or transmitted response. At this point the global transfer matrix  $\mathbf{M}_{\text{cav}}$  has already been assembled, so the additional step executes quickly even for very large grids.
- **Instantiate the cavity.** Construct a fresh `clsMyCavity` object and point it to a *shared* results folder (`folder`) so that all worker processes read and write the same matrix files, and a common temporary directory (`tmp_folder`) in which `clsBlockMatrix` can place its swap files. Finally set `.additional_steps=1` so that `.total_steps` correctly reports

`pre_steps + component_count + 1.` (Cf. lines 16–26.)

- **Build the cavity.** In real use you now initialize the spatial grid (pixel count, physical size) e.g. via `.grid.set_res(...)` and add all optical components (mirrors, lenses, propagation sections, ...) to the cavity’s component list via `.add_component(component)`. (Cf. lines 28–29.)
- **Define wavelength sweep.** Decide how many wavelengths you wish to scan (line 32) and assemble the corresponding vector (line 33). The length of this vector equals the number of independent simulations; each worker process may therefore pick up a different wavelength and run its own single-step sequence.
- **Task acquisition.** Each worker instantiates its own `clsTaskManager` (line 36) and requests the next unprocessed (`sim, step`) pair via `.get_next_task()` (line 37). The task manager ensures that (i) no two workers ever receive the *same* task, and (ii) tasks belonging to the *same* simulation are executed in the correct chronological order. If all tasks are currently taken, the call returns `(-1, -1)` after the configured wait time, signalling the worker to don’t do anything (line 38).
- **Per-simulation configuration.** Using the simulation index `sim`, the worker now (i) sets the wavelength `myCavity.Lambda = lambda_vec[sim]` and (ii) re-uses the same integer as `.UID`. The `.UID` is embedded by `.full_file_name(...)` in every filename the cavity writes, so intermediate files from different simulations never collide.
- **Single-step execution.** Calling `.single_step(...)` (i) loads the running cavity matrix from the previous step (if any), (ii) carries out the requested step (pre-step / component / additional step), (iii) persists the updated matrix back to disk. (line 41). The cavity therefore acts as its *own checkpoint-restart mechanism*; every worker is stateless apart from the files it leaves behind.
- **Task finalisation.** On success the worker records completion with `.end_task(...)` and exits. Another worker (or the same executable launched again) will pick up the next unfinished step until the entire wavelength sweep has been processed.

### `.UID`

#### **Read / write property.**

An integer identifier that is prepended to *every* filename created by `.full_file_name(...)`. It separates the persistent data belonging to different simulations that share the same output `.folder`. The default value is 0.

#### **Purpose**

By giving each simulation a distinct UID you ensure that all files produced through `.save_mat(...)`, `.save_M_bmat()`, ... are stored under unique names, e.g. "7\_M\_bmat.pkl", "23\_5\_R\_mat.pkl", and so on.

#### **Typical usage**

When running many wavelength points in parallel each worker sets `myCavity.UID = sim`, where `sim` is the simulation index obtained from `clsTaskManager`. For single, stand-alone simulations it is safe to keep the default value.

**.pre\_steps****Read / write property.**

Non-negative integer (default 0). When you call `.single_step(...)` with a parameter `step < .pre_steps`, the method calls the user-overridable hook `.pre_step(...)` and passes on the `step` parameter. Set `.pre_steps` to the number of pre-processing iterations you need; use 0 to disable the feature.

**Typical usage**

In single-step mode (see introduction) you might set `.pre_steps` to values  $> 0$  to get one or more preparatory calls of `.single_step(...)`, e.g. to pre-calculate and cache heavy transfer matrices before stepping through the actual components.

**.additional\_steps****Read / write property.**

Non-negative integer (default 1). When you call `.single_step(...)` with a parameter `step < .pre_steps + .component_count + .additional_steps` it loads the current total cavity transfer matrix and calls the user-overridable hook `.additional_step(...)`, passing on the index value `step - .component_count - .pre_steps`

**Typical usage**

In single-step mode (see introduction), set `.additional_steps` to the number of post-processing iterations you need. Put the *actual physics* of your study into the method `.additional_step(...)` of your sub-classed cavity - for example, create an incident field and evaluate and save the reflected or transmitted response.

**.total\_steps****Read-only property.**

Returns the total number of iterations that `.single_step(...)` requires for one complete run:

$$\text{.total\_steps} = \text{.component\_count} + \text{.pre\_steps} + \text{.additional\_steps}.$$

The value updates automatically whenever components are added or when `.pre_steps` or `.additional_steps` change; direct assignment to this property is not allowed.

**Typical usage**

Pass this value as `steps_per_simulation` parameter to the `clsTaskManager(...)` constructor to ensure each simulation processes exactly the required number of steps.

**.single\_step(step, reverse\_mul=False)****Method.**

Executes a single iteration of the “single step” cavity workflow and is the foundation of the *single-step mode* (see Introduction). Internally it decides which part of the workflow to run:

- `step < .pre_steps`: calls the user-overridable hook `.pre_step(...)` (*pre-processing phase*).
- `step < .component_count + .pre_steps`: processes exactly one optical component, updates `M_bmat_tot` (with optional `reverse_mul` order), then saves the result to disk.
- otherwise: loads the current total transfer matrix and invokes `.additional_step(...)` (*post-processing phase*).

### Parameters

- `step (int)` – Zero-based index of the iteration to execute. Valid range:  $0 \leq \text{step} < \text{.total_steps}$ .
- `reverse_mul (bool, default False)` – Controls the order in which the newly computed component matrix  $\mathbf{M}_i$  is merged into the cumulative transfer matrix  $\mathbf{M}_{\text{cav}}$ :
  - `False`:  $\mathbf{M}_{\text{cav}} \leftarrow \mathbf{M}_{\text{cav}} \mathbf{M}_i$  (default behavior).
  - `True`:  $\mathbf{M}_{\text{cav}} \leftarrow \mathbf{M}_i \mathbf{M}_{\text{cav}}$  (simulates a cavity with reversed component order).

### Return value

- `True` if the step was executed;
- `False` otherwise (e.g. if `step` is out of range or `.total_steps=0`.)

### Typical usage

Typically used in *single-step mode* (see Introduction): a `clsTaskManager(...)` instance provides the next `step` index, and the caller then executes `.single_step(step)` until all `.total_steps` iterations have been processed.

## `.single_step_mode`

### Read-only property.

Boolean flag that indicates whether the cavity is currently being processed in *single-step mode*. It is set to `True` automatically by `.single_step(...)` and reset to `False` by `.calc_M_bmat_tot(...)`.

## `.pre_step(step)`

### Method (hook).

Provides a user-defined pre-processing operation for a specific `step` index. The base implementation in `clsCavity` is empty; override it in a sub-class to perform actions such as pre-building heavy transfer matrices.

### When it is called

- Invoked automatically by `.single_step(...)` whenever `step <.pre_steps`.
- Called exactly once for each pre-processing iteration before any optical component is handled.

**Parameters**

- `step` (`int`) – index of the pre-processing iteration currently being executed, ranging from 0 to `.pre_steps`-1.

**Typical usage**

In single-step mode you derive a custom class from `clsCavity1path` or `clsCavity2path` and implement `pre_step` to pre-build and cache heavy transfer matrices for later use.

**`.additional_step(step)`****Method (hook).**

Provides a user-defined *post-processing* operation for a specific `step` index. The base implementation in `clsCavity` is empty; override it in a sub-class to perform tasks such as evaluating fields, collecting statistics, or writing results to disk.

**When it is called**

- Invoked automatically by `.single_step(...)` whenever `step>=.component_count+.pre_steps` and `step < .total_steps`.
- Called exactly once for each post-processing iteration after all optical components have been processed and the total transfer matrix has been loaded.

**Parameters**

- `step` (`int`) – index of the post-processing iteration currently being executed, ranging from 0 to `.additional_steps`-1.

**Typical usage**

Put the *actual physics* of your study into `.additional_step(...)` of your sub-classed cavity – for example, create an incident field and evaluate (and save) the reflected or transmitted response while running in single-step mode.

**F.1.13 Resonance Calculations****`.resonance_data_simple_cavity(R_left, R_right, length_opt, sym_phase = True)`****Method.**

Returns the resonance wavelength  $\lambda_c$  closest to the current wavelength `.Lambda`, the corresponding wavenumber  $k_c = 2\pi/\lambda_c$ , the longitudinal mode index  $l_{\text{mode}}$ , and the free-spectral-range  $\Delta\lambda$  for a two-mirror cavity of optical length  $L = \text{length\_opt}$ . When `sym_phase=True` (default) the phase offsets of *symmetric-phase* mirrors are included (see `clsMirrorBase2port.set_phys_behavior(...)`); otherwise the textbook calculation is used.

**Parameters**

- `R_left, R_right` – power reflectivities of the left and right mirrors ( $0 < R < 1$ ).
- `length_opt` – optical length  $L$  of the cavity (half round-trip).
- `sym_phase` (`bool`, default `True`) – whether to assume that the mirror phase shifts on partially reflective mirrors follow the symmetric phase convention; see

`clsMirrorBase2port.set_phys_behavior(...)` for more information.

### Return value

(`lambda_c`, `k_c`, `l_mode`, `lambda_period`).

### Formulae (`sym_phase = True`)

- Longitudinal resonance mode

$$l_{\text{mode}} = \left\lfloor \frac{1}{2\pi} \left[ \tan^{-1} \sqrt{\frac{1-R_L}{R_L}} + \tan^{-1} \sqrt{\frac{1-R_R}{R_R}} \right] + \frac{2L}{\lambda} \right\rfloor.$$

- Resonance wavenumber and wavelength

$$k_c = \frac{2\pi l_{\text{mode}} - \tan^{-1} \sqrt{\frac{1-R_L}{R_L}} - \tan^{-1} \sqrt{\frac{1-R_R}{R_R}}}{2L}, \quad \lambda_c = \frac{2\pi}{k_c}.$$

- Free-spectral-range

$$\Delta\lambda = \frac{2L}{l_{\text{mode}}} - \frac{2L}{l_{\text{mode}} + 1}.$$

### Formulae (`sym_phase = False`)

- Longitudinal resonance mode

$$l_{\text{mode}} = \left\lfloor \frac{2L}{\Lambda} - \frac{1}{2} \right\rfloor.$$

- Resonance wavelength and wavenumber

$$\lambda_c = \frac{4L}{2l_{\text{mode}} + 1}, \quad k_c = \frac{2\pi}{\lambda_c}.$$

- Free-spectral-range

$$\Delta\lambda = \lambda_c - \frac{4L}{2(l_{\text{mode}} + 1) + 1}.$$

### Typical usage

Perform a simulation at the physically exact resonance wavelength closest to the current wavelength.

```
.resonance_data_8f_cavity(R_left, R_center,
    R_right, f, sym_phase = True)
```

### Method.

Returns the resonance wavelength  $\lambda_c$  closest to the current wavelength `.Lambda`, the corresponding wavenumber  $k_c = 2\pi/\lambda_c$ , the longitudinal mode index `l_mode`, the free-spectral-range  $\Delta\lambda$ , and the length correction `l_corr` required for the *second* sub-cavity of an 8f resonator. When `sym_phase=True` (default) the phase offsets of *symmetric-phase* mirrors are included (see `clsMirrorBase2port.set_phys_behavior(...)`); otherwise the textbook calculation is used and  $l_{\text{corr}} = 0$ .

### Parameters

- `R_left` – power reflectivity of the left mirror.
- `R_center` – power reflectivity of the central mirror.

- `R_right` – power reflectivity of the right mirror.
- `f` – focal length  $f$ ; the two sub-cavities each have optical length  $4f$ .
- `sym_phase` (`bool`, default `True`) – whether to assume symmetric-phase behaviour for all mirrors.

**Return value**

`(lambda_c, k_c, l_mode, lambda_period, l_corr).`

**Formulae (`sym_phase = True`)**

- Longitudinal resonance mode

$$l_{\text{mode}} = \left\lfloor \frac{1}{2\pi} \left[ \tan^{-1} \sqrt{\frac{1-R_L}{R_L}} + \tan^{-1} \sqrt{\frac{1-R_C}{R_C}} \right] + \frac{8f}{\lambda} \right\rfloor.$$

- Resonance wavenumber and wavelength

$$k_c = \frac{2\pi}{\lambda_c}, \quad \lambda_c = \frac{2\pi}{k_c}.$$

- Free-spectral-range

$$\Delta\lambda = \frac{8f}{l_{\text{mode}}} - \frac{8f}{l_{\text{mode}} + 1}.$$

- Length correction for the second sub-cavity

$$l_{\text{corr}} = \frac{\tan^{-1} \sqrt{\frac{1-R_L}{R_L}} - \tan^{-1} \sqrt{\frac{1-R_C}{R_C}} - \tan^{-1} \sqrt{\frac{1-R_R}{\sqrt{R_R}}}}{2k_c}.$$

**Formulae (`sym_phase = False`)**

- Longitudinal resonance mode

$$l_{\text{mode}} = \left\lfloor \frac{8f}{\Lambda} - \frac{1}{2} \right\rfloor.$$

- Resonance wavelength and wavenumber

$$\lambda_c = \frac{8f}{l_{\text{mode}} + \frac{1}{2}}, \quad k_c = \frac{2\pi}{\lambda_c}.$$

- Free-spectral-range

$$\Delta\lambda = \lambda_c - \frac{8f}{l_{\text{mode}} + \frac{3}{2}},$$

- Length correction for the second sub-cavity

$$l_{\text{corr}} = 0.$$

**Typical usage**

Perform an  $8f$  cavity simulation at the physically exact resonance and adjust the second sub-cavity length by  $l_{\text{corr}}$  to maintain constructive interference.

**Note**

The correction  $l_{\text{corr}}$  is a first-order estimate; the true optimum may differ slightly and can be refined numerically if higher precision is required.

## F.2 Class `clsCavity1path`

The class `clsCavity1path` is a concrete subclass of `clsCavity` that represents a *linear* optical cavity – i.e. a resonator in which the light field performs round-trips along a single spatial path. “Linear” in this context means a straight, one-dimensional sequence of components; it does *not* exclude partially reflective elements inside that split the structure into several coupled sub-cavities.

By enabling the *Transmission–Behaves–Like–Reflection* mode for selected mirrors (see subsection D.9.5), also certain ring geometries – most notably bow-tie cavities – can be mapped onto an equivalent linear resonator and analyzed with `clsCavity1path`.

The functionality of the base class `clsCavity` (spatial grid, global transfer/scattering matrices, caching infrastructure, ...) is inherited unchanged. This section documents only the additional features relevant to single-path operation.

### Create and configure the cavity.

Before a simulation can be performed, the cavity needs to be created and configured.

1. Instantiate the cavity via the constructor `clsCavity1path(...)`.
2. Specify the vacuum wavelength with `.Lambda` or `.Lambda_nm` and set up the spatial grid (pixel count, physical size) through `.grid.set_opt_res_based_on_sidelength(...)` or `.grid.set_opt_res_tot_based_on_res_fov(...)`.
3. Create and append every optical element (mirrors, lenses, free-space sections, ...) in geometric order using `.add_component(component)`.

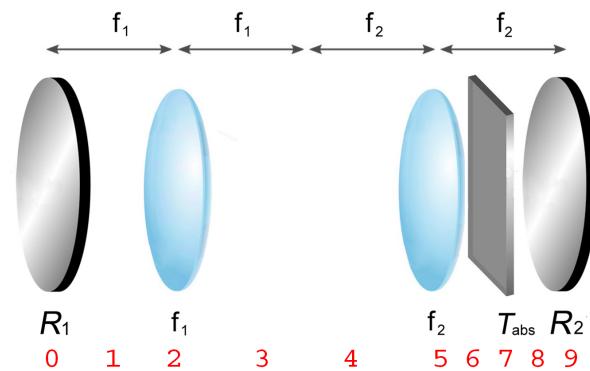


Figure F.1: Example of a simple linear cavity comprising two mirrors, two lenses, and an absorber, arranged in a  $4f$ -configuration.

Figure F.1 shows an example of a simple linear cavity that can be simulated using `clsCavity1path`. It consists of the following 10 components (numbered 0 to 9), which must be added using `.add_component(component)` in consecutive order:

0. A partially reflective input coupling mirror, simulated with `clsMirror`.
1. Free-space propagation over a distance  $f_1$ , simulated with `clsPropagation`.
2. A lens with focal length  $f_1$ , simulated with `clsThinLens`.

3. Another free-space propagation over distance  $f_1$ ; the same instance of `clsPropagation` as used in component 1 can be reused here.
4. Free-space propagation over distance  $f_2$ , simulated with `clsPropagation`.
5. A lens with focal length  $f_2$ , simulated with `clsThinLens`.
6. Propagation between lens  $f_2$  and the absorber  $T_{\text{abs}}$ , simulated with `clsPropagation`.
7. Propagation through the absorber, simulated with `clsPropagation`.
8. Propagation between the absorber  $T_{\text{abs}}$  and mirror  $R_2$ , simulated with `clsPropagation`.
9. A totally reflective end mirror  $R_2$ , simulated with `clsMirror`.

### Choose a simulation method.

These are the available simulation methods:

- **Direct field propagation**

It is possible to directly propagate an existing light field either left-to-right (LTR) or right-to-left (RTL) through a contiguous sequence of optical components via `.prop(...)`. Single reflections at any component are available through `.reflect(...)`. This variant is ideal for studying the behavior of individual elements or studying the behavior of a series of elements, but it does *not* form a closed resonator; multiple round-trips would have to be scripted manually.

- **A single, explicit round-trip**

With exactly two end mirrors you can simulate one complete LTR→RTL cycle using `.prop_round_trip_LTR_RTL(...)`. The light field to be propagated is passed on to this method, which returns the resulting light field.

- **Single explicit round-trip**

For a cavity that is bounded by exactly two mirrors you can track one complete LTR→RTL excursion with `.prop_round_trip_LTR_RTL(...)`. Pass the incident `clsLightField` as the first argument, followed by `idx1` and `idx2`, which identify the left and right mirror. The method produces a `clsLightField` representing the field immediately after the round-trip.

- **Multiple explicit round-trips**

With exactly two end mirrors you can also simulate an arbitrary number of round-trip cycles (as a finite approximation to the steady state) using the method `.prop_mult_round_trips_LTR_RTL(...)`. Incident fields can be assigned to the properties `.incident_field_left` and `.incident_field_right`; the resulting near-steady-state outputs appear in `.output_field_left` and `.output_field_right` after this method has finished computing.

- **Infinite round-trips (geometric-series approach)**

For a two-mirror cavity driven from one side you can dispense with iterative field propagation and work directly with transmission matrices:

1. `.get_round_trip_T_mat_LTR_RTL(...)` returns the *single-round-trip transmission matrix*  $\mathbf{T}_{\text{RT}}$ , mapping a field that starts on the right of the left mirror to the field that reappears there after one complete LTR→RTL excursion.

## 2. The infinite geometric series

$$\sum_{k=0}^{\infty} (\mathbf{R}_1 \mathbf{T}_{\text{RT}})^k = (\mathbb{1} - \mathbf{R}_1 \mathbf{T}_{\text{RT}})^{-1}$$

is evaluated internally by `.get_inf_round_trip_R_L_mat(...)` (where  $\mathbf{R}_1$  is the left-mirror reflection matrix). Using that result the method returns the exact left reflection matrix of the entire cavity,

$$\mathbf{R}_{\text{cav}}^{\text{left}} = \mathbf{R}_1 + \mathbf{T}_1 \mathbf{T}_{\text{RT}} (\mathbb{1} - \mathbf{R}_1 \mathbf{T}_{\text{RT}})^{-1} \mathbf{T}_1,$$

with  $\mathbf{T}_1$  the transmission matrix of the left mirror. This reflection matrix  $\mathbf{R}_{\text{cav}}^{\text{left}}$  can be applied to the left input field via `clsLightField.apply_TR_mat(...)`. Analogous routines exist for the RTL→LTR cycle and for the right-hand reflection matrix.

This matrix-series approach is numerically robust and sometimes faster than simulating many explicit cycles. Remember, however, that for an  $N \times N$  grid the matrices involved have size  $N^2 \times N^2$ ; large grids therefore impose a substantial memory and CPU load.

- **Transfer and scattering matrix formalism**

If the cavity contains additional “inner” mirrors it becomes a set of mutually coupled sub-cavities, and the full  $2 \times 2$  block transfer matrix of the entire component stack must be evaluated. Assign the driving fields to `.incident_field_left` and `.incident_field_right`; then simply access `.output_field_left` or `.output_field_right`. This request automatically invokes `.calc_M_bmat_tot(...)` to assemble the global transfer matrix, followed by the calculation of the cavity’s left and right reflection matrices `.R_L_mat_tot` and `.R_R_mat_tot`. The resulting steady-state output fields are returned in `.output_field_left` and `.output_field_right`. Although this method is considerably slower and more memory-intensive than the previous ones, it yields the complete response of an arbitrarily coupled multi-mirror cavity.

- **Steady state intracavity field**

Once the left and/or right output fields have been computed you can compute the steady-state intracavity field at an arbitrary interface index `bulk_idx`. Two formally equivalent methods are provided:

- `.calc_bulk_field_from_left(...)` uses the *left* incident–output pair to back-propagate the field from the outer left boundary up to `bulk_idx`. This is fastest when the target interface lies closer to the left mirror.
- `.calc_bulk_field_from_right(...)` does the same starting from the *right* side and is more efficient for positions closer to the right mirror.

Both routines store the decomposed contributions in `.bulk_field_LTR` (left-to-right field) and `.bulk_field_RTL` (right-to-left field); the superposition is available via `.bulk_field`. If no pre-computation has been performed, the computational load can even exceed that of the full transfer–scattering-matrix approach described above.

### F.2.1 Initialization

```
clsCavity1path(name, full_precision=True)
```

#### Constructor.

Initializes a single-path cavity subclass.

#### Parameters

- `name` – Human-readable identifier used in log messages and file names.
- `full_precision` – If `True` (default) internal arrays use `float64/complex128`; otherwise `float32/complex64` to reduce memory.

#### Behaviour

- Selects the numeric precision and allocates an empty `clsGrid` instance.
- Creates empty containers for left and right incident and output fields, and for the bulk field.
- Calls the constructor of `clsCavity(...)`.

#### Typical workflow

1. Specify the working wavelength via `.Lambda` or `.Lambda_nm`.
2. Initialize the sampling grid with `.grid.set_opt_res_based_on_sidelength(...)` or `.grid.set_opt_res_tot_based_on_res_fov(...)` or a related method.
3. Append components with `.add_component(component)` in the optical order they are encountered.

### F.2.2 Component Management

```
.add_component(component)
```

#### Method.

Appends a two-port optical element to the cavity's component list `clsCavity.components` and establishes the internal back-reference from the optical element to itself and its `.grid` instance via `clsOptComponent._connect_to_cavity(...)`.

#### Arguments

- `component`

An instance of `clsOptComponent2port` representing a mirror, lens, free-space section, aperture, ... The same object may be inserted multiple times if the optical layout contains identical segments; this is both valid and memory-efficient.

#### Behavior

1. The component is appended to `clsCavity.components`; the new element is thus accessible through `clsCavity.get_component(...)`.
2. `clsCavity.component_count` is implicitly incremented.

**Usage note**

Because the method merely registers the element, no optical calculation is triggered. Add all components in left-to-right order before invoking propagation or matrix routines.

```
.get_dist_phys(idx1, idx2)
```

**Method.**

Returns the cumulative *physical* length (in meters) of the cavity segment that starts at the *left* face of component `idx1` and ends at the *right* face of component `idx2`. The quantity is obtained by summing `dist_phys` of every component in the closed interval `[idx1, idx2]`.

**Arguments**

- `idx1`

Zero-based index of the first component (default 0). Values below 0 are clipped to the left edge.

- `idx2`

Zero-based index of the last component. The default 999 intentionally exceeds any realistic length and is clipped to the rightmost component.

**Return value**

A scalar floating-point number in meters.

```
.get_dist_opt(idx1, idx2)
```

**Method.**

Calculates the cumulative *optical* path length, in meters, from the left face of component `idx1` to the right face of component `idx2`. Each element contributes its individual `dist_opt` value, which already includes the effect of any refractive index inside the component.

**Arguments**

- `idx1`

Index of the first component (default 0). Negative values are clamped to the left edge of the cavity.

- `idx2`

Index of the last component (default 999). Values beyond the actual list length are clamped to the rightmost component.

**Return value**

A scalar floating-point number giving the total optical distance in meters.

### F.2.3 Propagation and Reflection

`.prop(field, idx_from, idx_to, direction)`

#### Method.

Propagates the specific light field (`field`) through a contiguous block of components in a single direction – either left-to-right (`Dir.LTR`) or right-to-left (`Dir.RTL`). Both boundary indices are inclusive, so setting `idx_from = idx_to` models transmission through one individual element. The routine automatically decides, for each interface, whether it is more efficient to work in position space or  $k$ -space; this choice is based on the stated preferences of the current component and its immediate neighbor.

#### Arguments

- `field`

Instance of `clsLightField` that provides the incident complex amplitudes. The object is left unmodified.

- `idx_from, idx_to`

Zero-based indices delimiting the propagation span. The method swaps the values if they are out of order and clamps them to the valid range  $[0, .component\_count - 1]$ .

- `direction`

Propagation direction: (`Dir.LTR` or `Dir.RTL`).

#### Return value

A new `clsLightField` containing the transmitted field at the *exit* face of the last processed component.

`.reflect(field, idx, side)`

#### Method.

Returns the field obtained by reflecting `field` at component `idx` on the selected interface (`Side.LEFT` or `Side.RIGHT`). If the index is outside the valid range, or if the cavity contains no component, the result is a zero field.

#### Arguments

- `field` – incident `clsLightField` (input is unchanged).
- `idx` – zero-based index of the target component.
- `side` – interface at which the reflection occurs.

#### Algorithm

- *Mirrors, curved mirrors, amplitude scalers.* These classes implement their own `.reflect(...)` method; the call is forwarded directly.
- *All other components.* The routine multiplies the incident field by the component's respective reflection matrix (`clsOptComponent2port.R_L_mat_tot` or `clsOptComponent2port.R_R_mat_tot`, respectively). Because this matrix must be generated on demand, the operation is significantly slower and more memory-

intensive than the specialized path above.

### Return value

A new `clsLightField` instance containing the reflected light field.

`.get_T_mat_LTR(idx_from, idx_to)`

### Method.

Builds the cumulative left-to-right transmission matrix  $\mathbf{T}_{\text{seg}}^{\text{LTR}}$  for a user-defined segment of the cavity. The transmission matrix maps an arbitrary incoming field on the *left* face of component `idx_from` to the field that exits on the *right* face of component `idx_to`.

### Arguments

- `idx_from` – index of the first component.
- `idx_to` – index of the last component (inclusive). If the two indices are given in reverse order they are swapped internally.

### Implementation details

For each component in the interval the routine multiplies the running product by `clsOptComponent2port.T_LTR_mat_tot` of that component. When `idx_from = 0` and `idx_to = component_count-1` the result is identical to the class-wide property `clsCavity.T_LTR_mat_tot`, which is cached after the first evaluation. In contrast, `.get_T_mat_LTR(...)` always builds the matrix on the fly.

### Return value

A matrix of size  $N^2 \times N^2$  that transfers complex amplitudes from left to right across the specified segment. If the cavity is empty the function returns the scalar 1 (identity).

`.get_T_mat_RTL(idx_from, idx_to)`

### Method.

Builds the cumulative right-to-left transmission matrix  $\mathbf{T}_{\text{seg}}^{\text{RTL}}$  for a user-defined segment of the cavity. The transmission matrix maps an arbitrary incoming field on the *right* face of component `idx_from` to the field that exits on the *left* face of component `idx_to`.

### Arguments

- `idx_from` – index of the first component.
- `idx_to` – index of the last component (inclusive). If the two indices are given in reverse order they are swapped internally.

### Implementation details

For each component in the interval the routine multiplies the running product by `clsOptComponent2port.T_RTL_mat_tot` of that component. When `idx_from = component_count-1` and `idx_to = 0` the result is identical to the class-wide property `clsCavity.T_RTL_mat_tot`, which is cached after the first evaluation. In contrast, `.get_T_mat_RTL(...)` always builds the matrix on the fly.

### Return value

A matrix of size  $N^2 \times N^2$  that transfers complex amplitudes from right to left across the

specified segment. If the cavity is empty the function returns the scalar 1 (identity).

#### F.2.4 Round-Trip Utilities

```
.prop_round_trip_LTR_RTL(field, idx1, idx2, incl_left_refl)
```

##### Method.

Performs a single round-trip propagation that starts on the *right* face of component `idx1`, travels left-to-right to component `idx2`, reflects on its `Side.LEFT` interface, and returns right-to-left to component `idx1`. Optionally the field is reflected once more on the `Side.RIGHT` interface of component `idx1`.

##### Arguments

- `field` – incident `clsLightField`.
- `idx1` – index of the first (left) boundary component.
- `idx2` – index of the second (right) boundary component. The two indices are swapped internally if supplied out of order.
- `incl_left_refl` – when `True` the field is finally reflected on the right face of component `idx1` before the routine returns.

##### Implementation details

1. Left-to-right propagation is delegated to `.prop(...)` ( $idx1+1, idx2-1, \text{Dir.LTR}$ ).
2. The field is reflected on the left side of component `idx2` via `.reflect(...)`.
3. Right-to-left propagation uses `.prop(...)` ( $idx2-1, idx1+1, \text{Dir.RTL}$ ).
4. If `incl_left_refl` is `True`, the routine multiplies the result with `clsOptComponent2port.R_R_mat_tot` of component `idx1`.

##### Return value

A new `clsLightField` representing the field after the complete round-trip (and the optional final reflection). If the cavity is empty the function returns a black field, i.e. the zero matrix.

```
.prop_mult_round_trips_LTR_RTL(no_of_trips, bulk_field_pos = 0,
                                print_progress = False,
                                plot_left_out_progress = False,
                                plot_right_out_progress = False,
                                plot_bulk_LTR_progress = False,
                                plot_bulk_RTL_progress = False,
                                idx1 = 0, idx2 = 999,
                                x_plot_shift = 0, y_plot_shift = 0)
```

##### Method.

Propagates the intracavity field through a user-specified number of left-to-right / right-to-left round-trips. The left incident field enters `clsCavity1path` at the *left* interface of component `idx1`; the right incident field enters at the *right* interface of component `idx2`. After `no_of_trips` cycles the routine stores `.output_field_left` and `.output_field_right` as the effective steady-state outputs. If `bulk_field_pos` lies

inside the indices [idx1–1, idx2] the intracavity field at that location is written to .bulk\_field\_LTR, .bulk\_field\_RTL, and .bulk\_field.

### Arguments

- **no\_of\_trips** (int)  
Number of round-trip cycles to simulate.
- **bulk\_field\_pos** (int)  
Component index whose *right* face is used for bulk-field sampling. Values of idx1 – 1 or idx2 place the probe just outside the left or right mirror, respectively.
- **print\_progress** (bool)  
If True, periodic text updates are sent to the `clsProgressPrinter` logger.
- **plot\_left\_out\_progress** (bool)  
Plot the evolving left output field after each status interval.
- **plot\_right\_out\_progress** (bool)  
Plot the evolving right output field after each status interval.
- **plot\_bulk\_LTR\_progress** (bool)  
Plot the LTR component of the bulk field when progress is printed.
- **plot\_bulk\_RTL\_progress** (bool)  
Plot the RTL component of the bulk field when progress is printed.
- **idx1** (int)  
Index of the left boundary mirror (inclusive).
- **idx2** (int)  
Index of the right boundary mirror (inclusive).
- **x\_plot\_shift, y\_plot\_shift** (int)  
Pixel offsets applied to all progress plots for visual clarity.

### Implementation details

The routine pre-computes the outer reflections and single-component transmissions at idx1 and idx2, then iterates no\_of\_trips times over the following sequence:

1. Add the left incident contribution and propagate LTR through the interior.
2. Optionally record bulk or right-hand outputs.
3. Reflect on the left side of component idx2.
4. Add the right incident contribution and propagate RTL back.
5. Optionally record bulk or left-hand outputs.
6. Reflect on the right side of component idx1.

Progress printing / plotting is throttled by .max\_time\_betw\_tic\_outputs. After the final cycle the last computed values are assigned to the public output- and bulk-field attributes.

### Return value

None. Results are accessed via .output\_field\_left, .output\_field\_right, .bulk\_field\_LTR, .bulk\_field\_RTL, and .bulk\_field.

```
.get_round_trip_T_mat_LTR_RTL(idx1, idx2, incl_left_refl)
```

### Method.

Returns the transmission matrix  $\mathbf{T}_{RT}$  for a single left-to-right / right-to-left round-trip that

1. starts on the *right* face of component `idx1`,
2. propagates LTR to the *left* face of component `idx2`,
3. reflects there,
4. propagates RTL back to the *right* face of component `idx1`,
5. and – if `incl_left_refl` is `True` – reflects once more on that interface.

### Arguments

- `idx1 (int)`

Index of the left boundary component; `-1` refers to the space just outside the cavity.

- `idx2 (int)`

Index of the right boundary component. If the two indices are supplied in reverse order they are swapped internally.

- `incl_left_refl (bool)`

When `True`, include the final reflection on the right face of component `idx1`. Setting this flag has no effect if `idx1 = -1`.

### Implementation details

The routine multiplies, in order,

1. the LTR transmission matrices `clsOptComponent2port.T_LTR_mat_tot` for all inner components `[idx1+1, idx2-1]`;
2. the left-side reflection matrix `clsOptComponent2port.R_L_mat_tot` of component `idx2`;
3. the RTL transmission matrices `clsOptComponent2port.T_RTL_mat_tot` for the return path `[idx2-1, idx1+1]`;
4. and, if requested, the right-side reflection matrix  $\mathbf{R}_R$  of component `idx1`.

### Return value

A transmission matrix of size  $N^2 \times N^2$  describing the net effect of the round-trip for any arbitrary light field. If the cavity is empty the function returns 1 (identity); if the round-trip terminates outside the cavity and `incl_left_refl` is `True`, it returns 0.

```
.get_inf_round_trip_R_L_mat(idx1, idx2)
```

### Method.

This method calls `.get_round_trip_T_mat_LTR_RTL(...)` to calculate the transmission matrix for a *single* LTR→RTL round-trip between component `idx1` and component `idx2`, and then – based on this – calculates the exact left reflection matrix

$\mathbf{R}_{\text{cav}}^{\text{left}}$  for an *infinite* sequence of LTR→RTL round-trips between component `idx1` and component `idx2`. The result can be applied directly to an incident field with `clsLightField.apply_TR_mat(...)`.

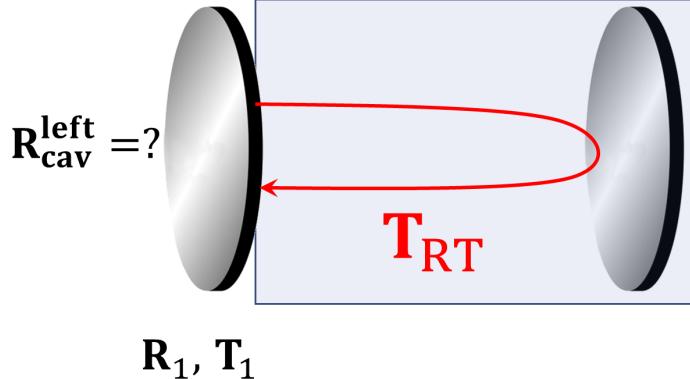


Figure F.2: Method `.get_inf_round_trip_R_L_mat(...)` calculates the left reflection matrix  $\mathbf{R}_{\text{cav}}^{\text{left}}$  from the single round-trip matrix  $\mathbf{T}_{\text{RT}}$ , as well as the left mirror's reflectivity  $\mathbf{R}_1$  and transmissivity  $\mathbf{T}_1$ .

### Arguments

- `idx1` (int)  
Index of the left boundary component.
- `idx2` (int)  
Index of the right boundary component (inclusive). If the two indices are given in reverse order they are swapped internally.

### Implementation details

1. Obtain the single-round-trip matrix  $\mathbf{T}_{\text{RT}}$  with `.get_round_trip_T_mat_LTR_RTL(...)` (`incl_left_refl = False`).
2. Form the product  $\mathbf{R}_1 \mathbf{T}_{\text{RT}}$  where  $\mathbf{R}_1$  is the right-side reflection matrix of component `idx1`.
3. Evaluate the infinite geometric series

$$(\mathbb{1} - \mathbf{R}_1 \mathbf{T}_{\text{RT}})^{-1}.$$

4. Assemble the final result

$$\mathbf{R}_{\text{cav}}^{\text{left}} = \mathbf{R}_1 + \mathbf{T}_1 \mathbf{T}_{\text{RT}} (\mathbb{1} - \mathbf{R}_1 \mathbf{T}_{\text{RT}})^{-1} \mathbf{T}_1,$$

where  $\mathbf{T}_1$  is the left-mirror transmission matrix.

### Return value

A matrix of size  $N^2 \times N^2$  giving the cavity's steady-state reflection on the left side. If the specified interval is empty (`idx2 ≤ idx1`) the function returns the scalar 0.

```
.get_round_trip_T_mat_RTL_LTR(idx1, idx2, incl_right_refl)
```

### Method.

Returns the transmission matrix  $\mathbf{T}_{RT}$  for a single right-to-left / left-to-right round-trip that

1. starts on the *left* face of component `idx2`,
2. propagates RTL to the *right* face of component `idx1`,
3. reflects there,
4. propagates LTR back to the *left* face of component `idx2`,
5. and – if `incl_right_refl` is `True` – reflects once more on that interface.

### Arguments

- `idx1` (`int`)  
Index of the left boundary component (inclusive).
- `idx2` (`int`)  
Index of the right boundary component; a value equal to `.component_count` refers to the space just outside the cavity. If the two indices are supplied in reverse order they are swapped internally.
- `incl_right_refl` (`bool`)  
When `True`, include the final reflection on the left face of component `idx2`. If `idx2 = component_count` this flag has no effect.

### Implementation details

The routine multiplies, in order,

1. the RTL transmission matrices `clsOptComponent2port.T_LTR_mat_tot` for all inner components  $[idx2-1, idx1+1]$ ;
2. the right-side reflection matrix `clsOptComponent2port.R_R_mat_tot` of component `idx1`;
3. the LTR transmission matrices `clsOptComponent2port.T_LTR_mat_tot` for the return path  $[idx1+1, idx2-1]$ ;
4. and, if requested, the left-side reflection matrix  $\mathbf{R}_L$  of component `idx2`.

### Return value

A transmission matrix of size  $N^2 \times N^2$  describing the net effect of the round-trip for any arbitrary light field. If the cavity is empty the function returns 1 (identity); if the round-trip terminates outside the cavity and `incl_right_refl` is `True`, it returns 0.

```
.get_inf_round_trip_R_R_mat(idx1, idx2)
```

### Method.

This method calls `.get_round_trip_T_mat_RTL_LTR(...)` to calculate the transmission matrix for a *single* RTL→LTR round-trip between component `idx1` and component `idx2`, and then – based on this – calculates the exact right reflection matrix

$\mathbf{R}_{\text{cav}}^{\text{right}}$  for an *infinite* sequence of RTL→LTR round-trips between the same two components. The result can be applied directly to an incident field with the method `clsLightField.apply_TR_mat(...)`.

### Arguments

- `idx1 (int)`  
Index of the left boundary component.
- `idx2 (int)`  
Index of the right boundary component (inclusive). If the two indices are given in reverse order they are swapped internally.

### Implementation details

1. Obtain the single-round-trip matrix  $\mathbf{T}_{\text{RT}}$  with `.get_round_trip_T_mat_RTL_LTR(...)` (`incl_right_refl = False`).
2. Form the product  $\mathbf{R}_2 \mathbf{T}_{\text{RT}}$  where  $\mathbf{R}_2$  is the left-side reflection matrix of component `idx2`.
3. Evaluate the infinite geometric series

$$(\mathbb{1} - \mathbf{R}_2 \mathbf{T}_{\text{RT}})^{-1}.$$

4. Assemble the final result

$$\mathbf{R}_{\text{cav}}^{\text{right}} = \mathbf{R}_2 + \mathbf{T}_2 \mathbf{T}_{\text{RT}} (\mathbb{1} - \mathbf{R}_2 \mathbf{T}_{\text{RT}})^{-1} \mathbf{T}_2,$$

where  $\mathbf{T}_2$  is the transmission matrix of the right mirror (component `idx2`).

### Return value

A matrix of size  $N^2 \times N^2$  giving the cavity's steady-state reflection on the right side. If the specified interval is empty (`idx2 ≤ idx1`) the function returns the scalar 0.

## F.2.5 Incident Fields

### `.incident_field_left`

#### Property.

Holds the left-hand incident field as a `clsLightField` instance (or `None` when no drive is present). Assigning a new value clears all cached results so that the next call to any output- or bulk-field routine triggers a fresh calculation.

#### Setter side effects

- Resets `.output_field_left` and `.output_field_right`.
- Resets `.bulk_field_LTR`, `.bulk_field_RTL`, and `.bulk_field_pos`.

#### Used by

- `.calc_output_field_left()` and `.calc_output_field_right()` (steady-state outputs).
- `.prop_mult_round_trips_LTR_RTL(...)` (iterative round-trip simulation).

- `.calc_bulk_field_from_left(...)` and `.calc_bulk_field_from_right(...)` (intracavity sampling).
- Any call path that eventually accesses `.output_field_left` or `.output_field_right`.

### `.incident_field_right`

#### **Property.**

Holds the right-hand incident field as a `clsLightField` instance (or `None` when no drive is present). Assigning a new value clears all cached results so that the next call to any output- or bulk-field routine triggers a fresh calculation.

#### **Setter side effects**

- Resets `.output_field_left` and `.output_field_right`.
- Resets `.bulk_field_LTR`, `.bulk_field RTL`, and `.bulk_field_pos`.

#### **Used by**

- `.calc_output_field_left()` and `.calc_output_field_right()`.
- `.prop_mult_round_trips_LTR_RTL(...)`.
- `.calc_bulk_field_from_left(...)` and `.calc_bulk_field_from_right(...)`.
- Any access to `.output_field_left` or `.output_field_right`.

## F.2.6 Output Fields

### `.output_field_left`

#### **Property.**

Returns the steady-state field that leaves the cavity on the *left* side. If the internal cache is `None` the value is generated on demand by calling `.calc_output_field_left()`; subsequent accesses reuse the stored result until any incident field is modified.

#### **Computed by**

- `.calc_output_field_left()` (invoked automatically when the cache is empty).

#### **Invalidated by**

- Writing to `.incident_field_left`.
- Writing to `.incident_field_right`.
- Calling `.clear_results()`.

### `.calc_output_field_left()`

#### **Method.**

Builds `.output_field_left` from the current incident drives. A fresh `clsLightField` instance is created, named “Output Field Left”, and incrementally filled with

- the left incident field multiplied by the cavity’s left reflection matrix `clsCavity.R_L_mat_tot`,
- the right incident field multiplied by the cavity’s RTL transmission matrix `clsCavity.T_RTL_mat_tot`.

Both `clsCavity.R_L_mat_tot` and `clsCavity.T_RTL_mat_tot` are inherited from `clsCavity`; if their cached values are `None` they are generated on demand via `.calc_R_L_mat_tot()` and `.calc_T_RTL_mat_tot()`.

#### **Return value**

`None`. The resulting field is stored internally and becomes available through `.output_field_left`.

### **`.output_field_right`**

#### **Property.**

Returns the steady-state field that leaves the cavity on the *right* side. If the internal cache is `None` the value is generated on demand by calling `.calc_output_field_right()`; subsequent accesses reuse the stored result until any incident field is modified.

#### **Computed by**

- `.calc_output_field_right()` (invoked automatically when the cache is empty).

#### **Invalidated by**

- Writing to `.incident_field_left`.
- Writing to `.incident_field_right`.
- Calling `.clear_results()`.

### **`.calc_output_field_right()`**

#### **Method.**

Builds `.output_field_right` from the current incident drives. A fresh `clsLightField` instance is created, named “Output Field Right”, and incrementally filled with

- the right incident field multiplied by the cavity’s right reflection matrix `clsCavity.R_R_mat_tot`,
- the left incident field multiplied by the cavity’s LTR transmission matrix `clsCavity.T_LTR_mat_tot`.

Both `clsCavity.R_R_mat_tot` and `clsCavity.T_LTR_mat_tot` are inherited from `clsCavity`; if their cached values are `None` they are generated on demand via `.calc_R_R_mat_tot()` and `.calc_T_LTR_mat_tot()`.

#### **Return value**

`None`. The resulting field is stored internally and becomes available through `.output_field_right`.

### F.2.7 Intracavity Field

`.calc_bulk_field_from_left(idx)`

#### Method.

Computes the steady-state intracavity field that appears *immediately to the right* of component `idx` by back-propagating the left incident–output pair. The resulting directional components are stored and made available in `.bulk_field_LTR` (LTR) and `.bulk_field_RTL` (RTL); the superposition of the LTR and RTL fields can be obtained via `.bulk_field`.

#### Arguments

- `idx` (`int`)

Index of the interface of interest. Values  $\leq -1$  select the left exterior; values  $\geq .component\_count - 1$  select the right exterior.

#### Implementation details

1. Special-case indices outside the cavity reuse the already available incident/output fields.
2. For interior positions the method builds the cumulative inverse transfer matrix  $M^{-1}$  from `idx` down to the leftmost component.
3. The four  $2 \times 2$  sub-blocks of  $M^{-1}$  are extracted and applied to the stored left output and incident fields to obtain the LTR and RTL intracavity field to the right of component `idx`.
4. Stores the results in `.bulk_field_LTR`, `.bulk_field_RTL` and `.bulk_field`, and sets `.bulk_field_pos` to `idx`.

Because the full inverse transfer matrix must be assembled, the runtime and memory footprint can exceed that of the global transfer–scattering-matrix approach if no prior calculation has been performed.

#### Return value

None. The directional bulk fields are stored internally and can be accessed through the properties mentioned above.

`.calc_bulk_field_from_right(idx)`

#### Method.

Computes the steady-state intracavity field that appears *immediately to the right* of component `idx` by back-propagating the right incident–output pair. The resulting directional components are stored and made available in `.bulk_field_LTR` (LTR) and `.bulk_field_RTL` (RTL); the superposition of the LTR and RTL fields can be obtained via `.bulk_field`.

#### Arguments

- `idx` (`int`) –

Index of the interface of interest. Values  $\leq -1$  select the left exterior; values  $\geq .component\_count - 1$  select the right exterior.

### Implementation details

1. Special-case indices outside the cavity reuse the already available incident/output fields.
2. For interior positions the method builds the cumulative forward transfer matrix  $\mathbf{M}$  from `idx+1` up to the rightmost component.
3. The four  $2 \times 2$  sub-blocks of  $\mathbf{M}$  are extracted and applied to the stored right incident and output fields to obtain the LTR and RTL intracavity field to the right of component `idx`.
4. Stores the results in `.bulk_field_LTR`, `.bulk_field_RTL` and `.bulk_field`, and sets `.bulk_field_pos` to `idx`.

Because the full transfer matrix must be assembled, the runtime and memory footprint can exceed that of the global transfer-scattering-matrix approach if no prior calculation has been performed.

### Return value

None. The directional bulk fields are stored internally and can be accessed through the properties mentioned above.

#### `.bulk_field_pos`

##### Read-only property.

Returns the component index `idx` for which `.bulk_field_LTR` or `.bulk_field_RTL` were last computed. The index refers to the field *immediately to the right* of the component; a value of `-1` denotes the left exterior, while `.component_count-1` denotes the right exterior. The property is `None` until either `.calc_bulk_field_from_left(...)` or `.calc_bulk_field_from_right(...)` is executed.

##### Set by

- `.calc_bulk_field_from_left(...)`.
- `.calc_bulk_field_from_right(...)`.

##### Cleared by

- Writing to `.incident_field_left`.
- Writing to `.incident_field_right`.
- Calling `.clear_results()`.

#### `.bulk_field_LTR`

##### Read-only property.

Returns the left-to-right component of the intracavity bulk field that was generated by the most recent call to `.calc_bulk_field_from_left(...)` or `.calc_bulk_field_from_right(...)`. The value is a `clsLightField` instance; it is `None` until either method has been executed.

##### Cleared by

- Writing to `.incident_field_left`.

- Writing to `.incident_field_right`.
- Calling `.clear_results()`.

#### `.bulk_field_RTL`

##### **Read-only property.**

Returns the right-to-left component of the intracavity bulk field that was generated by the most recent call to `.calc_bulk_field_from_left(...)` or `.calc_bulk_field_from_right(...)`. The value is a `clsLightField` instance; it is `None` until either method has been executed.

##### **Cleared by**

- Writing to `.incident_field_left`.
- Writing to `.incident_field_right`.
- Calling `.clear_results()`.

#### `.bulk_field`

##### **Read-only property.**

Returns the superposition of the left-to-right and right-to-left bulk fields that were generated by the most recent call to `.calc_bulk_field_from_left(...)` or `.calc_bulk_field_from_right(...)`. The value is a `clsLightField` instance; if both directional components are `None` the returned field is empty (black).

##### **Cleared by**

- Writing to `.incident_field_left`.
- Writing to `.incident_field_right`.
- Calling `.clear_results()`.

### F.2.8 Reflection and Transmission Matrices

#### `.calc_R_L_mat_tot()`

##### **Method.**

Calculates the left reflection matrix `clsCavity.R_L_mat_tot`, which corresponds to the upper-left block of the cavity's scattering matrix `clsCavity.S_bmat_tot`. The result is cached and made available in `clsCavity.R_L_mat_tot`.

##### **Implementation details**

1. If the global transfer matrix `clsCavity.M_bmat_tot` is `None`, it is generated on demand with `clsCavity.calc_M_bmat_tot(...)`.
2. Let  $\mathbf{M}$  be that transfer matrix. Following the block conversion rules in [4], the left-hand reflection matrix is obtained from  $\mathbf{M}$  via

$$\mathbf{R}_L = \mathbf{M}_{12} \mathbf{M}_{22}^{-1},$$

where the first index denotes the block row and the second index the block column.

3. The result is stored and made available in `clsCavity.R_L_mat_tot`.

#### **Return value**

None. The matrix is stored internally and can be retrieved from the property mentioned above.

#### `.calc_R_R_mat_tot()`

##### **Method.**

Calculates the right reflection matrix `clsCavity.R_R_mat_tot`, which corresponds to the bottom-right block of the cavity's scattering matrix `clsCavity.S_bmat_tot`. The result is cached and made available in `clsCavity.R_R_mat_tot`.

##### **Implementation details**

1. If the global transfer matrix `clsCavity.M_bmat_tot` is `None`, it is generated on demand with `clsCavity.calc_M_bmat_tot(...)`.
2. Let  $\mathbf{M}$  be that transfer matrix. Following the block conversion rules in [4], the right-hand reflection matrix is obtained from  $\mathbf{M}$  via

$$\mathbf{R}_R = -\mathbf{M}_{22}^{-1} \mathbf{M}_{21},$$

where the first index denotes the block row and the second index the block column.

3. The result is stored and made available in `clsCavity.R_R_mat_tot`.

#### **Return value**

None. The matrix is stored internally and can be retrieved from the property mentioned above.

#### `.calc_T_LTR_mat_tot()`

##### **Method.**

Calculates the left-to-right transmission matrix `clsCavity.T_LTR_mat_tot`, which corresponds to the bottom-left block of the cavity's scattering matrix `clsCavity.S_bmat_tot`. The result is cached and made available in `clsCavity.T_LTR_mat_tot`.

##### **Implementation details**

1. If the global transfer matrix `clsCavity.M_bmat_tot` is `None`, it is generated on demand with `clsCavity.calc_M_bmat_tot(...)`.
2. Let  $\mathbf{M}$  be that transfer matrix. Following the block conversion rules in [4], the left-to-right transmission matrix is obtained from  $\mathbf{M}$  via

$$\mathbf{T}_{LTR} = \mathbf{M}_{22}^{-1},$$

where the first index denotes the block row and the second index the block column.

3. The result is stored and made available in `clsCavity.T_LTR_mat_tot`.

#### **Return value**

None. The matrix is stored internally and can be retrieved from the property mentioned above.

### `.calc_T_RTL_mat_tot()`

#### **Method.**

Calculates the right-to-left transmission matrix `clsCavity.T_RTL_mat_tot`, which corresponds to the top-right block of the cavity's scattering matrix `clsCavity.S_bmat_tot`. The result is cached and made available in `clsCavity.T_RTL_mat_tot`.

#### **Implementation details**

1. If the global transfer matrix `clsCavity.M_bmat_tot` is `None`, it is generated on demand with `clsCavity.calc_M_bmat_tot(...)`.
2. Let  $\mathbf{M}$  be that transfer matrix. Following the block conversion rules in [4], the right-to-left transmission matrix is obtained from  $\mathbf{M}$  via

$$\mathbf{T}_{\text{RTL}} = \mathbf{M}_{11} - \mathbf{M}_{12} \mathbf{M}_{22}^{-1} \mathbf{M}_{21}.$$

3. The result is stored and made available in `clsCavity.T_RTL_mat_tot`.

#### **Return value**

`None`. The matrix is stored internally and can be retrieved from the property mentioned above.

### F.2.9 Full Transfer and Scattering Block Matrix

The full transfer block matrix for the whole cavity can be calculated via the base class method `clsCavity.calc_M_bmat_tot(...)` and accessed via `clsCavity.M_bmat_tot`. Similarly, the full scattering block matrix for the whole cavity can be accessed via the base class property `clsCavity.S_bmat_tot`; however, the method to calculate this scattering matrix is implemented here in `clsCavity1path`.

### `.calc_S_bmat_tot()`

#### **Method.**

Calculates the full scattering block matrix `clsCavity.S_bmat_tot` of the cavity from the cavity's transfer block matrix `clsCavity.M_bmat_tot`, using the matrix conversion formulas given in [4]. The result is cached and made available in `clsCavity.S_bmat_tot`.

#### **Implementation details**

1. If any of the sub-blocks of the scattering matrix is still undefined, it is computed using the following expressions derived from the total transfer matrix  $\mathbf{M}$  of the cavity:

$$\mathbf{R}_L = \mathbf{M}_{12} \mathbf{M}_{22}^{-1}$$

$$\mathbf{T}_{\text{RTL}} = \mathbf{M}_{11} - \mathbf{M}_{12} \mathbf{M}_{22}^{-1} \mathbf{M}_{21}$$

$$\mathbf{T}_{\text{LTR}} = \mathbf{M}_{22}^{-1}$$

$$\mathbf{R}_R = -\mathbf{M}_{22}^{-1} \mathbf{M}_{21}$$

2. These four blocks are assembled into the full scattering block matrix:

$$\mathbf{S} = \begin{pmatrix} \mathbf{R}_L & \mathbf{T}_{\text{RTL}} \\ \mathbf{T}_{\text{LTR}} & \mathbf{R}_R \end{pmatrix}$$

**Return value**

None. The result is stored internally and can be retrieved from `clsCavity.S_bmat_tot`.

**F.2.10 Caching and Memory**

`.clear_results()`

**Method.**

Clears all internally cached field results, including output and bulk fields. This method should be called explicitly if you want to discard previous results and force a fresh computation in subsequent method calls.

**Affected fields**

- `.output_field_left`
- `.output_field_right`
- `.bulk_field_LTR`
- `.bulk_field_RTL`
- `.bulk_field_pos`

**Return value**

None.

**F.2.11 File I/O Utilities**

`.save_R_L_mat_tot()`

**Method.**

Saves the cached left full resolution reflection matrix `clsCavity.R_L_mat_tot` to disk. The routine is a convenience wrapper around the base-class method `clsCavity.save_mat(...)`, which gets called with the base file name "R\_L\_mat\_tot". The complete file name and path is then generated by `clsCavity._full_file_name(...)`.

`.load_R_L_mat_tot()`

**Method.**

Loads the left full-resolution reflection matrix `clsCavity.R_L_mat_tot` that was previously saved via `.save_R_L_mat_tot()`, using the base-class method `clsCavity.load_mat(...)`.

`.save_R_L_mat_fov()`

**Method.**

Saves the cached left field-of-view resolution reflection matrix `clsCavity.R_L_mat_fov` to disk. The routine is a convenience wrapper around the base-class method `clsCavity.save_mat(...)`, which gets called with the base file name "R\_L\_mat\_fov".

The complete file name and path is then generated by `clsCavity._full_file_name(...)`.

### `.load_R_L_mat_fov()`

#### **Method.**

Loads the left field-of-view reflection matrix `clsCavity.R_L_mat_fov` that was previously saved via `.save_R_L_mat_fov()`, using the base-class method `clsCavity.load_mat(...)`.

### `.save_R_R_mat_tot()`

#### **Method.**

Saves the cached right full resolution reflection matrix `clsCavity.R_R_mat_tot` to disk. The routine is a convenience wrapper around the base-class method `clsCavity.save_mat(...)`, which gets called with the base file name "R\_R\_mat\_tot". The complete file name and path is then generated by `clsCavity._full_file_name(...)`.

### `.load_R_R_mat_tot()`

#### **Method.**

Loads the right full-resolution reflection matrix `clsCavity.R_R_mat_tot` that was previously saved via `.save_R_R_mat_tot()`, using the base-class method `clsCavity.load_mat(...)`.

### `.save_R_R_mat_fov()`

#### **Method.**

Saves the cached right field-of-view resolution reflection matrix `clsCavity.R_R_mat_fov` to disk. The routine is a convenience wrapper around the base-class method `clsCavity.save_mat(...)`, which gets called with the base file name "R\_R\_mat\_fov". The complete file name and path is then generated by `clsCavity._full_file_name(...)`.

### `.load_R_R_mat_fov()`

#### **Method.**

Loads the right field-of-view reflection matrix `clsCavity.R_R_mat_fov` that was previously saved via `.save_R_R_mat_fov()`, using the base-class method `clsCavity.load_mat(...)`.

### `.save_T_LTR_mat_tot()`

#### **Method.**

Saves the cached left-to-right full resolution transmission matrix `clsCavity.T_LTR_mat_tot` to disk. The routine is a convenience wrapper around the base-class method `clsCavity.save_mat(...)`, which gets called with the base

file name "T\_LTR\_mat\_tot". The complete file name and path is then generated by `clsCavity._full_file_name(...)`.

#### `.load_T_LTR_mat_tot()`

##### **Method.**

Loads the left-to-right full resolution transmission matrix `clsCavity.T_LTR_mat_tot` that was previously saved via `.save_T_LTR_mat_tot()`, using the base-class method `clsCavity.load_mat(...)`.

#### `.save_T_LTR_mat_fov()`

##### **Method.**

Saves the cached left-to-right field-of-view resolution transmission matrix `clsCavity.T_LTR_mat_fov` to disk. The routine is a convenience wrapper around the base-class method `clsCavity.save_mat(...)`, which gets called with the base file name "T\_LTR\_mat\_fov". The complete file name and path is then generated by `clsCavity._full_file_name(...)`.

#### `.load_T_LTR_mat_fov()`

##### **Method.**

Loads the left-to-right field-of-view transmission matrix `clsCavity.T_LTR_mat_fov` that was previously saved via `.save_T_LTR_mat_fov()`, using the base-class method `clsCavity.load_mat(...)`.

#### `.save_T_RTL_mat_tot()`

##### **Method.**

Saves the cached right-to-left full resolution transmission matrix `clsCavity.T_RTL_mat_tot` to disk. The routine is a convenience wrapper around the base-class method `clsCavity.save_mat(...)`, which gets called with the base file name "T\_RTL\_mat\_tot". The complete file name and path is then generated by `clsCavity._full_file_name(...)`.

#### `.load_T_RTL_mat_tot()`

##### **Method.**

Loads the right-to-left full resolution transmission matrix `clsCavity.T_RTL_mat_tot` that was previously saved via `.save_T_RTL_mat_tot()`, using the base-class method `clsCavity.load_mat(...)`.

#### `.save_T_RTL_mat_fov()`

##### **Method.**

Saves the cached right-to-left field-of-view resolution transmission matrix `clsCavity.T_RTL_mat_fov` to disk. The routine is a convenience wrapper around the base-class method `clsCavity.save_mat(...)`, which gets called with the base

file name "T\_RTL\_mat\_fov". The complete file name and path is then generated by `clsCavity._full_file_name(...)`.

`.load_T_RTL_mat_fov()`

#### Method.

Loads the right-to-left field-of-view transmission matrix `clsCavity.T_RTL_mat_fov` that was previously saved via `.save_T_RTL_mat_fov()`, using the base-class method `clsCavity.load_mat(...)`.

## F.3 Class `clsCavity2path`

The class `clsCavity2path` is a concrete subclass of `clsCavity` that represents a *ring* cavity with two distinct optical paths, labelled `Path.A` and `Path.B`. A typical example is the ring cavity shown in Figure F.3. Each path may contain its own sequence of 2-port components, and the two paths can be coupled through dedicated 4-port elements such as `clsBeamSplitterMirror`.

All generic functionality of the base class `clsCavity` – spatial grid management, global transfer/scattering matrices, caching infrastructure, ... – is inherited unchanged. The following sections document only the additional features relevant to two-path operation.

#### Create and configure the cavity.

Before a simulation can be performed, the cavity must be created and configured:

1. Instantiate the cavity via the constructor `clsCavity2path(...)`.
2. Specify the vacuum wavelength with `.Lambda` or `.Lambda_nm` and set up the spatial grid (pixel count, physical size) through `.grid.set_opt_res_based_on_sidelength(...)` or `.grid.set_opt_res_tot_based_on_res_fov(...)`.
3. Create and append every optical element in geometric order:
  - Use `.add_4port_component(component)` for 4-port devices that couple the two paths (e.g. `clsBeamSplitterMirror`, `clsTransmissionMixer`, ...).
  - Use `.add_2port_component(A, B)` for individual 2-port components placed in `Path.A` and/or `Path.B` (mirrors, lenses, free-space sections, ...).

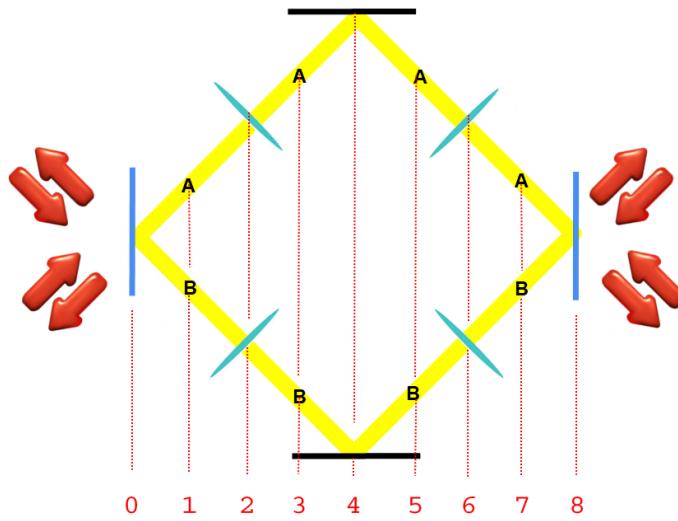


Figure F.3: Example of a simple ring cavity comprising two beam splitter mirrors (left and right side), four lenses, and two turning mirrors (top and bottom), arranged in a  $4f$ -configuration.

Figure F.3 shows an example of a simple two-path ring cavity that can be simulated using `clsCavity2path`. It consists of the following 9 4-port-components (numbered 0 to 8), which must be added in consecutive order using either `.add_4port_component(component)` for components that link the two paths (e.g., beam splitter mirrors), or `.add_2port_component(A, B)` for components that lie on a single path only.

0. Beam-splitter mirror coupling paths A and B, simulated with `clsBeamSplitterMirror` (`.add_4port_component(component)`).
1. Identical free-space section on both paths, realised via `clsOptComponentAdapter` that hosts a single `clsPropagation` instance and assigns it to `Path.A` and `Path.B` (`.add_2port_component(A, B)`).
2. Common lens of focal length  $f$  on both paths, again wrapped in `clsOptComponentAdapter` and hosting a `clsThinLens` (`.add_2port_component(A, B)`).
3. Second identical free-space section in both paths; the same `clsPropagation` instance used in item 1 can be reused here.
4. Top and bottom turning mirrors, implemented with `clsTransmissionMixer` (`.add_2port_component(A, B)`).
5. Third identical free-space section in both paths (identical to item 1).
6. Second common lens of focal length  $f$  in both paths (identical to item 2).
7. Fourth identical free-space section in both paths (identical to item 1).
8. Second beam-splitter mirror, identical to item 0 (`.add_4port_component(component)`).

### Choose a simulation method.

The following simulation routes are available for `clsCavity2path`:

- **Direct field propagation (diagnostics only)**

A pair of pre-existing fields – one assigned to `Path.A`, the other to `Path.B` – can be propagated left-to-right (LTR) or right-to-left (RTL) through any contiguous sequence of components via `.prop(...)`. Because `clsCavity2path` provides no `.reflect()` helper, this option is intended for studying the response of individual elements or arbitrary sections of the resonator; closed round-trips have to be scripted manually.

- **Transfer and scattering matrix formalism**

For real cavity simulations the full  $4 \times 4$  block transfer matrix of the entire stack must be evaluated.

1. Assign the driving incident fields for each relevant side and path via `.set_incident_field(...)`.
2. Retreive the corresponding output at any relevant side and for any relevant path with `.get_output_field(...)`.

The first call of `.get_output_field(...)` implicitly invokes `.calc_M_bmat_tot(...)` to assemble the global transfer matrix, followed by the calculation of the  $2 \times 2$  reflection and transmission block matrices `.R_L_mat_tot`, `.R_R_mat_tot`, `.T_LTR_mat_tot`, and `.T RTL_mat_tot`. Although slower and more memory-intensive than plain propagation, this route yields the complete steady-state response of an arbitrarily coupled two-path cavity.

- **Steady-state intracavity field**

You may sample the intracavity field at an arbitrary interface index `bulk_idx`. Two formally equivalent helpers exist:

- `.calc_bulk_field_from_left(...)`: back-propagates the *left* incident–output pair and is most efficient for positions nearer the left coupler.
- `.calc_bulk_field_from_right(...)`: analogous routine starting from the *right* side; faster for positions closer to the right coupler.

After the routine has finished, the directional contributions can be retrieved via `.get_bulk_field_LTR(...)` and `.get_bulk_field_RTL(...)`; the superposition is returned by `.get_bulk_field(...)`. If no transfer matrix has been cached beforehand, these calls can be as expensive as (or even slower than) the full transfer-scattering evaluation described above.

### F.3.1 Initialization

```
clsCavity2path(name, full_precision=True)
```

#### Constructor.

Initializes a two-path cavity subclass.

#### Parameters

- `name`  
Human-readable identifier used in log messages and file names.
- `full_precision`

If `True` (default) internal arrays use `float64/complex128`; otherwise `float32/complex64` to reduce memory.

### Behaviour

- Selects the numeric precision and creates an empty `clsGrid` instance.
- Creates separate empty containers for the left and right incident and output fields on `Path.A` and `Path.B`.
- Creates empty container for the left-to-right and right-to-left bulk fields in both paths.
- Invokes the base-class constructor `clsCavity(...)` to complete initialization.

### Typical workflow

1. Specify the working wavelength via `.Lambda` or `.Lambda_nm`.
2. Initialize the sampling grid via `.grid.set_opt_res_based_on_sidelength(...)` or `.grid.set_opt_res_tot_based_on_res_fov(...)`.
3. Append optical elements in geometric order:
  - Use `.add_4port_component(component)` for 4-port couplers that link the two paths.
  - Use `.add_2port_component(A, B)` for individual 2-port components placed on `Path.A` and/or `Path.B`.

## F.3.2 Component Management

### `.add_4port_component(component)`

#### Method.

Appends a *four-port* optical element (which is any sub-class of `clsOptComponent4port`, e.g. an instance of `clsBeamSplitterMirror`) to the cavity's component list `clsCavity.components` and establishes the internal back-reference from the optical element to `clsCavity2path` and its `.grid` instance via `clsOptComponent._connect_to_cavity(...)`.

#### Arguments

- `component`

An instance of `clsOptComponent4port` that couples `Path.A` and `Path.B`. The same object may be reused multiple times if the layout contains identical couplers.

#### Behavior

1. The component is appended to `clsCavity.components`; it can subsequently be accessed through `clsCavity.get_component(...)`.
2. `clsCavity.component_count` is implicitly incremented.

#### Usage note

This routine merely registers the four-port component. No optical computation is

triggered. Insert all components in geometric order before calling propagation or matrix routines.

`.add_2port_component(A, B)`

#### Method.

Registers up to two independent two-port elements – one assigned to `Path.A`, one to `Path.B` – at the *same list position* within the cavity’s component sequence. (The actual physical distances travelled in the two paths may differ; only the ordering index is shared.) Internally the pair is wrapped in a `clsOptComponentAdapter` so that the cavity still sees a single component entry.

#### Arguments

- A  
An instance of `clsOptComponent2port` to be placed on path A, or `None`.
- B  
An instance of `clsOptComponent2port` to be placed on path B, or `None`.

#### Behavior

1. If both arguments are `None` the call returns immediately (no-op).
2. The method searches the existing component list for an `clsOptComponentAdapter` that already pairs the same objects; if found, that adapter is reused and merely referenced again.
3. Otherwise a new `clsOptComponentAdapter` is created, appended to `clsCavity.components`, and connected to the grid via `.connect_to_cavity(...)`. The supplied two-port component(s) are then attached to the adapter on their respective paths. `clsCavity.component_count` is incremented accordingly.

#### Usage note

It is perfectly valid (and memory-efficient) to pass the *same* `clsPropagation`, `clsThinLens`, or similar instance for multiple positions if the optical layout repeats identical segments.

### F.3.3 Propagation

`.prop(in_A, in_B, idx_from, idx_to, direction)`

#### Method.

Propagates a pair of light fields – `in_A` on `Path.A` and `in_B` on `Path.B` – through a contiguous block of components in a single direction, either left-to-right (`Dir.LTR`) or right-to-left (`Dir.RTL`). Both boundary indices are inclusive; setting `idx_from = idx_to` propagates the fields through one individual element. The routine internally selects the most efficient representation (position space or  $k$ -space) for every interface, based on the declared preferences of the current component and its neighbour.

**Arguments**

- **in\_A**  
`clsLightField` instance supplying the incident amplitudes on path A, or `None` (interpreted as a black field).
- **in\_B**  
Same for path B.
- **idx\_from, idx\_to**  
Zero-based indices delimiting the propagation span. The method swaps the values if necessary and clamps them to the valid range [0, `clsCavity.component_count`–1].
- **direction**  
Propagation direction: `Dir.LTR` or `Dir.RTL`.

**Return value**

Two `clsLightField` objects (`out_A, out_B`) containing the transmitted complex amplitudes on paths A and B, respectively.

**.get\_dist\_phys(idx1, idx2)**

**Method.**

Returns the cumulative *physical* length (in meters) of the cavity segment that starts at the *left* face of component `idx1` and ends at the *right* face of component `idx2`, *separately* for path `Path.A` and path `Path.B`.

**Arguments**

- **idx1**  
Zero-based index of the first component (default 0). Values below 0 are clipped to the leftmost component.
- **idx2**  
Zero-based index of the last component. The default value 999 intentionally exceeds any realistic length and is clipped to the rightmost component.

**Return value**

A tuple ( $L_A, L_B$ ) giving the physical length of the selected segment on path A and path B, respectively, in meters.

**.get\_dist\_opt(idx1, idx2)**

**Method.**

Calculates the cumulative *optical* path length (in meters) for the segment that begins at the left face of component `idx1` and ends at the right face of component `idx2`, reported separately for path `Path.A` and path `Path.B`. Each element contributes its individual `dist_opt` pair, which already accounts for any refractive index present inside that component.

**Arguments**

- `idx1`

Index of the first component (default 0). Negative values are clamped to the left edge of the cavity.

- `idx2`

Index of the last component (default 999). Values beyond the list length are clamped to the rightmost component.

**Return value**

A tuple  $(L_A, L_B)$  containing the optical distance on path A and path B, respectively, in meters.

**F.3.4 Incident Fields****`.set_incident_field(side, path, field)`****Method.**

Assigns a driving field to one of the four input ports of a two-path cavity: (`Side.LEFT` versus `Side.RIGHT`) and (`Path.A` versus `Path.B`).

**Arguments**

- `side`

Port location (`Side.LEFT` or `Side.RIGHT`).

- `path`

Optical branch (`Path.A` or `Path.B`).

- `field (clsLightField)`

Complex-amplitude distribution to be injected. Pass `None` to clear a previously defined field.

**Behavior**

1. Stores the field in the corresponding internal slot.
2. Invalidates every quantity that depends on the incident field:
  - all cached output fields on both sides and paths,
  - both directional bulk-field components,
  - the cached bulk-field position tag.

Subsequent calls to `.get_output_field(...)` or the bulk-field routines will therefore trigger a fresh computation.

**Return value**

None.

**`.get_incident_field(side, path)`****Method.**

Retrieves the light field that is currently assigned to the specified input port of the two-path cavity.

**Arguments**

- **side**  
Port location (`Side.LEFT` or `Side.RIGHT`).
- **path**  
Optical branch (`Path.A` or `Path.B`).

**Return value**

The `clsLightField` instance previously supplied via `.set_incident_field(...)`, or `None` if no field is defined for that port.

**F.3.5 Output Fields**

`.get_output_field(side, path)`

**Method.**

Returns the steady-state output field for any of the four cavity ports (`Side.LEFT` / `Side.RIGHT`, `Path.A` / `Path.B`). If the requested quantity is not yet cached the routine delegates the calculation to the private helper `._calc_output_field(...)`, and caches the result.

**Arguments**

- **side** – Output port location (`Side.LEFT` or `Side.RIGHT`).
- **path** – Optical path (`Path.A` or `Path.B`).

**Behavior**

1. Looks up the cached field corresponding to `(side, path)`.
2. If the cache entry is `None`, invokes `._calc_output_field(...)` to assemble the field from the current incident fields and the  $2 \times 2$  reflection / transmission block matrices `clsCavity.R_L_mat_tot`, `clsCavity.R_R_mat_tot`, `clsCavity.T_LTR_mat_tot`, and `clsCavity.T RTL_mat_tot` (generated on demand if not available).
3. Returns the (now cached) `clsLightField` instance.

**Return value**

A `clsLightField` containing the complex amplitudes at the requested output port.

`._calc_output_field(side, path)`

**Method.**

Internal method that assembles and caches the steady-state output field for a specific port of a two-path cavity. Depending on the requested `(side, path)` the routine extracts either the first or the second row of the relevant  $2 \times 2$  reflection and transmission block matrices and applies those sub-blocks to the presently stored left and right incident fields  $\mathbf{a}_{in}^{L,A}$ ,  $\mathbf{a}_{in}^{L,B}$ ,  $\mathbf{a}_{in}^{R,A}$ , and  $\mathbf{a}_{in}^{R,B}$ .

### Arguments

- `side` – `Side.LEFT` or `Side.RIGHT`.
- `path` – `Path.A` (row 0) or `Path.B` (row 1).

### Computation pattern

`Side.LEFT / Path.A:`

$$\mathbf{a}_{\text{out}}^{\text{L,A}} = \mathbf{R}_L^{(1,1)} \mathbf{a}_{\text{in}}^{\text{L,A}} + \mathbf{R}_L^{(1,2)} \mathbf{a}_{\text{in}}^{\text{L,B}} + \mathbf{T}_{\text{RTL}}^{(1,1)} \mathbf{a}_{\text{in}}^{\text{R,A}} + \mathbf{T}_{\text{RTL}}^{(1,2)} \mathbf{a}_{\text{in}}^{\text{R,B}}$$

`Side.LEFT / Path.B:`

$$\mathbf{a}_{\text{out}}^{\text{L,B}} = \mathbf{R}_L^{(2,1)} \mathbf{a}_{\text{in}}^{\text{L,A}} + \mathbf{R}_L^{(2,2)} \mathbf{a}_{\text{in}}^{\text{L,B}} + \mathbf{T}_{\text{RTL}}^{(2,1)} \mathbf{a}_{\text{in}}^{\text{R,A}} + \mathbf{T}_{\text{RTL}}^{(2,2)} \mathbf{a}_{\text{in}}^{\text{R,B}}$$

`Side.RIGHT / Path.A:`

$$\mathbf{a}_{\text{out}}^{\text{R,A}} = \mathbf{T}_{\text{LTR}}^{(1,1)} \mathbf{a}_{\text{in}}^{\text{L,A}} + \mathbf{T}_{\text{LTR}}^{(1,2)} \mathbf{a}_{\text{in}}^{\text{L,B}} + \mathbf{R}_R^{(1,1)} \mathbf{a}_{\text{in}}^{\text{R,A}} + \mathbf{R}_R^{(1,2)} \mathbf{a}_{\text{in}}^{\text{R,B}}$$

`Side.RIGHT / Path.B:`

$$\mathbf{a}_{\text{out}}^{\text{R,B}} = \mathbf{T}_{\text{LTR}}^{(2,1)} \mathbf{a}_{\text{in}}^{\text{L,A}} + \mathbf{T}_{\text{LTR}}^{(2,2)} \mathbf{a}_{\text{in}}^{\text{L,B}} + \mathbf{R}_R^{(2,1)} \mathbf{a}_{\text{in}}^{\text{R,A}} + \mathbf{R}_R^{(2,2)} \mathbf{a}_{\text{in}}^{\text{R,B}}$$

Here  $\mathbf{R}_L$ ,  $\mathbf{R}_R$ ,  $\mathbf{T}_{\text{LTR}}$ , and  $\mathbf{T}_{\text{RTL}}$  denote the  $2 \times 2$  block matrices `clsCavity.R_L_mat_tot`, `.R_R_mat_tot`, `.T_LTR_mat_tot`, and `.T RTL_mat_tot`; if any of them is still `None` they are first generated via the corresponding `calc_...` methods. The notation  $\mathbf{R}_L^{(i,j)}$ , for instance, refers to the submatrix in the  $i$ -th row and  $j$ -th column of the  $\mathbf{R}_L$  block matrix.  $\mathbf{a}_{\text{in}}^{\text{L,A}}$  and  $\mathbf{a}_{\text{in}}^{\text{L,B}}$  denote the left-incident fields in paths A and B, respectively, as vectors of Fourier coefficients; similarly,  $\mathbf{a}_{\text{in}}^{\text{R,A}}$  and  $\mathbf{a}_{\text{in}}^{\text{R,B}}$  denote the right-incident fields. The corresponding output field vectors are written as  $\mathbf{a}_{\text{out}}^{\text{L,A}}$ ,  $\mathbf{a}_{\text{out}}^{\text{L,B}}$ ,  $\mathbf{a}_{\text{out}}^{\text{R,A}}$ , and  $\mathbf{a}_{\text{out}}^{\text{R,B}}$ .

### Caching

The synthesized `clsLightField` is stored in the appropriate private cache slot so that subsequent calls to `.get_output_field(...)` can return it instantly.

### Return value

None. The field is stored internally and can be accessed via `.get_output_field(...)`.

#### F.3.6 Intracavity Field

`.calc_bulk_field_from_left(idx)`

### Method.

Back-propagates the stored *left* incident–output pair to obtain the steady-state intracavity field *immediately to the right* of component `idx` for both paths. The left-to-right and right-to-left contributions can be accessed after the computation via `.get_bulk_field_LTR(...)` and `.get_bulk_field_RTL(...)`; the coherent superposition of these directional contributions is available through `.get_bulk_field(...)`.

## Arguments

- `idx (int)`

Interface index of interest. Values  $\leq -1$  reference the left exterior, while values  $\geq .\text{component\_count}-1$  address the right exterior.

## Implementation details

1. **Boundary shortcuts.** Indices outside the cavity return the corresponding incident and output fields without further processing.
2. **Inverse transfer matrix stack.** The routine forms the cumulative inverse stack

$$\mathbf{M}_\Sigma^{-1} = \mathbf{M}_{\text{idx}}^{-1} \mathbf{M}_{\text{idx}-1}^{-1} \dots \mathbf{M}_0^{-1},$$

where each  $\mathbf{M}_k^{-1}$  is the inverse  $4 \times 4$  transfer block matrix of component  $k$ , obtained from `clsOptComponent4port.inv_M_bmat_tot`. The product aggregates the inverse transfer matrices from the left boundary up to the interface immediately right of component `idx`.

3. **Field reconstruction.** The block matrix  $\mathbf{M}_\Sigma^{-1}$  consists of sixteen sub-blocks  $\mathbf{M}_{(p,q)}^{-1}$  ( $p, q \in \{1, 2, 3, 4\}$ ). Applying the first-row blocks to the left output vectors and the second-row blocks to the left incident vectors yields the RTL contributions; the third and fourth rows analogously yield the LTR contributions for paths A and B:

$$\begin{aligned}\mathbf{a}_{\text{RTL}}^{\text{A}} &= \mathbf{M}_{(1,1)}^{-1} \mathbf{a}_{\text{out}}^{\text{L,A}} + \mathbf{M}_{(1,2)}^{-1} \mathbf{a}_{\text{out}}^{\text{L,B}} + \mathbf{M}_{(1,3)}^{-1} \mathbf{a}_{\text{in}}^{\text{L,A}} + \mathbf{M}_{(1,4)}^{-1} \mathbf{a}_{\text{in}}^{\text{L,B}}, \\ \mathbf{a}_{\text{RTL}}^{\text{B}} &= \mathbf{M}_{(2,1)}^{-1} \mathbf{a}_{\text{out}}^{\text{L,A}} + \mathbf{M}_{(2,2)}^{-1} \mathbf{a}_{\text{out}}^{\text{L,B}} + \mathbf{M}_{(2,3)}^{-1} \mathbf{a}_{\text{in}}^{\text{L,A}} + \mathbf{M}_{(2,4)}^{-1} \mathbf{a}_{\text{in}}^{\text{L,B}}, \\ \mathbf{a}_{\text{LTR}}^{\text{A}} &= \mathbf{M}_{(3,1)}^{-1} \mathbf{a}_{\text{out}}^{\text{L,A}} + \mathbf{M}_{(3,2)}^{-1} \mathbf{a}_{\text{out}}^{\text{L,B}} + \mathbf{M}_{(3,3)}^{-1} \mathbf{a}_{\text{in}}^{\text{L,A}} + \mathbf{M}_{(3,4)}^{-1} \mathbf{a}_{\text{in}}^{\text{L,B}}, \\ \mathbf{a}_{\text{LTR}}^{\text{B}} &= \mathbf{M}_{(4,1)}^{-1} \mathbf{a}_{\text{out}}^{\text{L,A}} + \mathbf{M}_{(4,2)}^{-1} \mathbf{a}_{\text{out}}^{\text{L,B}} + \mathbf{M}_{(4,3)}^{-1} \mathbf{a}_{\text{in}}^{\text{L,A}} + \mathbf{M}_{(4,4)}^{-1} \mathbf{a}_{\text{in}}^{\text{L,B}}.\end{aligned}$$

4. **Caching.** The four directional fields are wrapped in fresh `clsLightField` instances and stored in the appropriate private cache slots, and `.bulk_field_pos` is set to `idx`.

Because the full inverse transfer stack must be assembled, the runtime and memory footprint can even exceed those of a global transfer-scattering-matrix evaluation when no prior data is cached.

## Return value

None. The four directional bulk fields are cached internally and can be retrieved via `.get_bulk_field_LTR(...)`, `.get_bulk_field_RTL(...)`, and `.get_bulk_field(...)`.

`.calc_bulk_field_from_right(idx)`

## Method.

Back-propagates the stored *right* incident-output pair to obtain the steady-state intracavity field *immediately to the right* of component `idx` for both paths. The left-to-right and right-to-left contributions can be accessed after the computation via `.get_bulk_field_LTR(...)` and `.get_bulk_field_RTL(...)`; the coherent superposition of these directional contributions is available through `.get_bulk_field(...)`.

### Arguments

- `idx (int)`

Interface index of interest. Values  $\leq -1$  reference the left exterior, while values  $\geq .\text{component\_count}-1$  address the right exterior.

### Implementation details

1. *Boundary shortcuts.* Indices outside the cavity return the corresponding incident and output fields without further processing.
2. *Forward transfer matrix stack.* The routine forms the cumulative stack

$$\mathbf{M}_\Sigma = \mathbf{M}_{\text{idx}+1} \mathbf{M}_{\text{idx}+2} \dots \mathbf{M}_{N-1},$$

where each  $\mathbf{M}_k$  is the  $4 \times 4$  transfer block matrix of component  $k$ , obtained from `clsOptComponent4port.M.bmat_tot`. The product aggregates the transfer from the interface immediately right of component `idx` to the right boundary of the cavity.

3. *Field reconstruction.* Writing  $\mathbf{M}_\Sigma$  in four  $2 \times 2$  block rows, application of the first two rows to the right incident vectors and the last two rows to the right output vectors yields the RTL and LTR contributions,

$$\begin{aligned}\mathbf{a}_{\text{RTL}}^A &= \mathbf{M}_{(1,1)} \mathbf{a}_{\text{in}}^{R,A} + \mathbf{M}_{(1,2)} \mathbf{a}_{\text{in}}^{R,B} + \mathbf{M}_{(1,3)} \mathbf{a}_{\text{out}}^{R,A} + \mathbf{M}_{(1,4)} \mathbf{a}_{\text{out}}^{R,B}, \\ \mathbf{a}_{\text{RTL}}^B &= \mathbf{M}_{(2,1)} \mathbf{a}_{\text{in}}^{R,A} + \mathbf{M}_{(2,2)} \mathbf{a}_{\text{in}}^{R,B} + \mathbf{M}_{(2,3)} \mathbf{a}_{\text{out}}^{R,A} + \mathbf{M}_{(2,4)} \mathbf{a}_{\text{out}}^{R,B}, \\ \mathbf{a}_{\text{LTR}}^A &= \mathbf{M}_{(3,1)} \mathbf{a}_{\text{in}}^{R,A} + \mathbf{M}_{(3,2)} \mathbf{a}_{\text{in}}^{R,B} + \mathbf{M}_{(3,3)} \mathbf{a}_{\text{out}}^{R,A} + \mathbf{M}_{(3,4)} \mathbf{a}_{\text{out}}^{R,B}, \\ \mathbf{a}_{\text{LTR}}^B &= \mathbf{M}_{(4,1)} \mathbf{a}_{\text{in}}^{R,A} + \mathbf{M}_{(4,2)} \mathbf{a}_{\text{in}}^{R,B} + \mathbf{M}_{(4,3)} \mathbf{a}_{\text{out}}^{R,A} + \mathbf{M}_{(4,4)} \mathbf{a}_{\text{out}}^{R,B}.\end{aligned}$$

Here  $\mathbf{M}_{(p,q)}$  denotes the  $2 \times 2$  sub-block in row  $p$ , column  $q$  of  $\mathbf{M}_\Sigma$ .

4. *Caching.* The four directional fields are wrapped in fresh `clsLightField` instances, stored in the appropriate private cache slots, and `.bulk_field_pos` is set to `idx`.

Because the full transfer stack must be assembled, the runtime and memory footprint can even exceed those of a global transfer-scattering-matrix evaluation when no prior data is cached.

### Return value

None. The four directional bulk fields are cached internally and can be retrieved via `.get_bulk_field_LTR(...)`, `.get_bulk_field_RTL(...)`, and `.get_bulk_field(...)`.

#### `.bulk_field_pos`

##### Read-only property.

Returns the component index `idx` for which the most-recent intracavity bulk field was computed. The index refers to the field *immediately to the right* of the component;  $-1$  designates the left exterior, whereas `.component_count` $-1$  designates the right exterior. The value is `None` until either `.calc_bulk_field_from_left(...)` or `.calc_bulk_field_from_right(...)` has been executed.

**Set by**

- `.calc_bulk_field_from_left(...).`
- `.calc_bulk_field_from_right(...).`

**Cleared by**

- Calling `.set_incident_field(...).`
- Calling `.clear_results().`

**`.get_bulk_field_LTR(path)`****Method.**

Returns the left-to-right component of the intracavity bulk field for the requested path.

**Arguments**

- `path` – `Path.A` or `Path.B` selects the desired optical branch.

**Behaviour**

The method simply returns the cached `clsLightField` instance that was generated by the most-recent call to `.calc_bulk_field_from_left(...)` or `.calc_bulk_field_from_right(...)`. If no bulk-field routine has been executed yet, the return value is `None`.

**Cleared by**

- Calling `.set_incident_field(...).`
- Calling `.clear_results().`

**`.get_bulk_field_RTL(path)`****Method.**

Returns the right-to-left component of the intracavity bulk field for the requested path.

**Arguments**

- `path` – `Path.A` or `Path.B` selects the desired optical branch.

**Behaviour**

The method simply returns the cached `clsLightField` instance that was generated by the most-recent call to `.calc_bulk_field_from_left(...)` or `.calc_bulk_field_from_right(...)`. If no bulk-field routine has been executed yet, the return value is `None`.

**Cleared by**

- Calling `.set_incident_field(...).`
- Calling `.clear_results().`

**`.get_bulk_field(path)`****Method.**

Returns the coherent superposition of the left-to-right and right-to-left bulk fields for the requested path.

**Arguments**

- `path` – `Path.A` or `Path.B` selects the desired optical branch.

**Behaviour**

Produces and returns a `clsLightField` instance whose complex amplitudes equal the sum of `.get_bulk_field.LTR(...)` and `.get_bulk_field.RTL(...)` of the respective path. Directional components that are `None` are ignored; if both are `None` the returned field is empty (black).

**Cleared by**

- Calling `.set_incident_field(...)`.
- Calling `.clear_results()`.

**F.3.7 Reflection and Transmission Matrices****`.calc_R_L_mat_tot()`****Method.**

Calculates the right reflection  $2 \times 2$  block matrix `clsCavity.R_R_mat_tot`, which corresponds to the bottom-right quadrant of the cavity's  $4 \times 4$  scattering block matrix `clsCavity.S_bmat_tot`. The result is cached and made available in `clsCavity.R_R_mat_tot`.

**Implementation details**

1. If the global transfer block matrix `clsCavity.M_bmat_tot` is `None`, it is generated on demand with `clsCavity.calc_M_bmat_tot(...)`.
2. Let  $\mathbf{M}$  be that  $4 \times 4$  transfer block matrix. Following the block conversion rules in [4], the  $2 \times 2$  right-hand reflection block matrix is obtained from  $\mathbf{M}$  via

$$\mathbf{R}_R = -\mathbf{M}_{IV}^{-1} \mathbf{M}_{II},$$

with

$$\mathbf{M}_{II} = \begin{pmatrix} \mathbf{M}_{13} & \mathbf{M}_{14} \\ \mathbf{M}_{23} & \mathbf{M}_{24} \end{pmatrix}, \quad \mathbf{M}_{IV} = \begin{pmatrix} \mathbf{M}_{33} & \mathbf{M}_{34} \\ \mathbf{M}_{43} & \mathbf{M}_{44} \end{pmatrix}$$

where the first index denotes the block row and the second index the block column.

3. The result is stored and made available in `clsCavity.R_R_mat_tot`.

**Return value**

`None`. The matrix is stored internally and can be retrieved from the property mentioned above.

### `.calc_R_R_mat_tot()`

#### **Method.**

Calculates the right reflection  $2 \times 2$  block matrix `clsCavity.R_R_mat_tot`, which corresponds to the bottom-right quadrant of the cavity's  $4 \times 4$  scattering block matrix `clsCavity.S_bmat_tot`. The result is cached and made available in `clsCavity.R_R_mat_tot`.

#### **Implementation details**

1. If the global transfer block matrix `clsCavity.M_bmat_tot` is `None`, it is generated on demand with `clsCavity.calc_M_bmat_tot(...)`.
2. Let  $\mathbf{M}$  be that  $4 \times 4$  transfer block matrix. Following the block conversion rules in [4], the  $2 \times 2$  right-hand reflection block matrix is obtained from  $\mathbf{M}$  via

$$\mathbf{R}_R = -\mathbf{M}_{IV}^{-1} \mathbf{M}_{III},$$

with

$$\mathbf{M}_{III} = \begin{pmatrix} \mathbf{M}_{31} & \mathbf{M}_{32} \\ \mathbf{M}_{41} & \mathbf{M}_{42} \end{pmatrix}, \quad \mathbf{M}_{IV} = \begin{pmatrix} \mathbf{M}_{33} & \mathbf{M}_{34} \\ \mathbf{M}_{43} & \mathbf{M}_{44} \end{pmatrix}$$

where the first index denotes the block row and the second index the block column.

3. The result is stored and made available in `clsCavity.R_R_mat_tot`.

#### **Return value**

`None`. The matrix is stored internally and can be retrieved from the property mentioned above.

### `.calc_T_LTR_mat_tot()`

#### **Method.**

Calculates the left-to-right  $2 \times 2$  transmission block matrix `clsCavity.T_LTR_mat_tot`, which corresponds to the bottom-left quadrant of the cavity's  $4 \times 4$  scattering block matrix `clsCavity.S_bmat_tot`. The result is cached and made available in `clsCavity.T_LTR_mat_tot`.

#### **Implementation details**

1. If the global transfer block matrix `clsCavity.M_bmat_tot` is `None`, it is generated on demand with `clsCavity.calc_M_bmat_tot(...)`.
2. Let  $\mathbf{M}$  be that  $4 \times 4$  transfer block matrix. Following the block conversion rules in [4], the  $2 \times 2$  left-to-right transmission block matrix is obtained from  $\mathbf{M}$  via

$$\mathbf{T}_{LTR} = \mathbf{M}_{IV}^{-1},$$

with

$$\mathbf{M}_{IV} = \begin{pmatrix} \mathbf{M}_{33} & \mathbf{M}_{34} \\ \mathbf{M}_{43} & \mathbf{M}_{44} \end{pmatrix}$$

where the first index denotes the block row and the second index the block column.

3. The result is stored and made available in `clsCavity.T_LTR_mat_tot`.

**Return value**

None. The matrix is stored internally and can be retrieved from the property mentioned above.

`.calc_T_RTL_mat_tot()`

**Method.**

Calculates the right-to-left  $2 \times 2$  transmission block matrix `clsCavity.T_RTL_mat_tot`, which corresponds to the top-right quadrant of the cavity's  $4 \times 4$  scattering block matrix `clsCavity.S_bmat_tot`. The result is cached and made available in `clsCavity.T_RTL_mat_tot`.

**Implementation details**

1. If the global transfer block matrix `clsCavity.M_bmat_tot` is `None`, it is generated on demand with `clsCavity.calc_M_bmat_tot(...)`.
2. Let  $\mathbf{M}$  be that  $4 \times 4$  transfer block matrix. Following the block conversion rules in [4], the  $2 \times 2$  right-to-left transmission block matrix is obtained from  $\mathbf{M}$  via

$$\mathbf{T}_{\text{RTL}} = \mathbf{M}_I - \mathbf{M}_{II} \mathbf{M}_{IV}^{-1} \mathbf{M}_{III}.$$

with

$$\begin{aligned}\mathbf{M}_I &= \begin{pmatrix} \mathbf{M}_{11} & \mathbf{M}_{12} \\ \mathbf{M}_{21} & \mathbf{M}_{22} \end{pmatrix}, & \mathbf{M}_{II} &= \begin{pmatrix} \mathbf{M}_{13} & \mathbf{M}_{14} \\ \mathbf{M}_{23} & \mathbf{M}_{24} \end{pmatrix}, \\ \mathbf{M}_{III} &= \begin{pmatrix} \mathbf{M}_{31} & \mathbf{M}_{32} \\ \mathbf{M}_{41} & \mathbf{M}_{42} \end{pmatrix}, & \mathbf{M}_{IV} &= \begin{pmatrix} \mathbf{M}_{33} & \mathbf{M}_{34} \\ \mathbf{M}_{43} & \mathbf{M}_{44} \end{pmatrix}\end{aligned}$$

3. The result is stored and made available in `clsCavity.T_RTL_mat_tot`.

**Return value**

None. The matrix is stored internally and can be retrieved from the property mentioned above.

### F.3.8 Full Transfer and Scattering Block Matrix

`.calc_S_bmat_tot()`

**Method.**

Calculates the full  $4 \times 4$  scattering block matrix `clsCavity.S_bmat_tot` of the cavity using the  $2 \times 2$  reflection and transmission block matrices  $\mathbf{R}_L$ ,  $\mathbf{R}_R$ ,  $\mathbf{T}_{\text{LTR}}$ , and  $\mathbf{T}_{\text{RTL}}$  as calculated by `.calc_R_L_mat_tot()`, `.calc_R_R_mat_tot()`, `.calc_T_LTR_mat_tot()`, and `.calc_T_RTL_mat_tot()`. The result is cached and made available in `clsCavity.S_bmat_tot`.

**Implementation details** These four  $2 \times 2$  blocks are assembled into the full  $4 \times 4$  scattering block matrix:

$$\mathbf{S} = \begin{pmatrix} \mathbf{R}_L & \mathbf{T}_{\text{RTL}} \\ \mathbf{T}_{\text{LTR}} & \mathbf{R}_R \end{pmatrix}$$

**Return value**

None. The result is stored internally and can be retrieved from `clsCavity.S_bmat_tot`.

### F.3.9 Caching and Memory

`.clear_results()`

#### Method.

Clears every cached output or intracavity field produced by previous simulations, including all four output ports and the directional intracavity bulk fields. Invoke this routine whenever the input fields or the optical layout has changed and you wish to enforce a fresh re-calculation.

#### Affected fields

- `.get_output_field(...)`
- `.get_bulk_field_LTR(...)`
- `.get_bulk_field_RTL(...)`
- `.get_bulk_field(...)`

#### Return value

None.

# Bibliography

- [1] Helmut Hörner, Lena Wild, Yevgeny Slobodkin, Gil Weinberg, Ori Katz, and Stefan Rotter. Coherent perfect absorption of arbitrary wavefronts at an exceptional point. *Physical Review Letters*, 133(17):173801, October 2024.
- [2] Yevgeny Slobodkin, Gil Weinberg, Helmut Hörner, Kevin Pichler, Stefan Rotter, and Ori Katz. Massively degenerate coherent perfect absorber for arbitrary wavefronts. *Science*, 377(6609):995–998, August 2022.
- [3] David Voelz. *Computational fourier optics : a MATLAB tutorial*. SPIE Press, Bellingham, Wash, 2011.
- [4] J. Frei, Xiao-Ding Cai, and S. Muller. Multiport s-parameter and t-parameter conversion with symmetry extension. *IEEE Transactions on Microwave Theory and Techniques*, 56(11):2493–2504, nov 2008.
- [5] Franklin A. Graybill. *Matrices with Applications in Statistics (Duxbury Classic)*. Duxbury Press, 2001.
- [6] Tzon-Tzer Lu and Sheng-Hua Shiou. Inverses of  $2 \times 2$  block matrices. *Computers and Mathematics with Applications*, 43(1–2):119–129, January 2002.
- [7] Helmut Hörner. Theory of a massively parallel coherent perfect absorber in a degenerate 4f cavity. 2021.
- [8] Anthony E. Siegman. *Lasers (Revised)*. UNIVERSITY SCIENCE BOOKS, October 1986.