

# C++ Library of the Linear Conjugate Gradient Methods (LibLCG)

Yi Zhang



<b>1 Data Structure Index</b>	<b>1</b>
1.1 Data Structures	1
<b>2 File Index</b>	<b>3</b>
2.1 File List	3
<b>3 Data Structure Documentation</b>	<b>5</b>
3.1 CLCG_CUDA_Solver Class Reference	5
3.1.1 Detailed Description	6
3.1.2 Constructor & Destructor Documentation	6
3.1.2.1 CLCG_CUDA_Solver()	6
3.1.2.2 ~CLCG_CUDA_Solver()	6
3.1.3 Member Function Documentation	6
3.1.3.1 _AxProduct()	6
3.1.3.2 _MxProduct()	7
3.1.3.3 _Progress()	7
3.1.3.4 AxProduct()	8
3.1.3.5 Minimize()	9
3.1.3.6 MinimizePreconditioned()	9
3.1.3.7 MxProduct()	10
3.1.3.8 Progress()	10
3.1.3.9 set_clcg_parameter()	11
3.1.3.10 set_report_interval()	11
3.1.3.11 silent()	11
3.1.4 Field Documentation	12
3.1.4.1 inter_	12
3.1.4.2 param_	12
3.1.4.3 silent_	12
3.2 CLCG_CUDAF_Solver Class Reference	12
3.2.1 Detailed Description	13
3.2.2 Constructor & Destructor Documentation	13
3.2.2.1 CLCG_CUDAF_Solver()	13
3.2.2.2 ~CLCG_CUDAF_Solver()	13
3.2.3 Member Function Documentation	13
3.2.3.1 _AxProduct()	13
3.2.3.2 _MxProduct()	14
3.2.3.3 _Progress()	14
3.2.3.4 AxProduct()	15
3.2.3.5 Minimize()	16
3.2.3.6 MinimizePreconditioned()	16
3.2.3.7 MxProduct()	17
3.2.3.8 Progress()	17
3.2.3.9 set_clcg_parameter()	18

3.2.3.10 set_report_interval()	18
3.2.3.11 silent()	18
3.2.4 Field Documentation	19
3.2.4.1 inter_	19
3.2.4.2 param_	19
3.2.4.3 silent_	19
3.3 CLCG_EIGEN_Solver Class Reference	19
3.3.1 Detailed Description	20
3.3.2 Constructor & Destructor Documentation	20
3.3.2.1 CLCG_EIGEN_Solver()	20
3.3.2.2 ~CLCG_EIGEN_Solver()	20
3.3.3 Member Function Documentation	20
3.3.3.1 _AxProduct()	20
3.3.3.2 _MxProduct()	21
3.3.3.3 _Progress()	21
3.3.3.4 AxProduct()	22
3.3.3.5 Minimize()	22
3.3.3.6 MinimizePreconditioned()	22
3.3.3.7 MxProduct()	23
3.3.3.8 Progress()	23
3.3.3.9 set_clcg_parameter()	24
3.3.3.10 set_report_interval()	24
3.3.3.11 silent()	24
3.3.4 Field Documentation	24
3.3.4.1 inter_	25
3.3.4.2 param_	25
3.3.4.3 silent_	25
3.4 clcg_para Struct Reference	25
3.4.1 Detailed Description	25
3.4.2 Field Documentation	25
3.4.2.1 abs_diff	26
3.4.2.2 epsilon	26
3.4.2.3 max_iterations	26
3.5 CLCG_Solver Class Reference	26
3.5.1 Detailed Description	27
3.5.2 Constructor & Destructor Documentation	27
3.5.2.1 CLCG_Solver()	27
3.5.2.2 ~CLCG_Solver()	27
3.5.3 Member Function Documentation	27
3.5.3.1 _AxProduct()	27
3.5.3.2 _Progress()	28
3.5.3.3 AxProduct()	28

3.5.3.4 Minimize()	29
3.5.3.5 Progress()	29
3.5.3.6 set_clcg_parameter()	30
3.5.3.7 set_report_interval()	30
3.5.3.8 silent()	30
3.5.4 Field Documentation	30
3.5.4.1 inter_	30
3.5.4.2 param_	31
3.5.4.3 silent_	31
3.6 LCG_CUDA_Solver Class Reference	31
3.6.1 Detailed Description	32
3.6.2 Constructor & Destructor Documentation	32
3.6.2.1 LCG_CUDA_Solver()	32
3.6.2.2 ~LCG_CUDA_Solver()	32
3.6.3 Member Function Documentation	32
3.6.3.1 _AxProduct()	32
3.6.3.2 _MxProduct()	33
3.6.3.3 _Progress()	33
3.6.3.4 AxProduct()	34
3.6.3.5 Minimize()	34
3.6.3.6 MinimizeConstrained()	35
3.6.3.7 MinimizePreconditioned()	36
3.6.3.8 MxProduct()	36
3.6.3.9 Progress()	37
3.6.3.10 set_lcg_parameter()	37
3.6.3.11 set_report_interval()	37
3.6.3.12 silent()	38
3.6.4 Field Documentation	38
3.6.4.1 inter_	38
3.6.4.2 param_	38
3.6.4.3 silent_	38
3.7 LCG_EIGEN_Solver Class Reference	38
3.7.1 Detailed Description	39
3.7.2 Constructor & Destructor Documentation	39
3.7.2.1 LCG_EIGEN_Solver()	40
3.7.2.2 ~LCG_EIGEN_Solver()	40
3.7.3 Member Function Documentation	40
3.7.3.1 _AxProduct()	40
3.7.3.2 _MxProduct()	40
3.7.3.3 _Progress()	41
3.7.3.4 AxProduct()	41
3.7.3.5 Minimize()	41

3.7.3.6 MinimizeConstrained()	42
3.7.3.7 MinimizePreconditioned()	42
3.7.3.8 MxProduct()	43
3.7.3.9 Progress()	43
3.7.3.10 set_lcg_parameter()	44
3.7.3.11 set_report_interval()	44
3.7.3.12 silent()	44
3.7.4 Field Documentation	44
3.7.4.1 inter_	44
3.7.4.2 param_	44
3.7.4.3 silent_	45
3.8 lcg_para Struct Reference	45
3.8.1 Detailed Description	45
3.8.2 Field Documentation	45
3.8.2.1 abs_diff	45
3.8.2.2 beta	45
3.8.2.3 epsilon	46
3.8.2.4 max_ iterations	46
3.8.2.5 maxi_m	46
3.8.2.6 restart_epsilon	46
3.8.2.7 sigma	46
3.8.2.8 step	46
3.9 LCG_Solver Class Reference	47
3.9.1 Detailed Description	48
3.9.2 Constructor & Destructor Documentation	48
3.9.2.1 LCG_Solver()	48
3.9.2.2 ~LCG_Solver()	48
3.9.3 Member Function Documentation	48
3.9.3.1 _AxProduct()	48
3.9.3.2 _MxProduct()	48
3.9.3.3 _Progress()	49
3.9.3.4 AxProduct()	49
3.9.3.5 Minimize()	50
3.9.3.6 MinimizeConstrained()	50
3.9.3.7 MinimizePreconditioned()	51
3.9.3.8 MxProduct()	51
3.9.3.9 Progress()	52
3.9.3.10 set_lcg_parameter()	52
3.9.3.11 set_report_interval()	52
3.9.3.12 silent()	53
3.9.4 Field Documentation	53
3.9.4.1 inter_	53

3.9.4.2 param_ . . . . .	53
3.9.4.3 silent_ . . . . .	53
<b>4 File Documentation</b>	<b>55</b>
4.1 algebra.cpp File Reference . . . . .	55
4.1.1 Function Documentation . . . . .	56
4.1.1.1 lcg_abs() . . . . .	56
4.1.1.2 lcg_dot() . . . . .	56
4.1.1.3 lcg_free() [1/2] . . . . .	57
4.1.1.4 lcg_free() [2/2] . . . . .	57
4.1.1.5 lcg_malloc() [1/2] . . . . .	57
4.1.1.6 lcg_malloc() [2/2] . . . . .	58
4.1.1.7 lcg_matvec() . . . . .	58
4.1.1.8 lcg_matvec_coo() . . . . .	58
4.1.1.9 lcg_max() . . . . .	59
4.1.1.10 lcg_min() . . . . .	59
4.1.1.11 lcg_set2box() . . . . .	60
4.1.1.12 lcg_squaredl2norm() . . . . .	60
4.1.1.13 lcg_vecrnd() [1/2] . . . . .	61
4.1.1.14 lcg_vecrnd() [2/2] . . . . .	61
4.1.1.15 lcg_vecset() [1/2] . . . . .	61
4.1.1.16 lcg_vecset() [2/2] . . . . .	62
4.2 algebra.h File Reference . . . . .	62
4.2.1 Typedef Documentation . . . . .	63
4.2.1.1 lcg_float . . . . .	63
4.2.2 Enumeration Type Documentation . . . . .	64
4.2.2.1 clcg_complex_e . . . . .	64
4.2.2.2 lcg_matrix_e . . . . .	64
4.2.3 Function Documentation . . . . .	64
4.2.3.1 lcg_abs() . . . . .	64
4.2.3.2 lcg_dot() . . . . .	65
4.2.3.3 lcg_free() [1/2] . . . . .	65
4.2.3.4 lcg_free() [2/2] . . . . .	65
4.2.3.5 lcg_malloc() [1/2] . . . . .	66
4.2.3.6 lcg_malloc() [2/2] . . . . .	66
4.2.3.7 lcg_matvec() . . . . .	66
4.2.3.8 lcg_matvec_coo() . . . . .	67
4.2.3.9 lcg_max() . . . . .	67
4.2.3.10 lcg_min() . . . . .	68
4.2.3.11 lcg_set2box() . . . . .	68
4.2.3.12 lcg_squaredl2norm() . . . . .	69
4.2.3.13 lcg_vecrnd() [1/2] . . . . .	69

4.2.3.14 lcg_vecrnd() [2/2]	69
4.2.3.15 lcg_vecset() [1/2]	70
4.2.3.16 lcg_vecset() [2/2]	70
4.3 algebra_cuda.h File Reference	71
4.3.1 Function Documentation	71
4.3.1.1 lcg_set2box_cuda()	71
4.3.1.2 lcg_smDcsr_get_diagonal()	72
4.3.1.3 lcg_vecDvecD_element_wise()	72
4.3.1.4 lcg_vecMvecD_element_wise()	73
4.4 algebra_eigen.cpp File Reference	73
4.4.1 Function Documentation	73
4.4.1.1 lcg_set2box_eigen()	73
4.5 algebra_eigen.h File Reference	74
4.5.1 Function Documentation	74
4.5.1.1 lcg_set2box_eigen()	74
4.6 clcg.cpp File Reference	74
4.6.1 Typedef Documentation	75
4.6.1.1 clcg_solver_ptr	75
4.6.2 Function Documentation	75
4.6.2.1 clbicg()	75
4.6.2.2 clbicg_symmetric()	76
4.6.2.3 clbicgstab()	76
4.6.2.4 clcg_solver()	76
4.6.2.5 clcgs()	77
4.6.2.6 cltfqmr()	77
4.7 clcg.h File Reference	77
4.7.1 Typedef Documentation	78
4.7.1.1 clcg_axfunc_ptr	78
4.7.1.2 clcg_progress_ptr	78
4.7.2 Function Documentation	78
4.7.2.1 clcg_solver()	79
4.8 clcg_cuda.h File Reference	79
4.8.1 Typedef Documentation	80
4.8.1.1 clcg_axfunc_cuda_ptr	80
4.8.1.2 clcg_progress_cuda_ptr	80
4.8.2 Function Documentation	81
4.8.2.1 clcg_solver_cuda()	81
4.8.2.2 clcg_solver_preconditioned_cuda()	81
4.9 clcg_cudaf.h File Reference	82
4.9.1 Typedef Documentation	83
4.9.1.1 clcg_axfunc_cudaf_ptr	83
4.9.1.2 clcg_progress_cudaf_ptr	83



4.9.2 Function Documentation	84
4.9.2.1 clcg_solver_cuda()	84
4.9.2.2 clcg_solver_preconditioned_cuda()	84
4.10 clcg_eigen.cpp File Reference	85
4.10.1 Typedef Documentation	86
4.10.1.1 eigen_preconditioned_solver_ptr	86
4.10.1.2 eigen_solver_ptr	86
4.10.2 Function Documentation	86
4.10.2.1 clbicg()	86
4.10.2.2 clbicg_symmetric()	87
4.10.2.3 clcg_solver_eigen()	87
4.10.2.4 clcg_solver_preconditioned_eigen()	87
4.10.2.5 clcgs()	88
4.10.2.6 clpbicg()	88
4.10.2.7 clpcg()	89
4.10.2.8 cltfqmr()	89
4.11 clcg_eigen.h File Reference	89
4.11.1 Typedef Documentation	90
4.11.1.1 clcg_axfunc_eigen_ptr	90
4.11.1.2 clcg_progress_eigen_ptr	90
4.11.2 Function Documentation	90
4.11.2.1 clcg_solver_eigen()	91
4.11.2.2 clcg_solver_preconditioned_eigen()	91
4.12 config.h File Reference	92
4.12.1 Macro Definition Documentation	92
4.12.1.1 LibLCG_CUDA	92
4.12.1.2 LibLCG_EIGEN	92
4.12.1.3 LibLCG_OPENMP	92
4.12.1.4 LibLCG_STD_COMPLEX	93
4.13 lcg.cpp File Reference	93
4.13.1 Typedef Documentation	94
4.13.1.1 lcg_solver_ptr	94
4.13.1.2 lcg_solver_ptr2	94
4.13.2 Function Documentation	95
4.13.2.1 lbicgstab()	95
4.13.2.2 lbicgstab2()	95
4.13.2.3 lcg()	96
4.13.2.4 lcg_solver()	97
4.13.2.5 lcg_solver_constrained()	97
4.13.2.6 lcg_solver_preconditioned()	98
4.13.2.7 lcgs()	99
4.13.2.8 lpcg()	100

4.13.2.9 lpg()	100
4.13.2.10 lspg()	101
4.14 lcg.h File Reference	102
4.14.1 Typedef Documentation	103
4.14.1.1 lcg_axfunc_ptr	103
4.14.1.2 lcg_progress_ptr	103
4.14.2 Function Documentation	103
4.14.2.1 lcg()	103
4.14.2.2 lcg_solver()	104
4.14.2.3 lcg_solver_constrained()	105
4.14.2.4 lcg_solver_preconditioned()	106
4.14.2.5 lcgs()	106
4.15 lcg_complex.cpp File Reference	107
4.15.1 Function Documentation	108
4.15.1.1 clcg_conjugate()	108
4.15.1.2 clcg_dot()	109
4.15.1.3 clcg_free() [1/2]	109
4.15.1.4 clcg_free() [2/2]	109
4.15.1.5 clcg_inner()	110
4.15.1.6 clcg_malloc() [1/2]	110
4.15.1.7 clcg_malloc() [2/2]	110
4.15.1.8 clcg_matvec()	111
4.15.1.9 clcg_module()	111
4.15.1.10 clcg_set()	112
4.15.1.11 clcg_square()	112
4.15.1.12 clcg_vecrnd() [1/2]	112
4.15.1.13 clcg_vecrnd() [2/2]	113
4.15.1.14 clcg_vecset() [1/2]	113
4.15.1.15 clcg_vecset() [2/2]	113
4.16 lcg_complex.h File Reference	114
4.16.1 Typedef Documentation	114
4.16.1.1 lcg_complex	115
4.16.2 Function Documentation	115
4.16.2.1 clcg_conjugate()	115
4.16.2.2 clcg_dot()	115
4.16.2.3 clcg_free() [1/2]	115
4.16.2.4 clcg_free() [2/2]	116
4.16.2.5 clcg_inner()	116
4.16.2.6 clcg_malloc() [1/2]	117
4.16.2.7 clcg_malloc() [2/2]	117
4.16.2.8 clcg_matvec()	117
4.16.2.9 clcg_module()	118

4.16.2.10 <code>clcg_set()</code> . . . . .	118
4.16.2.11 <code>clcg_square()</code> . . . . .	118
4.16.2.12 <code>clcg_vecrnd()</code> [1/2] . . . . .	119
4.16.2.13 <code>clcg_vecrnd()</code> [2/2] . . . . .	119
4.16.2.14 <code>clcg_vecset()</code> [1/2] . . . . .	119
4.16.2.15 <code>clcg_vecset()</code> [2/2] . . . . .	120
4.17 <code>lcg_complex_cuda.h</code> File Reference . . . . .	120
4.17.1 Function Documentation . . . . .	121
4.17.1.1 <code>clcg_Cdiff()</code> . . . . .	122
4.17.1.2 <code>clcg_Cscale()</code> . . . . .	122
4.17.1.3 <code>clcg_Csqrt()</code> . . . . .	122
4.17.1.4 <code>clcg_Csum()</code> . . . . .	123
4.17.1.5 <code>clcg_free_cuda()</code> . . . . .	123
4.17.1.6 <code>clcg_malloc_cuda()</code> . . . . .	123
4.17.1.7 <code>clcg_smCcoo_row2col()</code> . . . . .	124
4.17.1.8 <code>clcg_smCcsr_get_diagonal()</code> . . . . .	124
4.17.1.9 <code>clcg_smZcoo_row2col()</code> . . . . .	125
4.17.1.10 <code>clcg_smZcsr_get_diagonal()</code> . . . . .	125
4.17.1.11 <code>clcg_vecC_conjugate()</code> . . . . .	126
4.17.1.12 <code>clcg_vecDvecC_element_wise()</code> . . . . .	126
4.17.1.13 <code>clcg_vecDvecZ_element_wise()</code> . . . . .	127
4.17.1.14 <code>clcg_vecMvecC_element_wise()</code> . . . . .	127
4.17.1.15 <code>clcg_vecMvecZ_element_wise()</code> . . . . .	128
4.17.1.16 <code>clcg_vecset_cuda()</code> . . . . .	128
4.17.1.17 <code>clcg_vecZ_conjugate()</code> . . . . .	129
4.17.1.18 <code>clcg_Zdiff()</code> . . . . .	129
4.17.1.19 <code>clcg_Zscale()</code> . . . . .	130
4.17.1.20 <code>clcg_Zsqrt()</code> . . . . .	130
4.17.1.21 <code>clcg_Zsum()</code> . . . . .	130
4.17.1.22 <code>cuda2lcg_complex()</code> . . . . .	131
4.17.1.23 <code>lcg2cuda_complex()</code> . . . . .	131
4.18 <code>lcg_cuda.h</code> File Reference . . . . .	131
4.18.1 Typedef Documentation . . . . .	132
4.18.1.1 <code>lcg_axfunc_cuda_ptr</code> . . . . .	132
4.18.1.2 <code>lcg_progress_cuda_ptr</code> . . . . .	133
4.18.2 Function Documentation . . . . .	133
4.18.2.1 <code>lcg_solver_constrained_cuda()</code> . . . . .	133
4.18.2.2 <code>lcg_solver_cuda()</code> . . . . .	134
4.18.2.3 <code>lcg_solver_preconditioned_cuda()</code> . . . . .	135
4.19 <code>lcg_eigen.cpp</code> File Reference . . . . .	135
4.19.1 Typedef Documentation . . . . .	136
4.19.1.1 <code>eigen_solver_ptr</code> . . . . .	136

4.19.1.2 <a href="#">eigen_solver_ptr2</a>	137
4.19.2 <a href="#">Function Documentation</a>	137
4.19.2.1 <a href="#">lbicgstab()</a>	138
4.19.2.2 <a href="#">lbicgstab2()</a>	138
4.19.2.3 <a href="#">lcg()</a>	139
4.19.2.4 <a href="#">lcg_solver_constrained_eigen()</a>	139
4.19.2.5 <a href="#">lcg_solver_eigen()</a>	140
4.19.2.6 <a href="#">lcg_solver_preconditioned_eigen()</a>	141
4.19.2.7 <a href="#">lcgs()</a>	141
4.19.2.8 <a href="#">lpcg()</a>	142
4.19.2.9 <a href="#">lpg()</a>	143
4.19.2.10 <a href="#">lspg()</a>	143
4.20 <a href="#">lcg_eigen.h File Reference</a>	144
4.20.1 <a href="#">Typedef Documentation</a>	145
4.20.1.1 <a href="#">lcg_axfunc_eigen_ptr</a>	145
4.20.1.2 <a href="#">lcg_progress_eigen_ptr</a>	145
4.20.2 <a href="#">Function Documentation</a>	145
4.20.2.1 <a href="#">lcg_solver_constrained_eigen()</a>	145
4.20.2.2 <a href="#">lcg_solver_eigen()</a>	146
4.20.2.3 <a href="#">lcg_solver_preconditioned_eigen()</a>	147
4.21 <a href="#">preconditioner.cpp File Reference</a>	147
4.21.1 <a href="#">Function Documentation</a>	148
4.21.1.1 <a href="#">lcg_full_rank_coo()</a>	148
4.21.1.2 <a href="#">lcg_incomplete_Cholesky_full_coo()</a>	148
4.21.1.3 <a href="#">lcg_incomplete_Cholesky_half_buffersize_coo()</a>	149
4.21.1.4 <a href="#">lcg_incomplete_Cholesky_half_coo()</a>	149
4.21.1.5 <a href="#">lcg_solve_lower_triangle_coo()</a>	150
4.21.1.6 <a href="#">lcg_solve_upper_triangle_coo()</a>	151
4.22 <a href="#">preconditioner.h File Reference</a>	151
4.22.1 <a href="#">Function Documentation</a>	152
4.22.1.1 <a href="#">lcg_full_rank_coo()</a>	152
4.22.1.2 <a href="#">lcg_incomplete_Cholesky_full_coo()</a>	152
4.22.1.3 <a href="#">lcg_incomplete_Cholesky_half_buffersize_coo()</a>	153
4.22.1.4 <a href="#">lcg_incomplete_Cholesky_half_coo()</a>	153
4.22.1.5 <a href="#">lcg_solve_lower_triangle_coo()</a>	154
4.22.1.6 <a href="#">lcg_solve_upper_triangle_coo()</a>	154
4.23 <a href="#">preconditioner_cuda.h File Reference</a>	155
4.23.1 <a href="#">Function Documentation</a>	155
4.23.1.1 <a href="#">clcg_incomplete_Cholesky_cuda_full()</a>	155
4.23.1.2 <a href="#">clcg_incomplete_Cholesky_cuda_half()</a> [1/2]	156
4.23.1.3 <a href="#">clcg_incomplete_Cholesky_cuda_half()</a> [2/2]	156
4.23.1.4 <a href="#">clcg_incomplete_Cholesky_cuda_half_buffersize()</a>	157

4.24 preconditioner_eigen.cpp File Reference	158
4.24.1 Typedef Documentation	159
4.24.1.1 triplet_bl	159
4.24.1.2 triplet_cd	159
4.24.1.3 triplet_d	159
4.24.2 Function Documentation	159
4.24.2.1 clcg_Cholesky()	159
4.24.2.2 clcg_incomplete_Cholesky()	160
4.24.2.3 clcg_incomplete_LU()	160
4.24.2.4 clcg_invert_lower_triangle()	160
4.24.2.5 clcg_invert_upper_triangle()	161
4.24.2.6 clcg_solve_lower_triangle()	161
4.24.2.7 clcg_solve_upper_triangle()	161
4.24.2.8 lcg_Cholesky()	162
4.24.2.9 lcg_incomplete_Cholesky()	162
4.24.2.10 lcg_incomplete_LU()	162
4.24.2.11 lcg_invert_lower_triangle()	163
4.24.2.12 lcg_invert_upper_triangle()	163
4.24.2.13 lcg_solve_lower_triangle()	163
4.24.2.14 lcg_solve_upper_triangle()	164
4.25 preconditioner_eigen.h File Reference	164
4.25.1 Function Documentation	165
4.25.1.1 clcg_Cholesky()	165
4.25.1.2 clcg_incomplete_Cholesky()	165
4.25.1.3 clcg_incomplete_LU()	166
4.25.1.4 clcg_invert_lower_triangle()	166
4.25.1.5 clcg_invert_upper_triangle()	167
4.25.1.6 clcg_solve_lower_triangle()	167
4.25.1.7 clcg_solve_upper_triangle()	167
4.25.1.8 lcg_Cholesky()	168
4.25.1.9 lcg_incomplete_Cholesky()	168
4.25.1.10 lcg_incomplete_LU()	168
4.25.1.11 lcg_invert_lower_triangle()	169
4.25.1.12 lcg_invert_upper_triangle()	169
4.25.1.13 lcg_solve_lower_triangle()	169
4.25.1.14 lcg_solve_upper_triangle()	170
4.26 solver.cpp File Reference	170
4.27 solver.h File Reference	170
4.28 solver_cuda.h File Reference	171
4.29 solver_eigen.cpp File Reference	171
4.30 solver_eigen.h File Reference	171
4.31 util.cpp File Reference	171

---

4.31.1 Function Documentation . . . . .	172
4.31.1.1 clcg_default_parameters() . . . . .	172
4.31.1.2 clcg_error_str() . . . . .	172
4.31.1.3 clcg_select_solver() . . . . .	173
4.31.1.4 lcg_default_parameters() . . . . .	173
4.31.1.5 lcg_error_str() . . . . .	173
4.31.1.6 lcg_select_solver() . . . . .	174
4.32 util.h File Reference . . . . .	174
4.32.1 Enumeration Type Documentation . . . . .	175
4.32.1.1 clcg_return_enum . . . . .	175
4.32.1.2 clcg_solver_enum . . . . .	176
4.32.1.3 lcg_return_enum . . . . .	176
4.32.1.4 lcg_solver_enum . . . . .	177
4.32.2 Function Documentation . . . . .	177
4.32.2.1 clcg_default_parameters() . . . . .	177
4.32.2.2 clcg_error_str() . . . . .	177
4.32.2.3 clcg_select_solver() . . . . .	178
4.32.2.4 lcg_default_parameters() . . . . .	178
4.32.2.5 lcg_error_str() . . . . .	178
4.32.2.6 lcg_select_solver() . . . . .	179
<b>Index</b>	<b>181</b>

# Chapter 1

## Data Structure Index

### 1.1 Data Structures

Here are the data structures with brief descriptions:

<a href="#">CLCG_CUDA_Solver</a>		
Complex linear conjugate gradient solver class	. . . . .	5
<a href="#">CLCG_CUDAF_Solver</a>		
Complex linear conjugate gradient solver class	. . . . .	12
<a href="#">CLCG_EIGEN_Solver</a>		
Complex linear conjugate gradient solver class	. . . . .	19
<a href="#">clcg_para</a>		
Parameters of the conjugate gradient methods	. . . . .	25
<a href="#">CLCG_Solver</a>		
Complex linear conjugate gradient solver class	. . . . .	26
<a href="#">LCG_CUDA_Solver</a>		
Linear conjugate gradient solver class	. . . . .	31
<a href="#">LCG_EIGEN_Solver</a>		
Linear conjugate gradient solver class	. . . . .	38
<a href="#">lcg_para</a>		
Parameters of the conjugate gradient methods	. . . . .	45
<a href="#">LCG_Solver</a>		
Linear conjugate gradient solver class	. . . . .	47





## Chapter 2

# File Index

### 2.1 File List

Here is a list of all files with brief descriptions:

algebra.cpp	55
algebra.h	62
algebra_cuda.h	71
algebra_eigen.cpp	73
algebra_eigen.h	74
clcg.cpp	74
clcg.h	77
clcg_cuda.h	79
clcg_cudaf.h	82
clcg_eigen.cpp	85
clcg_eigen.h	89
config.h	92
lcg.cpp	93
lcg.h	102
lcg_complex.cpp	107
lcg_complex.h	114
lcg_complex_cuda.h	120
lcg_cuda.h	131
lcg_eigen.cpp	135
lcg_eigen.h	144
preconditioner.cpp	147
preconditioner.h	151
preconditioner_cuda.h	155
preconditioner_eigen.cpp	158
preconditioner_eigen.h	164
solver.cpp	170
solver.h	170
solver_cuda.h	171
solver_eigen.cpp	171
solver_eigen.h	171
util.cpp	171
util.h	174



## Chapter 3

# Data Structure Documentation

### 3.1 CLCG\_CUDA\_Solver Class Reference

Complex linear conjugate gradient solver class.

```
#include <solver_cuda.h>
```

#### Public Member Functions

- [CLCG\\_CUDA\\_Solver](#) ()
- virtual [~CLCG\\_CUDA\\_Solver](#) ()
- virtual void [AxProduct](#) (cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, cusparseDnVecDescr\_t x, cusparseDnVecDescr\_t prod\_Ax, const int n\_size, const int nz\_size, cusparseOperation\_t oper\_t)=0  
*Virtual function of the product of  $A*x$ .*
- virtual void [MxProduct](#) (cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, cusparseDnVecDescr\_t x, cusparseDnVecDescr\_t prod\_Mx, const int n\_size, const int nz\_size, cusparseOperation\_t oper\_t)=0  
*Virtual function of the product of  $M^{-1}*x$ .*
- virtual int [Progress](#) (const cuDoubleComplex \*m, const [lcg\\_float](#) converge, const [clcg\\_para](#) \*param, const int n\_size, const int nz\_size, const int k)  
*Virtual function of the process monitoring.*
- void [silent](#) ()  
*Do not report any processes.*
- void [set\\_report\\_interval](#) (unsigned int inter)  
*Set the interval to run the process monitoring function.*
- void [set\\_clcg\\_parameter](#) (const [clcg\\_para](#) &in\_param)  
*Set the parameters of the algorithms.*
- void [Minimize](#) (cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, cuDoubleComplex \*x, cuDoubleComplex \*b, const int n\_size, const int nz\_size, [clcg\\_solver\\_enum](#) solver\_id=CLCG\_BICG, bool verbose=true, bool er\_throw=false)  
*Run the constrained minimizing process.*
- void [MinimizePreconditioned](#) (cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, cuDoubleComplex \*x, cuDoubleComplex \*b, const int n\_size, const int nz\_size, [clcg\\_solver\\_enum](#) solver\_id=CLCG\_PCG, bool verbose=true, bool er\_throw=false)  
*Run the preconditioned minimizing process.*

## Static Public Member Functions

- static void [\\_AxProduct](#) (void \*instance, cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, cusparseDnVecDescr\_t x, cusparseDnVecDescr\_t prod\_Ax, const int n\_size, const int nz\_size, cusparseOperation\_t oper\_t)  
*Interface of the virtual function of the product of  $A*x$ .*
- static void [\\_MxProduct](#) (void \*instance, cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, cusparseDnVecDescr\_t x, cusparseDnVecDescr\_t prod\_Mx, const int n\_size, const int nz\_size, cusparseOperation\_t oper\_t)  
*Interface of the virtual function of the product of  $M^{-1}*x$ .*
- static int [\\_Progress](#) (void \*instance, const cuDoubleComplex \*m, const [lcg\\_float](#) converge, const [clcg\\_para](#) \*param, const int n\_size, const int nz\_size, const int k)  
*Interface of the virtual function of the process monitoring.*

## Protected Attributes

- [clcg\\_para](#) param\_
- unsigned int [inter\\_](#)
- bool [silent\\_](#)

### 3.1.1 Detailed Description

Complex linear conjugate gradient solver class.

### 3.1.2 Constructor & Destructor Documentation

#### 3.1.2.1 CLCG\_CUDA\_Solver()

```
CLCG_CUDA_Solver::CLCG_CUDA_Solver ( )
```

#### 3.1.2.2 ~CLCG\_CUDA\_Solver()

```
virtual CLCG_CUDA_Solver::~~CLCG_CUDA_Solver ( ) [inline], [virtual]
```

### 3.1.3 Member Function Documentation

#### 3.1.3.1 \_AxProduct()

```
static void CLCG_CUDA_Solver::_AxProduct (
    void * instance,
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    cusparseDnVecDescr_t x,
    cusparseDnVecDescr_t prod_Ax,
    const int n_size,
    const int nz_size,
    cusparseOperation_t oper_t ) [inline], [static]
```

Interface of the virtual function of the product of  $A*x$ .

## Parameters

<i>instance</i>	User data sent to identify the function address
<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x[in]</i>	Pointer of the multiplier
<i>prod_Ax[out]</i>	Pointer of the product
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>oper_t</i>	Cuspars operator. This parameter is not need by the algorithm. It is passed for CUDA usages

## 3.1.3.2 \_MxProduct()

```
static void CLCG_CUDA_Solver::_MxProduct (
    void * instance,
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    cusparseDnVecDescr_t x,
    cusparseDnVecDescr_t prod_Mx,
    const int n_size,
    const int nz_size,
    cusparseOperation_t oper_t ) [inline], [static]
```

Interface of the virtual function of the product of  $M^{-1} \cdot x$ .

## Parameters

<i>instance</i>	User data sent to identify the function address
<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x[in]</i>	Pointer of the multiplier
<i>prod_Mx[out]</i>	Pointer of the product
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>oper_t</i>	Cuspars operator. This parameter is not need by the algorithm. It is passed for CUDA usages

## 3.1.3.3 \_Progress()

```
static int CLCG_CUDA_Solver::_Progress (
    void * instance,
    const cuDoubleComplex * m,
    const lcg_float converge,
    const clcg_para * param,
```

```

    const int n_size,
    const int nz_size,
    const int k ) [inline], [static]

```

Interface of the virtual function of the process monitoring.

#### Parameters

<i>instance</i>	User data sent to identify the function address
<i>m</i>	Pointer of the current solution
<i>converge</i>	Current value of the convergence
<i>param</i>	Pointer of the parameters used in the algorithms
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>k</i>	Current iteration times

#### Returns

int Status of the process

#### 3.1.3.4 AxProduct()

```

virtual void CLCG_CUDA_Solver::AxProduct (
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    cusparseDnVecDescr_t x,
    cusparseDnVecDescr_t prod_Ax,
    const int n_size,
    const int nz_size,
    cusparseOperation_t oper_t ) [pure virtual]

```

Virtual function of the product of  $A \cdot x$ .

#### Parameters

<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x[in]</i>	Pointer of the multiplier
<i>prod_Ax[out]</i>	Pointer of the product
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>oper_t</i>	Cusparse operator. This parameter is not need by the algorithm. It is passed for CUDA usages

### 3.1.3.5 Minimize()

```
void CLCG_CUDA_Solver::Minimize (
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    cuDoubleComplex * x,
    cuDoubleComplex * b,
    const int n_size,
    const int nz_size,
    clcg_solver_enum solver_id = CLCG_BICG,
    bool verbose = true,
    bool er_throw = false )
```

Run the constrained minimizing process.

#### Parameters

<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x</i>	Pointer of the solution vector
<i>b</i>	Pointer of the targeting vector
<i>n_size</i>	Size of the solution vector
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>solver_id</i>	Solver type
<i>verbose</i>	Report more information of the full process
<i>er_throw</i>	Instead of showing error messages on screen, throw them out using std::exception

### 3.1.3.6 MinimizePreconditioned()

```
void CLCG_CUDA_Solver::MinimizePreconditioned (
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    cuDoubleComplex * x,
    cuDoubleComplex * b,
    const int n_size,
    const int nz_size,
    clcg_solver_enum solver_id = CLCG_PCG,
    bool verbose = true,
    bool er_throw = false )
```

Run the preconditioned minimizing process.

#### Parameters

<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x</i>	Pointer of the solution vector
<i>b</i>	Pointer of the targeting vector
<i>n_size</i>	Size of the solution vector

## Parameters

<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>solver_id</i>	Solver type
<i>verbose</i>	Report more information of the full process
<i>er_throw</i>	Instead of showing error messages on screen, throw them out using <code>std::exception</code>

**3.1.3.7 MxProduct()**

```
virtual void CLCG_CUDA_Solver::MxProduct (
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    cusparseDnVecDescr_t x,
    cusparseDnVecDescr_t prod_Mx,
    const int n_size,
    const int nz_size,
    cusparseOperation_t oper_t ) [pure virtual]
```

Virtual function of the product of  $M^{-1} * x$ .

## Parameters

<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x[in]</i>	Pointer of the multiplier
<i>prod_Mx[out]</i>	Pointer of the product
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>oper_t</i>	Cusparse operator. This parameter is not need by the algorithm. It is passed for CUDA usages

**3.1.3.8 Progress()**

```
virtual int CLCG_CUDA_Solver::Progress (
    const cuDoubleComplex * m,
    const lcg_float converge,
    const clcg_para * param,
    const int n_size,
    const int nz_size,
    const int k ) [virtual]
```

Virtual function of the process monitoring.



## Parameters

<i>m</i>	Pointer of the current solution
<i>converge</i>	Current value of the convergence
<i>param</i>	Pointer of the parameters used in the algorithms
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>k</i>	Current iteration times

## Returns

int Status of the process

## 3.1.3.9 set\_clcg\_parameter()

```
void CLCG_CUDA_Solver::set_clcg_parameter (
    const clcg_para & in_param )
```

Set the parameters of the algorithms.

## Parameters

<i>in_param</i>	the input parameters
-----------------	----------------------

## 3.1.3.10 set\_report\_interval()

```
void CLCG_CUDA_Solver::set_report_interval (
    unsigned int inter )
```

Set the interval to run the process monitoring function.

## Parameters

<i>inter</i>	the interval
--------------	--------------

## 3.1.3.11 silent()

```
void CLCG_CUDA_Solver::silent ( )
```

Do not report any processes.

### 3.1.4 Field Documentation

#### 3.1.4.1 inter\_

```
unsigned int CLCG_CUDA_Solver::inter_ [protected]
```

#### 3.1.4.2 param\_

```
clcg_para CLCG_CUDA_Solver::param_ [protected]
```

#### 3.1.4.3 silent\_

```
bool CLCG_CUDA_Solver::silent_ [protected]
```

The documentation for this class was generated from the following file:

- [solver\\_cuda.h](#)

## 3.2 CLCG\_CUDAF\_Solver Class Reference

Complex linear conjugate gradient solver class.

```
#include <solver_cuda.h>
```

### Public Member Functions

- [CLCG\\_CUDAF\\_Solver](#) ()
- virtual [~CLCG\\_CUDAF\\_Solver](#) ()
- virtual void [AxProduct](#) (cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, cusparseDnVecDescr\_t x, cusparseDnVecDescr\_t prod\_Ax, const int n\_size, const int nz\_size, cusparseOperation\_t oper\_t)=0  
*Virtual function of the product of  $A \cdot x$ .*
- virtual void [MxProduct](#) (cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, cusparseDnVecDescr\_t x, cusparseDnVecDescr\_t prod\_Mx, const int n\_size, const int nz\_size, cusparseOperation\_t oper\_t)=0  
*Virtual function of the product of  $M^{\wedge} - 1 \cdot x$ .*
- virtual int [Progress](#) (const cuComplex \*m, const float converge, const [clcg\\_para](#) \*param, const int n\_size, const int nz\_size, const int k)  
*Virtual function of the process monitoring.*
- void [silent](#) ()  
*Do not report any processes.*
- void [set\\_report\\_interval](#) (unsigned int inter)  
*Set the interval to run the process monitoring function.*
- void [set\\_clcg\\_parameter](#) (const [clcg\\_para](#) &in\_param)  
*Set the parameters of the algorithms.*
- void [Minimize](#) (cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, cuComplex \*x, cuComplex \*b, const int n\_size, const int nz\_size, [clcg\\_solver\\_enum](#) solver\_id=CLCG\_BICG, bool verbose=true, bool er\_throw=false)  
*Run the constrained minimizing process.*
- void [MinimizePreconditioned](#) (cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, cuComplex \*x, cuComplex \*b, const int n\_size, const int nz\_size, [clcg\\_solver\\_enum](#) solver\_id=CLCG\_PCG, bool verbose=true, bool er\_throw=false)  
*Run the preconditioned minimizing process.*

## Static Public Member Functions

- static void [\\_AxProduct](#) (void \*instance, cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, cusparseDnVecDescr\_t x, cusparseDnVecDescr\_t prod\_Ax, const int n\_size, const int nz\_size, cusparseOperation\_t oper\_t)  
*Interface of the virtual function of the product of  $A*x$ .*
- static void [\\_MxProduct](#) (void \*instance, cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, cusparseDnVecDescr\_t x, cusparseDnVecDescr\_t prod\_Mx, const int n\_size, const int nz\_size, cusparseOperation\_t oper\_t)  
*Interface of the virtual function of the product of  $M^{-1}*x$ .*
- static int [\\_Progress](#) (void \*instance, const cuComplex \*m, const float converge, const [clcg\\_para](#) \*param, const int n\_size, const int nz\_size, const int k)  
*Interface of the virtual function of the process monitoring.*

## Protected Attributes

- [clcg\\_para](#) param\_
- unsigned int [inter\\_](#)
- bool [silent\\_](#)

## 3.2.1 Detailed Description

Complex linear conjugate gradient solver class.

## 3.2.2 Constructor & Destructor Documentation

### 3.2.2.1 CLCG\_CUDAF\_Solver()

```
CLCG_CUDAF_Solver::CLCG_CUDAF_Solver ( )
```

### 3.2.2.2 ~CLCG\_CUDAF\_Solver()

```
virtual CLCG_CUDAF_Solver::~~CLCG_CUDAF_Solver ( ) [inline], [virtual]
```

## 3.2.3 Member Function Documentation

### 3.2.3.1 \_AxProduct()

```
static void CLCG_CUDAF_Solver::_AxProduct (
    void * instance,
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    cusparseDnVecDescr_t x,
    cusparseDnVecDescr_t prod_Ax,
    const int n_size,
    const int nz_size,
    cusparseOperation_t oper_t ) [inline], [static]
```

Interface of the virtual function of the product of  $A*x$ .

## Parameters

<i>instance</i>	User data sent to identify the function address
<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x[in]</i>	Pointer of the multiplier
<i>prod_Ax[out]</i>	Pointer of the product
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>oper_t</i>	Cuspars operator. This parameter is not need by the algorithm. It is passed for CUDA usages

3.2.3.2 **\_MxProduct()**

```
static void CLCG_CUDAF_Solver::_MxProduct (
    void * instance,
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    cusparseDnVecDescr_t x,
    cusparseDnVecDescr_t prod_Mx,
    const int n_size,
    const int nz_size,
    cusparseOperation_t oper_t ) [inline], [static]
```

Interface of the virtual function of the product of  $M^{-1} \cdot x$ .

## Parameters

<i>instance</i>	User data sent to identify the function address
<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x[in]</i>	Pointer of the multiplier
<i>prod_Mx[out]</i>	Pointer of the product
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>oper_t</i>	Cuspars operator. This parameter is not need by the algorithm. It is passed for CUDA usages

3.2.3.3 **\_Progress()**

```
static int CLCG_CUDAF_Solver::_Progress (
    void * instance,
    const cuComplex * m,
    const float converge,
    const clcg\_para * param,
```

```

    const int n_size,
    const int nz_size,
    const int k ) [inline], [static]

```

Interface of the virtual function of the process monitoring.

#### Parameters

<i>instance</i>	User data sent to identify the function address
<i>m</i>	Pointer of the current solution
<i>converge</i>	Current value of the convergence
<i>param</i>	Pointer of the parameters used in the algorithms
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>k</i>	Current iteration times

#### Returns

int Status of the process

#### 3.2.3.4 AxProduct()

```

virtual void CLCG_CUDAF_Solver::AxProduct (
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    cusparseDnVecDescr_t x,
    cusparseDnVecDescr_t prod_Ax,
    const int n_size,
    const int nz_size,
    cusparseOperation_t oper_t ) [pure virtual]

```

Virtual function of the product of  $A \times x$ .

#### Parameters

<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x[in]</i>	Pointer of the multiplier
<i>prod_Ax[out]</i>	Pointer of the product
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>oper_t</i>	Cusparse operator. This parameter is not need by the algorithm. It is passed for CUDA usages

### 3.2.3.5 Minimize()

```
void CLCG_CUDAF_Solver::Minimize (
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    cuComplex * x,
    cuComplex * b,
    const int n_size,
    const int nz_size,
    clcg_solver_enum solver_id = CLCG_BICG,
    bool verbose = true,
    bool er_throw = false )
```

Run the constrained minimizing process.

#### Parameters

<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x</i>	Pointer of the solution vector
<i>b</i>	Pointer of the targeting vector
<i>n_size</i>	Size of the solution vector
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>solver_id</i>	Solver type
<i>verbose</i>	Report more information of the full process
<i>er_throw</i>	Instead of showing error messages on screen, throw them out using <code>std::exception</code>

### 3.2.3.6 MinimizePreconditioned()

```
void CLCG_CUDAF_Solver::MinimizePreconditioned (
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    cuComplex * x,
    cuComplex * b,
    const int n_size,
    const int nz_size,
    clcg_solver_enum solver_id = CLCG_PCG,
    bool verbose = true,
    bool er_throw = false )
```

Run the preconditioned minimizing process.

#### Parameters

<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x</i>	Pointer of the solution vector
<i>b</i>	Pointer of the targeting vector
<i>n_size</i>	Size of the solution vector

## Parameters

<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>solver_id</i>	Solver type
<i>verbose</i>	Report more information of the full process
<i>er_throw</i>	Instead of showing error messages on screen, throw them out using <code>std::exception</code>

**3.2.3.7 MxProduct()**

```
virtual void CLCG_CUDAF_Solver::MxProduct (
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    cusparseDnVecDescr_t x,
    cusparseDnVecDescr_t prod_Mx,
    const int n_size,
    const int nz_size,
    cusparseOperation_t oper_t ) [pure virtual]
```

Virtual function of the product of  $M^{-1} * x$ .

## Parameters

<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x[in]</i>	Pointer of the multiplier
<i>prod_Mx[out]</i>	Pointer of the product
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>oper_t</i>	Cusparse operator. This parameter is not need by the algorithm. It is passed for CUDA usages

**3.2.3.8 Progress()**

```
virtual int CLCG_CUDAF_Solver::Progress (
    const cuComplex * m,
    const float converge,
    const clcg_para * param,
    const int n_size,
    const int nz_size,
    const int k ) [virtual]
```

Virtual function of the process monitoring.

**Parameters**

<i>m</i>	Pointer of the current solution
<i>converge</i>	Current value of the convergence
<i>param</i>	Pointer of the parameters used in the algorithms
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>k</i>	Current iteration times

**Returns**

int Status of the process

**3.2.3.9 set\_clcg\_parameter()**

```
void CLCG_CUDA_F_Solver::set_clcg_parameter (
    const clcg_para & in_param )
```

Set the parameters of the algorithms.

**Parameters**

<i>in_param</i>	the input parameters
-----------------	----------------------

**3.2.3.10 set\_report\_interval()**

```
void CLCG_CUDA_F_Solver::set_report_interval (
    unsigned int inter )
```

Set the interval to run the process monitoring function.

**Parameters**

<i>inter</i>	the interval
--------------	--------------

**3.2.3.11 silent()**

```
void CLCG_CUDA_F_Solver::silent ( )
```

Do not report any processes.



### 3.2.4 Field Documentation

#### 3.2.4.1 inter\_

```
unsigned int CLCG_CUDA_Solver::inter_ [protected]
```

#### 3.2.4.2 param\_

```
clcg_para CLCG_CUDA_Solver::param_ [protected]
```

#### 3.2.4.3 silent\_

```
bool CLCG_CUDA_Solver::silent_ [protected]
```

The documentation for this class was generated from the following file:

- [solver\\_cuda.h](#)

## 3.3 CLCG\_EIGEN\_Solver Class Reference

Complex linear conjugate gradient solver class.

```
#include <solver_eigen.h>
```

### Public Member Functions

- [CLCG\\_EIGEN\\_Solver](#) ()
- virtual [~CLCG\\_EIGEN\\_Solver](#) ()
- virtual void [AxProduct](#) (const Eigen::VectorXcd &x, Eigen::VectorXcd &prod\_Ax, [lcg\\_matrix\\_e](#) layout, [clcg\\_complex\\_e](#) conjugate)=0  
*Interface of the virtual function of the product of  $A \cdot x$ .*
- virtual void [MxProduct](#) (const Eigen::VectorXcd &x, Eigen::VectorXcd &prod\_Mx, [lcg\\_matrix\\_e](#) layout, [clcg\\_complex\\_e](#) conjugate)=0  
*Interface of the virtual function of the product of  $M^{-1} \cdot x$ .*
- virtual int [Progress](#) (const Eigen::VectorXcd \*m, const [lcg\\_float](#) converge, const [clcg\\_para](#) \*param, const int k)  
*Virtual function of the process monitoring.*
- void [silent](#) ()  
*Do not report any processes.*
- void [set\\_report\\_interval](#) (unsigned int inter)  
*Set the interval to run the process monitoring function.*
- void [set\\_clcg\\_parameter](#) (const [clcg\\_para](#) &in\_param)  
*Set the interval to run the process monitoring function.*
- void [Minimize](#) (Eigen::VectorXcd &m, const Eigen::VectorXcd &b, [clcg\\_solver\\_enum](#) solver\_id=[CLCG\\_CGS](#), bool verbose=true, bool er\_throw=false)  
*Run the minimizing process.*
- void [MinimizePreconditioned](#) (Eigen::VectorXcd &m, const Eigen::VectorXcd &b, [clcg\\_solver\\_enum](#) solver\_id=[CLCG\\_PBICG](#), bool verbose=true, bool er\_throw=false)  
*Run the preconditioned minimizing process.*

## Static Public Member Functions

- static void [\\_AxProduct](#) (void \*instance, const Eigen::VectorXcd &x, Eigen::VectorXcd &prod\_Ax, [lcg\\_matrix\\_e](#) layout, [clcg\\_complex\\_e](#) conjugate)  
*Interface of the virtual function of the product of  $A*x$ .*
- static void [\\_MxProduct](#) (void \*instance, const Eigen::VectorXcd &x, Eigen::VectorXcd &prod\_Mx, [lcg\\_matrix\\_e](#) layout, [clcg\\_complex\\_e](#) conjugate)  
*Interface of the virtual function of the product of  $M^{-1}*x$ .*
- static int [\\_Progress](#) (void \*instance, const Eigen::VectorXcd \*m, const [lcg\\_float](#) converge, const [clcg\\_para](#) \*param, const int k)  
*Interface of the virtual function of the process monitoring.*

## Protected Attributes

- [clcg\\_para](#) param\_
- unsigned int [inter\\_](#)
- bool [silent\\_](#)

### 3.3.1 Detailed Description

Complex linear conjugate gradient solver class.

### 3.3.2 Constructor & Destructor Documentation

#### 3.3.2.1 CLCG\_EIGEN\_Solver()

```
CLCG_EIGEN_Solver::CLCG_EIGEN_Solver ( )
```

#### 3.3.2.2 ~CLCG\_EIGEN\_Solver()

```
virtual CLCG_EIGEN_Solver::~~CLCG_EIGEN_Solver ( ) [inline], [virtual]
```

### 3.3.3 Member Function Documentation

#### 3.3.3.1 \_AxProduct()

```
static void CLCG_EIGEN_Solver::_AxProduct (
    void * instance,
    const Eigen::VectorXcd & x,
    Eigen::VectorXcd & prod_Ax,
    lcg\_matrix\_e layout,
    clcg\_complex\_e conjugate ) [inline], [static]
```

Interface of the virtual function of the product of  $A*x$ .

## Parameters

<i>instance</i>	User data sent to identify the function address
<i>x[in]</i>	Pointer of the multiplier
<i>prod_Ax[out]</i>	Pointer of the product
<i>layout</i>	Layout of the kernel matrix. This is passed for the <a href="#">clcg_matvec()</a> function
<i>conjugate</i>	Welther to use conjugate of the kernel matrix. This is passed for the <a href="#">clcg_matvec()</a> function

## 3.3.3.2 \_MxProduct()

```
static void CLCG_EIGEN_Solver::_MxProduct (
    void * instance,
    const Eigen::VectorXcd & x,
    Eigen::VectorXcd & prod_Mx,
    lcg_matrix_e layout,
    clcg_complex_e conjugate ) [inline], [static]
```

Interface of the virtual function of the product of  $M^{-1} * x$ .

## Parameters

<i>instance</i>	User data sent to identify the function address
<i>x[in]</i>	Pointer of the multiplier
<i>prod_Mx[out]</i>	Pointer of the product
<i>layout</i>	Layout of the kernel matrix. This is passed for the <a href="#">clcg_matvec()</a> function
<i>conjugate</i>	Welther to use conjugate of the kernel matrix. This is passed for the <a href="#">clcg_matvec()</a> function

## 3.3.3.3 \_Progress()

```
static int CLCG_EIGEN_Solver::_Progress (
    void * instance,
    const Eigen::VectorXcd * m,
    const lcg_float converge,
    const clcg_para * param,
    const int k ) [inline], [static]
```

Interface of the virtual function of the process monitoring.

## Parameters

<i>instance</i>	User data sent to identify the function address
<i>m</i>	Pointer of the current solution
<i>converge</i>	Current value of the convergence
<i>param</i>	Pointer of the parameters used in the algorithms
<i>k</i>	Current iteration times

**Returns**

int Status of the process

**3.3.3.4 AxProduct()**

```
virtual void CLCG_EIGEN_Solver::AxProduct (
    const Eigen::VectorXcd & x,
    Eigen::VectorXcd & prod_Ax,
    clcg_matrix_e layout,
    clcg_complex_e conjugate ) [pure virtual]
```

Interface of the virtual function of the product of  $A \cdot x$ .

**Parameters**

<i>x[in]</i>	Pointer of the multiplier
<i>prod_Ax[out]</i>	Pointer of the product
<i>layout</i>	Layout of the kernel matrix. This is passed for the <a href="#">clcg_matvec()</a> function
<i>conjugate</i>	Welther to use conjugate of the kernel matrix. This is passed for the <a href="#">clcg_matvec()</a> function

**3.3.3.5 Minimize()**

```
void CLCG_EIGEN_Solver::Minimize (
    Eigen::VectorXcd & m,
    const Eigen::VectorXcd & b,
    clcg_solver_enum solver_id = CLCG_CGS,
    bool verbose = true,
    bool er_throw = false )
```

Run the minimizing process.

**Parameters**

<i>m</i>	Pointer of the solution vector
<i>b</i>	Pointer of the targeting vector
<i>solver_id</i>	Solver type
<i>verbose</i>	Report more information of the full process
<i>er_throw</i>	Instead of showing error messages on screen, throw them out using <code>std::exception</code>

**3.3.3.6 MinimizePreconditioned()**

```
void CLCG_EIGEN_Solver::MinimizePreconditioned (
```

```

Eigen::VectorXcd & m,
const Eigen::VectorXcd & b,
clcg_solver_enum solver_id = CLCG_PBICG,
bool verbose = true,
bool er_throw = false )

```

Run the preconditioned minimizing process.

#### Parameters

<i>m</i>	Pointer of the solution vector
<i>b</i>	Pointer of the targeting vector
<i>solver_id</i>	Solver type
<i>verbose</i>	Report more information of the full process
<i>er_throw</i>	Instead of showing error messages on screen, throw them out using <code>std::exception</code>

#### 3.3.3.7 MxProduct()

```

virtual void CLCG_EIGEN_Solver::MxProduct (
    const Eigen::VectorXcd & x,
    Eigen::VectorXcd & prod_Mx,
    lcg_matrix_e layout,
    clcg_complex_e conjugate ) [pure virtual]

```

Interface of the virtual function of the product of  $M^{-1} * x$ .

#### Parameters

<i>x[in]</i>	Pointer of the multiplier
<i>prod_Mx[out]</i>	Pointer of the product
<i>layout</i>	Layout of the kernel matrix. This is passed for the <a href="#">clcg_matvec()</a> function
<i>conjugate</i>	Whether to use conjugate of the kernel matrix. This is passed for the <a href="#">clcg_matvec()</a> function

#### 3.3.3.8 Progress()

```

int CLCG_EIGEN_Solver::Progress (
    const Eigen::VectorXcd * m,
    const lcg_float converge,
    const clcg_para * param,
    const int k ) [virtual]

```

Virtual function of the process monitoring.

#### Parameters

<i>m</i>	Pointer of the current solution
----------	---------------------------------

**Parameters**

<i>converge</i>	Current value of the convergence
<i>param</i>	Pointer of the parameters used in the algorithms
<i>k</i>	Current iteration times

**Returns**

int Status of the process

**3.3.3.9 set\_clcg\_parameter()**

```
void CLCG_EIGEN_Solver::set_clcg_parameter (
    const clcg_para & in_param )
```

Set the interval to run the process monitoring function.

**Parameters**

<i>inter</i>	the interval
--------------	--------------

**3.3.3.10 set\_report\_interval()**

```
void CLCG_EIGEN_Solver::set_report_interval (
    unsigned int inter )
```

Set the interval to run the process monitoring function.

**Parameters**

<i>inter</i>	the interval
--------------	--------------

**3.3.3.11 silent()**

```
void CLCG_EIGEN_Solver::silent ( )
```

Do not report any processes.

**3.3.4 Field Documentation**

#### 3.3.4.1 inter\_

```
unsigned int CLCG_EIGEN_Solver::inter_ [protected]
```

#### 3.3.4.2 param\_

```
clcg_para CLCG_EIGEN_Solver::param_ [protected]
```

#### 3.3.4.3 silent\_

```
bool CLCG_EIGEN_Solver::silent_ [protected]
```

The documentation for this class was generated from the following files:

- [solver\\_eigen.h](#)
- [solver\\_eigen.cpp](#)

## 3.4 clcg\_para Struct Reference

Parameters of the conjugate gradient methods.

```
#include <util.h>
```

### Data Fields

- int [max\\_iterations](#)
- [lcg\\_float](#) epsilon
- int [abs\\_diff](#)

#### 3.4.1 Detailed Description

Parameters of the conjugate gradient methods.

#### 3.4.2 Field Documentation

### 3.4.2.1 abs\_diff

```
int clcg_para::abs_diff
```

Whether to use absolute mean differences (AMD) between  $|Ax - B|$  to evaluate the process. The default value is false which means the gradient based evaluating method is used. The AMD based method will be used if this variable is set to true. This parameter is only applied to the non-constrained methods.

### 3.4.2.2 epsilon

```
lcg_float clcg_para::epsilon
```

Epsilon for convergence test. This parameter determines the accuracy with which the solution is to be found. A minimization terminates when  $\|g\|/\max(\|x\|, 1.0) \leq \epsilon$  or  $|Ax - B| \leq \epsilon$  for the [lcg\\_solver\(\)](#) function, where  $\|\cdot\|$  denotes the Euclidean (L2) norm and  $|\cdot|$  denotes the L1 norm. The default value of epsilon is 1e-6. For box-constrained methods, the convergence test is implemented using  $\|P(m-g) - m\| \leq \epsilon$ , in which P is the projector that transfers m into the constrained domain.

### 3.4.2.3 max\_iterations

```
int clcg_para::max_iterations
```

Maximal iteration times. The process will continue till the convergence is met if this option is set to zero (default).

The documentation for this struct was generated from the following file:

- [util.h](#)

## 3.5 CLCG\_Solver Class Reference

Complex linear conjugate gradient solver class.

```
#include <solver.h>
```

### Public Member Functions

- [CLCG\\_Solver](#) ()
- virtual [~CLCG\\_Solver](#) ()
- virtual void [AxProduct](#) (const [lcg\\_complex](#) \*x, [lcg\\_complex](#) \*prod\_Ax, const int x\_size, [lcg\\_matrix\\_e](#) layout, [clcg\\_complex\\_e](#) conjugate)=0  
*Interface of the virtual function of the product of A\*x.*
- virtual int [Progress](#) (const [lcg\\_complex](#) \*m, const [lcg\\_float](#) converge, const [clcg\\_para](#) \*param, const int n\_↔ size, const int k)  
*Interface of the virtual function of the process monitoring.*
- void [silent](#) ()  
*Do not report any processes.*
- void [set\\_report\\_interval](#) (unsigned int inter)  
*Set the interval to run the process monitoring function.*
- void [set\\_clcg\\_parameter](#) (const [clcg\\_para](#) &in\_param)  
*Set the parameters of the algorithms.*
- void [Minimize](#) ([lcg\\_complex](#) \*m, const [lcg\\_complex](#) \*b, int x\_size, [clcg\\_solver\\_enum](#) solver\_id=CLCG\_CGS, bool verbose=true, bool er\_throw=false)  
*Run the minimizing process.*



## Static Public Member Functions

- static void `_AxProduct` (void \*instance, const `lcg_complex` \*x, `lcg_complex` \*prod\_Ax, const int x\_size, `lcg_matrix_e` layout, `clcg_complex_e` conjugate)

*Interface of the virtual function of the product of  $A*x$ .*

- static int `_Progress` (void \*instance, const `lcg_complex` \*m, const `lcg_float` converge, const `clcg_para` \*param, const int n\_size, const int k)

*Interface of the virtual function of the process monitoring.*

## Protected Attributes

- `clcg_para` param\_
- unsigned int `inter_`
- bool `silent_`

### 3.5.1 Detailed Description

Complex linear conjugate gradient solver class.

### 3.5.2 Constructor & Destructor Documentation

#### 3.5.2.1 CLCG\_Solver()

```
CLCG_Solver::CLCG_Solver ( )
```

#### 3.5.2.2 ~CLCG\_Solver()

```
virtual CLCG_Solver::~~CLCG_Solver ( ) [inline], [virtual]
```

### 3.5.3 Member Function Documentation

#### 3.5.3.1 \_AxProduct()

```
static void CLCG_Solver::_AxProduct (
    void * instance,
    const lcg_complex * x,
    lcg_complex * prod_Ax,
    const int x_size,
    lcg_matrix_e layout,
    clcg_complex_e conjugate ) [inline], [static]
```

Interface of the virtual function of the product of  $A*x$ .

**Parameters**

<i>instance</i>	User data sent to identify the function address
<i>x[in]</i>	Pointer of the multiplier
<i>prod_Ax[out]</i>	Pointer of the product
<i>x_size</i>	Size of the array
<i>layout</i>	Layout of the kernel matrix. This is passed for the <a href="#">clcg_matvec()</a> function
<i>conjugate</i>	Welther to use conjugate of the kernel matrix. This is passed for the <a href="#">clcg_matvec()</a> function

**3.5.3.2 \_Progress()**

```
static int CLCG_Solver::_Progress (
    void * instance,
    const lcg_complex * m,
    const lcg_float converge,
    const clcg_para * param,
    const int n_size,
    const int k ) [inline], [static]
```

Interface of the virtual function of the process monitoring.

**Parameters**

<i>instance</i>	User data sent to identify the function address
<i>m</i>	Pointer of the current solution
<i>converge</i>	Current value of the convergence
<i>param</i>	Pointer of the parameters used in the algorithms
<i>n_size</i>	Size of the solution
<i>k</i>	Current iteration times

**Returns**

int Status of the process

**3.5.3.3 AxProduct()**

```
virtual void CLCG_Solver::AxProduct (
    const lcg_complex * x,
    lcg_complex * prod_Ax,
    const int x_size,
    lcg_matrix_e layout,
    clcg_complex_e conjugate ) [pure virtual]
```

Interface of the virtual function of the product of A\*x.

## Parameters

<i>x[in]</i>	Pointer of the multiplier
<i>prod_Ax[out]</i>	Pointer of the product
<i>x_size</i>	Size of the array
<i>layout</i>	Layout of the kernel matrix. This is passed for the <a href="#">clcg_matvec()</a> function
<i>conjugate</i>	Welther to use conjugate of the kernel matrix. This is passed for the <a href="#">clcg_matvec()</a> function

## 3.5.3.4 Minimize()

```
void CLCG_Solver::Minimize (
    lcg_complex * m,
    const lcg_complex * b,
    int x_size,
    clcg_solver_enum solver_id = CLCG_CGS,
    bool verbose = true,
    bool er_throw = false )
```

Run the minimizing process.

## Parameters

<i>m</i>	Pointer of the solution vector
<i>b</i>	Pointer of the targeting vector
<i>x_size</i>	Size of the solution vector
<i>solver_id</i>	Solver type
<i>verbose</i>	Report more information of the full process
<i>er_throw</i>	Instead of showing error messages on screen, throw them out using <code>std::exception</code>

## 3.5.3.5 Progress()

```
int CLCG_Solver::Progress (
    const lcg_complex * m,
    const lcg_float converge,
    const clcg_para * param,
    const int n_size,
    const int k ) [virtual]
```

Interface of the virtual function of the process monitoring.

## Parameters

<i>m</i>	Pointer of the current solution
<i>converge</i>	Current value of the convergence
<i>param</i>	Pointer of the parameters used in the algorithms
<i>n_size</i>	Size of the solution
<i>k</i>	Current iteration times

**Returns**

int Status of the process

**3.5.3.6 set\_clcg\_parameter()**

```
void CLCG_Solver::set_clcg_parameter (
    const clcg_para & in_param )
```

Set the parameters of the algorithms.

**Parameters**

<i>in_param</i>	the input parameters
-----------------	----------------------

**3.5.3.7 set\_report\_interval()**

```
void CLCG_Solver::set_report_interval (
    unsigned int inter )
```

Set the interval to run the process monitoring function.

**Parameters**

<i>inter</i>	the interval
--------------	--------------

**3.5.3.8 silent()**

```
void CLCG_Solver::silent ( )
```

Do not report any processes.

**3.5.4 Field Documentation****3.5.4.1 inter\_**

```
unsigned int CLCG_Solver::inter_ [protected]
```

## 3.5.4.2 param\_

```
lcg_para LCG_Solver::param_ [protected]
```

## 3.5.4.3 silent\_

```
bool LCG_Solver::silent_ [protected]
```

The documentation for this class was generated from the following files:

- [solver.h](#)
- [solver.cpp](#)

## 3.6 LCG\_CUDA\_Solver Class Reference

Linear conjugate gradient solver class.

```
#include <solver_cuda.h>
```

### Public Member Functions

- [LCG\\_CUDA\\_Solver](#) ()
- virtual [~LCG\\_CUDA\\_Solver](#) ()
- virtual void [AxProduct](#) (cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, cusparseDnVecDescr\_t x, cusparseDnVecDescr\_t prod\_Ax, const int n\_size, const int nz\_size)=0  
*Virtual function of the product of  $A \cdot x$ .*
- virtual void [MxProduct](#) (cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, cusparseDnVecDescr\_t x, cusparseDnVecDescr\_t prod\_Mx, const int n\_size, const int nz\_size)=0  
*Virtual function of the product of  $M^{-1} \cdot x$ .*
- virtual int [Progress](#) (const [lcg\\_float](#) \*m, const [lcg\\_float](#) converge, const [lcg\\_para](#) \*param, const int n\_size, const int nz\_size, const int k)  
*Virtual function of the process monitoring.*
- void [silent](#) ()  
*Do not report any processes.*
- void [set\\_report\\_interval](#) (unsigned int inter)  
*Set the interval to run the process monitoring function.*
- void [set\\_lcg\\_parameter](#) (const [lcg\\_para](#) &in\_param)  
*Set the parameters of the algorithms.*
- void [Minimize](#) (cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, [lcg\\_float](#) \*x, [lcg\\_float](#) \*b, const int n\_size, const int nz\_size, [lcg\\_solver\\_enum](#) solver\_id=[LCG\\_CG](#), bool verbose=true, bool er\_throw=false)  
*Run the constrained minimizing process.*
- void [MinimizePreconditioned](#) (cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, [lcg\\_float](#) \*x, [lcg\\_float](#) \*b, const int n\_size, const int nz\_size, [lcg\\_solver\\_enum](#) solver\_id=[LCG\\_CG](#), bool verbose=true, bool er\_throw=false)  
*Run the preconditioned minimizing process.*
- void [MinimizeConstrained](#) (cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, [lcg\\_float](#) \*x, const [lcg\\_float](#) \*b, const [lcg\\_float](#) \*low, const [lcg\\_float](#) \*hig, const int n\_size, const int nz\_size, [lcg\\_solver\\_enum](#) solver\_id=[LCG\\_PG](#), bool verbose=true, bool er\_throw=false)  
*Run the constrained minimizing process.*

## Static Public Member Functions

- static void [\\_AxProduct](#) (void \*instance, cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, cusparseDnVecDescr\_t x, cusparseDnVecDescr\_t prod\_Ax, const int n\_size, const int nz\_size)  
*Interface of the virtual function of the product of  $A*x$ .*
- static void [\\_MxProduct](#) (void \*instance, cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, cusparseDnVecDescr\_t x, cusparseDnVecDescr\_t prod\_Mx, const int n\_size, const int nz\_size)  
*Interface of the virtual function of the product of  $M^{\wedge}-1*x$ .*
- static int [\\_Progress](#) (void \*instance, const [lcg\\_float](#) \*m, const [lcg\\_float](#) converge, const [lcg\\_para](#) \*param, const int n\_size, const int nz\_size, const int k)  
*Interface of the virtual function of the process monitoring.*

## Protected Attributes

- [lcg\\_para](#) param\_
- unsigned int [inter\\_](#)
- bool [silent\\_](#)

### 3.6.1 Detailed Description

Linear conjugate gradient solver class.

### 3.6.2 Constructor & Destructor Documentation

#### 3.6.2.1 LCG\_CUDA\_Solver()

```
LCG_CUDA_Solver::LCG_CUDA_Solver ( )
```

#### 3.6.2.2 ~LCG\_CUDA\_Solver()

```
virtual LCG_CUDA_Solver::~~LCG_CUDA_Solver ( ) [inline], [virtual]
```

### 3.6.3 Member Function Documentation

#### 3.6.3.1 \_AxProduct()

```
static void LCG_CUDA_Solver::_AxProduct (
    void * instance,
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    cusparseDnVecDescr_t x,
    cusparseDnVecDescr_t prod_Ax,
    const int n_size,
    const int nz_size ) [inline], [static]
```

Interface of the virtual function of the product of  $A*x$ .

## Parameters

<i>instance</i>	User data sent to identify the function address
<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x[in]</i>	Pointer of the multiplier
<i>prod_Ax[out]</i>	Pointer of the product
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages

## 3.6.3.2 \_MxProduct()

```
static void LCG_CUDA_Solver::_MxProduct (
    void * instance,
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    cusparseDnVecDescr_t x,
    cusparseDnVecDescr_t prod_Mx,
    const int n_size,
    const int nz_size ) [inline], [static]
```

Interface of the virtual function of the product of  $M^{-1} * x$ .

## Parameters

<i>instance</i>	User data sent to identify the function address
<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x[in]</i>	Pointer of the multiplier
<i>prod_Mx[out]</i>	Pointer of the product
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages

## 3.6.3.3 \_Progress()

```
static int LCG_CUDA_Solver::_Progress (
    void * instance,
    const lcg_float * m,
    const lcg_float converge,
    const lcg_para * param,
    const int n_size,
    const int nz_size,
    const int k ) [inline], [static]
```

Interface of the virtual function of the process monitoring.

**Parameters**

<i>instance</i>	User data sent to identify the function address
<i>m</i>	Pointer of the current solution
<i>converge</i>	Current value of the convergence
<i>param</i>	Pointer of the parameters used in the algorithms
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>k</i>	Current iteration times

**Returns**

int Status of the process

**3.6.3.4 AxProduct()**

```
virtual void LCG_CUDA_Solver::AxProduct (
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    cusparseDnVecDescr_t x,
    cusparseDnVecDescr_t prod_Ax,
    const int n_size,
    const int nz_size ) [pure virtual]
```

Virtual function of the product of  $A \cdot x$ .

**Parameters**

<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x[in]</i>	Pointer of the multiplier
<i>prod_Ax[out]</i>	Pointer of the product
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages

**3.6.3.5 Minimize()**

```
void LCG_CUDA_Solver::Minimize (
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    lcg_float * x,
    lcg_float * b,
    const int n_size,
```



```

const int nz_size,
lcg_solver_enum solver_id = LCG_CG,
bool verbose = true,
bool er_throw = false )

```

Run the constrained minimizing process.

#### Parameters

<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x</i>	Pointer of the solution vector
<i>b</i>	Pointer of the targeting vector
<i>n_size</i>	Size of the solution vector
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>solver_id</i>	Solver type
<i>verbose</i>	Report more information of the full process
<i>er_throw</i>	Instead of showing error messages on screen, throw them out using <code>std::exception</code>

#### 3.6.3.6 MinimizeConstrained()

```

void LCG_CUDA_Solver::MinimizeConstrained (
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    lcg_float * x,
    const lcg_float * b,
    const lcg_float * low,
    const lcg_float * hig,
    const int n_size,
    const int nz_size,
    lcg_solver_enum solver_id = LCG_PG,
    bool verbose = true,
    bool er_throw = false )

```

Run the constrained minimizing process.

#### Parameters

<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x</i>	Pointer of the solution vector
<i>b</i>	Pointer of the targeting vector
<i>low</i>	Lower bound of the solution vector
<i>hig</i>	Higher bound of the solution vector
<i>n_size</i>	Size of the solution vector
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>solver_id</i>	Solver type
<i>verbose</i>	Report more information of the full process
<i>er_throw</i>	Instead of showing error messages on screen, throw them out using <code>std::exception</code>

### 3.6.3.7 MinimizePreconditioned()

```
void LCG_CUDA_Solver::MinimizePreconditioned (
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    lcg_float * x,
    lcg_float * b,
    const int n_size,
    const int nz_size,
    lcg_solver_enum solver_id = LCG_CG,
    bool verbose = true,
    bool er_throw = false )
```

Run the preconditioned minimizing process.

#### Parameters

<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x</i>	Pointer of the solution vector
<i>b</i>	Pointer of the targeting vector
<i>n_size</i>	Size of the solution vector
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>solver_id</i>	Solver type
<i>verbose</i>	Report more information of the full process
<i>er_throw</i>	Instead of showing error messages on screen, throw them out using std::exception

### 3.6.3.8 MxProduct()

```
virtual void LCG_CUDA_Solver::MxProduct (
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    cusparseDnVecDescr_t x,
    cusparseDnVecDescr_t prod_Mx,
    const int n_size,
    const int nz_size ) [pure virtual]
```

Virtual function of the product of  $M^{-1} * x$ .

#### Parameters

<i>cub_handle</i>	Handler of the CuBLAS library
<i>cus_handle</i>	Handler of the CuSparse library
<i>x[in]</i>	Pointer of the multiplier
<i>prod_Mx[out]</i>	Pointer of the product
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages

### 3.6.3.9 Progress()

```
virtual int LCG_CUDA_Solver::Progress (
    const lcg_float * m,
    const lcg_float converge,
    const lcg_para * param,
    const int n_size,
    const int nz_size,
    const int k ) [virtual]
```

Virtual function of the process monitoring.

#### Parameters

<i>m</i>	Pointer of the current solution
<i>converge</i>	Current value of the convergence
<i>param</i>	Pointer of the parameters used in the algorithms
<i>n_size</i>	Size of the solution
<i>nz_size</i>	Non-zero size of the sparse kernel matrix. This parameter is not need by the algorithm. It is passed for CUDA usages
<i>k</i>	Current iteration times

#### Returns

int Status of the process

### 3.6.3.10 set\_lcg\_parameter()

```
void LCG_CUDA_Solver::set_lcg_parameter (
    const lcg_para & in_param )
```

Set the parameters of the algorithms.

#### Parameters

<i>in_param</i>	the input parameters
-----------------	----------------------

### 3.6.3.11 set\_report\_interval()

```
void LCG_CUDA_Solver::set_report_interval (
    unsigned int inter )
```

Set the interval to run the process monitoring function.

#### Parameters

<i>inter</i>	the interval
--------------	--------------

#### 3.6.3.12 `silent()`

```
void LCG_CUDA_Solver::silent ( )
```

Do not report any processes.

### 3.6.4 Field Documentation

#### 3.6.4.1 `inter_`

```
unsigned int LCG_CUDA_Solver::inter_ [protected]
```

#### 3.6.4.2 `param_`

```
lcg_para LCG_CUDA_Solver::param_ [protected]
```

#### 3.6.4.3 `silent_`

```
bool LCG_CUDA_Solver::silent_ [protected]
```

The documentation for this class was generated from the following file:

- [solver\\_cuda.h](#)

## 3.7 LCG\_EIGEN\_Solver Class Reference

Linear conjugate gradient solver class.

```
#include <solver_eigen.h>
```

## Public Member Functions

- [LCG\\_EIGEN\\_Solver](#) ()
- virtual [~LCG\\_EIGEN\\_Solver](#) ()
- virtual void [AxProduct](#) (const Eigen::VectorXd &x, Eigen::VectorXd &prod\_Ax)=0  
*Virtual function of the product of  $A*x$ .*
- virtual void [MxProduct](#) (const Eigen::VectorXd &x, Eigen::VectorXd &prod\_Mx)=0  
*Virtual function of the product of  $M^{-1}*x$ .*
- virtual int [Progress](#) (const Eigen::VectorXd \*m, const [lcg\\_float](#) converge, const [lcg\\_para](#) \*param, const int k)  
*Virtual function of the process monitoring.*
- void [silent](#) ()  
*Do not report any processes.*
- void [set\\_report\\_interval](#) (unsigned int inter)  
*Set the interval to run the process monitoring function.*
- void [set\\_lcg\\_parameter](#) (const [lcg\\_para](#) &in\_param)  
*Set the parameters of the algorithms.*
- void [Minimize](#) (Eigen::VectorXd &m, const Eigen::VectorXd &b, [lcg\\_solver\\_enum](#) solver\_id=[LCG\\_CG](#), bool verbose=true, bool er\_throw=false)  
*Run the minimizing process.*
- void [MinimizePreconditioned](#) (Eigen::VectorXd &m, const Eigen::VectorXd &b, [lcg\\_solver\\_enum](#) solver\_id=[LCG\\_PCG](#), bool verbose=true, bool er\_throw=false)  
*Run the preconditioned minimizing process.*
- void [MinimizeConstrained](#) (Eigen::VectorXd &m, const Eigen::VectorXd &B, const Eigen::VectorXd &low, const Eigen::VectorXd &high, [lcg\\_solver\\_enum](#) solver\_id=[LCG\\_PG](#), bool verbose=true, bool er\_throw=false)  
*Run the constrained minimizing process.*

## Static Public Member Functions

- static void [\\_AxProduct](#) (void \*instance, const Eigen::VectorXd &x, Eigen::VectorXd &prod\_Ax)  
*Interface of the virtual function of the product of  $A*x$ .*
- static void [\\_MxProduct](#) (void \*instance, const Eigen::VectorXd &x, Eigen::VectorXd &prod\_Mx)  
*Interface of the virtual function of the product of  $M^{-1}*x$ .*
- static int [\\_Progress](#) (void \*instance, const Eigen::VectorXd \*m, const [lcg\\_float](#) converge, const [lcg\\_para](#) \*param, const int k)  
*Interface of the virtual function of the process monitoring.*

## Protected Attributes

- [lcg\\_para](#) param\_
- unsigned int [inter\\_](#)
- bool [silent\\_](#)

### 3.7.1 Detailed Description

Linear conjugate gradient solver class.

### 3.7.2 Constructor & Destructor Documentation

### 3.7.2.1 LCG\_EIGEN\_Solver()

```
LCG_EIGEN_Solver::LCG_EIGEN_Solver ( )
```

### 3.7.2.2 ~LCG\_EIGEN\_Solver()

```
virtual LCG_EIGEN_Solver::~~LCG_EIGEN_Solver ( ) [inline], [virtual]
```

## 3.7.3 Member Function Documentation

### 3.7.3.1 \_AxProduct()

```
static void LCG_EIGEN_Solver::_AxProduct (
    void * instance,
    const Eigen::VectorXd & x,
    Eigen::VectorXd & prod_Ax ) [inline], [static]
```

Interface of the virtual function of the product of  $A \cdot x$ .

#### Parameters

<i>instance</i>	User data sent to identify the function address
<i>x[in]</i>	Pointer of the multiplier
<i>prod_Ax[out]</i>	Pointer of the product

### 3.7.3.2 \_MxProduct()

```
static void LCG_EIGEN_Solver::_MxProduct (
    void * instance,
    const Eigen::VectorXd & x,
    Eigen::VectorXd & prod_Mx ) [inline], [static]
```

Interface of the virtual function of the product of  $M^{-1} \cdot x$ .

#### Parameters

<i>instance</i>	User data sent to identify the function address
<i>x[in]</i>	Pointer of the multiplier
<i>prod_Mx[out]</i>	Pointer of the product

### 3.7.3.3 \_Progress()

```
static int LCG_EIGEN_Solver::_Progress (
    void * instance,
    const Eigen::VectorXd * m,
    const lcg_float converge,
    const lcg_para * param,
    const int k ) [inline], [static]
```

Interface of the virtual function of the process monitoring.

#### Parameters

<i>instance</i>	User data sent to identify the function address
<i>m</i>	Pointer of the current solution
<i>converge</i>	Current value of the convergence
<i>param</i>	Pointer of the parameters used in the algorithms
<i>k</i>	Current iteration times

#### Returns

int Status of the process

### 3.7.3.4 AxProduct()

```
virtual void LCG_EIGEN_Solver::AxProduct (
    const Eigen::VectorXd & x,
    Eigen::VectorXd & prod_Ax ) [pure virtual]
```

Virtual function of the product of  $A \cdot x$ .

#### Parameters

<i>x[in]</i>	Pointer of the multiplier
<i>prod_Ax[out]</i>	Pointer of the product

### 3.7.3.5 Minimize()

```
void LCG_EIGEN_Solver::Minimize (
    Eigen::VectorXd & m,
    const Eigen::VectorXd & b,
    lcg_solver_enum solver_id = LCG_CG,
```

```
bool verbose = true,
bool er_throw = false )
```

Run the minimizing process.

#### Parameters

<i>m</i>	Pointer of the solution vector
<i>b</i>	Pointer of the targeting vector
<i>solver</i> ↔ <i>_id</i>	Solver type
<i>verbose</i>	Report more information of the full process
<i>er_throw</i>	Instead of showing error messages on screen, throw them out using <code>std::exception</code>

### 3.7.3.6 MinimizeConstrained()

```
void LCG_EIGEN_Solver::MinimizeConstrained (
    Eigen::VectorXd & m,
    const Eigen::VectorXd & B,
    const Eigen::VectorXd & low,
    const Eigen::VectorXd & hig,
    lcg_solver_enum solver_id = LCG_PG,
    bool verbose = true,
    bool er_throw = false )
```

Run the constrained minimizing process.

#### Parameters

<i>m</i>	Pointer of the solution vector
<i>b</i>	Pointer of the targeting vector
<i>low</i>	Lower bound of the solution vector
<i>hig</i>	Higher bound of the solution vector
<i>solver</i> ↔ <i>_id</i>	Solver type
<i>verbose</i>	Report more information of the full process
<i>er_throw</i>	Instead of showing error messages on screen, throw them out using <code>std::exception</code>

### 3.7.3.7 MinimizePreconditioned()

```
void LCG_EIGEN_Solver::MinimizePreconditioned (
    Eigen::VectorXd & m,
    const Eigen::VectorXd & b,
    lcg_solver_enum solver_id = LCG_PCG,
    bool verbose = true,
    bool er_throw = false )
```

Run the preconditioned minimizing process.



## Parameters

<i>m</i>	Pointer of the solution vector
<i>b</i>	Pointer of the targeting vector
<i>solver</i> ↔ <i>_id</i>	Solver type
<i>verbose</i>	Report more information of the full process
<i>er_throw</i>	Instead of showing error messages on screen, throw them out using <code>std::exception</code>

## 3.7.3.8 MxProduct()

```
virtual void LCG_EIGEN_Solver::MxProduct (
    const Eigen::VectorXd & x,
    Eigen::VectorXd & prod_Mx ) [pure virtual]
```

Virtual function of the product of  $M^{-1} \cdot x$ .

## Parameters

<i>x[in]</i>	Pointer of the multiplier
<i>prod_Mx[out]</i>	Pointer of the product

## 3.7.3.9 Progress()

```
int LCG_EIGEN_Solver::Progress (
    const Eigen::VectorXd * m,
    const lcg_float converge,
    const lcg_para * param,
    const int k ) [virtual]
```

Virtual function of the process monitoring.

## Parameters

<i>m</i>	Pointer of the current solution
<i>converge</i>	Current value of the convergence
<i>param</i>	Pointer of the parameters used in the algorithms
<i>k</i>	Current iteration times

## Returns

int Status of the process

### 3.7.3.10 set\_lcg\_parameter()

```
void LCG_EIGEN_Solver::set_lcg_parameter (
    const lcg\_para & in_param )
```

Set the parameters of the algorithms.

#### Parameters

<i>in_param</i>	the input parameters
-----------------	----------------------

### 3.7.3.11 set\_report\_interval()

```
void LCG_EIGEN_Solver::set_report_interval (
    unsigned int inter )
```

Set the interval to run the process monitoring function.

#### Parameters

<i>inter</i>	the interval
--------------	--------------

### 3.7.3.12 silent()

```
void LCG_EIGEN_Solver::silent ( )
```

Do not report any processes.

## 3.7.4 Field Documentation

### 3.7.4.1 inter\_

```
unsigned int LCG_EIGEN_Solver::inter_ [protected]
```

### 3.7.4.2 param\_

```
lcg\_para LCG_EIGEN_Solver::param_ [protected]
```

### 3.7.4.3 silent\_

```
bool LCG_EIGEN_Solver::silent_ [protected]
```

The documentation for this class was generated from the following files:

- [solver\\_eigen.h](#)
- [solver\\_eigen.cpp](#)

## 3.8 lcg\_para Struct Reference

Parameters of the conjugate gradient methods.

```
#include <util.h>
```

### Data Fields

- int [max\\_iterations](#)
- [lcg\\_float](#) [epsilon](#)
- int [abs\\_diff](#)
- [lcg\\_float](#) [restart\\_epsilon](#)
- [lcg\\_float](#) [step](#)
- [lcg\\_float](#) [sigma](#)
- [lcg\\_float](#) [beta](#)
- int [maxi\\_m](#)

### 3.8.1 Detailed Description

Parameters of the conjugate gradient methods.

### 3.8.2 Field Documentation

#### 3.8.2.1 abs\_diff

```
int lcg_para::abs_diff
```

Whether to use absolute mean differences (AMD) between  $|Ax - B|$  to evaluate the process. The default value is false which means the gradient based evaluating method is used. The AMD based method will be used if this variable is set to true. This parameter is only applied to the non-constrained methods.

#### 3.8.2.2 beta

```
lcg\_float lcg_para::beta
```

descending ratio for conducting the non-monotonic linear search. The range of this variable is (0, 1). The default is given as 0.9

### 3.8.2.3 epsilon

```
lcg_float lcg_para::epsilon
```

Epsilon for convergence test. This parameter determines the accuracy with which the solution is to be found. A minimization terminates when  $\|g\|/\max(\|x\|, 1.0) \leq \text{epsilon}$  or  $\sqrt{\|g\|}/N \leq \text{epsilon}$  for the `lcg_solver()` function, where  $\|\cdot\|$  denotes the Euclidean (L2) norm. The default value of epsilon is 1e-6.

### 3.8.2.4 max\_iterations

```
int lcg_para::max_iterations
```

Maximal iteration times. The process will continue till the convergence is met if this option is set to zero (default).

### 3.8.2.5 maxi\_m

```
int lcg_para::maxi_m
```

The maximal record times of the objective values for the SPG method. The method use the objective values from the most recent maxi\_m times to preform the non-monotonic linear search. The default value is 10.

### 3.8.2.6 restart\_epsilon

```
lcg_float lcg_para::restart_epsilon
```

Restart epsilon for the LCG\_BICGSTAB2 algorithm. The default value is 1e-6

### 3.8.2.7 sigma

```
lcg_float lcg_para::sigma
```

multiplier for updating solutions with the spectral projected gradient method. The range of this variable is (0, 1). The default is given as 0.95

### 3.8.2.8 step

```
lcg_float lcg_para::step
```

Initial step length for the project gradient method. The default is 1.0

The documentation for this struct was generated from the following file:

- [util.h](#)

## 3.9 LCG\_Solver Class Reference

Linear conjugate gradient solver class.

```
#include <solver.h>
```

### Public Member Functions

- [LCG\\_Solver](#) ()
- virtual [~LCG\\_Solver](#) ()
- virtual void [AxProduct](#) (const [lcg\\_float](#) \*a, [lcg\\_float](#) \*b, const int num)=0  
*Virtual function of the product of  $A*x$ .*
- virtual void [MxProduct](#) (const [lcg\\_float](#) \*a, [lcg\\_float](#) \*b, const int num)=0  
*Virtual function of the product of  $M^{-1}*x$ .*
- virtual int [Progress](#) (const [lcg\\_float](#) \*m, const [lcg\\_float](#) converge, const [lcg\\_para](#) \*param, const int n\_size, const int k)  
*Virtual function of the process monitoring.*
- void [silent](#) ()  
*Do not report any processes.*
- void [set\\_report\\_interval](#) (unsigned int inter)  
*Set the interval to run the process monitoring function.*
- void [set\\_lcg\\_parameter](#) (const [lcg\\_para](#) &in\_param)  
*Set the parameters of the algorithms.*
- void [Minimize](#) ([lcg\\_float](#) \*m, const [lcg\\_float](#) \*b, int x\_size, [lcg\\_solver\\_enum](#) solver\_id=[LCG\\_CG](#), bool verbose=true, bool er\_throw=false)  
*Run the minimizing process.*
- void [MinimizePreconditioned](#) ([lcg\\_float](#) \*m, const [lcg\\_float](#) \*b, int x\_size, [lcg\\_solver\\_enum](#) solver\_id=[LCG\\_PCG](#), bool verbose=true, bool er\_throw=false)  
*Run the preconditioned minimizing process.*
- void [MinimizeConstrained](#) ([lcg\\_float](#) \*m, const [lcg\\_float](#) \*b, const [lcg\\_float](#) \*low, const [lcg\\_float](#) \*hig, int x\_size, [lcg\\_solver\\_enum](#) solver\_id=[LCG\\_PG](#), bool verbose=true, bool er\_throw=false)  
*Run the constrained minimizing process.*

### Static Public Member Functions

- static void [\\_AxProduct](#) (void \*instance, const [lcg\\_float](#) \*a, [lcg\\_float](#) \*b, const int num)  
*Interface of the virtual function of the product of  $A*x$ .*
- static void [\\_MxProduct](#) (void \*instance, const [lcg\\_float](#) \*a, [lcg\\_float](#) \*b, const int num)  
*Interface of the virtual function of the product of  $M^{-1}*x$ .*
- static int [\\_Progress](#) (void \*instance, const [lcg\\_float](#) \*m, const [lcg\\_float](#) converge, const [lcg\\_para](#) \*param, const int n\_size, const int k)  
*Interface of the virtual function of the process monitoring.*

### Protected Attributes

- [lcg\\_para](#) param\_
- unsigned int inter\_
- bool silent\_

### 3.9.1 Detailed Description

Linear conjugate gradient solver class.

### 3.9.2 Constructor & Destructor Documentation

#### 3.9.2.1 LCG\_Solver()

```
LCG_Solver::LCG_Solver ( )
```

#### 3.9.2.2 ~LCG\_Solver()

```
virtual LCG_Solver::~~LCG_Solver ( ) [inline], [virtual]
```

### 3.9.3 Member Function Documentation

#### 3.9.3.1 \_AxProduct()

```
static void LCG_Solver::_AxProduct (
    void * instance,
    const lcg_float * a,
    lcg_float * b,
    const int num ) [inline], [static]
```

Interface of the virtual function of the product of  $A \cdot x$ .

##### Parameters

<i>instance</i>	User data sent to identify the function address
<i>a[in]</i>	Pointer of the multiplier
<i>b[out]</i>	Pointer of the product
<i>num</i>	Size of the array

#### 3.9.3.2 \_MxProduct()

```
static void LCG_Solver::_MxProduct (
    void * instance,
```

```
const lcg_float * a,
lcg_float * b,
const int num ) [inline], [static]
```

Interface of the virtual function of the product of  $M^{-1} * x$ .

#### Parameters

<i>instance</i>	User data sent to identify the function address
<i>a[in]</i>	Pointer of the multiplier
<i>b[out]</i>	Pointer of the product
<i>num</i>	Size of the array

### 3.9.3.3 \_Progress()

```
static int LCG_Solver::_Progress (
    void * instance,
    const lcg_float * m,
    const lcg_float converge,
    const lcg_para * param,
    const int n_size,
    const int k ) [inline], [static]
```

Interface of the virtual function of the process monitoring.

#### Parameters

<i>instance</i>	User data sent to identify the function address
<i>m</i>	Pointer of the current solution
<i>converge</i>	Current value of the convergence
<i>param</i>	Pointer of the parameters used in the algorithms
<i>n_size</i>	Size of the solution
<i>k</i>	Current iteration times

#### Returns

int Status of the process

### 3.9.3.4 AxProduct()

```
virtual void LCG_Solver::AxProduct (
    const lcg_float * a,
    lcg_float * b,
    const int num ) [pure virtual]
```

Virtual function of the product of  $A * x$ .

## Parameters

<i>a[in]</i>	Pointer of the multiplier
<i>b[out]</i>	Pointer of the product
<i>num</i>	Size of the array

**3.9.3.5 Minimize()**

```
void LCG_Solver::Minimize (
    lcg_float * m,
    const lcg_float * b,
    int x_size,
    lcg_solver_enum solver_id = LCG_CG,
    bool verbose = true,
    bool er_throw = false )
```

Run the minimizing process.

## Parameters

<i>m</i>	Pointer of the solution vector
<i>b</i>	Pointer of the targeting vector
<i>x_size</i>	Size of the solution vector
<i>solver_id</i>	Solver type
<i>verbose</i>	Report more information of the full process
<i>er_throw</i>	Instead of showing error messages on screen, throw them out using std::exception

**3.9.3.6 MinimizeConstrained()**

```
void LCG_Solver::MinimizeConstrained (
    lcg_float * m,
    const lcg_float * b,
    const lcg_float * low,
    const lcg_float * hig,
    int x_size,
    lcg_solver_enum solver_id = LCG_PG,
    bool verbose = true,
    bool er_throw = false )
```

Run the constrained minimizing process.

## Parameters

<i>m</i>	Pointer of the solution vector
<i>b</i>	Pointer of the targeting vector



## Parameters

<i>low</i>	Lower bound of the solution vector
<i>hig</i>	Higher bound of the solution vector
<i>x_size</i>	Size of the solution vector
<i>solver</i> ↔ <i>_id</i>	Solver type
<i>verbose</i>	Report more information of the full process
<i>er_throw</i>	Instead of showing error messages on screen, throw them out using <code>std::exception</code>

**3.9.3.7 MinimizePreconditioned()**

```
void LCG_Solver::MinimizePreconditioned (
    lcg_float * m,
    const lcg_float * b,
    int x_size,
    lcg_solver_enum solver_id = LCG_PCG,
    bool verbose = true,
    bool er_throw = false )
```

Run the preconditioned minimizing process.

## Parameters

<i>m</i>	Pointer of the solution vector
<i>b</i>	Pointer of the targeting vector
<i>x_size</i>	Size of the solution vector
<i>solver</i> ↔ <i>_id</i>	Solver type
<i>verbose</i>	Report more information of the full process
<i>er_throw</i>	Instead of showing error messages on screen, throw them out using <code>std::exception</code>

**3.9.3.8 MxProduct()**

```
virtual void LCG_Solver::MxProduct (
    const lcg_float * a,
    lcg_float * b,
    const int num ) [pure virtual]
```

Virtual function of the product of  $M^{-1} \cdot x$ .

## Parameters

<i>a[in]</i>	Pointer of the multiplier
<i>b[out]</i>	Pointer of the product
<i>num</i>	Size of the array

### 3.9.3.9 Progress()

```
int LCG_Solver::Progress (
    const lcg_float * m,
    const lcg_float converge,
    const lcg_para * param,
    const int n_size,
    const int k ) [virtual]
```

Virtual function of the process monitoring.

#### Parameters

<i>m</i>	Pointer of the current solution
<i>converge</i>	Current value of the convergence
<i>param</i>	Pointer of the parameters used in the algorithms
<i>n_size</i>	Size of the solution
<i>k</i>	Current iteration times

#### Returns

int Status of the process

### 3.9.3.10 set\_lcg\_parameter()

```
void LCG_Solver::set_lcg_parameter (
    const lcg_para & in_param )
```

Set the parameters of the algorithms.

#### Parameters

<i>in_param</i>	the input parameters
-----------------	----------------------

### 3.9.3.11 set\_report\_interval()

```
void LCG_Solver::set_report_interval (
    unsigned int inter )
```

Set the interval to run the process monitoring function.

## Parameters

<i>inter</i>	the interval
--------------	--------------

**3.9.3.12 silent()**

```
void LCG_Solver::silent ( )
```

Do not report any processes.

**3.9.4 Field Documentation****3.9.4.1 inter\_**

```
unsigned int LCG_Solver::inter_ [protected]
```

**3.9.4.2 param\_**

```
lcg_para LCG_Solver::param_ [protected]
```

**3.9.4.3 silent\_**

```
bool LCG_Solver::silent_ [protected]
```

The documentation for this class was generated from the following files:

- [solver.h](#)
- [solver.cpp](#)



## Chapter 4

# File Documentation

### 4.1 algebra.cpp File Reference

```
#include "ctime"  
#include "random"  
#include "algebra.h"  
#include "omp.h"
```

#### Functions

- [lcg\\_float lcg\\_abs](#) ([lcg\\_float](#) a)  
*Return absolute value.*
- [lcg\\_float lcg\\_max](#) ([lcg\\_float](#) a, [lcg\\_float](#) b)  
*Return the bigger value.*
- [lcg\\_float lcg\\_min](#) ([lcg\\_float](#) a, [lcg\\_float](#) b)  
*Return the smaller value.*
- [lcg\\_float lcg\\_set2box](#) ([lcg\\_float](#) low, [lcg\\_float](#) hig, [lcg\\_float](#) a, bool low\_bound, bool hig\_bound)  
*Set the input value within a box constraint.*
- [lcg\\_float \\*](#) [lcg\\_malloc](#) (int n)  
*Locate memory for a lcg\_float pointer type.*
- [lcg\\_float \\*\\*](#) [lcg\\_malloc](#) (int m, int n)  
*Locate memory for a lcg\_float second pointer type.*
- void [lcg\\_free](#) ([lcg\\_float](#) \*x)  
*Destroy memory used by the lcg\_float type array.*
- void [lcg\\_free](#) ([lcg\\_float](#) \*\*x, int m)  
*Destroy memory used by the 2D lcg\_float type array.*
- void [lcg\\_vecset](#) ([lcg\\_float](#) \*a, [lcg\\_float](#) b, int size)  
*set a vector's value*
- void [lcg\\_vecset](#) ([lcg\\_float](#) \*\*a, [lcg\\_float](#) b, int m, int n)  
*set a 2d vector's value*
- void [lcg\\_vecrnd](#) ([lcg\\_float](#) \*a, [lcg\\_float](#) l, [lcg\\_float](#) h, int size)  
*set a vector using random values*
- void [lcg\\_vecrnd](#) ([lcg\\_float](#) \*\*a, [lcg\\_float](#) l, [lcg\\_float](#) h, int m, int n)  
*set a 2D vector using random values*

- double `lcg_squaredl2norm` (`lcg_float` \*a, int n)  
*calculate the squared L2 norm of the input vector*
- void `lcg_dot` (`lcg_float` &ret, const `lcg_float` \*a, const `lcg_float` \*b, int size)  
*calculate dot product of two real vectors*
- void `lcg_matvec` (`lcg_float` \*\*A, const `lcg_float` \*x, `lcg_float` \*Ax, int m\_size, int n\_size, `lcg_matrix_e` layout)  
*calculate product of a real matrix and a vector*
- void `lcg_matvec_coo` (const int \*row, const int \*col, const `lcg_float` \*Mat, const `lcg_float` \*V, `lcg_float` \*p, int M, int N, int nz\_size, bool pre\_position)  
*Calculate the product of a sparse matrix multiplied by a vector. The matrix is stored in the COO format.*

## 4.1.1 Function Documentation

### 4.1.1.1 `lcg_abs()`

```
lcg_float lcg_abs (
    lcg_float a )
```

Return absolute value.

#### Parameters

in	<i>a</i>	input value
----	----------	-------------

#### Returns

The absolute value

### 4.1.1.2 `lcg_dot()`

```
void lcg_dot (
    lcg_float & ret,
    const lcg_float * a,
    const lcg_float * b,
    int size )
```

calculate dot product of two real vectors

#### Parameters

in	<i>a</i>	pointer of the vector a
in	<i>b</i>	pointer of the vector b
in	<i>size</i>	size of the vector

**Returns**

dot product

**4.1.1.3 lcg\_free() [1/2]**

```
void lcg_free (
    lcg_float ** x,
    int m )
```

Destroy memory used by the 2D lcg\_float type array.

**Parameters**

x	Pointer of the array.
---	-----------------------

**4.1.1.4 lcg\_free() [2/2]**

```
void lcg_free (
    lcg_float * x )
```

Destroy memory used by the lcg\_float type array.

**Parameters**

x	Pointer of the array.
---	-----------------------

**4.1.1.5 lcg\_malloc() [1/2]**

```
lcg_float** lcg_malloc (
    int m,
    int n )
```

Locate memory for a lcg\_float second pointer type.

**Parameters**

in	n	Size of the lcg_float array.
----	---	------------------------------

**Returns**

Pointer of the array's location.

#### 4.1.1.6 lcg\_malloc() [2/2]

```
lcg_float* lcg_malloc (
    int n )
```

Locate memory for a lcg\_float pointer type.

##### Parameters

in	<i>n</i>	Size of the lcg_float array.
----	----------	------------------------------

##### Returns

Pointer of the array's location.

#### 4.1.1.7 lcg\_matvec()

```
void lcg_matvec (
    lcg_float ** A,
    const lcg_float * x,
    lcg_float * Ax,
    int m_size,
    int n_size,
    lcg_matrix_e layout = MatNormal )
```

calculate product of a real matrix and a vector

Different configurations: layout=Normal -> A layout=Transpose -> A<sup>T</sup>

##### Parameters

	<i>A</i>	matrix A
in	<i>x</i>	vector x
	<i>Ax</i>	product of Ax
in	<i>m_size</i>	row size of A
in	<i>n_size</i>	column size of A
in	<i>layout</i>	layout of A used for multiplication. Must be Normal or Transpose

#### 4.1.1.8 lcg\_matvec\_coo()

```
void lcg_matvec_coo (
    const int * row,
```



```

const int * col,
const lcg_float * Mat,
const lcg_float * V,
lcg_float * p,
int M,
int N,
int nz_size,
bool pre_position = false )

```

Calculate the product of a sparse matrix multiplied by a vector. The matrix is stored in the COO format.

#### Parameters

<i>row</i>	Row index of the input sparse matrix.
<i>col</i>	Column index of the input sparse matrix.
<i>Mat</i>	Non-zero values of the input sparse matrix.
<i>V</i>	Multiplier vector
<i>p</i>	Output product
<i>M</i>	Row number of the sparse matrix
<i>N</i>	Column number of the sparse matrix
<i>nz_size</i>	Non-zero size of the matrix
<i>pre_position</i>	If true, the multiplier is seen as a row vector. Otherwise, it is treated as a column vector.

#### 4.1.1.9 lcg\_max()

```

lcg_float lcg_max (
    lcg_float a,
    lcg_float b )

```

Return the bigger value.

#### Parameters

in	<i>a</i>	input value
in	<i>b</i>	input value

#### Returns

The bigger value

#### 4.1.1.10 lcg\_min()

```

lcg_float lcg_min (
    lcg_float a,
    lcg_float b )

```

Return the smaller value.

**Parameters**

<i>in</i>	<i>a</i>	input value
<i>in</i>	<i>b</i>	input value

**Returns**

The smaller value

**4.1.1.11 lcg\_set2box()**

```
lcg_float lcg_set2box (
    lcg_float low,
    lcg_float hig,
    lcg_float a,
    bool low_bound = true,
    bool hig_bound = true )
```

Set the input value within a box constraint.

**Parameters**

<i>a</i>	low boundary
<i>b</i>	high boundary
<i>in</i>	input value
<i>low_bound</i>	Whether to include the low boundary value
<i>hig_bound</i>	Whether to include the high boundary value

**Returns**

box constrained value

**4.1.1.12 lcg\_squaredl2norm()**

```
double lcg_squaredl2norm (
    lcg_float * a,
    int n )
```

calculate the squared L2 norm of the input vector

**Parameters**

<i>a</i>	pointer of the vector
<i>n</i>	size of the vector

**Returns**

double L2 norm

**4.1.1.13 lcg\_vecrnd() [1/2]**

```
void lcg_vecrnd (
    lcg_float ** a,
    lcg_float l,
    lcg_float h,
    int m,
    int n )
```

set a 2D vector using random values

**Parameters**

	<i>a</i>	pointer of the vector
in	<i>l</i>	the lower bound of random values
in	<i>h</i>	the higher bound of random values
in	<i>m</i>	row size of the vector
in	<i>n</i>	column size of the vector

**4.1.1.14 lcg\_vecrnd() [2/2]**

```
void lcg_vecrnd (
    lcg_float * a,
    lcg_float l,
    lcg_float h,
    int size )
```

set a vector using random values

**Parameters**

	<i>a</i>	pointer of the vector
in	<i>l</i>	the lower bound of random values
in	<i>h</i>	the higher bound of random values
in	<i>size</i>	size of the vector

**4.1.1.15 lcg\_vecset() [1/2]**

```
void lcg_vecset (
    lcg_float ** a,
```

```
lcg_float b,
int m,
int n )
```

set a 2d vector's value

#### Parameters

	<i>a</i>	pointer of the matrix
in	<i>b</i>	initial value
in	<i>m</i>	row size of the matrix
in	<i>n</i>	column size of the matrix

#### 4.1.1.16 lcg\_vecset() [2/2]

```
void lcg_vecset (
    lcg_float * a,
    lcg_float b,
    int size )
```

set a vector's value

#### Parameters

	<i>a</i>	pointer of the vector
in	<i>b</i>	initial value
in	<i>size</i>	vector size

## 4.2 algebra.h File Reference

```
#include "config.h"
```

### Typedefs

- typedef double [lcg\\_float](#)

*A simple definition of the float type we use here. Easy to change in the future. Right now it is just an alias of double.*

### Enumerations

- enum [lcg\\_matrix\\_e](#) { [MatNormal](#), [MatTranspose](#) }  
*Matrix layouts.*
- enum [clcg\\_complex\\_e](#) { [NonConjugate](#), [Conjugate](#) }  
*Conjugate types for a complex number.*

## Functions

- `lcg_float` `lcg_abs` (`lcg_float` a)  
*Return absolute value.*
- `lcg_float` `lcg_max` (`lcg_float` a, `lcg_float` b)  
*Return the bigger value.*
- `lcg_float` `lcg_min` (`lcg_float` a, `lcg_float` b)  
*Return the smaller value.*
- `lcg_float` `lcg_set2box` (`lcg_float` low, `lcg_float` hig, `lcg_float` a, bool low\_bound=true, bool hig\_bound=true)  
*Set the input value within a box constraint.*
- `lcg_float *` `lcg_malloc` (int n)  
*Locate memory for a `lcg_float` pointer type.*
- `lcg_float **` `lcg_malloc` (int m, int n)  
*Locate memory for a `lcg_float` second pointer type.*
- void `lcg_free` (`lcg_float *`x)  
*Destroy memory used by the `lcg_float` type array.*
- void `lcg_free` (`lcg_float **`x, int m)  
*Destroy memory used by the 2D `lcg_float` type array.*
- void `lcg_vecset` (`lcg_float *`a, `lcg_float` b, int size)  
*set a vector's value*
- void `lcg_vecset` (`lcg_float **`a, `lcg_float` b, int m, int n)  
*set a 2d vector's value*
- void `lcg_vecrnd` (`lcg_float *`a, `lcg_float` l, `lcg_float` h, int size)  
*set a vector using random values*
- void `lcg_vecrnd` (`lcg_float **`a, `lcg_float` l, `lcg_float` h, int m, int n)  
*set a 2D vector using random values*
- double `lcg_squaredl2norm` (`lcg_float *`a, int n)  
*calculate the squared L2 norm of the input vector*
- void `lcg_dot` (`lcg_float` &ret, const `lcg_float *`a, const `lcg_float *`b, int size)  
*calculate dot product of two real vectors*
- void `lcg_matvec` (`lcg_float **`A, const `lcg_float *`x, `lcg_float *`Ax, int m\_size, int n\_size, `lcg_matrix_e` layout=MatNormal)  
*calculate product of a real matrix and a vector*
- void `lcg_matvec_coo` (const int \*row, const int \*col, const `lcg_float *`Mat, const `lcg_float *`V, `lcg_float *`p, int M, int N, int nz\_size, bool pre\_position=false)  
*Calculate the product of a sparse matrix multiplied by a vector. The matrix is stored in the COO format.*

### 4.2.1 Typedef Documentation

#### 4.2.1.1 `lcg_float`

```
typedef double lcg_float
```

A simple definition of the float type we use here. Easy to change in the future. Right now it is just an alias of double.

## 4.2.2 Enumeration Type Documentation

### 4.2.2.1 clcg\_complex\_e

enum `clcg_complex_e`

Conjugate types for a complex number.

Enumerator

NonConjugate	
Conjugate	

### 4.2.2.2 lcg\_matrix\_e

enum `lcg_matrix_e`

Matrix layouts.

Enumerator

MatNormal	
MatTranspose	

## 4.2.3 Function Documentation

### 4.2.3.1 lcg\_abs()

```
lcg_float lcg_abs (  
    lcg_float a )
```

Return absolute value.

Parameters

in	<i>a</i>	input value
----	----------	-------------

**Returns**

The absolute value

**4.2.3.2 lcg\_dot()**

```
void lcg_dot (
    lcg_float & ret,
    const lcg_float * a,
    const lcg_float * b,
    int size )
```

calculate dot product of two real vectors

**Parameters**

in	<i>a</i>	pointer of the vector a
in	<i>b</i>	pointer of the vector b
in	<i>size</i>	size of the vector

**Returns**

dot product

**4.2.3.3 lcg\_free() [1/2]**

```
void lcg_free (
    lcg_float ** x,
    int m )
```

Destroy memory used by the 2D lcg\_float type array.

**Parameters**

<i>x</i>	Pointer of the array.
----------	-----------------------

**4.2.3.4 lcg\_free() [2/2]**

```
void lcg_free (
    lcg_float * x )
```

Destroy memory used by the lcg\_float type array.

**Parameters**

$x$	Pointer of the array.
-----	-----------------------

**4.2.3.5 lcg\_malloc() [1/2]**

```
lcg_float** lcg_malloc (
    int m,
    int n )
```

Locate memory for a lcg\_float second pointer type.

**Parameters**

in	$n$	Size of the lcg_float array.
----	-----	------------------------------

**Returns**

Pointer of the array's location.

**4.2.3.6 lcg\_malloc() [2/2]**

```
lcg_float* lcg_malloc (
    int n )
```

Locate memory for a lcg\_float pointer type.

**Parameters**

in	$n$	Size of the lcg_float array.
----	-----	------------------------------

**Returns**

Pointer of the array's location.

**4.2.3.7 lcg\_matvec()**

```
void lcg_matvec (
    lcg_float ** A,
    const lcg_float * x,
    lcg_float * Ax,
```



```

    int m_size,
    int n_size,
    lcg_matrix_e layout = MatNormal )

```

calculate product of a real matrix and a vector

Different configurations: layout=Normal -> A layout=Transpose ->  $A^T$

#### Parameters

	<i>A</i>	matrix A
in	<i>x</i>	vector x
	<i>Ax</i>	product of Ax
in	<i>m_size</i>	row size of A
in	<i>n_size</i>	column size of A
in	<i>layout</i>	layout of A used for multiplication. Must be Normal or Transpose

#### 4.2.3.8 lcg\_matvec\_coo()

```

void lcg_matvec_coo (
    const int * row,
    const int * col,
    const lcg_float * Mat,
    const lcg_float * V,
    lcg_float * p,
    int M,
    int N,
    int nz_size,
    bool pre_position = false )

```

Calculate the product of a sparse matrix multiplied by a vector. The matrix is stored in the COO format.

#### Parameters

<i>row</i>	Row index of the input sparse matrix.
<i>col</i>	Column index of the input sparse matrix.
<i>Mat</i>	Non-zero values of the input sparse matrix.
<i>V</i>	Multiplier vector
<i>p</i>	Output product
<i>M</i>	Row number of the sparse matrix
<i>N</i>	Column number of the sparse matrix
<i>nz_size</i>	Non-zero size of the matrix
<i>pre_position</i>	If true, the multiplier is seen as a row vector. Otherwise, it is treated as a column vector.

#### 4.2.3.9 lcg\_max()

```

lcg_float lcg_max (

```

```
lcg_float a,
lcg_float b )
```

Return the bigger value.

#### Parameters

in	<i>a</i>	input value
in	<i>b</i>	input value

#### Returns

The bigger value

#### 4.2.3.10 lcg\_min()

```
lcg_float lcg_min (
    lcg_float a,
    lcg_float b )
```

Return the smaller value.

#### Parameters

in	<i>a</i>	input value
in	<i>b</i>	input value

#### Returns

The smaller value

#### 4.2.3.11 lcg\_set2box()

```
lcg_float lcg_set2box (
    lcg_float low,
    lcg_float hig,
    lcg_float a,
    bool low_bound = true,
    bool hig_bound = true )
```

Set the input value within a box constraint.

#### Parameters

<i>a</i>	low boundary
<i>b</i>	high boundary
<i>in</i>	input value
<i>low_bound</i>	Whether to include the low boundary value
<i>hig_bound</i>	Whether to include the high boundary value

**Returns**

box constrained value

**4.2.3.12 lcg\_squaredl2norm()**

```
double lcg_squaredl2norm (
    lcg_float * a,
    int n )
```

calculate the squared L2 norm of the input vector

**Parameters**

<i>a</i>	pointer of the vector
<i>n</i>	size of the vector

**Returns**

double L2 norm

**4.2.3.13 lcg\_vecrnd() [1/2]**

```
void lcg_vecrnd (
    lcg_float ** a,
    lcg_float l,
    lcg_float h,
    int m,
    int n )
```

set a 2D vector using random values

**Parameters**

	<i>a</i>	pointer of the vector
in	<i>l</i>	the lower bound of random values
in	<i>h</i>	the higher bound of random values
in	<i>m</i>	row size of the vector
in	<i>n</i>	column size of the vector

**4.2.3.14 lcg\_vecrnd() [2/2]**

```
void lcg_vecrnd (
    lcg_float * a,
```

```

    lcg_float l,
    lcg_float h,
    int size )

```

set a vector using random values

#### Parameters

	<i>a</i>	pointer of the vector
in	<i>l</i>	the lower bound of random values
in	<i>h</i>	the higher bound of random values
in	<i>size</i>	size of the vector

#### 4.2.3.15 lcg\_vecset() [1/2]

```

void lcg_vecset (
    lcg_float ** a,
    lcg_float b,
    int m,
    int n )

```

set a 2d vector's value

#### Parameters

	<i>a</i>	pointer of the matrix
in	<i>b</i>	initial value
in	<i>m</i>	row size of the matrix
in	<i>n</i>	column size of the matrix

#### 4.2.3.16 lcg\_vecset() [2/2]

```

void lcg_vecset (
    lcg_float * a,
    lcg_float b,
    int size )

```

set a vector's value

#### Parameters

	<i>a</i>	pointer of the vector
in	<i>b</i>	initial value
in	<i>size</i>	vector size

## 4.3 algebra\_cuda.h File Reference

```
#include "algebra.h"
#include <cuda_runtime.h>
```

### Functions

- void `lcg_set2box_cuda` (const `lcg_float` \*low, const `lcg_float` \*hig, `lcg_float` \*a, int n, bool low\_bound=true, bool hig\_bound=true)  
*Set the input value within a box constraint.*
- void `lcg_smDcsr_get_diagonal` (const int \*A\_ptr, const int \*A\_col, const `lcg_float` \*A\_val, const int A\_len, `lcg_float` \*A\_diag, int bk\_size=1024)  
*Extract diagonal elements from a square CUDA sparse matrix that is formatted in the CSR format.*
- void `lcg_vecMvecD_element_wise` (const `lcg_float` \*a, const `lcg_float` \*b, `lcg_float` \*c, int n, int bk\_size=1024)  
*Element-wise muplication between two CUDA arries.*
- void `lcg_vecDvecD_element_wise` (const `lcg_float` \*a, const `lcg_float` \*b, `lcg_float` \*c, int n, int bk\_size=1024)  
*Element-wise division between two CUDA arries.*

### 4.3.1 Function Documentation

#### 4.3.1.1 `lcg_set2box_cuda()`

```
void lcg_set2box_cuda (
    const lcg_float * low,
    const lcg_float * hig,
    lcg_float * a,
    int n,
    bool low_bound = true,
    bool hig_bound = true )
```

Set the input value within a box constraint.

#### Parameters

<i>a</i>	low boundary
<i>b</i>	high boundary
<i>in</i>	input value
<i>low_bound</i>	Whether to include the low boundary value
<i>hig_bound</i>	Whether to include the high boundary value

#### Returns

box constrained value

#### 4.3.1.2 lcg\_smDcsr\_get\_diagonal()

```
void lcg_smDcsr_get_diagonal (
    const int * A_ptr,
    const int * A_col,
    const lcg_float * A_val,
    const int A_len,
    lcg_float * A_diag,
    int bk_size = 1024 )
```

Extract diagonal elements from a square CUDA sparse matrix that is formatted in the CSR format.

##### Note

This is a device side function. All memories must be allocated on the GPU device.

##### Parameters

in	<i>A_ptr</i>	Row index pointer
in	<i>A_col</i>	Column index
in	<i>A_val</i>	Non-zero values of the matrix
in	<i>A_len</i>	Dimension of the matrix
	<i>A_diag</i>	Output digonal elements
in	<i>bk_size</i>	Default CUDA block size.

#### 4.3.1.3 lcg\_vecDvecD\_element\_wise()

```
void lcg_vecDvecD_element_wise (
    const lcg_float * a,
    const lcg_float * b,
    lcg_float * c,
    int n,
    int bk_size = 1024 )
```

Element-wise division between two CUDA arries.

##### Note

This is a device side function. All memories must be allocated on the GPU device.

##### Parameters

in	<i>a</i>	Pointer of the input array
in	<i>b</i>	Pointer of the input array
	<i>c</i>	Pointer of the output array
in	<i>n</i>	Length of the arraies
in	<i>bk_size</i>	Default CUDA block size.

#### 4.3.1.4 lcg\_vecMvecD\_element\_wise()

```
void lcg_vecMvecD_element_wise (
    const lcg_float * a,
    const lcg_float * b,
    lcg_float * c,
    int n,
    int bk_size = 1024 )
```

Element-wise muplication between two CUDA arries.

##### Note

This is a device side function. All memories must be allocated on the GPU device.

##### Parameters

in	<i>a</i>	Pointer of the input array
in	<i>b</i>	Pointer of the input array
	<i>c</i>	Pointer of the output array
in	<i>n</i>	Length of the arraies
in	<i>bk_size</i>	Default CUDA block size.

## 4.4 algebra\_eigen.cpp File Reference

```
#include "algebra_eigen.h"
```

### Functions

- void [lcg\\_set2box\\_eigen](#) (const Eigen::VectorXd &low, const Eigen::VectorXd &hig, Eigen::VectorXd m)  
*Set the input value within a box constraint.*

#### 4.4.1 Function Documentation

##### 4.4.1.1 lcg\_set2box\_eigen()

```
void lcg_set2box_eigen (
    const Eigen::VectorXd & low,
    const Eigen::VectorXd & hig,
    Eigen::VectorXd m )
```

Set the input value within a box constraint.

## Parameters

<i>low_bound</i>	Whether to include the low boundary value
<i>hig_bound</i>	Whether to include the high boundary value
<i>m</i>	Returned values

## 4.5 algebra\_eigen.h File Reference

```
#include "algebra.h"
#include "Eigen/Dense"
```

### Functions

- void [lcg\\_set2box\\_eigen](#) (const Eigen::VectorXd &low, const Eigen::VectorXd &hig, Eigen::VectorXd m)  
*Set the input value within a box constraint.*

### 4.5.1 Function Documentation

#### 4.5.1.1 lcg\_set2box\_eigen()

```
void lcg_set2box_eigen (
    const Eigen::VectorXd & low,
    const Eigen::VectorXd & hig,
    Eigen::VectorXd m )
```

Set the input value within a box constraint.

## Parameters

<i>low_bound</i>	Whether to include the low boundary value
<i>hig_bound</i>	Whether to include the high boundary value
<i>m</i>	Returned values

## 4.6 clcg.cpp File Reference

```
#include "clcg.h"
#include "cmath"
#include "config.h"
#include "omp.h"
```



## Typedefs

- typedef int(\* [clcg\\_solver\\_ptr](#)) ([clcg\\_axfunc\\_ptr](#) Afp, [clcg\\_progress\\_ptr](#) Pfp, [lcg\\_complex](#) \*m, const [lcg\\_complex](#) \*B, const int n\_size, const [clcg\\_para](#) \*param, void \*instance)

## Functions

- int [clbicg](#) ([clcg\\_axfunc\\_ptr](#) Afp, [clcg\\_progress\\_ptr](#) Pfp, [lcg\\_complex](#) \*m, const [lcg\\_complex](#) \*B, const int n\_size, const [clcg\\_para](#) \*param, void \*instance)
- int [clbicg\\_symmetric](#) ([clcg\\_axfunc\\_ptr](#) Afp, [clcg\\_progress\\_ptr](#) Pfp, [lcg\\_complex](#) \*m, const [lcg\\_complex](#) \*B, const int n\_size, const [clcg\\_para](#) \*param, void \*instance)
- int [clcgs](#) ([clcg\\_axfunc\\_ptr](#) Afp, [clcg\\_progress\\_ptr](#) Pfp, [lcg\\_complex](#) \*m, const [lcg\\_complex](#) \*B, const int n\_size, const [clcg\\_para](#) \*param, void \*instance)
- int [clbicgstab](#) ([clcg\\_axfunc\\_ptr](#) Afp, [clcg\\_progress\\_ptr](#) Pfp, [lcg\\_complex](#) \*m, const [lcg\\_complex](#) \*B, const int n\_size, const [clcg\\_para](#) \*param, void \*instance)
- int [cltfqmr](#) ([clcg\\_axfunc\\_ptr](#) Afp, [clcg\\_progress\\_ptr](#) Pfp, [lcg\\_complex](#) \*m, const [lcg\\_complex](#) \*B, const int n\_size, const [clcg\\_para](#) \*param, void \*instance)
- int [clcg\\_solver](#) ([clcg\\_axfunc\\_ptr](#) Afp, [clcg\\_progress\\_ptr](#) Pfp, [lcg\\_complex](#) \*m, const [lcg\\_complex](#) \*B, const int n\_size, const [clcg\\_para](#) \*param, void \*instance, [clcg\\_solver\\_enum](#) solver\_id)

*A combined complex conjugate gradient solver function.*

### 4.6.1 Typedef Documentation

#### 4.6.1.1 [clcg\\_solver\\_ptr](#)

```
typedef int(* clcg\_solver\_ptr) (clcg\_axfunc\_ptr Afp, clcg\_progress\_ptr Pfp, lcg\_complex *m,
const lcg\_complex *B, const int n_size, const clcg\_para *param, void *instance)
```

### 4.6.2 Function Documentation

#### 4.6.2.1 [clbicg\(\)](#)

```
int clbicg (
    clcg\_axfunc\_ptr Afp,
    clcg\_progress\_ptr Pfp,
    lcg\_complex * m,
    const lcg\_complex * B,
    const int n_size,
    const clcg\_para * param,
    void * instance )
```

#### 4.6.2.2 clbicg\_symmetric()

```
int clbicg_symmetric (
    clcg_axfunc_ptr Afp,
    clcg_progress_ptr Pfp,
    lcg_complex * m,
    const lcg_complex * B,
    const int n_size,
    const clcg_para * param,
    void * instance )
```

#### 4.6.2.3 clbicgstab()

```
int clbicgstab (
    clcg_axfunc_ptr Afp,
    clcg_progress_ptr Pfp,
    lcg_complex * m,
    const lcg_complex * B,
    const int n_size,
    const clcg_para * param,
    void * instance )
```

#### 4.6.2.4 clcg\_solver()

```
int clcg_solver (
    clcg_axfunc_ptr Afp,
    clcg_progress_ptr Pfp,
    lcg_complex * m,
    const lcg_complex * B,
    const int n_size,
    const clcg_para * param,
    void * instance,
    clcg_solver_enum solver_id = CLCG_BICG )
```

A combined complex conjugate gradient solver function.

##### Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.

### Returns

Status of the function.

#### 4.6.2.5 clcgs()

```
int clcgs (
    clcg_axfunc_ptr Afp,
    clcg_progress_ptr Pfp,
    lcg_complex * m,
    const lcg_complex * B,
    const int n_size,
    const clcg_para * param,
    void * instance )
```

#### 4.6.2.6 cltfqmr()

```
int cltfqmr (
    clcg_axfunc_ptr Afp,
    clcg_progress_ptr Pfp,
    lcg_complex * m,
    const lcg_complex * B,
    const int n_size,
    const clcg_para * param,
    void * instance )
```

## 4.7 clcg.h File Reference

```
#include "lcg_complex.h"
#include "util.h"
```

### Typedefs

- typedef void(\* [clcg\\_axfunc\\_ptr](#)) (void \*instance, const [lcg\\_complex](#) \*x, [lcg\\_complex](#) \*prod\_Ax, const int x↔\_size, [lcg\\_matrix\\_e](#) layout, [clcg\\_complex\\_e](#) conjugate)  
*Callback interface for calculating the complex product of a N\*N matrix 'A' multiplied by a complex vertical vector 'x'.*
- typedef int(\* [clcg\\_progress\\_ptr](#)) (void \*instance, const [lcg\\_complex](#) \*m, const [lcg\\_float](#) converge, const [clcg\\_para](#) \*param, const int n\_size, const int k)  
*Callback interface for monitoring the progress and terminate the iteration if necessary.*

### Functions

- int [clcg\\_solver](#) ([clcg\\_axfunc\\_ptr](#) Afp, [clcg\\_progress\\_ptr](#) Pfp, [lcg\\_complex](#) \*m, const [lcg\\_complex](#) \*B, const int n\_size, const [clcg\\_para](#) \*param, void \*instance, [clcg\\_solver\\_enum](#) solver\_id=CLCG\_BICG)  
*A combined complex conjugate gradient solver function.*

## 4.7.1 Typedef Documentation

### 4.7.1.1 `clcg_axfunc_ptr`

```
typedef void(* clcg_axfunc_ptr) (void *instance, const lcg_complex *x, lcg_complex *prod_Ax,
const int x_size, lcg_matrix_e layout, clcg_complex_e conjugate)
```

Callback interface for calculating the complex product of a N\*N matrix 'A' multiplied by a complex vertical vector 'x'.

#### Parameters

<i>instance</i>	The user data sent for the <code>clcg_solver()</code> functions by the client.
<i>x</i>	Multiplier of the Ax product.
<i>Ax</i>	Product of A multiplied by x.
<i>x_size</i>	Size of x and column/row numbers of A.
<i>layout</i>	Whether to use the transpose of A for calculation.
<i>conjugate</i>	Whether to use the conjugate of A for calculation.

### 4.7.1.2 `clcg_progress_ptr`

```
typedef int(* clcg_progress_ptr) (void *instance, const lcg_complex *m, const lcg_float converge,
const clcg_para *param, const int n_size, const int k)
```

Callback interface for monitoring the progress and terminate the iteration if necessary.

#### Parameters

<i>instance</i>	The user data sent for the <code>clcg_solver()</code> functions by the client.
<i>m</i>	The current solutions.
<i>converge</i>	The current value evaluating the iteration progress.
<i>n_size</i>	The size of the variables
<i>k</i>	The iteration count.

#### Return values

<i>int</i>	Zero to continue the optimization process. Returning a non-zero value will terminate the optimization process.
------------	--

## 4.7.2 Function Documentation

## 4.7.2.1 clcg\_solver()

```
int clcg_solver (
    clcg_axfunc_ptr Afp,
    clcg_progress_ptr Pfp,
    lcg_complex * m,
    const lcg_complex * B,
    const int n_size,
    const clcg_para * param,
    void * instance,
    clcg_solver_enum solver_id = CLCG_BICG )
```

A combined complex conjugate gradient solver function.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.

## Returns

Status of the function.

## 4.8 clcg\_cuda.h File Reference

```
#include "util.h"
#include "lcg_complex_cuda.h"
#include <cublas_v2.h>
#include <cusparse_v2.h>
```

## Typedefs

- typedef void(\* [clcg\\_axfunc\\_cuda\\_ptr](#)) (void \*instance, cublasHandle\_t cub\_handle, cusparseHandle\_t cusparse\_handle, cusparseDnVecDescr\_t x, cusparseDnVecDescr\_t prod\_Ax, const int n\_size, const int nz\_size, cusparseOperation\_t oper\_t)  
*Callback interface for calculating the product of a N\*N matrix 'A' multiplied by a vertical vector 'x'. Note that both A and x are hosted on the GPU device.*
- typedef int(\* [clcg\\_progress\\_cuda\\_ptr](#)) (void \*instance, const cuDoubleComplex \*m, const [lcg\\_float](#) converge, const [clcg\\_para](#) \*param, const int n\_size, const int nz\_size, const int k)  
*Callback interface for monitoring the progress and terminate the iteration if necessary. Note that m is hosted on the GPU device.*

## Functions

- int [clcg\\_solver\\_cuda](#) ([clcg\\_axfunc\\_cuda\\_ptr](#) Afp, [clcg\\_progress\\_cuda\\_ptr](#) Pfp, cuDoubleComplex \*m, const cuDoubleComplex \*B, const int n\_size, const int nz\_size, const [clcg\\_para](#) \*param, void \*instance, cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, [clcg\\_solver\\_enum](#) solver\_id=CLCG\_BICG)  
A combined conjugate gradient solver function. Note that both m and B are hosted on the GPU device.
- int [clcg\\_solver\\_preconditioned\\_cuda](#) ([clcg\\_axfunc\\_cuda\\_ptr](#) Afp, [clcg\\_axfunc\\_cuda\\_ptr](#) Mfp, [clcg\\_progress\\_cuda\\_ptr](#) Pfp, cuDoubleComplex \*m, const cuDoubleComplex \*B, const int n\_size, const int nz\_size, const [clcg\\_para](#) \*param, void \*instance, cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, [clcg\\_solver\\_enum](#) solver\_id=CLCG\_PCG)  
A combined conjugate gradient solver function. Note that both m and B are hosted on the GPU device.

## 4.8.1 Typedef Documentation

### 4.8.1.1 clcg\_axfunc\_cuda\_ptr

```
typedef void(* clcg_axfunc_cuda_ptr) (void *instance, cublasHandle_t cub_handle, cusparseHandle_t cus_handle, cusparseDnVecDescr_t x, cusparseDnVecDescr_t prod_Ax, const int n_size, const int nz_size, cusparseOperation_t oper_t)
```

Callback interface for calculating the product of a N\*N matrix 'A' multiplied by a vertical vector 'x'. Note that both A and x are hosted on the GPU device.

#### Parameters

<i>instance</i>	The user data sent for the <a href="#">lcg_solver_cuda()</a> functions by the client.
<i>cub_handle</i>	Handler of the cublas object.
<i>cus_handle</i>	Handlee of the cusparse object.
<i>x</i>	Multiplier of the Ax product.
<i>Ax</i>	Product of A multiplied by x.
<i>n_size</i>	Size of x and column/row numbers of A.

### 4.8.1.2 clcg\_progress\_cuda\_ptr

```
typedef int(* clcg_progress_cuda_ptr) (void *instance, const cuDoubleComplex *m, const lcg\_float converge, const clcg\_para *param, const int n_size, const int nz_size, const int k)
```

Callback interface for monitoring the progress and terminate the iteration if necessary. Note that m is hosted on the GPU device.

#### Parameters

<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> functions by the client.
<i>m</i>	The current solutions.
<i>converge</i>	The current value evaluating the iteration progress.
<i>n_size</i>	The size of the variables
<i>k</i>	The iteration count.

## Return values

<i>int</i>	Zero to continue the optimization process. Returning a non-zero value will terminate the optimization process.
------------	--

## 4.8.2 Function Documentation

## 4.8.2.1 clcg\_solver\_cuda()

```
int clcg_solver_cuda (
    clcg_axfunc_cuda_ptr Afp,
    clcg_progress_cuda_ptr Pfp,
    cuDoubleComplex * m,
    const cuDoubleComplex * B,
    const int n_size,
    const int nz_size,
    const clcg_para * param,
    void * instance,
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    clcg_solver_enum solver_id = CLCG_BICG )
```

A combined conjugate gradient solver function. Note that both m and B are hosted on the GPU device.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client.
	<i>cub_handle</i>	Handler of the cublas object.
	<i>cus_handle</i>	Handlee of the cusparse object. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_BICG.

## Returns

Status of the function.

## 4.8.2.2 clcg\_solver\_preconditioned\_cuda()

```
int clcg_solver_preconditioned_cuda (
    clcg_axfunc_cuda_ptr Afp,
```

```

    clcg_axfunc_cuda_ptr Mfp,
    clcg_progress_cuda_ptr Pfp,
    cuDoubleComplex * m,
    const cuDoubleComplex * B,
    const int n_size,
    const int nz_size,
    const clcg_para * param,
    void * instance,
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    clcg_solver_enum solver_id = CLCG_PCG )

```

A combined conjugate gradient solver function. Note that both m and B are hosted on the GPU device.

#### Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Mfp</i>	Callback function for calculating the product of 'Mx' for preconditioning.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client.
	<i>cub_handle</i>	Handler of the cublas object.
	<i>cus_handle</i>	Handlee of the cusparse object. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.

#### Returns

Status of the function.

## 4.9 clcg\_cudaf.h File Reference

```

#include "util.h"
#include "lcg_complex_cuda.h"
#include <cublas_v2.h>
#include <cusparse_v2.h>

```

### Typedefs

- typedef void(\* [clcg\\_axfunc\\_cudaf\\_ptr](#)) (void \*instance, cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_↵\_handle, cusparseDnVecDescr\_t x, cusparseDnVecDescr\_t prod\_Ax, const int n\_size, const int nz\_size, cusparseOperation\_t oper\_t)  
*Callback interface for calculating the product of a N\*N matrix 'A' multiplied by a vertical vector 'x'. Note that both A and x are hosted on the GPU device.*
- typedef int(\* [clcg\\_progress\\_cudaf\\_ptr](#)) (void \*instance, const cuComplex \*m, const float converge, const [clcg\\_para](#) \*param, const int n\_size, const int nz\_size, const int k)  
*Callback interface for monitoring the progress and terminate the iteration if necessary. Note that m is hosted on the GPU device.*



## Functions

- int [clcg\\_solver\\_cuda](#) ([clcg\\_axfunc\\_cudaf\\_ptr](#) Afp, [clcg\\_progress\\_cudaf\\_ptr](#) Pfp, cuComplex \*m, const cuComplex \*B, const int n\_size, const int nz\_size, const [clcg\\_para](#) \*param, void \*instance, cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, [clcg\\_solver\\_enum](#) solver\_id=[CLCG\\_BICG](#))  
A combined conjugate gradient solver function. Note that both m and B are hosted on the GPU device.
- int [clcg\\_solver\\_preconditioned\\_cuda](#) ([clcg\\_axfunc\\_cudaf\\_ptr](#) Afp, [clcg\\_axfunc\\_cudaf\\_ptr](#) Mfp, [clcg\\_progress\\_cudaf\\_ptr](#) Pfp, cuComplex \*m, const cuComplex \*B, const int n\_size, const int nz\_size, const [clcg\\_para](#) \*param, void \*instance, cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, [clcg\\_solver\\_enum](#) solver\_id=[CLCG\\_PCG](#))  
A combined conjugate gradient solver function. Note that both m and B are hosted on the GPU device.

### 4.9.1 Typedef Documentation

#### 4.9.1.1 clcg\_axfunc\_cudaf\_ptr

```
typedef void(* clcg_axfunc_cudaf_ptr) (void *instance, cublasHandle_t cub_handle, cusparseHandle_t cus_handle, cusparseDnVecDescr_t x, cusparseDnVecDescr_t prod_Ax, const int n_size, const int nz_size, cusparseOperation_t oper_t)
```

Callback interface for calculating the product of a N\*N matrix 'A' multiplied by a vertical vector 'x'. Note that both A and x are hosted on the GPU device.

##### Parameters

<i>instance</i>	The user data sent for the <a href="#">lcg_solver_cuda()</a> functions by the client.
<i>cub_handle</i>	Handler of the cublas object.
<i>cus_handle</i>	Handlee of the cusparse object.
<i>x</i>	Multiplier of the Ax product.
<i>Ax</i>	Product of A multiplied by x.
<i>n_size</i>	Size of x and column/row numbers of A.

#### 4.9.1.2 clcg\_progress\_cudaf\_ptr

```
typedef int(* clcg_progress_cudaf_ptr) (void *instance, const cuComplex *m, const float converge, const clcg\_para *param, const int n_size, const int nz_size, const int k)
```

Callback interface for monitoring the progress and terminate the iteration if necessary. Note that m is hosted on the GPU device.

##### Parameters

<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> functions by the client.
<i>m</i>	The current solutions.
<i>converge</i>	The current value evaluating the iteration progress.
<i>n_size</i>	The size of the variables
<i>k</i>	The iteration count.

## Return values

<i>int</i>	Zero to continue the optimization process. Returning a non-zero value will terminate the optimization process.
------------	--

## 4.9.2 Function Documentation

### 4.9.2.1 clcg\_solver\_cuda()

```
int clcg_solver_cuda (
    clcg_axfunc_cuda_ptr Afp,
    clcg_progress_cuda_ptr Pfp,
    cuComplex * m,
    const cuComplex * B,
    const int n_size,
    const int nz_size,
    const clcg_para * param,
    void * instance,
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    clcg_solver_enum solver_id = CLCG_BICG )
```

A combined conjugate gradient solver function. Note that both m and B are hosted on the GPU device.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client.
	<i>cub_handle</i>	Handler of the cublas object.
	<i>cus_handle</i>	Handlee of the cusparse object. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_BICG.

## Returns

Status of the function.

### 4.9.2.2 clcg\_solver\_preconditioned\_cuda()

```
int clcg_solver_preconditioned_cuda (
    clcg_axfunc_cuda_ptr Afp,
```

```

    clcg_axfunc_cuda_ptr Mfp,
    clcg_progress_cuda_ptr Pfp,
    cuComplex * m,
    const cuComplex * B,
    const int n_size,
    const int nz_size,
    const clcg_para * param,
    void * instance,
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    clcg_solver_enum solver_id = CLCG_PCG )

```

A combined conjugate gradient solver function. Note that both m and B are hosted on the GPU device.

#### Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Mfp</i>	Callback function for calculating the product of 'Mx' for preconditioning.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client.
	<i>cub_handle</i>	Handler of the cublas object.
	<i>cus_handle</i>	Handlee of the cusparse object. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.

#### Returns

Status of the function.

## 4.10 clcg\_eigen.cpp File Reference

```

#include "cmath"
#include "ctime"
#include "iostream"
#include "clcg_eigen.h"
#include "config.h"
#include "omp.h"

```

#### Typedefs

- typedef int(\* [eigen\\_solver\\_ptr](#)) (clcg\_axfunc\_eigen\_ptr Afp, clcg\_progress\_eigen\_ptr Pfp, Eigen::VectorXcd &m, const Eigen::VectorXcd &B, const [clcg\\_para](#) \*param, void \*instance)
- typedef int(\* [eigen\\_preconditioned\\_solver\\_ptr](#)) (clcg\_axfunc\_eigen\_ptr Afp, [clcg\\_axfunc\\_eigen\\_ptr](#) Mfp, [clcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXcd &m, const Eigen::VectorXcd &B, const [clcg\\_para](#) \*param, void \*instance)

## Functions

- `int clbicg (clcg_axfunc_eigen_ptr Afp, clcg_progress_eigen_ptr Pfp, Eigen::VectorXcd &m, const Eigen::VectorXcd &B, const clcg_para *param, void *instance)`
- `int clbicg_symmetric (clcg_axfunc_eigen_ptr Afp, clcg_progress_eigen_ptr Pfp, Eigen::VectorXcd &m, const Eigen::VectorXcd &B, const clcg_para *param, void *instance)`
- `int clcgs (clcg_axfunc_eigen_ptr Afp, clcg_progress_eigen_ptr Pfp, Eigen::VectorXcd &m, const Eigen::VectorXcd &B, const clcg_para *param, void *instance)`
- `int cltfqmr (clcg_axfunc_eigen_ptr Afp, clcg_progress_eigen_ptr Pfp, Eigen::VectorXcd &m, const Eigen::VectorXcd &B, const clcg_para *param, void *instance)`
- `int clcg_solver_eigen (clcg_axfunc_eigen_ptr Afp, clcg_progress_eigen_ptr Pfp, Eigen::VectorXcd &m, const Eigen::VectorXcd &B, const clcg_para *param, void *instance, clcg_solver_enum solver_id)`  
*A combined conjugate gradient solver function.*
- `int clpcg (clcg_axfunc_eigen_ptr Afp, clcg_axfunc_eigen_ptr Mfp, clcg_progress_eigen_ptr Pfp, Eigen::VectorXcd &m, const Eigen::VectorXcd &B, const clcg_para *param, void *instance)`
- `int clpbicg (clcg_axfunc_eigen_ptr Afp, clcg_axfunc_eigen_ptr Mfp, clcg_progress_eigen_ptr Pfp, Eigen::VectorXcd &m, const Eigen::VectorXcd &B, const clcg_para *param, void *instance)`
- `int clcg_solver_preconditioned_eigen (clcg_axfunc_eigen_ptr Afp, clcg_axfunc_eigen_ptr Mfp, clcg_progress_eigen_ptr Pfp, Eigen::VectorXcd &m, const Eigen::VectorXcd &B, const clcg_para *param, void *instance, clcg_solver_enum solver_id)`  
*A combined conjugate gradient solver function.*

## 4.10.1 Typedef Documentation

### 4.10.1.1 eigen\_preconditioned\_solver\_ptr

```
typedef int(* eigen_preconditioned_solver_ptr) (clcg_axfunc_eigen_ptr Afp, clcg_axfunc_eigen_ptr
Mfp, clcg_progress_eigen_ptr Pfp, Eigen::VectorXcd &m, const Eigen::VectorXcd &B, const clcg_para
*param, void *instance)
```

### 4.10.1.2 eigen\_solver\_ptr

```
typedef int(* eigen_solver_ptr) (clcg_axfunc_eigen_ptr Afp, clcg_progress_eigen_ptr Pfp, Eigen↵
::VectorXcd &m, const Eigen::VectorXcd &B, const clcg_para *param, void *instance)
```

## 4.10.2 Function Documentation

### 4.10.2.1 clbicg()

```
int clbicg (
    clcg_axfunc_eigen_ptr Afp,
    clcg_progress_eigen_ptr Pfp,
    Eigen::VectorXcd & m,
    const Eigen::VectorXcd & B,
    const clcg_para * param,
    void * instance )
```

## 4.10.2.2 clbicg\_symmetric()

```
int clbicg_symmetric (
    clcg_axfunc_eigen_ptr Afp,
    clcg_progress_eigen_ptr Pfp,
    Eigen::VectorXcd & m,
    const Eigen::VectorXcd & B,
    const clcg_para * param,
    void * instance )
```

## 4.10.2.3 clcg\_solver\_eigen()

```
int clcg_solver_eigen (
    clcg_axfunc_eigen_ptr Afp,
    clcg_progress_eigen_ptr Pfp,
    Eigen::VectorXcd & m,
    const Eigen::VectorXcd & B,
    const clcg_para * param,
    void * instance,
    clcg_solver_enum solver_id = CLCG_CGS )
```

A combined conjugate gradient solver function.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the solver function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is CLCG_CGS.

## Returns

Status of the function.

## 4.10.2.4 clcg\_solver\_preconditioned\_eigen()

```
int clcg_solver_preconditioned_eigen (
    clcg_axfunc_eigen_ptr Afp,
    clcg_axfunc_eigen_ptr Mfp,
    clcg_progress_eigen_ptr Pfp,
    Eigen::VectorXcd & m,
    const Eigen::VectorXcd & B,
```

```

const clcg_para * param,
void * instance,
clcg_solver_enum solver_id = CLCG_PBICG )

```

A combined conjugate gradient solver function.

#### Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Mfp</i>	Callback function for calculating the product of ' $M^{-1}x$ ', in which M is the preconditioning matrix
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the solver function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. the value must CLCG_PBICG (default) or CLCG_PCG.

#### Returns

Status of the function.

#### 4.10.2.5 clcgs()

```

int clcgs (
    clcg_axfunc_eigen_ptr Afp,
    clcg_progress_eigen_ptr Pfp,
    Eigen::VectorXcd & m,
    const Eigen::VectorXcd & B,
    const clcg_para * param,
    void * instance )

```

#### 4.10.2.6 clpbicg()

```

int clpbicg (
    clcg_axfunc_eigen_ptr Afp,
    clcg_axfunc_eigen_ptr Mfp,
    clcg_progress_eigen_ptr Pfp,
    Eigen::VectorXcd & m,
    const Eigen::VectorXcd & B,
    const clcg_para * param,
    void * instance )

```

## 4.10.2.7 clpcg()

```
int clpcg (
    clcg_axfunc_eigen_ptr Afp,
    clcg_axfunc_eigen_ptr Mfp,
    clcg_progress_eigen_ptr Pfp,
    Eigen::VectorXcd & m,
    const Eigen::VectorXcd & B,
    const clcg_para * param,
    void * instance )
```

## 4.10.2.8 cltfqmr()

```
int cltfqmr (
    clcg_axfunc_eigen_ptr Afp,
    clcg_progress_eigen_ptr Pfp,
    Eigen::VectorXcd & m,
    const Eigen::VectorXcd & B,
    const clcg_para * param,
    void * instance )
```

## 4.11 clcg\_eigen.h File Reference

```
#include "util.h"
#include "complex"
#include "Eigen/Dense"
```

## Typedefs

- typedef void(\* [clcg\\_axfunc\\_eigen\\_ptr](#)) (void \*instance, const Eigen::VectorXcd &x, Eigen::VectorXcd &prod, [\\_Ax](#), [lcg\\_matrix\\_e](#) layout, [clcg\\_complex\\_e](#) conjugate)  
*Callback interface for calculating the product of a N\*N matrix 'A' multiplied by a vertical vector 'x'.*
- typedef int(\* [clcg\\_progress\\_eigen\\_ptr](#)) (void \*instance, const Eigen::VectorXcd \*m, const [lcg\\_float](#) converge, const [clcg\\_para](#) \*param, const int k)  
*Callback interface for monitoring the progress and terminate the iteration if necessary.*

## Functions

- int [clcg\\_solver\\_eigen](#) ([clcg\\_axfunc\\_eigen\\_ptr](#) Afp, [clcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXcd &m, const Eigen::VectorXcd &B, const [clcg\\_para](#) \*param, void \*instance, [clcg\\_solver\\_enum](#) solver\_id=[CLCG\\_CGS](#))  
*A combined conjugate gradient solver function.*
- int [clcg\\_solver\\_preconditioned\\_eigen](#) ([clcg\\_axfunc\\_eigen\\_ptr](#) Afp, [clcg\\_axfunc\\_eigen\\_ptr](#) Mfp, [clcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXcd &m, const Eigen::VectorXcd &B, const [clcg\\_para](#) \*param, void \*instance, [clcg\\_solver\\_enum](#) solver\_id=[CLCG\\_PBICG](#))  
*A combined conjugate gradient solver function.*

### 4.11.1 Typedef Documentation

#### 4.11.1.1 `clcg_axfunc_eigen_ptr`

```
typedef void(* clcg_axfunc_eigen_ptr) (void *instance, const Eigen::VectorXcd &x, Eigen::↵
VectorXcd &prod_Ax, lcg\_matrix\_e layout, clcg\_complex\_e conjugate)
```

Callback interface for calculating the product of a N\*N matrix 'A' multiplied by a vertical vector 'x'.

##### Parameters

<i>instance</i>	The user data sent for the solver functions by the client.
<i>x</i>	Multiplier of the Ax product.
<i>Ax</i>	Product of A multiplied by x.
<i>layout</i>	layout information of the matrix A passed by the solver functions.
<i>conjugate</i>	Layout information of the matrix A passed by the solver functions.

#### 4.11.1.2 `clcg_progress_eigen_ptr`

```
typedef int(* clcg_progress_eigen_ptr) (void *instance, const Eigen::VectorXcd *m, const lcg\_float
converge, const lcg\_para *param, const int k)
```

Callback interface for monitoring the progress and terminate the iteration if necessary.

##### Parameters

<i>instance</i>	The user data sent for the solver functions by the client.
<i>m</i>	The current solutions.
<i>converge</i>	The current value evaluating the iteration progress.
<i>param</i>	The parameter object passed by the solver functions.
<i>k</i>	The iteration count.

##### Return values

<i>int</i>	Zero to continue the optimization process. Returning a non-zero value will terminate the optimization process.
------------	--

### 4.11.2 Function Documentation



## 4.11.2.1 clcg\_solver\_eigen()

```
int clcg_solver_eigen (
    clcg_axfunc_eigen_ptr Afp,
    clcg_progress_eigen_ptr Pfp,
    Eigen::VectorXcd & m,
    const Eigen::VectorXcd & B,
    const clcg_para * param,
    void * instance,
    clcg_solver_enum solver_id = CLCG_CGS )
```

A combined conjugate gradient solver function.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the solver function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is CLCG_CGS.

## Returns

Status of the function.

## 4.11.2.2 clcg\_solver\_preconditioned\_eigen()

```
int clcg_solver_preconditioned_eigen (
    clcg_axfunc_eigen_ptr Afp,
    clcg_axfunc_eigen_ptr Mfp,
    clcg_progress_eigen_ptr Pfp,
    Eigen::VectorXcd & m,
    const Eigen::VectorXcd & B,
    const clcg_para * param,
    void * instance,
    clcg_solver_enum solver_id = CLCG_PBICG )
```

A combined conjugate gradient solver function.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Mfp</i>	Callback function for calculating the product of 'M <sup>-1</sup> x', in which M is the preconditioning matrix
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.

## Parameters

	<i>B</i>	Objective vector of the linear system.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the solver function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>solver↔ _id</i>	Solver type used to solve the linear system. the value must CLCG_PBICG (default) or CLCG_PCG.

## Returns

Status of the function.

## 4.12 config.h File Reference

### Macros

- `#define LibLCG_OPENMP`
- `#define LibLCG_EIGEN`
- `#define LibLCG_STD_COMPLEX`
- `#define LibLCG_CUDA`

#### 4.12.1 Macro Definition Documentation

##### 4.12.1.1 LibLCG\_CUDA

```
#define LibLCG_CUDA
```

##### 4.12.1.2 LibLCG\_EIGEN

```
#define LibLCG_EIGEN
```

##### 4.12.1.3 LibLCG\_OPENMP

```
#define LibLCG_OPENMP
```

## 4.12.1.4 LibLCG\_STD\_COMPLEX

```
#define LibLCG_STD_COMPLEX
```

## 4.13 lcg.cpp File Reference

```
#include "lcg.h"
#include "cmath"
#include "config.h"
#include "omp.h"
```

## Typedefs

- typedef int(\* [lcg\\_solver\\_ptr](#)) ([lcg\\_axfunc\\_ptr](#) Afp, [lcg\\_progress\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const int n\_size, const [lcg\\_para](#) \*param, void \*instance)  
*Callback interface of the conjugate gradient solver.*
- typedef int(\* [lcg\\_solver\\_ptr2](#)) ([lcg\\_axfunc\\_ptr](#) Afp, [lcg\\_progress\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const [lcg\\_float](#) \*low, const [lcg\\_float](#) \*hig, const int n\_size, const [lcg\\_para](#) \*param, void \*instance)  
*A combined conjugate gradient solver function.*

## Functions

- int [lbicgstab](#) ([lcg\\_axfunc\\_ptr](#) Afp, [lcg\\_progress\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const int n\_size, const [lcg\\_para](#) \*param, void \*instance)  
*Biconjugate gradient method.*
- int [lbicgstab2](#) ([lcg\\_axfunc\\_ptr](#) Afp, [lcg\\_progress\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const int n\_size, const [lcg\\_para](#) \*param, void \*instance)  
*Biconjugate gradient method 2.*
- int [lcg\\_solver](#) ([lcg\\_axfunc\\_ptr](#) Afp, [lcg\\_progress\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const int n\_size, const [lcg\\_para](#) \*param, void \*instance, [lcg\\_solver\\_enum](#) solver\_id)  
*A combined conjugate gradient solver function.*
- int [lpcg](#) ([lcg\\_axfunc\\_ptr](#) Afp, [lcg\\_axfunc\\_ptr](#) Mfp, [lcg\\_progress\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const int n\_size, const [lcg\\_para](#) \*param, void \*instance)  
*Preconditioned conjugate gradient method.*
- int [lcg\\_solver\\_preconditioned](#) ([lcg\\_axfunc\\_ptr](#) Afp, [lcg\\_axfunc\\_ptr](#) Mfp, [lcg\\_progress\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const int n\_size, const [lcg\\_para](#) \*param, void \*instance, [lcg\\_solver\\_enum](#) solver\_id)  
*A combined conjugate gradient solver function.*
- int [lpg](#) ([lcg\\_axfunc\\_ptr](#) Afp, [lcg\\_progress\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const [lcg\\_float](#) \*low, const [lcg\\_float](#) \*hig, const int n\_size, const [lcg\\_para](#) \*param, void \*instance)  
*Conjugate gradient method with projected gradient for inequality constraints.*
- int [lspg](#) ([lcg\\_axfunc\\_ptr](#) Afp, [lcg\\_progress\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const [lcg\\_float](#) \*low, const [lcg\\_float](#) \*hig, const int n\_size, const [lcg\\_para](#) \*param, void \*instance)  
*Conjugate gradient method with projected gradient for inequality constraints.*
- int [lcg\\_solver\\_constrained](#) ([lcg\\_axfunc\\_ptr](#) Afp, [lcg\\_progress\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const [lcg\\_float](#) \*low, const [lcg\\_float](#) \*hig, const int n\_size, const [lcg\\_para](#) \*param, void \*instance, [lcg\\_solver\\_enum](#) solver\_id)  
*A combined conjugate gradient solver function with inequality constraints.*
- int [lcg](#) ([lcg\\_axfunc\\_ptr](#) Afp, [lcg\\_progress\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const int n\_size, const [lcg\\_para](#) \*param, void \*instance, [lcg\\_float](#) \*Gk, [lcg\\_float](#) \*Dk, [lcg\\_float](#) \*ADk)  
*Standalone function of the Linear Conjugate Gradient algorithm.*
- int [lcgs](#) ([lcg\\_axfunc\\_ptr](#) Afp, [lcg\\_progress\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const int n\_size, const [lcg\\_para](#) \*param, void \*instance, [lcg\\_float](#) \*RK, [lcg\\_float](#) \*ROT, [lcg\\_float](#) \*PK, [lcg\\_float](#) \*AX, [lcg\\_float](#) \*UK, [lcg\\_float](#) \*QK, [lcg\\_float](#) \*WK)  
*Standalone function of the Conjugate Gradient Squared algorithm.*

### 4.13.1 Typedef Documentation

#### 4.13.1.1 lcg\_solver\_ptr

```
typedef int(* lcg_solver_ptr) (lcg_axfunc_ptr Afp, lcg_progress_ptr Pfp, lcg_float *m, const
lcg_float *B, const int n_size, const lcg_para *param, void *instance)
```

Callback interface of the conjugate gradient solver.

##### Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.

##### Returns

Status of the function.

#### 4.13.1.2 lcg\_solver\_ptr2

```
typedef int(* lcg_solver_ptr2) (lcg_axfunc_ptr Afp, lcg_progress_ptr Pfp, lcg_float *m, const
lcg_float *B, const lcg_float *low, const lcg_float *hig, const int n_size, const lcg_para
*param, void *instance)
```

A combined conjugate gradient solver function.

##### Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>low</i>	The lower boundary of the acceptable solution.
in	<i>hig</i>	The higher boundary of the acceptable solution.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.
	<i>P</i>	Precondition vector (optional expect for the LCG_PCG method). The default value is nullptr.

## Returns

Status of the function.

## 4.13.2 Function Documentation

### 4.13.2.1 lbicgstab()

```
int lbicgstab (
    lcg_axfunc_ptr Afp,
    lcg_progress_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
    const int n_size,
    const lcg_para * param,
    void * instance )
```

Biconjugate gradient method.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>P</i>	Precondition vector (optional expect for the LCG_PCG method). The default value is nullptr.

## Returns

Status of the function.

### 4.13.2.2 lbicgstab2()

```
int lbicgstab2 (
    lcg_axfunc_ptr Afp,
    lcg_progress_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
    const int n_size,
    const lcg_para * param,
    void * instance )
```

Biconjugate gradient method 2.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>P</i>	Precondition vector (optional expect for the LCG_PCG method). The default value is nullptr.

## Returns

Status of the function.

4.13.2.3 [lcg\(\)](#)

```
int lcg (
    lcg_axfunc_ptr Afp,
    lcg_progress_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
    const int n_size,
    const lcg_para * param,
    void * instance,
    lcg_float * Gk = nullptr,
    lcg_float * Dk = nullptr,
    lcg_float * ADk = nullptr )
```

Standalone function of the Linear Conjugate Gradient algorithm.

## Note

To use the [lcg\(\)](#) function for massive inversions, it is better to provide external vectors Gk, Dk and ADk to avoid allocating and destroying temporary vectors.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector of the size n_size
in	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
in	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg()</a> function by the client. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>Gk</i>	Conjugate gradient vector of the size n_size. If this pointer is null, the function will create an internal vector instead.
	<i>Dk</i>	Directional gradient vector of the size n_size. If this pointer is null, the function will create an internal vector instead.
	<i>ADk</i>	Intermediate vector of the size n_size. If this pointer is null, the function will create an internal vector instead.

## Returns

Status of the function.

## 4.13.2.4 lcg\_solver()

```
int lcg_solver (
    lcg_axfunc_ptr Afp,
    lcg_progress_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
    const int n_size,
    const lcg_para * param,
    void * instance,
    lcg_solver_enum solver_id = LCG_CGS )
```

A combined conjugate gradient solver function.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.

## Returns

Status of the function.

## 4.13.2.5 lcg\_solver\_constrained()

```
int lcg_solver_constrained (
    lcg_axfunc_ptr Afp,
    lcg_progress_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
    const lcg_float * low,
    const lcg_float * hig,
    const int n_size,
    const lcg_para * param,
    void * instance,
    lcg_solver_enum solver_id = LCG_PG )
```

A combined conjugate gradient solver function with inequality constraints.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>low</i>	The lower boundary of the acceptable solution.
in	<i>hig</i>	The higher boundary of the acceptable solution.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver↔ _id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.
	<i>P</i>	Precondition vector (optional expect for the LCG_PCG method). The default value is NULL.

## Returns

Status of the function.

4.13.2.6 [lcg\\_solver\\_preconditioned\(\)](#)

```
int lcg_solver_preconditioned (
    lcg_axfunc_ptr Afp,
    lcg_axfunc_ptr Mfp,
    lcg_progress_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
    const int n_size,
    const lcg_para * param,
    void * instance,
    lcg_solver_enum solver_id = LCG_PCG )
```

A combined conjugate gradient solver function.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Mfp</i>	Callback function for calculating the product of ' $M^{-1}x$ ', in which M is the preconditioning matrix.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver↔ _id</i>	Solver type used to solve the linear system. The default value is LCG_PCG.



## Returns

Status of the function.

## 4.13.2.7 lcgs()

```
int lcgs (
    lcg_axfunc_ptr Afp,
    lcg_progress_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
    const int n_size,
    const lcg_para * param,
    void * instance,
    lcg_float * RK = nullptr,
    lcg_float * ROT = nullptr,
    lcg_float * PK = nullptr,
    lcg_float * AX = nullptr,
    lcg_float * UK = nullptr,
    lcg_float * QK = nullptr,
    lcg_float * WK = nullptr )
```

Standalone function of the Conjugate Gradient Squared algorithm.

## Note

Algorithm 2 in "Generalized conjugate gradient method" by Fokkema et al. (1996).

To use the `lcgs()` function for massive inversions, it is better to provide external vectors RK, ROT, PK, AX, UK, QK, and WK to avoid allocating and destroying temporary vectors.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <code>lcg_solver()</code> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>RK</i>	Intermediate vector of the size <i>n_size</i> . If this pointer is null, the function will create an internal vector instead.
	<i>ROT</i>	Intermediate vector of the size <i>n_size</i> . If this pointer is null, the function will create an internal vector instead.
	<i>PK</i>	Intermediate vector of the size <i>n_size</i> . If this pointer is null, the function will create an internal vector instead.
	<i>AX</i>	Intermediate vector of the size <i>n_size</i> . If this pointer is null, the function will create an internal vector instead.
	<i>UK</i>	Intermediate vector of the size <i>n_size</i> . If this pointer is null, the function will create an internal vector instead.
	<i>QK</i>	Intermediate vector of the size <i>n_size</i> . If this pointer is null, the function will create an internal vector instead.
	<i>WK</i>	Intermediate vector of the size <i>n_size</i> . If this pointer is null, the function will create an internal vector instead.
Generated by Doxygen		internal vector instead.

**Returns**

Status of the function.

**4.13.2.8 lpcg()**

```
int lpcg (
    lcg_axfunc_ptr Afp,
    lcg_axfunc_ptr Mfp,
    lcg_progress_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
    const int n_size,
    const lcg_para * param,
    void * instance )
```

Preconditioned conjugate gradient method.

**Note**

Algorithm 1 in "Preconditioned conjugate gradients for singular systems" by Kaasschieter (1988).

**Parameters**

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Mfp</i>	Callback function for calculating the product of ' $M^{-1}x$ ', in which M is the preconditioning matrix.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.

**Returns**

Status of the function.

**4.13.2.9 lpg()**

```
int lpg (
    lcg_axfunc_ptr Afp,
    lcg_progress_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
```

```

const lcg_float * low,
const lcg_float * hig,
const int n_size,
const lcg_para * param,
void * instance )

```

Conjugate gradient method with projected gradient for inequality constraints.

#### Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>low</i>	The lower boundary of the acceptable solution.
in	<i>hig</i>	The higher boundary of the acceptable solution.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>solver↔ _id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.
	<i>P</i>	Precondition vector (optional expect for the LCG_PCG method). The default value is nullptr.

#### Returns

Status of the function.

#### 4.13.2.10 lspg()

```

int lspg (
    lcg_axfunc_ptr Afp,
    lcg_progress_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
    const lcg_float * low,
    const lcg_float * hig,
    const int n_size,
    const lcg_para * param,
    void * instance )

```

Conjugate gradient method with projected gradient for inequality constraints.

#### Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>low</i>	The lower boundary of the acceptable solution.

## Parameters

in	<i>hig</i>	The higher boundary of the acceptable solution.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.
	<i>P</i>	Precondition vector (optional expect for the LCG_PCG method). The default value is nullptr.

## Returns

Status of the function.

## 4.14 lcg.h File Reference

```
#include "util.h"
```

### Typedefs

- typedef void(\* [lcg\\_axfunc\\_ptr](#)) (void \*instance, const [lcg\\_float](#) \*x, [lcg\\_float](#) \*prod\_Ax, const int n\_size)  
*Callback interface for calculating the product of a N\*N matrix 'A' multiplied by a vertical vector 'x'.*
- typedef int(\* [lcg\\_progress\\_ptr](#)) (void \*instance, const [lcg\\_float](#) \*m, const [lcg\\_float](#) converge, const [lcg\\_para](#) \*param, const int n\_size, const int k)  
*Callback interface for monitoring the progress and terminate the iteration if necessary.*

### Functions

- int [lcg\\_solver](#) ([lcg\\_axfunc\\_ptr](#) Afp, [lcg\\_progress\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const int n\_size, const [lcg\\_para](#) \*param, void \*instance, [lcg\\_solver\\_enum](#) solver\_id=LCG\_CGS)  
*A combined conjugate gradient solver function.*
- int [lcg\\_solver\\_preconditioned](#) ([lcg\\_axfunc\\_ptr](#) Afp, [lcg\\_axfunc\\_ptr](#) Mfp, [lcg\\_progress\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const int n\_size, const [lcg\\_para](#) \*param, void \*instance, [lcg\\_solver\\_enum](#) solver\_id=LCG\_PCG)  
*A combined conjugate gradient solver function.*
- int [lcg\\_solver\\_constrained](#) ([lcg\\_axfunc\\_ptr](#) Afp, [lcg\\_progress\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const [lcg\\_float](#) \*low, const [lcg\\_float](#) \*hig, const int n\_size, const [lcg\\_para](#) \*param, void \*instance, [lcg\\_solver\\_enum](#) solver\_id=LCG\_PG)  
*A combined conjugate gradient solver function with inequality constraints.*
- int [lcg](#) ([lcg\\_axfunc\\_ptr](#) Afp, [lcg\\_progress\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const int n\_size, const [lcg\\_para](#) \*param, void \*instance, [lcg\\_float](#) \*Gk=nullptr, [lcg\\_float](#) \*Dk=nullptr, [lcg\\_float](#) \*ADk=nullptr)  
*Standalone function of the Linear Conjugate Gradient algorithm.*
- int [lcgs](#) ([lcg\\_axfunc\\_ptr](#) Afp, [lcg\\_progress\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const int n\_size, const [lcg\\_para](#) \*param, void \*instance, [lcg\\_float](#) \*RK=nullptr, [lcg\\_float](#) \*ROT=nullptr, [lcg\\_float](#) \*PK=nullptr, [lcg\\_float](#) \*AX=nullptr, [lcg\\_float](#) \*UK=nullptr, [lcg\\_float](#) \*QK=nullptr, [lcg\\_float](#) \*WK=nullptr)  
*Standalone function of the Conjugate Gradient Squared algorithm.*

## 4.14.1 Typedef Documentation

### 4.14.1.1 lcg\_axfunc\_ptr

```
typedef void(* lcg_axfunc_ptr) (void *instance, const lcg_float *x, lcg_float *prod_Ax, const
int n_size)
```

Callback interface for calculating the product of a N\*N matrix 'A' multiplied by a vertical vector 'x'.

#### Parameters

<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> functions by the client.
<i>x</i>	Multiplier of the Ax product.
<i>Ax</i>	Product of A multiplied by x.
<i>n_size</i>	Size of x and column/row numbers of A.

### 4.14.1.2 lcg\_progress\_ptr

```
typedef int(* lcg_progress_ptr) (void *instance, const lcg_float *m, const lcg_float converge,
const lcg_para *param, const int n_size, const int k)
```

Callback interface for monitoring the progress and terminate the iteration if necessary.

#### Parameters

<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> functions by the client.
<i>m</i>	The current solutions.
<i>converge</i>	The current value evaluating the iteration progress.
<i>n_size</i>	The size of the variables
<i>k</i>	The iteration count.

#### Return values

<i>int</i>	Zero to continue the optimization process. Returning a non-zero value will terminate the optimization process.
------------	--

## 4.14.2 Function Documentation

### 4.14.2.1 lcg()

```
int lcg (
    lcg_axfunc_ptr Afp,
```

```

    lcg_progress_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
    const int n_size,
    const lcg_para * param,
    void * instance,
    lcg_float * Gk = nullptr,
    lcg_float * Dk = nullptr,
    lcg_float * ADk = nullptr )

```

Standalone function of the Linear Conjugate Gradient algorithm.

#### Note

To use the `lcg()` function for massive inversions, it is better to provide external vectors Gk, Dk and ADk to avoid allocating and destroying temporary vectors.

#### Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector of the size n_size
in	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
in	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <code>lcg()</code> function by the client. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>Gk</i>	Conjugate gradient vector of the size n_size. If this pointer is null, the function will create an internal vector instead.
	<i>Dk</i>	Directional gradient vector of the size n_size. If this pointer is null, the function will create an internal vector instead.
	<i>ADk</i>	Intermediate vector of the size n_size. If this pointer is null, the function will create an internal vector instead.

#### Returns

Status of the function.

#### 4.14.2.2 lcg\_solver()

```

int lcg_solver (
    lcg_axfunc_ptr Afp,
    lcg_progress_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
    const int n_size,
    const lcg_para * param,
    void * instance,
    lcg_solver_enum solver_id = LCG_CGS )

```

A combined conjugate gradient solver function.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.

## Returns

Status of the function.

## 4.14.2.3 lcg\_solver\_constrained()

```
int lcg_solver_constrained (
    lcg_axfunc_ptr Afp,
    lcg_progress_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
    const lcg_float * low,
    const lcg_float * hig,
    const int n_size,
    const lcg_para * param,
    void * instance,
    lcg_solver_enum solver_id = LCG_PG )
```

A combined conjugate gradient solver function with inequality constraints.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>low</i>	The lower boundary of the acceptable solution.
in	<i>hig</i>	The higher boundary of the acceptable solution.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.
	<i>P</i>	Precondition vector (optional expect for the LCG_PCG method). The default value is NULL.

**Returns**

Status of the function.

**4.14.2.4 lcg\_solver\_preconditioned()**

```
int lcg_solver_preconditioned (
    lcg_axfunc_ptr Afp,
    lcg_axfunc_ptr Mfp,
    lcg_progress_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
    const int n_size,
    const lcg_para * param,
    void * instance,
    lcg_solver_enum solver_id = LCG_PCG )
```

A combined conjugate gradient solver function.

**Parameters**

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Mfp</i>	Callback function for calculating the product of ' $M^{-1}x$ ', in which M is the preconditioning matrix.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_PCG.

**Returns**

Status of the function.

**4.14.2.5 lcgs()**

```
int lcgs (
    lcg_axfunc_ptr Afp,
    lcg_progress_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
    const int n_size,
    const lcg_para * param,
```



```

void * instance,
lcg_float * RK = nullptr,
lcg_float * ROT = nullptr,
lcg_float * PK = nullptr,
lcg_float * AX = nullptr,
lcg_float * UK = nullptr,
lcg_float * QK = nullptr,
lcg_float * WK = nullptr )

```

Standalone function of the Conjugate Gradient Squared algorithm.

#### Note

Algorithm 2 in "Generalized conjugate gradient method" by Fokkema et al. (1996).

To use the [lcgs\(\)](#) function for massive inversions, it is better to provide external vectors RK, ROT, PK, AX, UK, QK, and WK to avoid allocating and destroying temporary vectors.

#### Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>RK</i>	Intermediate vector of the size n_size. If this pointer is null, the function will create an internal vector instead.
	<i>ROT</i>	Intermediate vector of the size n_size. If this pointer is null, the function will create an internal vector instead.
	<i>PK</i>	Intermediate vector of the size n_size. If this pointer is null, the function will create an internal vector instead.
	<i>AX</i>	Intermediate vector of the size n_size. If this pointer is null, the function will create an internal vector instead.
	<i>UK</i>	Intermediate vector of the size n_size. If this pointer is null, the function will create an internal vector instead.
	<i>QK</i>	Intermediate vector of the size n_size. If this pointer is null, the function will create an internal vector instead.
	<i>WK</i>	Intermediate vector of the size n_size. If this pointer is null, the function will create an internal vector instead.

#### Returns

Status of the function.

## 4.15 lcg\_complex.cpp File Reference

```

#include "cmath"
#include "ctime"
#include "random"
#include "lcg_complex.h"
#include "omp.h"

```

## Functions

- `lcg_complex * clcg_malloc (int n)`  
*Locate memory for a `lcg_complex` pointer type.*
- `lcg_complex ** clcg_malloc (int m, int n)`  
*Locate memory for a `lcg_complex` second pointer type.*
- `void clcg_free (lcg_complex *x)`  
*Destroy memory used by the `lcg_complex` type array.*
- `void clcg_free (lcg_complex **x, int m)`  
*Destroy memory used by the 2D `lcg_complex` type array.*
- `void clcg_vecset (lcg_complex *a, lcg_complex b, int size)`  
*set a complex vector's value*
- `void clcg_vecset (lcg_complex **a, lcg_complex b, int m, int n)`  
*set a 2d complex vector's value*
- `void clcg_set (lcg_complex *a, lcg_float r, lcg_float i)`  
*setup a complex number*
- `lcg_float clcg_square (const lcg_complex *a)`  
*Calculate the squared module of a complex number.*
- `lcg_float clcg_module (const lcg_complex *a)`  
*Calculate the module of a complex number.*
- `lcg_complex clcg_conjugate (const lcg_complex *a)`  
*Calculate the conjugate of a complex number.*
- `void clcg_vecrnd (lcg_complex *a, lcg_complex l, lcg_complex h, int size)`  
*set a complex vector using random values*
- `void clcg_vecrnd (lcg_complex **a, lcg_complex l, lcg_complex h, int m, int n)`  
*set a 2D complex vector using random values*
- `void clcg_dot (lcg_complex &ret, const lcg_complex *a, const lcg_complex *b, int size)`  
*calculate dot product of two complex vectors*
- `void clcg_inner (lcg_complex &ret, const lcg_complex *a, const lcg_complex *b, int size)`  
*calculate inner product of two complex vectors*
- `void clcg_matvec (lcg_complex **A, const lcg_complex *x, lcg_complex *Ax, int m_size, int n_size, lcg_matrix_e layout, lcg_complex_e conjugate)`  
*calculate product of a complex matrix and a complex vector*

### 4.15.1 Function Documentation

#### 4.15.1.1 `clcg_conjugate()`

```
lcg_complex clcg_conjugate (
    const lcg_complex * a )
```

Calculate the conjugate of a complex number.

#### Returns

The complex conjugate.

## 4.15.1.2 clcg\_dot()

```
void clcg_dot (
    lcg_complex & ret,
    const lcg_complex * a,
    const lcg_complex * b,
    int size )
```

calculate dot product of two complex vectors

the product of two complex vectors are defined as  $\langle a, b \rangle = \sum \{a_i \cdot b_i\}$

## Parameters

in	<i>a</i>	complex vector a
in	<i>b</i>	complex vector b
in	<i>x_size</i>	size of the vector

## Returns

product

## 4.15.1.3 clcg\_free() [1/2]

```
void clcg_free (
    lcg_complex ** x,
    int m )
```

Destroy memory used by the 2D lcg\_complex type array.

## Parameters

<i>x</i>	Pointer of the array.
----------	-----------------------

## 4.15.1.4 clcg\_free() [2/2]

```
void clcg_free (
    lcg_complex * x )
```

Destroy memory used by the lcg\_complex type array.

## Parameters

<i>x</i>	Pointer of the array.
----------	-----------------------

#### 4.15.1.5 clcg\_inner()

```
void clcg_inner (
    lcg_complex & ret,
    const lcg_complex * a,
    const lcg_complex * b,
    int size )
```

calculate inner product of two complex vectors

the product of two complex vectors are defined as  $\langle a, b \rangle = \sum \{\bar{a}_i \cdot b_i\}$

##### Parameters

in	<i>a</i>	complex vector a
in	<i>b</i>	complex vector b
in	<i>x_size</i>	size of the vector

##### Returns

product

#### 4.15.1.6 clcg\_malloc() [1/2]

```
lcg_complex** clcg_malloc (
    int m,
    int n )
```

Locate memory for a lcg\_complex second pointer type.

##### Parameters

in	<i>n</i>	Size of the lcg_float array.
----	----------	------------------------------

##### Returns

Pointer of the array's location.

#### 4.15.1.7 clcg\_malloc() [2/2]

```
lcg_complex* clcg_malloc (
    int n )
```

Locate memory for a lcg\_complex pointer type.

## Parameters

in	$n$	Size of the lcg_float array.
----	-----	------------------------------

## Returns

Pointer of the array's location.

## 4.15.1.8 clcg\_matvec()

```
void clcg_matvec (
    lcg_complex ** A,
    const lcg_complex * x,
    lcg_complex * Ax,
    int m_size,
    int n_size,
    lcg_matrix_e layout = MatNormal,
    clcg_complex_e conjugate = NonConjugate )
```

calculate product of a complex matrix and a complex vector

the product of two complex vectors are defined as  $\langle a, b \rangle = \sum \{\bar{a}_i \cdot b_i\}$ . Different configurations:  
 layout=Normal,conjugate=false -> A layout=Transpose,conjugate=false ->  $A^T$  layout=Normal,conjugate=true ->  $\bar{A}$  layout=Transpose,conjugate=true ->  $A^H$

## Parameters

	$A$	complex matrix A
in	$x$	complex vector x
	$Ax$	product of Ax
in	$m\_size$	row size of A
in	$n\_size$	column size of A
in	$layout$	layout of A used for multiplication. Must be Normal or Transpose
in	$conjugate$	whether to use the complex conjugate of A for calculation

## 4.15.1.9 clcg\_module()

```
lcg_float clcg_module (
    const lcg_complex * a )
```

Calculate the module of a complex number.

## Returns

The module

#### 4.15.1.10 `clcg_set()`

```
void clcg_set (
    lcg_complex * a,
    lcg_float r,
    lcg_float i )
```

setup a complex number

##### Parameters

in	<i>r</i>	The real part of the complex number
in	<i>i</i>	The imaginary part of the complex number

#### 4.15.1.11 `clcg_square()`

```
lcg_float clcg_square (
    const lcg_complex * a )
```

Calculate the squared module of a complex number.

##### Returns

The module

#### 4.15.1.12 `clcg_vecrnd()` [1/2]

```
void clcg_vecrnd (
    lcg_complex ** a,
    lcg_complex l,
    lcg_complex h,
    int m,
    int n )
```

set a 2D complex vector using random values

##### Parameters

	<i>a</i>	pointer of the vector
in	<i>l</i>	the lower bound of random values
in	<i>h</i>	the higher bound of random values
in	<i>m</i>	row size of the vector
in	<i>n</i>	column size of the vector

**4.15.1.13 clcg\_vecrnd()** [2/2]

```
void clcg_vecrnd (
    lcg_complex * a,
    lcg_complex l,
    lcg_complex h,
    int size )
```

set a complex vector using random values

**Parameters**

	<i>a</i>	pointer of the vector
in	<i>l</i>	the lower bound of random values
in	<i>h</i>	the higher bound of random values
in	<i>size</i>	size of the vector

**4.15.1.14 clcg\_vecset()** [1/2]

```
void clcg_vecset (
    lcg_complex ** a,
    lcg_complex b,
    int m,
    int n )
```

set a 2d complex vector's value

**Parameters**

	<i>a</i>	pointer of the matrix
in	<i>b</i>	initial value
in	<i>m</i>	row size of the matrix
in	<i>n</i>	column size of the matrix

**4.15.1.15 clcg\_vecset()** [2/2]

```
void clcg_vecset (
    lcg_complex * a,
    lcg_complex b,
    int size )
```

set a complex vector's value

**Parameters**

	<i>a</i>	pointer of the vector
in	<i>b</i>	initial value
in	<i>size</i>	vector size

## 4.16 lcg\_complex.h File Reference

```
#include "iostream"
#include "algebra.h"
#include "complex"
```

### Typedefs

- typedef std::complex< [lcg\\_float](#) > [lcg\\_complex](#)

### Functions

- [lcg\\_complex](#) \* [clcg\\_malloc](#) (int n)  
*Locate memory for a lcg\_complex pointer type.*
- [lcg\\_complex](#) \*\* [clcg\\_malloc](#) (int m, int n)  
*Locate memory for a lcg\_complex second pointer type.*
- void [clcg\\_free](#) ([lcg\\_complex](#) \*x)  
*Destroy memory used by the lcg\_complex type array.*
- void [clcg\\_free](#) ([lcg\\_complex](#) \*\*x, int m)  
*Destroy memory used by the 2D lcg\_complex type array.*
- void [clcg\\_vecset](#) ([lcg\\_complex](#) \*a, [lcg\\_complex](#) b, int size)  
*set a complex vector's value*
- void [clcg\\_vecset](#) ([lcg\\_complex](#) \*\*a, [lcg\\_complex](#) b, int m, int n)  
*set a 2d complex vector's value*
- void [clcg\\_set](#) ([lcg\\_complex](#) \*a, [lcg\\_float](#) r, [lcg\\_float](#) i)  
*setup a complex number*
- [lcg\\_float](#) [clcg\\_square](#) (const [lcg\\_complex](#) \*a)  
*Calculate the squared module of a complex number.*
- [lcg\\_float](#) [clcg\\_module](#) (const [lcg\\_complex](#) \*a)  
*Calculate the module of a complex number.*
- [lcg\\_complex](#) [clcg\\_conjugate](#) (const [lcg\\_complex](#) \*a)  
*Calculate the conjugate of a complex number.*
- void [clcg\\_vecrnd](#) ([lcg\\_complex](#) \*a, [lcg\\_complex](#) l, [lcg\\_complex](#) h, int size)  
*set a complex vector using random values*
- void [clcg\\_vecrnd](#) ([lcg\\_complex](#) \*\*a, [lcg\\_complex](#) l, [lcg\\_complex](#) h, int m, int n)  
*set a 2D complex vector using random values*
- void [clcg\\_dot](#) ([lcg\\_complex](#) &ret, const [lcg\\_complex](#) \*a, const [lcg\\_complex](#) \*b, int size)  
*calculate dot product of two complex vectors*
- void [clcg\\_inner](#) ([lcg\\_complex](#) &ret, const [lcg\\_complex](#) \*a, const [lcg\\_complex](#) \*b, int size)  
*calculate inner product of two complex vectors*
- void [clcg\\_matvec](#) ([lcg\\_complex](#) \*\*A, const [lcg\\_complex](#) \*x, [lcg\\_complex](#) \*Ax, int m\_size, int n\_size, [lcg\\_matrix\\_e](#) layout=MatNormal, [lcg\\_complex\\_e](#) conjugate=NonConjugate)  
*calculate product of a complex matrix and a complex vector*

#### 4.16.1 Typedef Documentation



### 4.16.1.1 lcg\_complex

```
typedef std::complex<lcg_float> lcg_complex
```

## 4.16.2 Function Documentation

### 4.16.2.1 clcg\_conjugate()

```
lcg_complex clcg_conjugate (
    const lcg_complex * a )
```

Calculate the conjugate of a complex number.

#### Returns

The complex conjugate.

### 4.16.2.2 clcg\_dot()

```
void clcg_dot (
    lcg_complex & ret,
    const lcg_complex * a,
    const lcg_complex * b,
    int size )
```

calculate dot product of two complex vectors

the product of two complex vectors are defined as  $\langle a, b \rangle = \sum \{a_i \cdot b_i\}$

#### Parameters

in	<i>a</i>	complex vector a
in	<i>b</i>	complex vector b
in	<i>x_size</i>	size of the vector

#### Returns

product

### 4.16.2.3 clcg\_free() [1/2]

```
void clcg_free (
```

```

    lcg_complex ** x,
    int m )

```

Destroy memory used by the 2D lcg\_complex type array.

#### Parameters

<i>x</i>	Pointer of the array.
----------	-----------------------

#### 4.16.2.4 clcg\_free() [2/2]

```

void clcg_free (
    lcg_complex * x )

```

Destroy memory used by the lcg\_complex type array.

#### Parameters

<i>x</i>	Pointer of the array.
----------	-----------------------

#### 4.16.2.5 clcg\_inner()

```

void clcg_inner (
    lcg_complex & ret,
    const lcg_complex * a,
    const lcg_complex * b,
    int size )

```

calculate inner product of two complex vectors

the product of two complex vectors are defined as  $\langle a, b \rangle = \sum \{\bar{a}_i \cdot b_i\}$

#### Parameters

in	<i>a</i>	complex vector a
in	<i>b</i>	complex vector b
in	<i>x_size</i>	size of the vector

#### Returns

product

**4.16.2.6 clcg\_malloc() [1/2]**

```
lcg_complex** clcg_malloc (
    int m,
    int n )
```

Locate memory for a lcg\_complex second pointer type.

**Parameters**

in	<i>n</i>	Size of the lcg_float array.
----	----------	------------------------------

**Returns**

Pointer of the array's location.

**4.16.2.7 clcg\_malloc() [2/2]**

```
lcg_complex* clcg_malloc (
    int n )
```

Locate memory for a lcg\_complex pointer type.

**Parameters**

in	<i>n</i>	Size of the lcg_float array.
----	----------	------------------------------

**Returns**

Pointer of the array's location.

**4.16.2.8 clcg\_matvec()**

```
void clcg_matvec (
    lcg_complex ** A,
    const lcg_complex * x,
    lcg_complex * Ax,
    int m_size,
    int n_size,
    lcg_matrix_e layout = MatNormal,
    clcg_complex_e conjugate = NonConjugate )
```

calculate product of a complex matrix and a complex vector

the product of two complex vectors are defined as  $\langle a, b \rangle = \sum \{\bar{a}_i \cdot b_i\}$ . Different configurations:  
 layout=Normal,conjugate=false -> A layout=Transpose,conjugate=false ->  $A^T$  layout=Normal,conjugate=true ->  $\bar{A}$   
 layout=Transpose,conjugate=true ->  $A^H$

## Parameters

	$A$	complex matrix A
in	$x$	complex vector x
	$Ax$	product of Ax
in	$m\_size$	row size of A
in	$n\_size$	column size of A
in	$layout$	layout of A used for multiplication. Must be Normal or Transpose
in	$conjugate$	whether to use the complex conjugate of A for calculation

4.16.2.9 `clcg_module()`

```
lcg_float clcg_module (
    const lcg_complex * a )
```

Calculate the module of a complex number.

## Returns

The module

4.16.2.10 `clcg_set()`

```
void clcg_set (
    lcg_complex * a,
    lcg_float r,
    lcg_float i )
```

setup a complex number

## Parameters

in	$r$	The real part of the complex number
in	$i$	The imaginary part of the complex number

4.16.2.11 `clcg_square()`

```
lcg_float clcg_square (
    const lcg_complex * a )
```

Calculate the squared module of a complex number.

**Returns**

The module

**4.16.2.12 clcg\_vecrnd() [1/2]**

```
void clcg_vecrnd (
    lcg_complex ** a,
    lcg_complex l,
    lcg_complex h,
    int m,
    int n )
```

set a 2D complex vector using random values

**Parameters**

	<i>a</i>	pointer of the vector
in	<i>l</i>	the lower bound of random values
in	<i>h</i>	the higher bound of random values
in	<i>m</i>	row size of the vector
in	<i>n</i>	column size of the vector

**4.16.2.13 clcg\_vecrnd() [2/2]**

```
void clcg_vecrnd (
    lcg_complex * a,
    lcg_complex l,
    lcg_complex h,
    int size )
```

set a complex vector using random values

**Parameters**

	<i>a</i>	pointer of the vector
in	<i>l</i>	the lower bound of random values
in	<i>h</i>	the higher bound of random values
in	<i>size</i>	size of the vector

**4.16.2.14 clcg\_vecset() [1/2]**

```
void clcg_vecset (
    lcg_complex ** a,
```

```

    lcg_complex b,
    int m,
    int n )

```

set a 2d complex vector's value

#### Parameters

	<i>a</i>	pointer of the matrix
in	<i>b</i>	initial value
in	<i>m</i>	row size of the matrix
in	<i>n</i>	column size of the matrix

#### 4.16.2.15 clcg\_vecset() [2/2]

```

void clcg_vecset (
    lcg_complex * a,
    lcg_complex b,
    int size )

```

set a complex vector's value

#### Parameters

	<i>a</i>	pointer of the vector
in	<i>b</i>	initial value
in	<i>size</i>	vector size

## 4.17 lcg\_complex\_cuda.h File Reference

```

#include "lcg_complex.h"
#include <cuda_runtime.h>
#include <cuComplex.h>

```

### Functions

- [lcg\\_complex\\_cuda2lcg\\_complex](#) (cuDoubleComplex a)  
*Convert cuda complex number to lcg complex number.*
- cuDoubleComplex [lcg2cuda\\_complex](#) (lcg\_complex a)  
*Convert lcg complex number to CUDA complex number.*
- cuDoubleComplex \* [clcg\\_malloc\\_cuda](#) (size\_t n)  
*Locate memory for a cuDoubleComplex pointer type.*
- void [clcg\\_free\\_cuda](#) (cuDoubleComplex \*x)  
*Destroy memory used by the cuDoubleComplex type array.*
- void [clcg\\_vecset\\_cuda](#) (cuDoubleComplex \*a, cuDoubleComplex b, size\_t size)

- set a complex vector's value*

  - `cuComplex clcg_Cscale` (`lcg_float` s, `cuComplex` a)

*Host side function for scale a cuDoubleComplex object.*
- `cuComplex clcg_Csum` (`cuComplex` a, `cuComplex` b)

*Calculate the sum of two cuda complex number. This is a host side function.*
- `cuComplex clcg_Cdiff` (`cuComplex` a, `cuComplex` b)

*Calculate the difference of two cuda complex number. This is a host side function.*
- `cuComplex clcg_Csqrt` (`cuComplex` a)

*Calculate the sqrt() of a cuda complex number.*
- `cuDoubleComplex clcg_Zscale` (`lcg_float` s, `cuDoubleComplex` a)

*Host side function for scale a cuDoubleComplex object.*
- `cuDoubleComplex clcg_Zsum` (`cuDoubleComplex` a, `cuDoubleComplex` b)

*Calculate the sum of two cuda complex number. This is a host side function.*
- `cuDoubleComplex clcg_Zdiff` (`cuDoubleComplex` a, `cuDoubleComplex` b)

*Calculate the difference of two cuda complex number. This is a host side function.*
- `cuDoubleComplex clcg_Zsqrt` (`cuDoubleComplex` a)

*Calculate the sqrt() of a cuda complex number.*
- `void clcg_smCcoo_row2col` (`const int *A_row`, `const int *A_col`, `const cuComplex *A`, `int N`, `int nz`, `int *Ac_row`, `int *Ac_col`, `cuComplex *Ac_val`)

*Convert the indexing sequence of a sparse matrix from the row-major to col-major format.*
- `void clcg_smZcoo_row2col` (`const int *A_row`, `const int *A_col`, `const cuDoubleComplex *A`, `int N`, `int nz`, `int *Ac_row`, `int *Ac_col`, `cuDoubleComplex *Ac_val`)

*Convert the indexing sequence of a sparse matrix from the row-major to col-major format.*
- `void clcg_smCcsr_get_diagonal` (`const int *A_ptr`, `const int *A_col`, `const cuComplex *A_val`, `const int A_len`, `cuComplex *A_diag`, `int bk_size=1024`)

*Extract diagonal elements from a square CUDA sparse matrix that is formatted in the CSR format.*
- `void clcg_smZcsr_get_diagonal` (`const int *A_ptr`, `const int *A_col`, `const cuDoubleComplex *A_val`, `const int A_len`, `cuDoubleComplex *A_diag`, `int bk_size=1024`)

*Extract diagonal elements from a square CUDA sparse matrix that is formatted in the CSR format.*
- `void clcg_vecMvecC_element_wise` (`const cuComplex *a`, `const cuComplex *b`, `cuComplex *c`, `int n`, `int bk_size=1024`)

*Element-wise muplication between two CUDA arries.*
- `void clcg_vecMvecZ_element_wise` (`const cuDoubleComplex *a`, `const cuDoubleComplex *b`, `cuDoubleComplex *c`, `int n`, `int bk_size=1024`)

*Element-wise muplication between two CUDA arries.*
- `void clcg_vecDvecC_element_wise` (`const cuComplex *a`, `const cuComplex *b`, `cuComplex *c`, `int n`, `int bk_size=1024`)

*Element-wise division between two CUDA arries.*
- `void clcg_vecDvecZ_element_wise` (`const cuDoubleComplex *a`, `const cuDoubleComplex *b`, `cuDoubleComplex *c`, `int n`, `int bk_size=1024`)

*Element-wise division between two CUDA arries.*
- `void clcg_vecC_conjugate` (`const cuComplex *a`, `cuComplex *ca`, `int n`, `int bk_size=1024`)

*Return complex conjugates of an input CUDA complex array.*
- `void clcg_vecZ_conjugate` (`const cuDoubleComplex *a`, `cuDoubleComplex *ca`, `int n`, `int bk_size=1024`)

*Return complex conjugates of an input CUDA complex array.*

### 4.17.1 Function Documentation

#### 4.17.1.1 clcg\_Cdiff()

```
cuComplex clcg_Cdiff (
    cuComplex a,
    cuComplex b )
```

Calculate the difference of two cuda complex number. This is a host side function.

##### Parameters

<i>a</i>	Complex number
<i>b</i>	Complex number

##### Returns

cuComplex Difference of the input complex number

#### 4.17.1.2 clcg\_Cscale()

```
cuComplex clcg_Cscale (
    lcg_float s,
    cuComplex a )
```

Host side function for scale a cuDoubleComplex object.

##### Parameters

<i>s</i>	scale factor
<i>a</i>	Complex number

##### Returns

cuComplex scaled complex number

#### 4.17.1.3 clcg\_Csqrt()

```
cuComplex clcg_Csqrt (
    cuComplex a )
```

Calculate the sqrt() of a cuda complex number.

##### Parameters

<i>a</i>	Complex number
----------	----------------



**Returns**

cuComplex root value

**4.17.1.4 clcg\_Csum()**

```
cuComplex clcg_Csum (
    cuComplex a,
    cuComplex b )
```

Calculate the sum of two cuda complex number. This is a host side function.

**Parameters**

<i>a</i>	Complex number
<i>b</i>	Complex number

**Returns**

cuComplex Sum of the input complex number

**4.17.1.5 clcg\_free\_cuda()**

```
void clcg_free_cuda (
    cuDoubleComplex * x )
```

Destroy memory used by the cuDoubleComplex type array.

**Parameters**

<i>x</i>	Pointer of the array.
----------	-----------------------

**4.17.1.6 clcg\_malloc\_cuda()**

```
cuDoubleComplex* clcg_malloc_cuda (
    size_t n )
```

Locate memory for a cuDoubleComplex pointer type.

**Parameters**

in	<i>n</i>	Size of the lcg_float array.
----	----------	------------------------------

**Returns**

Pointer of the array's location.

**4.17.1.7 clcg\_smCcoo\_row2col()**

```
void clcg_smCcoo_row2col (
    const int * A_row,
    const int * A_col,
    const cuComplex * A,
    int N,
    int nz,
    int * Ac_row,
    int * Ac_col,
    cuComplex * Ac_val )
```

Convert the indexing sequence of a sparse matrix from the row-major to col-major format.

**Note**

The sparse matrix is stored in the COO format. This is a host side function.

**Parameters**

<i>A_row</i>	Row index
<i>A_col</i>	Column index
<i>A</i>	Non-zero values of the matrix
<i>N</i>	Row/column length of A
<i>nz</i>	Number of the non-zero values in A
<i>Ac_row</i>	Output row index
<i>Ac_col</i>	Output column index
<i>Ac_val</i>	Non-zero values of the output matrix

**4.17.1.8 clcg\_smCcsr\_get\_diagonal()**

```
void clcg_smCcsr_get_diagonal (
    const int * A_ptr,
    const int * A_col,
    const cuComplex * A_val,
    const int A_len,
    cuComplex * A_diag,
    int bk_size = 1024 )
```

Extract diagonal elements from a square CUDA sparse matrix that is formatted in the CSR format.

**Note**

This is a device side function. All memories must be allocated on the GPU device.

## Parameters

in	<i>A_ptr</i>	Row index pointer
in	<i>A_col</i>	Column index
in	<i>A_val</i>	Non-zero values of the matrix
in	<i>A_len</i>	Dimension of the matrix
	<i>A_diag</i>	Output digonal elements
in	<i>bk_size</i>	Default CUDA block size.

## 4.17.1.9 clcg\_smZcoo\_row2col()

```
void clcg_smZcoo_row2col (
    const int * A_row,
    const int * A_col,
    const cuDoubleComplex * A,
    int N,
    int nz,
    int * Ac_row,
    int * Ac_col,
    cuDoubleComplex * Ac_val )
```

Convert the indexing sequence of a sparse matrix from the row-major to col-major format.

## Note

The sparse matrix is stored in the COO foramt. This is a host side function.

## Parameters

<i>A_row</i>	Row index
<i>A_col</i>	Column index
<i>A</i>	Non-zero values of the matrix
<i>N</i>	Row/column length of A
<i>nz</i>	Number of the non-zero values in A
<i>Ac_row</i>	Output row index
<i>Ac_col</i>	Output column index
<i>Ac_val</i>	Non-zero values of the output matrix

## 4.17.1.10 clcg\_smZcsr\_get\_diagonal()

```
void clcg_smZcsr_get_diagonal (
    const int * A_ptr,
    const int * A_col,
    const cuDoubleComplex * A_val,
    const int A_len,
```

```
cuDoubleComplex * A_diag,
int bk_size = 1024 )
```

Extract diagonal elements from a square CUDA sparse matrix that is formatted in the CSR format.

#### Note

This is a device side function. All memories must be allocated on the GPU device.

#### Parameters

in	<i>A_ptr</i>	Row index pointer
in	<i>A_col</i>	Column index
in	<i>A_val</i>	Non-zero values of the matrix
in	<i>A_len</i>	Dimension of the matrix
	<i>A_diag</i>	Output digonal elements
in	<i>bk_size</i>	Default CUDA block size.

#### 4.17.1.11 clcg\_vecC\_conjugate()

```
void clcg_vecC_conjugate (
    const cuComplex * a,
    cuComplex * ca,
    int n,
    int bk_size = 1024 )
```

Return complex conjugates of an input CUDA complex array.

#### Parameters

	<i>a</i>	Pointer of the input arra
	<i>ca</i>	Pointer of the output array
	<i>n</i>	Length of the arraies
in	<i>bk_size</i>	Default CUDA block size.

#### 4.17.1.12 clcg\_vecDvecC\_element\_wise()

```
void clcg_vecDvecC_element_wise (
    const cuComplex * a,
    const cuComplex * b,
    cuComplex * c,
    int n,
    int bk_size = 1024 )
```

Element-wise division between two CUDA arries.

**Note**

This is a device side function. All memories must be allocated on the GPU device.

**Parameters**

in	<i>a</i>	Pointer of the input array
in	<i>b</i>	Pointer of the input array
	<i>c</i>	Pointer of the output array
in	<i>n</i>	Length of the arraies
in	<i>bk_size</i>	Default CUDA block size.

**4.17.1.13 `clcg_vecDvecZ_element_wise()`**

```
void clcg_vecDvecZ_element_wise (
    const cuDoubleComplex * a,
    const cuDoubleComplex * b,
    cuDoubleComplex * c,
    int n,
    int bk_size = 1024 )
```

Element-wise division between two CUDA arries.

**Note**

This is a device side function. All memories must be allocated on the GPU device.

**Parameters**

in	<i>a</i>	Pointer of the input array
in	<i>b</i>	Pointer of the input array
	<i>c</i>	Pointer of the output array
in	<i>n</i>	Length of the arraies
in	<i>bk_size</i>	Default CUDA block size.

**4.17.1.14 `clcg_vecMvecC_element_wise()`**

```
void clcg_vecMvecC_element_wise (
    const cuComplex * a,
    const cuComplex * b,
    cuComplex * c,
    int n,
    int bk_size = 1024 )
```

Element-wise muplication between two CUDA arries.

**Note**

This is a device side function. All memories must be allocated on the GPU device.

**Parameters**

in	<i>a</i>	Pointer of the input array
in	<i>b</i>	Pointer of the input array
	<i>c</i>	Pointer of the output array
in	<i>n</i>	Length of the arraies
in	<i>bk_size</i>	Default CUDA block size.

**4.17.1.15 clcg\_vecMvecZ\_element\_wise()**

```
void clcg_vecMvecZ_element_wise (
    const cuDoubleComplex * a,
    const cuDoubleComplex * b,
    cuDoubleComplex * c,
    int n,
    int bk_size = 1024 )
```

Element-wise muplication between two CUDA arries.

**Note**

This is a device side function. All memories must be allocated on the GPU device.

**Parameters**

in	<i>a</i>	Pointer of the input array
in	<i>b</i>	Pointer of the input array
	<i>c</i>	Pointer of the output array
in	<i>n</i>	Length of the arraies
in	<i>bk_size</i>	Default CUDA block size.

**4.17.1.16 clcg\_vecset\_cuda()**

```
void clcg_vecset_cuda (
    cuDoubleComplex * a,
    cuDoubleComplex b,
    size_t size )
```

set a complex vector's value

## Parameters

	<i>a</i>	pointer of the vector
in	<i>b</i>	initial value
in	<i>size</i>	vector size

**4.17.1.17 clcg\_vecZ\_conjugate()**

```
void clcg_vecZ_conjugate (
    const cuDoubleComplex * a,
    cuDoubleComplex * ca,
    int n,
    int bk_size = 1024 )
```

Return complex conjugates of an input CUDA complex array.

## Parameters

	<i>a</i>	Pointer of the input arra
	<i>ca</i>	Pointer of the output array
	<i>n</i>	Length of the arraies
in	<i>bk_size</i>	Default CUDA block size.

**4.17.1.18 clcg\_Zdiff()**

```
cuDoubleComplex clcg_Zdiff (
    cuDoubleComplex a,
    cuDoubleComplex b )
```

Calculate the difference of two cuda complex number. This is a host side function.

## Parameters

<i>a</i>	Complex number
<i>b</i>	Complex number

## Returns

cuDoubleComplex Difference of the input complex number

#### 4.17.1.19 clcg\_Zscale()

```
cuDoubleComplex clcg_Zscale (
    lcg_float s,
    cuDoubleComplex a )
```

Host side function for scale a cuDoubleComplex object.

##### Parameters

<i>s</i>	scale factor
<i>a</i>	Complex number

##### Returns

cuDoubleComplex scaled complex number

#### 4.17.1.20 clcg\_Zsqrt()

```
cuDoubleComplex clcg_Zsqrt (
    cuDoubleComplex a )
```

Calculate the sqrt() of a cuda complex number.

##### Parameters

<i>a</i>	Complex number
----------	----------------

##### Returns

cuDoubleComplex root value

#### 4.17.1.21 clcg\_Zsum()

```
cuDoubleComplex clcg_Zsum (
    cuDoubleComplex a,
    cuDoubleComplex b )
```

Calculate the sum of two cuda complex number. This is a host side function.

##### Parameters

<i>a</i>	Complex number
<i>b</i>	Complex number



**Returns**

cuDoubleComplex Sum of the input complex number

**4.17.1.22 cuda2lcg\_complex()**

```
lcg_complex cuda2lcg_complex (
    cuDoubleComplex a )
```

Convert cuda complex number to lcg complex number.

**Parameters**

<i>a</i>	CUDA complex number
----------	---------------------

**Returns**

lcg\_complex lcg complex number

**4.17.1.23 lcg2cuda\_complex()**

```
cuDoubleComplex lcg2cuda_complex (
    lcg_complex a )
```

Convert lcg complex number to CUDA complex number.

**Parameters**

<i>a</i>	lcg complex number
----------	--------------------

**Returns**

cuDoubleComplex CUDA complex number

## 4.18 lcg\_cuda.h File Reference

```
#include "util.h"
#include "algebra_cuda.h"
#include <cublas_v2.h>
#include <cusparse_v2.h>
```

## Typedefs

- typedef void(\* [lcg\\_axfunc\\_cuda\\_ptr](#)) (void \*instance, cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, cusparseDnVecDescr\_t x, cusparseDnVecDescr\_t prod\_Ax, const int n\_size, const int nz\_size)  
*Callback interface for calculating the product of a N\*N matrix 'A' multiplied by a vertical vector 'x'. Note that both A and x are hosted on the GPU device.*
- typedef int(\* [lcg\\_progress\\_cuda\\_ptr](#)) (void \*instance, const [lcg\\_float](#) \*m, const [lcg\\_float](#) converge, const [lcg\\_para](#) \*param, const int n\_size, const int nz\_size, const int k)  
*Callback interface for monitoring the progress and terminate the iteration if necessary. Note that m is hosted on the GPU device.*

## Functions

- int [lcg\\_solver\\_cuda](#) ([lcg\\_axfunc\\_cuda\\_ptr](#) Afp, [lcg\\_progress\\_cuda\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const int n\_size, const int nz\_size, const [lcg\\_para](#) \*param, void \*instance, cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, [lcg\\_solver\\_enum](#) solver\_id=[LCG\\_CG](#))  
*A combined conjugate gradient solver function. Note that both m and B are hosted on the GPU device.*
- int [lcg\\_solver\\_preconditioned\\_cuda](#) ([lcg\\_axfunc\\_cuda\\_ptr](#) Afp, [lcg\\_axfunc\\_cuda\\_ptr](#) Mfp, [lcg\\_progress\\_cuda\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const int n\_size, const int nz\_size, const [lcg\\_para](#) \*param, void \*instance, cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, [lcg\\_solver\\_enum](#) solver\_id=[LCG\\_PCG](#))  
*A combined conjugate gradient solver function. Note that both m and B are hosted on the GPU device.*
- int [lcg\\_solver\\_constrained\\_cuda](#) ([lcg\\_axfunc\\_cuda\\_ptr](#) Afp, [lcg\\_progress\\_cuda\\_ptr](#) Pfp, [lcg\\_float](#) \*m, const [lcg\\_float](#) \*B, const [lcg\\_float](#) \*low, const [lcg\\_float](#) \*hig, const int n\_size, const int nz\_size, const [lcg\\_para](#) \*param, void \*instance, cublasHandle\_t cub\_handle, cusparseHandle\_t cus\_handle, [lcg\\_solver\\_enum](#) solver\_id=[LCG\\_PG](#))  
*A combined conjugate gradient solver function. Note that both m and B are hosted on the GPU device.*

## 4.18.1 Typedef Documentation

### 4.18.1.1 [lcg\\_axfunc\\_cuda\\_ptr](#)

```
typedef void(* lcg\_axfunc\_cuda\_ptr) (void *instance, cublasHandle_t cub_handle, cusparseHandle_t cus_handle, cusparseDnVecDescr_t x, cusparseDnVecDescr_t prod_Ax, const int n_size, const int nz_size)
```

Callback interface for calculating the product of a N\*N matrix 'A' multiplied by a vertical vector 'x'. Note that both A and x are hosted on the GPU device.

#### Parameters

<i>instance</i>	The user data sent for the <a href="#">lcg_solver_cuda()</a> functions by the client.
<i>cub_handle</i>	Handler of the cublas object.
<i>cus_handle</i>	Handlee of the cusparse object.
<i>x</i>	Multiplier of the Ax product.
<i>Ax</i>	Product of A multiplied by x.
<i>n_size</i>	Size of x and column/row numbers of A.

### 4.18.1.2 lcg\_progress\_cuda\_ptr

```
typedef int(* lcg_progress_cuda_ptr) (void *instance, const lcg_float *m, const lcg_float converge,
const lcg_para *param, const int n_size, const int nz_size, const int k)
```

Callback interface for monitoring the progress and terminate the iteration if necessary. Note that m is hosted on the GPU device.

#### Parameters

<i>instance</i>	The user data sent for the <code>lcg_solver()</code> functions by the client.
<i>m</i>	The current solutions.
<i>converge</i>	The current value evaluating the iteration progress.
<i>n_size</i>	The size of the variables
<i>k</i>	The iteration count.

#### Return values

<i>int</i>	Zero to continue the optimization process. Returning a non-zero value will terminate the optimization process.
------------	--

## 4.18.2 Function Documentation

### 4.18.2.1 lcg\_solver\_constrained\_cuda()

```
int lcg_solver_constrained_cuda (
    lcg_axfunc_cuda_ptr Afp,
    lcg_progress_cuda_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
    const lcg_float * low,
    const lcg_float * hig,
    const int n_size,
    const int nz_size,
    const lcg_para * param,
    void * instance,
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    lcg_solver_enum solver_id = LCG_PG )
```

A combined conjugate gradient solver function. Note that both m and B are hosted on the GPU device.

#### Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Mfp</i>	Callback function for calculating the product of 'Mx' for preconditioning.

## Parameters

in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>low</i>	Lower bound of the acceptable solution.
	<i>hig</i>	Higher bound of the acceptable solution.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
in	<i>nz_size</i>	Size of the non-zero element of a cusparse object.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client.
	<i>cub_handle</i>	Handler of the cublas object.
	<i>cus_handle</i>	Handlee of the cusparse object. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.

## Returns

Status of the function.

4.18.2.2 [lcg\\_solver\\_cuda\(\)](#)

```
int lcg_solver_cuda (
    lcg_axfunc_cuda_ptr Afp,
    lcg_progress_cuda_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
    const int n_size,
    const int nz_size,
    const lcg_para * param,
    void * instance,
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    lcg_solver_enum solver_id = LCG_CG )
```

A combined conjugate gradient solver function. Note that both m and B are hosted on the GPU device.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client.
	<i>cub_handle</i>	Handler of the cublas object.
	<i>cus_handle</i>	Handlee of the cusparse object. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.

## Returns

Status of the function.

## 4.18.2.3 lcg\_solver\_preconditioned\_cuda()

```
int lcg_solver_preconditioned_cuda (
    lcg_axfunc_cuda_ptr Afp,
    lcg_axfunc_cuda_ptr Mfp,
    lcg_progress_cuda_ptr Pfp,
    lcg_float * m,
    const lcg_float * B,
    const int n_size,
    const int nz_size,
    const lcg_para * param,
    void * instance,
    cublasHandle_t cub_handle,
    cusparseHandle_t cus_handle,
    lcg_solver_enum solver_id = LCG_PCG )
```

A combined conjugate gradient solver function. Note that both m and B are hosted on the GPU device.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Mfp</i>	Callback function for calculating the product of 'Mx' for preconditioning.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
in	<i>nz_size</i>	Size of the non-zero element of a cusparse object.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client.
	<i>cub_handle</i>	Handler of the cublas object.
	<i>cus_handle</i>	Handlee of the cusparse object. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.

## Returns

Status of the function.

## 4.19 lcg\_eigen.cpp File Reference

```
#include "lcg_eigen.h"
#include "cmath"
#include "config.h"
#include "omp.h"
```

## Typedefs

- typedef int(\* [eigen\\_solver\\_ptr](#)) ([lcg\\_axfunc\\_eigen\\_ptr](#) Afp, [lcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXd &m, const Eigen::VectorXd &B, const [lcg\\_para](#) \*param, void \*instance)

*Callback interface of the conjugate gradient solver.*

- typedef int(\* [eigen\\_solver\\_ptr2](#)) ([lcg\\_axfunc\\_eigen\\_ptr](#) Afp, [lcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXd &m, const Eigen::VectorXd &B, const Eigen::VectorXd &low, const Eigen::VectorXd &high, const [lcg\\_para](#) \*param, void \*instance)

*A combined conjugate gradient solver function.*

## Functions

- int [lcg](#) ([lcg\\_axfunc\\_eigen\\_ptr](#) Afp, [lcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXd &m, const Eigen::VectorXd &B, const [lcg\\_para](#) \*param, void \*instance)

*Conjugate gradient method.*

- int [lcgs](#) ([lcg\\_axfunc\\_eigen\\_ptr](#) Afp, [lcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXd &m, const Eigen::VectorXd &B, const [lcg\\_para](#) \*param, void \*instance)

*Conjugate gradient squared method.*

- int [lbicgstab](#) ([lcg\\_axfunc\\_eigen\\_ptr](#) Afp, [lcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXd &m, const Eigen::VectorXd &B, const [lcg\\_para](#) \*param, void \*instance)

*Biconjugate gradient method.*

- int [lbicgstab2](#) ([lcg\\_axfunc\\_eigen\\_ptr](#) Afp, [lcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXd &m, const Eigen::VectorXd &B, const [lcg\\_para](#) \*param, void \*instance)

*Biconjugate gradient method 2.*

- int [lcg\\_solver\\_eigen](#) ([lcg\\_axfunc\\_eigen\\_ptr](#) Afp, [lcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXd &m, const Eigen::VectorXd &B, const [lcg\\_para](#) \*param, void \*instance, [lcg\\_solver\\_enum](#) solver\_id)

*A combined conjugate gradient solver function.*

- int [lpcg](#) ([lcg\\_axfunc\\_eigen\\_ptr](#) Afp, [lcg\\_axfunc\\_eigen\\_ptr](#) Mfp, [lcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXd &m, const Eigen::VectorXd &B, const [lcg\\_para](#) \*param, void \*instance)

*Preconditioned conjugate gradient method.*

- int [lcg\\_solver\\_preconditioned\\_eigen](#) ([lcg\\_axfunc\\_eigen\\_ptr](#) Afp, [lcg\\_axfunc\\_eigen\\_ptr](#) Mfp, [lcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXd &m, const Eigen::VectorXd &B, const [lcg\\_para](#) \*param, void \*instance, [lcg\\_solver\\_enum](#) solver\_id)

*A combined conjugate gradient solver function.*

- int [lpg](#) ([lcg\\_axfunc\\_eigen\\_ptr](#) Afp, [lcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXd &m, const Eigen::VectorXd &B, const Eigen::VectorXd &low, const Eigen::VectorXd &high, const [lcg\\_para](#) \*param, void \*instance)

*Conjugate gradient method with projected gradient for inequality constraints.*

- int [lspg](#) ([lcg\\_axfunc\\_eigen\\_ptr](#) Afp, [lcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXd &m, const Eigen::VectorXd &B, const Eigen::VectorXd &low, const Eigen::VectorXd &high, const [lcg\\_para](#) \*param, void \*instance)

*Conjugate gradient method with projected gradient for inequality constraints.*

- int [lcg\\_solver\\_constrained\\_eigen](#) ([lcg\\_axfunc\\_eigen\\_ptr](#) Afp, [lcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXd &m, const Eigen::VectorXd &B, const Eigen::VectorXd &low, const Eigen::VectorXd &high, const [lcg\\_para](#) \*param, void \*instance, [lcg\\_solver\\_enum](#) solver\_id)

*A combined conjugate gradient solver function with inequality constraints.*

## 4.19.1 Typedef Documentation

### 4.19.1.1 eigen\_solver\_ptr

```
typedef int(* eigen_solver_ptr) (lcg\_axfunc\_eigen\_ptr Afp, lcg\_progress\_eigen\_ptr Pfp, Eigen::VectorXd &m, const Eigen::VectorXd &B, const lcg\_para *param, void *instance)
```

Callback interface of the conjugate gradient solver.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.

## Returns

Status of the function.

## 4.19.1.2 eigen\_solver\_ptr2

```
typedef int(* eigen_solver_ptr2) (lcg_axfunc_eigen_ptr Afp, lcg_progress_eigen_ptr Pfp, Eigen↵
::VectorXd &m, const Eigen::VectorXd &B, const Eigen::VectorXd &low, const Eigen::VectorXd
&hig, const lcg_para *param, void *instance)
```

A combined conjugate gradient solver function.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>low</i>	The lower boundary of the acceptable solution.
in	<i>hig</i>	The higher boundary of the acceptable solution.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>solver↵ _id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.
	<i>P</i>	Precondition vector (optional expect for the LCG_PCG method). The default value is nullptr.

## Returns

Status of the function.

## 4.19.2 Function Documentation

#### 4.19.2.1 lbicgstab()

```
int lbicgstab (
    lcg_axfunc_eigen_ptr Afp,
    lcg_progress_eigen_ptr Pfp,
    Eigen::VectorXd & m,
    const Eigen::VectorXd & B,
    const lcg_para * param,
    void * instance )
```

Biconjugate gradient method.

##### Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>P</i>	Precondition vector (optional expect for the LCG_PCG method). The default value is nullptr.

##### Returns

Status of the function.

#### 4.19.2.2 lbicgstab2()

```
int lbicgstab2 (
    lcg_axfunc_eigen_ptr Afp,
    lcg_progress_eigen_ptr Pfp,
    Eigen::VectorXd & m,
    const Eigen::VectorXd & B,
    const lcg_para * param,
    void * instance )
```

Biconjugate gradient method 2.

##### Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>P</i>	Precondition vector (optional expect for the LCG_PCG method). The default value is nullptr.



**Returns**

Status of the function.

**4.19.2.3 lcg()**

```
int lcg (
    lcg_axfunc_eigen_ptr Afp,
    lcg_progress_eigen_ptr Pfp,
    Eigen::VectorXd & m,
    const Eigen::VectorXd & B,
    const lcg_para * param,
    void * instance )
```

Conjugate gradient method.

**Parameters**

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>P</i>	Precondition vector (optional expect for the LCG_PCG method). The default value is nullptr.

**Returns**

Status of the function.

**4.19.2.4 lcg\_solver\_constrained\_eigen()**

```
int lcg_solver_constrained_eigen (
    lcg_axfunc_eigen_ptr Afp,
    lcg_progress_eigen_ptr Pfp,
    Eigen::VectorXd & m,
    const Eigen::VectorXd & B,
    const Eigen::VectorXd & low,
    const Eigen::VectorXd & hig,
    const lcg_para * param,
    void * instance,
    lcg_solver_enum solver_id = LCG_PG )
```

A combined conjugate gradient solver function with inequality constraints.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>low</i>	The lower boundary of the acceptable solution.
in	<i>hig</i>	The higher boundary of the acceptable solution.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver↔ _id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.
	<i>P</i>	Precondition vector (optional expect for the LCG_PCG method). The default value is NULL.

## Returns

Status of the function.

4.19.2.5 [lcg\\_solver\\_eigen\(\)](#)

```
int lcg_solver_eigen (
    lcg_axfunc_eigen_ptr Afp,
    lcg_progress_eigen_ptr Pfp,
    Eigen::VectorXd & m,
    const Eigen::VectorXd & B,
    const lcg_para * param,
    void * instance,
    lcg_solver_enum solver_id = LCG_CG )
```

A combined conjugate gradient solver function.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver↔ _id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.

## Returns

Status of the function.

#### 4.19.2.6 lcg\_solver\_preconditioned\_eigen()

```
int lcg_solver_preconditioned_eigen (
    lcg_axfunc_eigen_ptr Afp,
    lcg_axfunc_eigen_ptr Mfp,
    lcg_progress_eigen_ptr Pfp,
    Eigen::VectorXd & m,
    const Eigen::VectorXd & B,
    const lcg_para * param,
    void * instance,
    lcg_solver_enum solver_id = LCG_PCG )
```

A combined conjugate gradient solver function.

##### Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Mfp</i>	Callback function for calculating the product of ' $M^{-1}x$ ', in which M is the preconditioning matrix.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_PCG.

##### Returns

Status of the function.

#### 4.19.2.7 lcgs()

```
int lcgs (
    lcg_axfunc_eigen_ptr Afp,
    lcg_progress_eigen_ptr Pfp,
    Eigen::VectorXd & m,
    const Eigen::VectorXd & B,
    const lcg_para * param,
    void * instance )
```

Conjugate gradient squared method.

##### Note

Algorithm 2 in "Generalized conjugate gradient method" by Fokkema et al. (1996).

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>P</i>	Precondition vector (optional expect for the LCG_PCG method). The default value is nullptr.

## Returns

Status of the function.

## 4.19.2.8 lpcg()

```
int lpcg (
    lcg_axfunc_eigen_ptr Afp,
    lcg_axfunc_eigen_ptr Mfp,
    lcg_progress_eigen_ptr Pfp,
    Eigen::VectorXd & m,
    const Eigen::VectorXd & B,
    const lcg_para * param,
    void * instance )
```

Preconditioned conjugate gradient method.

## Note

Algorithm 1 in "Preconditioned conjugate gradients for singular systems" by Kaasschieter (1988).

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>n_size</i>	Size of the solution vector and objective vector.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>P</i>	Precondition vector (optional expect for the LCG_PCG method). The default value is nullptr.

**Returns**

Status of the function.

**4.19.2.9 lpg()**

```
int lpg (
    lcg_axfunc_eigen_ptr Afp,
    lcg_progress_eigen_ptr Pfp,
    Eigen::VectorXd & m,
    const Eigen::VectorXd & B,
    const Eigen::VectorXd & low,
    const Eigen::VectorXd & hig,
    const lcg_para * param,
    void * instance )
```

Conjugate gradient method with projected gradient for inequality constraints.

**Parameters**

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>low</i>	The lower boundary of the acceptable solution.
in	<i>hig</i>	The higher boundary of the acceptable solution.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>solver↔ _id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.
	<i>P</i>	Precondition vector (optional expect for the LCG_PCG method). The default value is nullptr.

**Returns**

Status of the function.

**4.19.2.10 lspg()**

```
int lspg (
    lcg_axfunc_eigen_ptr Afp,
    lcg_progress_eigen_ptr Pfp,
    Eigen::VectorXd & m,
    const Eigen::VectorXd & B,
    const Eigen::VectorXd & low,
    const Eigen::VectorXd & hig,
    const lcg_para * param,
    void * instance )
```

Conjugate gradient method with projected gradient for inequality constraints.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>low</i>	The lower boundary of the acceptable solution.
in	<i>hig</i>	The higher boundary of the acceptable solution.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'nullptr' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.
	<i>P</i>	Precondition vector (optional expect for the LCG_PCG method). The default value is nullptr.

## Returns

Status of the function.

## 4.20 lcg\_eigen.h File Reference

```
#include "util.h"
#include "algebra_eigen.h"
```

### Typedefs

- typedef void(\* [lcg\\_axfunc\\_eigen\\_ptr](#)) (void \*instance, const Eigen::VectorXd &x, Eigen::VectorXd &prod\_Ax)  
*Callback interface for calculating the product of a N\*N matrix 'A' multiplied by a vertical vector 'x'.*
- typedef int(\* [lcg\\_progress\\_eigen\\_ptr](#)) (void \*instance, const Eigen::VectorXd \*m, const [lcg\\_float](#) converge, const [lcg\\_para](#) \*param, const int k)  
*Callback interface for monitoring the progress and terminate the iteration if necessary.*

### Functions

- int [lcg\\_solver\\_eigen](#) ([lcg\\_axfunc\\_eigen\\_ptr](#) Afp, [lcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXd &m, const Eigen::VectorXd &B, const [lcg\\_para](#) \*param, void \*instance, [lcg\\_solver\\_enum](#) solver\_id=LCG\_CG)  
*A combined conjugate gradient solver function.*
- int [lcg\\_solver\\_preconditioned\\_eigen](#) ([lcg\\_axfunc\\_eigen\\_ptr](#) Afp, [lcg\\_axfunc\\_eigen\\_ptr](#) Mfp, [lcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXd &m, const Eigen::VectorXd &B, const [lcg\\_para](#) \*param, void \*instance, [lcg\\_solver\\_enum](#) solver\_id=LCG\_PCG)  
*A combined conjugate gradient solver function.*
- int [lcg\\_solver\\_constrained\\_eigen](#) ([lcg\\_axfunc\\_eigen\\_ptr](#) Afp, [lcg\\_progress\\_eigen\\_ptr](#) Pfp, Eigen::VectorXd &m, const Eigen::VectorXd &B, const Eigen::VectorXd &low, const Eigen::VectorXd &hig, const [lcg\\_para](#) \*param, void \*instance, [lcg\\_solver\\_enum](#) solver\_id=LCG\_PG)  
*A combined conjugate gradient solver function with inequality constraints.*

## 4.20.1 Typedef Documentation

### 4.20.1.1 lcg\_axfunc\_eigen\_ptr

```
typedef void(* lcg_axfunc_eigen_ptr) (void *instance, const Eigen::VectorXd &x, Eigen::VectorXd
&prod_Ax)
```

Callback interface for calculating the product of a N\*N matrix 'A' multiplied by a vertical vector 'x'.

#### Parameters

<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> functions by the client.
<i>x</i>	Multiplier of the Ax product.
<i>Ax</i>	Product of A multiplied by x.

### 4.20.1.2 lcg\_progress\_eigen\_ptr

```
typedef int(* lcg_progress_eigen_ptr) (void *instance, const Eigen::VectorXd *m, const lcg_float
converge, const lcg_para *param, const int k)
```

Callback interface for monitoring the progress and terminate the iteration if necessary.

#### Parameters

<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> functions by the client.
<i>m</i>	The current solutions.
<i>converge</i>	The current value evaluating the iteration progress.
<i>k</i>	The iteration count.

#### Return values

<i>int</i>	Zero to continue the optimization process. Returning a non-zero value will terminate the optimization process.
------------	--

## 4.20.2 Function Documentation

### 4.20.2.1 lcg\_solver\_constrained\_eigen()

```
int lcg_solver_constrained_eigen (
    lcg_axfunc_eigen_ptr Afp,
```

```

    lcg_progress_eigen_ptr Pfp,
    Eigen::VectorXd & m,
    const Eigen::VectorXd & B,
    const Eigen::VectorXd & low,
    const Eigen::VectorXd & hig,
    const lcg_para * param,
    void * instance,
    lcg_solver_enum solver_id = LCG_PG )

```

A combined conjugate gradient solver function with inequality constraints.

#### Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
in	<i>low</i>	The lower boundary of the acceptable solution.
in	<i>hig</i>	The higher boundary of the acceptable solution.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.
	<i>P</i>	Precondition vector (optional expect for the LCG_PCG method). The default value is NULL.

#### Returns

Status of the function.

#### 4.20.2.2 lcg\_solver\_eigen()

```

int lcg_solver_eigen (
    lcg_axfunc_eigen_ptr Afp,
    lcg_progress_eigen_ptr Pfp,
    Eigen::VectorXd & m,
    const Eigen::VectorXd & B,
    const lcg_para * param,
    void * instance,
    lcg_solver_enum solver_id = LCG_CG )

```

A combined conjugate gradient solver function.

#### Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_CGS.



## Returns

Status of the function.

## 4.20.2.3 lcg\_solver\_preconditioned\_eigen()

```
int lcg_solver_preconditioned_eigen (
    lcg_axfunc_eigen_ptr Afp,
    lcg_axfunc_eigen_ptr Mfp,
    lcg_progress_eigen_ptr Pfp,
    Eigen::VectorXd & m,
    const Eigen::VectorXd & B,
    const lcg_para * param,
    void * instance,
    lcg_solver_enum solver_id = LCG_PCG )
```

A combined conjugate gradient solver function.

## Parameters

in	<i>Afp</i>	Callback function for calculating the product of 'Ax'.
in	<i>Mfp</i>	Callback function for calculating the product of ' $M^{-1}x$ ', in which M is the preconditioning matrix.
in	<i>Pfp</i>	Callback function for monitoring the iteration progress.
	<i>m</i>	Initial solution vector.
	<i>B</i>	Objective vector of the linear system.
	<i>param</i>	Parameter setup for the conjugate gradient methods.
	<i>instance</i>	The user data sent for the <a href="#">lcg_solver()</a> function by the client. This variable is either 'this' for class member functions or 'NULL' for global functions.
	<i>solver_id</i>	Solver type used to solve the linear system. The default value is LCG_PCG.

## Returns

Status of the function.

## 4.21 preconditioner.cpp File Reference

```
#include "preconditioner.h"
#include "cmath"
#include "map"
```

## Functions

- void [lcg\\_incomplete\\_Cholesky\\_half\\_buffsize\\_coo](#) (const int \*row, const int \*col, int nz\_size, int \*lnz\_size)  
Return the number of non-zero elements in the lower triangular part of the input matrix.

- void `lcg_incomplete_cholesky_half_coo` (const int \*row, const int \*col, const `lcg_float` \*val, int N, int nz\_size, int lnz\_size, int \*IC\_row, int \*IC\_col, `lcg_float` \*IC\_val)  
Preform the incomplete Cholesky factorization for a sparse matrix that is saved in the COO format.
- void `lcg_incomplete_cholesky_full_coo` (const int \*row, const int \*col, const `lcg_float` \*val, int N, int nz\_size, int \*IC\_row, int \*IC\_col, `lcg_float` \*IC\_val)  
Preform the incomplete Cholesky factorization for a sparse matrix that is saved in the COO format.
- void `lcg_solve_upper_triangle_coo` (const int \*row, const int \*col, const `lcg_float` \*U, const `lcg_float` \*B, `lcg_float` \*x, int N, int nz\_size)  
Solve the linear system  $Ux = B$ , in which  $U$  is a upper triangle matrix.
- void `lcg_solve_lower_triangle_coo` (const int \*row, const int \*col, const `lcg_float` \*L, const `lcg_float` \*B, `lcg_float` \*x, int N, int nz\_size)  
Solve the linear system  $Lx = B$ , in which  $L$  is a lower triangle matrix.
- bool `lcg_full_rank_coo` (const int \*row, const int \*col, const `lcg_float` \*M, int N, int nz\_size)  
Check to see if a square matrix is full ranked or not. The sparse matrix is stored in the COO format.

## 4.21.1 Function Documentation

### 4.21.1.1 `lcg_full_rank_coo()`

```
bool lcg_full_rank_coo (
    const int * row,
    const int * col,
    const lcg_float * M,
    int N,
    int nz_size )
```

Check to see if a square matrix is full ranked or not. The sparse matrix is stored in the COO format.

#### Parameters

<i>row</i>	Row index of the input sparse matrix.
<i>col</i>	Column index of the input sparse matrix.
<i>M</i>	Non-zero values of the input sparse matrix.
<i>N</i>	Row/Column size of the sparse matrix.
<i>nz_size</i>	Length of the non-zero elements.

#### Returns

true The matrix is full ranked.  
false The matrix is not full ranked.

### 4.21.1.2 `lcg_incomplete_cholesky_full_coo()`

```
void lcg_incomplete_cholesky_full_coo (
    const int * row,
```

```

const int * col,
const lcg_float * val,
int N,
int nz_size,
int * IC_row,
int * IC_col,
lcg_float * IC_val )

```

Preform the incomplete Cholesky factorization for a sparse matrix that is saved in the COO format.

#### Note

The factorized lower and upper triangular matrixes are stored in the lower and upper triangular parts of the output matrix accordingly.

#### Parameters

<i>row</i>	Row index of the input sparse matrix.
<i>col</i>	Column index of the input sparse matrix.
<i>val</i>	Non-zero values of the input sparse matrix.
<i>N</i>	Row/Column size of the sparse matrix.
<i>nz_size</i>	Length of the non-zero elements.
<i>IC_row</i>	Row index of the factorized triangular sparse matrix.
<i>IC_col</i>	Column index of the factorized triangular sparse matrix.
<i>IC_val</i>	Non-zero values of the factorized triangular sparse matrix.

#### 4.21.1.3 lcg\_incomplete\_Cholesky\_half\_buffsize\_coo()

```

void lcg_incomplete_Cholesky_half_buffsize_coo (
    const int * row,
    const int * col,
    int nz_size,
    int * lnz_size )

```

Return the number of non-zero elements in the lower triangular part of the input matrix.

#### Parameters

<i>row[in]</i>	Row index of the input sparse matrix.
<i>col[in]</i>	Column index of the input sparse matrix.
<i>nz_size[in]</i>	Length of the non-zero elements.
<i>lnz_size[out]</i>	Length of the non-zero elements in the lower triangle

#### 4.21.1.4 lcg\_incomplete\_Cholesky\_half\_coo()

```

void lcg_incomplete_Cholesky_half_coo (

```

```

const int * row,
const int * col,
const lcg_float * val,
int N,
int nz_size,
int lnz_size,
int * IC_row,
int * IC_col,
lcg_float * IC_val )

```

Preform the incomplete Cholesky factorization for a sparse matrix that is saved in the COO format.

#### Note

Only the factorized lower triangular matrix is stored in the lower part of the output matrix accordingly.

#### Parameters

<i>row</i>	Row index of the input sparse matrix.
<i>col</i>	Column index of the input sparse matrix.
<i>val</i>	Non-zero values of the input sparse matrix.
<i>N</i>	Row/Column size of the sparse matrix.
<i>nz_size</i>	Length of the non-zero elements.
<i>lnz_size</i>	Legnth of the non-zero elements in the lower triangle
<i>IC_row</i>	Row index of the factorized triangular sparse matrix.
<i>IC_col</i>	Column index of the factorized triangular sparse matrix.
<i>IC_val</i>	Non-zero values of the factorized triangular sparse matrix.

#### 4.21.1.5 lcg\_solve\_lower\_triangle\_coo()

```

void lcg_solve_lower_triangle_coo (
    const int * row,
    const int * col,
    const lcg_float * L,
    const lcg_float * B,
    lcg_float * x,
    int N,
    int nz_size )

```

Solve the linear system  $Lx = B$ , in which  $L$  is a lower triangle matrix.

#### Parameters

<i>row</i>	Row index of the input sparse matrix.
<i>col</i>	Column index of the input sparse matrix.
<i>L</i>	Non-zero values of the input sparse matrix.
<i>B</i>	Object array.
<i>x</i>	The returned solution.
<i>N</i>	Row/Column size of the sparse matrix.
<i>nz_size</i>	Length of the non-zeor elements.

#### 4.21.1.6 lcg\_solve\_upper\_triangle\_coo()

```
void lcg_solve_upper_triangle_coo (
    const int * row,
    const int * col,
    const lcg_float * U,
    const lcg_float * B,
    lcg_float * x,
    int N,
    int nz_size )
```

Solve the linear system  $Ux = B$ , in which  $U$  is a upper triangle matrix.

##### Parameters

<i>row</i>	Row index of the input sparse matrix.
<i>col</i>	Column index of the input sparse matrix.
<i>U</i>	Non-zero values of the input sparse matrix.
<i>B</i>	Object array.
<i>x</i>	The returned solution.
<i>N</i>	Row/Column size of the sparse matrix.
<i>nz_size</i>	Length of the non-zeor elements.

## 4.22 preconditioner.h File Reference

```
#include "algebra.h"
```

### Functions

- void [lcg\\_incomplete\\_Cholesky\\_half\\_buffersize\\_coo](#) (const int \*row, const int \*col, int nz\_size, int \*lnz\_size)  
*Return the number of non-zero elements in the lower triangular part of the input matrix.*
- void [lcg\\_incomplete\\_Cholesky\\_half\\_coo](#) (const int \*row, const int \*col, const lcg\_float \*val, int N, int nz\_size, int lnz\_size, int \*IC\_row, int \*IC\_col, lcg\_float \*IC\_val)  
*Preform the incomplete Cholesky factorization for a sparse matrix that is saved in the COO format.*
- void [lcg\\_incomplete\\_Cholesky\\_full\\_coo](#) (const int \*row, const int \*col, const lcg\_float \*val, int N, int nz\_size, int \*IC\_row, int \*IC\_col, lcg\_float \*IC\_val)  
*Preform the incomplete Cholesky factorization for a sparse matrix that is saved in the COO format.*
- void [lcg\\_solve\\_upper\\_triangle\\_coo](#) (const int \*row, const int \*col, const lcg\_float \*U, const lcg\_float \*B, lcg\_float \*x, int N, int nz\_size)  
*Solve the linear system  $Ux = B$ , in which  $U$  is a upper triangle matrix.*
- void [lcg\\_solve\\_lower\\_triangle\\_coo](#) (const int \*row, const int \*col, const lcg\_float \*L, const lcg\_float \*B, lcg\_float \*x, int N, int nz\_size)  
*Solve the linear system  $Lx = B$ , in which  $L$  is a lower triangle matrix.*
- bool [lcg\\_full\\_rank\\_coo](#) (const int \*row, const int \*col, const lcg\_float \*M, int N, int nz\_size)  
*Check to see if a square matrix is full ranked or not. The sparse matrix is stored in the COO format.*

## 4.22.1 Function Documentation

### 4.22.1.1 lcg\_full\_rank\_coo()

```
bool lcg_full_rank_coo (
    const int * row,
    const int * col,
    const lcg_float * M,
    int N,
    int nz_size )
```

Check to see if a square matrix is full ranked or not. The sparse matrix is stored in the COO format.

#### Parameters

<i>row</i>	Row index of the input sparse matrix.
<i>col</i>	Column index of the input sparse matrix.
<i>M</i>	Non-zero values of the input sparse matrix.
<i>N</i>	Row/Column size of the sparse matrix.
<i>nz_size</i>	Length of the non-zeor elements.

#### Returns

`true` The matrix is full ranked.  
`false` The matrix is not full ranked.

### 4.22.1.2 lcg\_incomplete\_Cholesky\_full\_coo()

```
void lcg_incomplete_Cholesky_full_coo (
    const int * row,
    const int * col,
    const lcg_float * val,
    int N,
    int nz_size,
    int * IC_row,
    int * IC_col,
    lcg_float * IC_val )
```

Preform the incomplete Cholesky factorization for a sparse matrix that is saved in the COO format.

#### Note

The factorized lower and upper triangular matrixes are stored in the lower and upper triangular parts of the output matrix accordingly.

## Parameters

<i>row</i>	Row index of the input sparse matrix.
<i>col</i>	Column index of the input sparse matrix.
<i>val</i>	Non-zero values of the input sparse matrix.
<i>N</i>	Row/Column size of the sparse matrix.
<i>nz_size</i>	Length of the non-zero elements.
<i>IC_row</i>	Row index of the factorized triangular sparse matrix.
<i>IC_col</i>	Column index of the factorized triangular sparse matrix.
<i>IC_val</i>	Non-zero values of the factorized triangular sparse matrix.

4.22.1.3 `lcg_incomplete_Cholesky_half_buffsize_coo()`

```
void lcg_incomplete_Cholesky_half_buffsize_coo (
    const int * row,
    const int * col,
    int nz_size,
    int * lnz_size )
```

Return the number of non-zero elements in the lower triangular part of the input matrix.

## Parameters

<i>row[in]</i>	Row index of the input sparse matrix.
<i>col[in]</i>	Column index of the input sparse matrix.
<i>nz_size[in]</i>	Length of the non-zero elements.
<i>lnz_size[out]</i>	Length of the non-zero elements in the lower triangle

4.22.1.4 `lcg_incomplete_Cholesky_half_coo()`

```
void lcg_incomplete_Cholesky_half_coo (
    const int * row,
    const int * col,
    const lcg_float * val,
    int N,
    int nz_size,
    int lnz_size,
    int * IC_row,
    int * IC_col,
    lcg_float * IC_val )
```

Perform the incomplete Cholesky factorization for a sparse matrix that is saved in the COO format.

## Note

Only the factorized lower triangular matrix is stored in the lower part of the output matrix accordingly.

## Parameters

<i>row</i>	Row index of the input sparse matrix.
<i>col</i>	Column index of the input sparse matrix.
<i>val</i>	Non-zero values of the input sparse matrix.
<i>N</i>	Row/Column size of the sparse matrix.
<i>nz_size</i>	Length of the non-zero elements.
<i>lnz_size</i>	Legnth of the non-zero elements in the lower triangle
<i>IC_row</i>	Row index of the factorized triangular sparse matrix.
<i>IC_col</i>	Column index of the factorized triangular sparse matrix.
<i>IC_val</i>	Non-zero values of the factorized triangular sparse matrix.

4.22.1.5 `lcg_solve_lower_triangle_coo()`

```
void lcg_solve_lower_triangle_coo (
    const int * row,
    const int * col,
    const lcg_float * L,
    const lcg_float * B,
    lcg_float * x,
    int N,
    int nz_size )
```

Solve the linear system  $Lx = B$ , in which  $L$  is a lower triangle matrix.

## Parameters

<i>row</i>	Row index of the input sparse matrix.
<i>col</i>	Column index of the input sparse matrix.
<i>L</i>	Non-zero values of the input sparse matrix.
<i>B</i>	Object array.
<i>x</i>	The returned solution.
<i>N</i>	Row/Column size of the sparse matrix.
<i>nz_size</i>	Length of the non-zeor elements.

4.22.1.6 `lcg_solve_upper_triangle_coo()`

```
void lcg_solve_upper_triangle_coo (
    const int * row,
    const int * col,
    const lcg_float * U,
    const lcg_float * B,
    lcg_float * x,
    int N,
    int nz_size )
```

Solve the linear system  $Ux = B$ , in which  $U$  is a upper triangle matrix.



## Parameters

<i>row</i>	Row index of the input sparse matrix.
<i>col</i>	Column index of the input sparse matrix.
<i>U</i>	Non-zero values of the input sparse matrix.
<i>B</i>	Object array.
<i>x</i>	The returned solution.
<i>N</i>	Row/Column size of the sparse matrix.
<i>nz_size</i>	Length of the non-zero elements.

## 4.23 preconditioner\_cuda.h File Reference

```
#include "lcg_complex_cuda.h"
```

### Functions

- void [clcg\\_incomplete\\_Cholesky\\_cuda\\_half\\_buffsize](#) (const int \*row, const int \*col, int nz\_size, int \*lnz\_size)  
*Return the number of non-zero elements in the lower triangular part of the input matrix.*
- void [clcg\\_incomplete\\_Cholesky\\_cuda\\_half](#) (const int \*row, const int \*col, const cuComplex \*val, int N, int nz\_size, int lnz\_size, int \*IC\_row, int \*IC\_col, cuComplex \*IC\_val)  
*Perform the incomplete Cholesky factorization for a sparse matrix that is saved in the COO format.*
- void [clcg\\_incomplete\\_Cholesky\\_cuda\\_half](#) (const int \*row, const int \*col, const cuDoubleComplex \*val, int N, int nz\_size, int lnz\_size, int \*IC\_row, int \*IC\_col, cuDoubleComplex \*IC\_val)  
*Perform the incomplete Cholesky factorization for a sparse matrix that is saved in the COO format.*
- void [clcg\\_incomplete\\_Cholesky\\_cuda\\_full](#) (const int \*row, const int \*col, const cuDoubleComplex \*val, int N, int nz\_size, int \*IC\_row, int \*IC\_col, cuDoubleComplex \*IC\_val)  
*Perform the incomplete Cholesky factorization for a sparse matrix that is saved in the COO format.*

### 4.23.1 Function Documentation

#### 4.23.1.1 [clcg\\_incomplete\\_Cholesky\\_cuda\\_full\(\)](#)

```
void clcg_incomplete_Cholesky_cuda_full (
    const int * row,
    const int * col,
    const cuDoubleComplex * val,
    int N,
    int nz_size,
    int * IC_row,
    int * IC_col,
    cuDoubleComplex * IC_val )
```

Perform the incomplete Cholesky factorization for a sparse matrix that is saved in the COO format.

#### Note

The factorized lower and upper triangular matrixes are stored in the lower and upper triangular parts of the output matrix accordingly.

## Parameters

<i>row</i>	Row index of the input sparse matrix.
<i>col</i>	Column index of the input sparse matrix.
<i>val</i>	Non-zero values of the input sparse matrix.
<i>N</i>	Row/Column size of the sparse matrix.
<i>nz_size</i>	Length of the non-zero elements.
<i>IC_row</i>	Row index of the factorized triangular sparse matrix.
<i>IC_col</i>	Column index of the factorized triangular sparse matrix.
<i>IC_val</i>	Non-zero values of the factorized triangular sparse matrix.

4.23.1.2 `clcg_incomplete_Cholesky_cuda_half()` [1/2]

```
void clcg_incomplete_Cholesky_cuda_half (
    const int * row,
    const int * col,
    const cuComplex * val,
    int N,
    int nz_size,
    int lnz_size,
    int * IC_row,
    int * IC_col,
    cuComplex * IC_val )
```

Perform the incomplete Cholesky factorization for a sparse matrix that is saved in the COO format.

## Note

Only the factorized lower triangular matrix is stored in the lower part of the output matrix accordingly.

## Parameters

<i>row</i>	Row index of the input sparse matrix.
<i>col</i>	Column index of the input sparse matrix.
<i>val</i>	Non-zero values of the input sparse matrix.
<i>N</i>	Row/Column size of the sparse matrix.
<i>nz_size</i>	Length of the non-zero elements.
<i>lnz_size</i>	Length of the non-zero elements in the lower triangle
<i>IC_row</i>	Row index of the factorized triangular sparse matrix.
<i>IC_col</i>	Column index of the factorized triangular sparse matrix.
<i>IC_val</i>	Non-zero values of the factorized triangular sparse matrix.

4.23.1.3 `clcg_incomplete_Cholesky_cuda_half()` [2/2]

```
void clcg_incomplete_Cholesky_cuda_half (
```

```

const int * row,
const int * col,
const cuDoubleComplex * val,
int N,
int nz_size,
int lnz_size,
int * IC_row,
int * IC_col,
cuDoubleComplex * IC_val )

```

Preform the incomplete Cholesky factorization for a sparse matrix that is saved in the COO format.

#### Note

Only the factorized lower triangular matrix is stored in the lower part of the output matrix accordingly.

#### Parameters

<i>row</i>	Row index of the input sparse matrix.
<i>col</i>	Column index of the input sparse matrix.
<i>val</i>	Non-zero values of the input sparse matrix.
<i>N</i>	Row/Column size of the sparse matrix.
<i>nz_size</i>	Length of the non-zero elements.
<i>lnz_size</i>	Legnth of the non-zero elements in the lower triangle
<i>IC_row</i>	Row index of the factorized triangular sparse matrix.
<i>IC_col</i>	Column index of the factorized triangular sparse matrix.
<i>IC_val</i>	Non-zero values of the factorized triangular sparse matrix.

#### 4.23.1.4 clcg\_incomplete\_Cholesky\_cuda\_half\_buffsize()

```

void clcg_incomplete_Cholesky_cuda_half_buffsize (
    const int * row,
    const int * col,
    int nz_size,
    int * lnz_size )

```

Return the number of non-zero elements in the lower triangular part of the input matrix.

#### Parameters

<i>row[in]</i>	Row index of the input sparse matrix.
<i>col[in]</i>	Column index of the input sparse matrix.
<i>nz_size[in]</i>	Length of the non-zero elements.
<i>lnz_size[out]</i>	Legnth of the non-zero elements in the lower triangle

## 4.24 preconditioner\_eigen.cpp File Reference

```
#include "preconditioner_eigen.h"
#include "exception"
#include "stdexcept"
#include "vector"
#include "iostream"
```

### Typedefs

- typedef Eigen::Triplet< int > [triplet\\_bi](#)
- typedef Eigen::Triplet< double > [triplet\\_d](#)
- typedef Eigen::Triplet< std::complex< double > > [triplet\\_cd](#)

### Functions

- void [lcg\\_Cholesky](#) (const Eigen::MatrixXd &A, Eigen::MatrixXd &L)  
*Perform the Cholesky decomposition and return the lower triangular matrix.*
- void [clcg\\_Cholesky](#) (const Eigen::MatrixXcd &A, Eigen::MatrixXcd &L)  
*Perform the Cholesky decomposition and return the lower triangular matrix.*
- void [lcg\\_invert\\_lower\\_triangle](#) (const Eigen::MatrixXd &L, Eigen::MatrixXd &Linv)  
*Calculate the invert of a lower triangle matrix (Full rank only).*
- void [lcg\\_invert\\_upper\\_triangle](#) (const Eigen::MatrixXd &U, Eigen::MatrixXd &Uinv)  
*Calculate the invert of a upper triangle matrix (Full rank only).*
- void [clcg\\_invert\\_lower\\_triangle](#) (const Eigen::MatrixXcd &L, Eigen::MatrixXcd &Linv)  
*Calculate the invert of a lower triangle matrix (Full rank only).*
- void [clcg\\_invert\\_upper\\_triangle](#) (const Eigen::MatrixXcd &U, Eigen::MatrixXcd &Uinv)  
*Calculate the invert of a upper triangle matrix (Full rank only).*
- void [lcg\\_incomplete\\_Cholesky](#) (const Eigen::SparseMatrix< double, Eigen::RowMajor > &A, Eigen::SparseMatrix< double, Eigen::RowMajor > &L, size\_t fill)  
*Calculate the incomplete Cholesky decomposition and return the lower triangular matrix.*
- void [clcg\\_incomplete\\_Cholesky](#) (const Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > &A, Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > &L, size\_t fill)  
*Calculate the incomplete Cholesky decomposition and return the lower triangular matrix.*
- void [lcg\\_incomplete\\_LU](#) (const Eigen::SparseMatrix< double, Eigen::RowMajor > &A, Eigen::SparseMatrix< double, Eigen::RowMajor > &L, Eigen::SparseMatrix< double, Eigen::RowMajor > &U, size\_t fill)  
*Calculate the incomplete LU factorizations.*
- void [clcg\\_incomplete\\_LU](#) (const Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > &A, Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > &L, Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > &U, size\_t fill)  
*Calculate the incomplete LU factorizations.*
- void [lcg\\_solve\\_lower\\_triangle](#) (const Eigen::SparseMatrix< double, Eigen::RowMajor > &L, const Eigen::VectorXd &B, Eigen::VectorXd &X)  
*Solve the linear system  $Lx = B$ , in which  $L$  is a lower triangle matrix.*
- void [lcg\\_solve\\_upper\\_triangle](#) (const Eigen::SparseMatrix< double, Eigen::RowMajor > &U, const Eigen::VectorXd &B, Eigen::VectorXd &X)  
*Solve the linear system  $Ux = B$ , in which  $U$  is a upper triangle matrix.*
- void [clcg\\_solve\\_lower\\_triangle](#) (const Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > &L, const Eigen::VectorXcd &B, Eigen::VectorXcd &X)  
*Solve the linear system  $Lx = B$ , in which  $L$  is a lower triangle matrix.*
- void [clcg\\_solve\\_upper\\_triangle](#) (const Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > &U, const Eigen::VectorXcd &B, Eigen::VectorXcd &X)  
*Solve the linear system  $Ux = B$ , in which  $U$  is a upper triangle matrix.*

## 4.24.1 Typedef Documentation

### 4.24.1.1 triplet\_bl

```
typedef Eigen::Triplet<int> triplet_bl
```

### 4.24.1.2 triplet\_cd

```
typedef Eigen::Triplet<std::complex<double> > triplet_cd
```

### 4.24.1.3 triplet\_d

```
typedef Eigen::Triplet<double> triplet_d
```

## 4.24.2 Function Documentation

### 4.24.2.1 clcg\_Cholesky()

```
void clcg_Cholesky (
    const Eigen::MatrixXcd & A,
    Eigen::MatrixXcd & L )
```

Perform the Cholesky decomposition and return the lower triangular matrix.

#### Note

This could serve as a direct solver.

#### Parameters

in	$A$	The input matrix. Must be full rank and symmetric (aka. $A = A^T$ )
	$L$	The output low triangular matrix

#### 4.24.2.2 clcg\_incomplete\_Cholesky()

```
void clcg_incomplete_Cholesky (
    const Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > & A,
    Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > & L,
    size_t fill = 0 )
```

Calculate the incomplete Cholesky decomposition and return the lower triangular matrix.

##### Parameters

<i>in</i>	<i>A</i>	The input sparse matrix. Must be full rank and symmetric (aka. $A = A^T$ )
	<i>L</i>	The output lower triangular matrix
	<i>fill</i>	The fill-in number of the output sparse matrix. No fill-in reduction will be processed if this variable is set to zero.

#### 4.24.2.3 clcg\_incomplete\_LU()

```
void clcg_incomplete_LU (
    const Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > & A,
    Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > & L,
    Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > & U,
    size_t fill = 0 )
```

Calculate the incomplete LU factorizations.

##### Parameters

<i>A</i>	The input sparse matrix. Must be full rank.
<i>L</i>	The output lower triangular matrix.
<i>U</i>	The output upper triangular matrix.
<i>fill</i>	The fill-in number of the output sparse matrix. No fill-in reduction will be processed if this variable is set to zero.

#### 4.24.2.4 clcg\_invert\_lower\_triangle()

```
void clcg_invert_lower_triangle (
    const Eigen::MatrixXcd & L,
    Eigen::MatrixXcd & Linv )
```

Calculate the invert of a lower triangle matrix (Full rank only).

##### Parameters

<i>L</i>	The operating lower triangle matrix
<i>Linv</i>	The inverted lower triangle matrix

**4.24.2.5 clcg\_invert\_upper\_triangle()**

```
void clcg_invert_upper_triangle (
    const Eigen::MatrixXcd & U,
    Eigen::MatrixXcd & Uinv )
```

Calculate the invert of a upper triangle matrix (Full rank only).

**Parameters**

<i>U</i>	The operating upper triangle matrix
<i>Uinv</i>	The inverted upper triangle matrix

**4.24.2.6 clcg\_solve\_lower\_triangle()**

```
void clcg_solve_lower_triangle (
    const Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > & L,
    const Eigen::VectorXcd & B,
    Eigen::VectorXcd & X )
```

Solve the linear system  $Lx = B$ , in which L is a lower triangle matrix.

**Parameters**

<i>L</i>	The input lower triangle matrix
<i>B</i>	The object vector
<i>X</i>	The solution vector

**4.24.2.7 clcg\_solve\_upper\_triangle()**

```
void clcg_solve_upper_triangle (
    const Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > & U,
    const Eigen::VectorXcd & B,
    Eigen::VectorXcd & X )
```

Solve the linear system  $Ux = B$ , in which U is a upper triangle matrix.

**Parameters**

<i>U</i>	The input upper triangle matrix
<i>B</i>	The object vector
<i>X</i>	The solution vector

#### 4.24.2.8 lcg\_Cholesky()

```
void lcg_Cholesky (
    const Eigen::MatrixXd & A,
    Eigen::MatrixXd & L )
```

Perform the Cholesky decomposition and return the lower triangular matrix.

##### Note

This could serve as a direct solver.

##### Parameters

<i>A</i>	The input matrix. Must be full rank and symmetric (aka. $A = A^T$ )
<i>L</i>	The output low triangular matrix

#### 4.24.2.9 lcg\_incomplete\_Cholesky()

```
void lcg_incomplete_Cholesky (
    const Eigen::SparseMatrix< double, Eigen::RowMajor > & A,
    Eigen::SparseMatrix< double, Eigen::RowMajor > & L,
    size_t fill = 0 )
```

Calculate the incomplete Cholesky decomposition and return the lower triangular matrix.

##### Parameters

<i>in</i>	<i>A</i>	The input sparse matrix. Must be full rank and symmetric (aka. $A = A^T$ )
	<i>L</i>	The output lower triangular matrix
	<i>fill</i>	The fill-in number of the output sparse matrix. No fill-in reduction will be processed if this variable is set to zero.

#### 4.24.2.10 lcg\_incomplete\_LU()

```
void lcg_incomplete_LU (
    const Eigen::SparseMatrix< double, Eigen::RowMajor > & A,
    Eigen::SparseMatrix< double, Eigen::RowMajor > & L,
    Eigen::SparseMatrix< double, Eigen::RowMajor > & U,
    size_t fill = 0 )
```

Calculate the incomplete LU factorizations.



## Parameters

<i>A</i>	The input sparse matrix. Must be full rank.
<i>L</i>	The output lower triangular matrix.
<i>U</i>	The output upper triangular matrix.
<i>fill</i>	The fill-in number of the output sparse matrix. No fill-in reduction will be processed if this variable is set to zero.

## 4.24.2.11 lcg\_invert\_lower\_triangle()

```
void lcg_invert_lower_triangle (
    const Eigen::MatrixXd & L,
    Eigen::MatrixXd & Linv )
```

Calculate the invert of a lower triangle matrix (Full rank only).

## Parameters

<i>L</i>	The operating lower triangle matrix
<i>Linv</i>	The inverted lower triangle matrix

## 4.24.2.12 lcg\_invert\_upper\_triangle()

```
void lcg_invert_upper_triangle (
    const Eigen::MatrixXd & U,
    Eigen::MatrixXd & Uinv )
```

Calculate the invert of a upper triangle matrix (Full rank only).

## Parameters

<i>U</i>	The operating upper triangle matrix
<i>Uinv</i>	The inverted upper triangle matrix

## 4.24.2.13 lcg\_solve\_lower\_triangle()

```
void lcg_solve_lower_triangle (
    const Eigen::SparseMatrix< double, Eigen::RowMajor > & L,
    const Eigen::VectorXd & B,
    Eigen::VectorXd & X )
```

Solve the linear system  $Lx = B$ , in which  $L$  is a lower triangle matrix.

## Parameters

$L$	The input lower triangle matrix
$B$	The object vector
$X$	The solution vector

4.24.2.14 `lcg_solve_upper_triangle()`

```
void lcg_solve_upper_triangle (
    const Eigen::SparseMatrix< double, Eigen::RowMajor > & U,
    const Eigen::VectorXd & B,
    Eigen::VectorXd & X )
```

Solve the linear system  $Ux = B$ , in which  $U$  is a upper triangle matrix.

## Parameters

$U$	The input upper triangle matrix
$B$	The object vector
$X$	The solution vector

4.25 `preconditioner_eigen.h` File Reference

```
#include "complex"
#include "Eigen/Dense"
#include "Eigen/SparseCore"
```

## Functions

- void `lcg_Cholesky` (const Eigen::MatrixXd &A, Eigen::MatrixXd &L)  
*Perform the Cholesky decomposition and return the lower triangular matrix.*
- void `clcg_Cholesky` (const Eigen::MatrixXcd &A, Eigen::MatrixXcd &L)  
*Perform the Cholesky decomposition and return the lower triangular matrix.*
- void `lcg_invert_lower_triangle` (const Eigen::MatrixXd &L, Eigen::MatrixXd &Linv)  
*Calculate the invert of a lower triangle matrix (Full rank only).*
- void `lcg_invert_upper_triangle` (const Eigen::MatrixXd &U, Eigen::MatrixXd &Uinv)  
*Calculate the invert of a upper triangle matrix (Full rank only).*
- void `clcg_invert_lower_triangle` (const Eigen::MatrixXcd &L, Eigen::MatrixXcd &Linv)  
*Calculate the invert of a lower triangle matrix (Full rank only).*
- void `clcg_invert_upper_triangle` (const Eigen::MatrixXcd &U, Eigen::MatrixXcd &Uinv)  
*Calculate the invert of a upper triangle matrix (Full rank only).*
- void `lcg_incomplete_Cholesky` (const Eigen::SparseMatrix< double, Eigen::RowMajor > &A, Eigen::SparseMatrix< double, Eigen::RowMajor > &L, size\_t fill=0)  
*Calculate the incomplete Cholesky decomposition and return the lower triangular matrix.*

- void [clcg\\_incomplete\\_Cholesky](#) (const Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > &A, Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > &L, size\_t fill=0)  
Calculate the incomplete Cholesky decomposition and return the lower triangular matrix.
- void [lcg\\_incomplete\\_LU](#) (const Eigen::SparseMatrix< double, Eigen::RowMajor > &A, Eigen::SparseMatrix< double, Eigen::RowMajor > &L, Eigen::SparseMatrix< double, Eigen::RowMajor > &U, size\_t fill=0)  
Calculate the incomplete LU factorizations.
- void [clcg\\_incomplete\\_LU](#) (const Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > &A, Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > &L, Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > &U, size\_t fill=0)  
Calculate the incomplete LU factorizations.
- void [lcg\\_solve\\_lower\\_triangle](#) (const Eigen::SparseMatrix< double, Eigen::RowMajor > &L, const Eigen::VectorX< double > &B, Eigen::VectorX< double > &X)  
Solve the linear system  $Lx = B$ , in which  $L$  is a lower triangle matrix.
- void [lcg\\_solve\\_upper\\_triangle](#) (const Eigen::SparseMatrix< double, Eigen::RowMajor > &U, const Eigen::VectorX< double > &B, Eigen::VectorX< double > &X)  
Solve the linear system  $Ux = B$ , in which  $U$  is a upper triangle matrix.
- void [clcg\\_solve\\_lower\\_triangle](#) (const Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > &L, const Eigen::VectorXcd &B, Eigen::VectorXcd &X)  
Solve the linear system  $Lx = B$ , in which  $L$  is a lower triangle matrix.
- void [clcg\\_solve\\_upper\\_triangle](#) (const Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > &U, const Eigen::VectorXcd &B, Eigen::VectorXcd &X)  
Solve the linear system  $Ux = B$ , in which  $U$  is a upper triangle matrix.

## 4.25.1 Function Documentation

### 4.25.1.1 clcg\_Cholesky()

```
void clcg_Cholesky (
    const Eigen::MatrixXcd & A,
    Eigen::MatrixXcd & L )
```

Perform the Cholesky decomposition and return the lower triangular matrix.

#### Note

This could serve as a direct solver.

#### Parameters

in	$A$	The input matrix. Must be full rank and symmetric (aka. $A = A^T$ )
	$L$	The output low triangular matrix

### 4.25.1.2 clcg\_incomplete\_Cholesky()

```
void clcg_incomplete_Cholesky (
    const Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > & A,
```

```
Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > & L,
size_t fill = 0 )
```

Calculate the incomplete Cholesky decomposition and return the lower triangular matrix.

#### Parameters

<i>in</i>	<i>A</i>	The input sparse matrix. Must be full rank and symmetric (aka. $A = A^T$ )
	<i>L</i>	The output lower triangular matrix
	<i>fill</i>	The fill-in number of the output sparse matrix. No fill-in reduction will be processed if this variable is set to zero.

#### 4.25.1.3 clcg\_incomplete\_LU()

```
void clcg_incomplete_LU (
    const Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > & A,
    Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > & L,
    Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > & U,
    size_t fill = 0 )
```

Calculate the incomplete LU factorizations.

#### Parameters

<i>A</i>	The input sparse matrix. Must be full rank.
<i>L</i>	The output lower triangular matrix.
<i>U</i>	The output upper triangular matrix.
<i>fill</i>	The fill-in number of the output sparse matrix. No fill-in reduction will be processed if this variable is set to zero.

#### 4.25.1.4 clcg\_invert\_lower\_triangle()

```
void clcg_invert_lower_triangle (
    const Eigen::MatrixXcd & L,
    Eigen::MatrixXcd & Linv )
```

Calculate the invert of a lower triangle matrix (Full rank only).

#### Parameters

<i>L</i>	The operating lower triangle matrix
<i>Linv</i>	The inverted lower triangle matrix

## 4.25.1.5 clcg\_invert\_upper\_triangle()

```
void clcg_invert_upper_triangle (
    const Eigen::MatrixXcd & U,
    Eigen::MatrixXcd & Uinv )
```

Calculate the invert of a upper triangle matrix (Full rank only).

## Parameters

<i>U</i>	The operating upper triangle matrix
<i>Uinv</i>	The inverted upper triangle matrix

## 4.25.1.6 clcg\_solve\_lower\_triangle()

```
void clcg_solve_lower_triangle (
    const Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > & L,
    const Eigen::VectorXcd & B,
    Eigen::VectorXcd & X )
```

Solve the linear system  $Lx = B$ , in which  $L$  is a lower triangle matrix.

## Parameters

<i>L</i>	The input lower triangle matrix
<i>B</i>	The object vector
<i>X</i>	The solution vector

## 4.25.1.7 clcg\_solve\_upper\_triangle()

```
void clcg_solve_upper_triangle (
    const Eigen::SparseMatrix< std::complex< double >, Eigen::RowMajor > & U,
    const Eigen::VectorXcd & B,
    Eigen::VectorXcd & X )
```

Solve the linear system  $Ux = B$ , in which  $U$  is a upper triangle matrix.

## Parameters

<i>U</i>	The input upper triangle matrix
<i>B</i>	The object vector
<i>X</i>	The solution vector

#### 4.25.1.8 lcg\_Cholesky()

```
void lcg_Cholesky (
    const Eigen::MatrixXd & A,
    Eigen::MatrixXd & L )
```

Perform the Cholesky decomposition and return the lower triangular matrix.

##### Note

This could serve as a direct solver.

##### Parameters

<i>A</i>	The input matrix. Must be full rank and symmetric (aka. $A = A^T$ )
<i>L</i>	The output low triangular matrix

#### 4.25.1.9 lcg\_incomplete\_Cholesky()

```
void lcg_incomplete_Cholesky (
    const Eigen::SparseMatrix< double, Eigen::RowMajor > & A,
    Eigen::SparseMatrix< double, Eigen::RowMajor > & L,
    size_t fill = 0 )
```

Calculate the incomplete Cholesky decomposition and return the lower triangular matrix.

##### Parameters

<i>in</i>	<i>A</i>	The input sparse matrix. Must be full rank and symmetric (aka. $A = A^T$ )
	<i>L</i>	The output lower triangular matrix
	<i>fill</i>	The fill-in number of the output sparse matrix. No fill-in reduction will be processed if this variable is set to zero.

#### 4.25.1.10 lcg\_incomplete\_LU()

```
void lcg_incomplete_LU (
    const Eigen::SparseMatrix< double, Eigen::RowMajor > & A,
    Eigen::SparseMatrix< double, Eigen::RowMajor > & L,
    Eigen::SparseMatrix< double, Eigen::RowMajor > & U,
    size_t fill = 0 )
```

Calculate the incomplete LU factorizations.

##### Parameters

<i>A</i>	The input sparse matrix. Must be full rank.
----------	---

## Parameters

<i>L</i>	The output lower triangular matrix.
<i>U</i>	The output upper triangular matrix.
<i>fill</i>	The fill-in number of the output sparse matrix. No fill-in reduction will be processed if this variable is set to zero.

## 4.25.1.11 lcg\_invert\_lower\_triangle()

```
void lcg_invert_lower_triangle (
    const Eigen::MatrixXd & L,
    Eigen::MatrixXd & Linv )
```

Calculate the invert of a lower triangle matrix (Full rank only).

## Parameters

<i>L</i>	The operating lower triangle matrix
<i>Linv</i>	The inverted lower triangle matrix

## 4.25.1.12 lcg\_invert\_upper\_triangle()

```
void lcg_invert_upper_triangle (
    const Eigen::MatrixXd & U,
    Eigen::MatrixXd & Uinv )
```

Calculate the invert of a upper triangle matrix (Full rank only).

## Parameters

<i>U</i>	The operating upper triangle matrix
<i>Uinv</i>	The inverted upper triangle matrix

## 4.25.1.13 lcg\_solve\_lower\_triangle()

```
void lcg_solve_lower_triangle (
    const Eigen::SparseMatrix< double, Eigen::RowMajor > & L,
    const Eigen::VectorXd & B,
    Eigen::VectorXd & X )
```

Solve the linear system  $Lx = B$ , in which  $L$  is a lower triangle matrix.

**Parameters**

$L$	The input lower triangle matrix
$B$	The object vector
$X$	The solution vector

**4.25.1.14 lcg\_solve\_upper\_triangle()**

```
void lcg_solve_upper_triangle (
    const Eigen::SparseMatrix< double, Eigen::RowMajor > & U,
    const Eigen::VectorXd & B,
    Eigen::VectorXd & X )
```

Solve the linear system  $Ux = B$ , in which  $U$  is a upper triangle matrix.

**Parameters**

$U$	The input upper triangle matrix
$B$	The object vector
$X$	The solution vector

**4.26 solver.cpp File Reference**

```
#include "solver.h"
#include "ctime"
#include "iostream"
#include "config.h"
#include "omp.h"
```

**4.27 solver.h File Reference**

```
#include "lcg.h"
#include "clcg.h"
```

**Data Structures**

- class [LCG\\_Solver](#)  
*Linear conjugate gradient solver class.*
- class [CLCG\\_Solver](#)  
*Complex linear conjugate gradient solver class.*



## 4.28 solver\_cuda.h File Reference

```
#include "lcg_cuda.h"
#include "clcg_cuda.h"
#include "clcg_cudaf.h"
```

### Data Structures

- class [LCG\\_CUDA\\_Solver](#)  
*Linear conjugate gradient solver class.*
- class [CLCG\\_CUDAF\\_Solver](#)  
*Complex linear conjugate gradient solver class.*
- class [CLCG\\_CUDA\\_Solver](#)  
*Complex linear conjugate gradient solver class.*

## 4.29 solver\_eigen.cpp File Reference

```
#include "solver_eigen.h"
#include "cmath"
#include "ctime"
#include "iostream"
#include "config.h"
#include "omp.h"
```

## 4.30 solver\_eigen.h File Reference

```
#include "lcg_eigen.h"
#include "clcg_eigen.h"
```

### Data Structures

- class [LCG\\_EIGEN\\_Solver](#)  
*Linear conjugate gradient solver class.*
- class [CLCG\\_EIGEN\\_Solver](#)  
*Complex linear conjugate gradient solver class.*

## 4.31 util.cpp File Reference

```
#include "iostream"
#include "exception"
#include "stdexcept"
#include "util.h"
```

## Functions

- [lcg\\_para](#) [lcg\\_default\\_parameters](#) ()  
*Return a [lcg\\_para](#) type instance with default values.*
- [lcg\\_solver\\_enum](#) [lcg\\_select\\_solver](#) (std::string slr\_char)  
*Select a type of solver according to the name.*
- void [lcg\\_error\\_str](#) (int er\_index, bool er\_throw)  
*Display or throw out a string explanation for the [lcg\\_solver\(\)](#) function's return values.*
- [clcg\\_para](#) [clcg\\_default\\_parameters](#) ()  
*Return a [clcg\\_para](#) type instance with default values.*
- [clcg\\_solver\\_enum](#) [clcg\\_select\\_solver](#) (std::string slr\_char)  
*Select a type of solver according to the name.*
- void [clcg\\_error\\_str](#) (int er\_index, bool er\_throw)  
*Display or throw out a string explanation for the [clcg\\_solver\(\)](#) function's return values.*

### 4.31.1 Function Documentation

#### 4.31.1.1 [clcg\\_default\\_parameters\(\)](#)

```
clcg\_para clcg\_default\_parameters ( )
```

Return a [clcg\\_para](#) type instance with default values.

Users can use this function to get default parameters' value for the complex conjugate gradient methods.

##### Returns

A [clcg\\_para](#) type instance.

#### 4.31.1.2 [clcg\\_error\\_str\(\)](#)

```
void clcg\_error\_str (
    int er_index,
    bool er_throw = false )
```

Display or throw out a string explanation for the [clcg\\_solver\(\)](#) function's return values.

##### Parameters

in	<i>er_index</i>	The error index returned by the <a href="#">lcg_solver()</a> function.
in	<i>er_throw</i>	throw out a char string of the explanation.

**Returns**

A string explanation of the error.

**4.31.1.3 clcg\_select\_solver()**

```
clcg_solver_enum clcg_select_solver (
    std::string slr_char )
```

Select a type of solver according to the name.

**Parameters**

in	<i>slr_char</i>	Name of the solver
----	-----------------	--------------------

**Returns**

The clcg solver enum.

**4.31.1.4 lcg\_default\_parameters()**

```
lcg_para lcg_default_parameters ( )
```

Return a [lcg\\_para](#) type instance with default values.

Users can use this function to get default parameters' value for the conjugate gradient methods.

**Returns**

A [lcg\\_para](#) type instance.

**4.31.1.5 lcg\_error\_str()**

```
void lcg_error_str (
    int er_index,
    bool er_throw = false )
```

Display or throw out a string explanation for the [lcg\\_solver\(\)](#) function's return values.

**Parameters**

in	<i>er_index</i>	The error index returned by the <a href="#">lcg_solver()</a> function.
in	<i>er_throw</i>	throw out a char string of the explanation.

**Returns**

A string explanation of the error.

**4.31.1.6 lcg\_select\_solver()**

```
lcg_solver_enum lcg_select_solver (
    std::string slr_char )
```

Select a type of solver according to the name.

**Parameters**

in	slr_char	Name of the solver
----	----------	--------------------

**Returns**

The lcg solver enum.

**4.32 util.h File Reference**

```
#include "string"
#include "algebra.h"
```

**Data Structures**

- struct [lcg\\_para](#)  
*Parameters of the conjugate gradient methods.*
- struct [clcg\\_para](#)  
*Parameters of the conjugate gradient methods.*

**Enumerations**

- enum [lcg\\_solver\\_enum](#) {  
[LCG\\_CG](#), [LCG\\_PCG](#), [LCG\\_CGS](#), [LCG\\_BICGSTAB](#),  
[LCG\\_BICGSTAB2](#), [LCG\\_PG](#), [LCG\\_SPG](#) }  
*Types of method that could be recognized by the [lcg\\_solver\(\)](#) function.*
- enum [lcg\\_return\\_enum](#) {  
[LCG\\_SUCCESS](#) = 0, [LCG\\_CONVERGENCE](#) = 0, [LCG\\_STOP](#), [LCG\\_ALREADY\\_OPTIMIZED](#),  
[LCG\\_UNKNOWN\\_ERROR](#) = -1024, [LCG\\_INVILAD\\_VARIABLE\\_SIZE](#), [LCG\\_INVILAD\\_MAX\\_ITERATIONS](#),  
[LCG\\_INVILAD\\_EPSILON](#),  
[LCG\\_INVILAD\\_RESTART\\_EPSILON](#), [LCG\\_REACHED\\_MAX\\_ITERATIONS](#), [LCG\\_NULL\\_PRECONDITION\\_MATRIX](#),  
[LCG\\_NAN\\_VALUE](#),  
[LCG\\_INVALID\\_POINTER](#), [LCG\\_INVALID\\_LAMBDA](#), [LCG\\_INVALID\\_SIGMA](#), [LCG\\_INVALID\\_BETA](#),  
[LCG\\_INVALID\\_MAXIM](#), [LCG\\_SIZE\\_NOT\\_MATCH](#) }  
*return value of the [lcg\\_solver\(\)](#) function*

- enum `clcg_solver_enum` {  
`CLCG_BICG`, `CLCG_BICG_SYM`, `CLCG_CGS`, `CLCG_BICGSTAB`,  
`CLCG_TFQMR`, `CLCG_PCG`, `CLCG_PBICG` }  
*Types of method that could be recognized by the `clcg_solver()` function.*
- enum `clcg_return_enum` {  
`CLCG_SUCCESS` = 0, `CLCG_CONVERGENCE` = 0, `CLCG_STOP`, `CLCG_ALREADY_OPTIMIZED`,  
`CLCG_UNKNOWN_ERROR` = -1024, `CLCG_INVILAD_VARIABLE_SIZE`, `CLCG_INVILAD_MAX_ITERATIONS`,  
`CLCG_INVILAD_EPSILON`,  
`CLCG_REACHED_MAX_ITERATIONS`, `CLCG_NAN_VALUE`, `CLCG_INVALID_POINTER`, `CLCG_SIZE_NOT_MATCH`,  
`CLCG_UNKNOWN_SOLVER` }  
*return value of the `clcg_solver()` function*

## Functions

- `lcg_para lcg_default_parameters ()`  
*Return a `lcg_para` type instance with default values.*
- `lcg_solver_enum lcg_select_solver (std::string slr_char)`  
*Select a type of solver according to the name.*
- void `lcg_error_str` (int er\_index, bool er\_throw=false)  
*Display or throw out a string explanation for the `lcg_solver()` function's return values.*
- `clcg_para clcg_default_parameters ()`  
*Return a `clcg_para` type instance with default values.*
- `clcg_solver_enum clcg_select_solver (std::string slr_char)`  
*Select a type of solver according to the name.*
- void `clcg_error_str` (int er\_index, bool er\_throw=false)  
*Display or throw out a string explanation for the `clcg_solver()` function's return values.*

### 4.32.1 Enumeration Type Documentation

#### 4.32.1.1 `clcg_return_enum`

enum `clcg_return_enum`

return value of the `clcg_solver()` function

##### Enumerator

<code>CLCG_SUCCESS</code>	The solver function terminated successfully.
<code>CLCG_CONVERGENCE</code>	The iteration reached convergence.
<code>CLCG_STOP</code>	The iteration is stopped by the monitoring function.
<code>CLCG_ALREADY_OPTIMIZED</code>	The initial solution is already optimized.
<code>CLCG_UNKNOWN_ERROR</code>	Unknown error.
<code>CLCG_INVILAD_VARIABLE_SIZE</code>	The variable size is negative.
<code>CLCG_INVILAD_MAX_ITERATIONS</code>	The maximal iteration times is negative.
<code>CLCG_INVILAD_EPSILON</code>	The epsilon is negative.
<code>CLCG_REACHED_MAX_ITERATIONS</code>	Iteration reached maximal limit.
<code>CLCG_NAN_VALUE</code>	Nan value.
<code>CLCG_INVALID_POINTER</code>	Invalid pointer.
<code>CLCG_SIZE_NOT_MATCH</code>	Sizes of m and B do not match.
<code>CLCG_UNKNOWN_SOLVER</code>	Unknown solver.

#### 4.32.1.2 clcg\_solver\_enum

enum `clcg_solver_enum`

Types of method that could be recognized by the `clcg_solver()` function.

##### Enumerator

CLCG_BICG	Jacob's Bi-Conjugate Gradient Method
CLCG_BICG_SYM	Bi-Conjugate Gradient Method accelerated for complex symmetric A
CLCG_CGS	Conjugate Gradient Squared Method with real coefficients.
CLCG_BICGSTAB	Biconjugate gradient method.
CLCG_TFQMR	Quasi-Minimal Residual Method Transpose Free Quasi-Minimal Residual Method
CLCG_PCG	Preconditioned conjugate gradient
CLCG_PBCG	Preconditioned Bi-Conjugate Gradient Method

#### 4.32.1.3 lcg\_return\_enum

enum `lcg_return_enum`

return value of the `lcg_solver()` function

##### Enumerator

LCG_SUCCESS	The solver function terminated successfully.
LCG_CONVERGENCE	The iteration reached convergence.
LCG_STOP	The iteration is stopped by the monitoring function.
LCG_ALREADY_OPTIMIZED	The initial solution is already optimized.
LCG_UNKNOWN_ERROR	Unknown error.
LCG_INVILAD_VARIABLE_SIZE	The variable size is negative.
LCG_INVILAD_MAX_ITERATIONS	The maximal iteration times is negative.
LCG_INVILAD_EPSILON	The epsilon is negative.
LCG_INVILAD_RESTART_EPSILON	The restart epsilon is negative.
LCG_REACHED_MAX_ITERATIONS	Iteration reached maximal limit.
LCG_NULL_PRECONDITION_MATRIX	Null precondition matrix.
LCG_NAN_VALUE	Nan value.
LCG_INVALID_POINTER	Invalid pointer.
LCG_INVALID_LAMBDA	Invalid range for lambda.
LCG_INVALID_SIGMA	Invalid range for sigma.
LCG_INVALID_BETA	Invalid range for beta.
LCG_INVALID_MAXIM	Invalid range for maxi_m.
LCG_SIZE_NOT_MATCH	Sizes of m and B do not match.

#### 4.32.1.4 lcg\_solver\_enum

```
enum lcg_solver_enum
```

Types of method that could be recognized by the `lcg_solver()` function.

Enumerator

LCG_CG	Conjugate gradient method.
LCG_PCG	Preconditioned conjugate gradient method.
LCG_CGS	Conjugate gradient squared method.
LCG_BICGSTAB	Biconjugate gradient method.
LCG_BICGSTAB2	Biconjugate gradient method with restart.
LCG_PG	Conjugate gradient method with projected gradient for inequality constraints. This algorithm comes without non-monotonic linear search for the step length.
LCG_SPG	Conjugate gradient method with spectral projected gradient for inequality constraints. This algorithm comes with non-monotonic linear search for the step length.

### 4.32.2 Function Documentation

#### 4.32.2.1 clcg\_default\_parameters()

```
clcg_para clcg_default_parameters ( )
```

Return a `clcg_para` type instance with default values.

Users can use this function to get default parameters' value for the complex conjugate gradient methods.

Returns

A `clcg_para` type instance.

#### 4.32.2.2 clcg\_error\_str()

```
void clcg_error_str (
    int er_index,
    bool er_throw = false )
```

Display or throw out a string explanation for the `clcg_solver()` function's return values.

**Parameters**

in	<i>er_index</i>	The error index returned by the <a href="#">lcg_solver()</a> function.
in	<i>er_throw</i>	throw out a char string of the explanation.

**Returns**

A string explanation of the error.

**4.32.2.3 clcg\_select\_solver()**

```
clcg_solver_enum clcg_select_solver (
    std::string slr_char )
```

Select a type of solver according to the name.

**Parameters**

in	<i>slr_char</i>	Name of the solver
----	-----------------	--------------------

**Returns**

The clcg solver enum.

**4.32.2.4 lcg\_default\_parameters()**

```
lcg_para lcg_default_parameters ( )
```

Return a [lcg\\_para](#) type instance with default values.

Users can use this function to get default parameters' value for the conjugate gradient methods.

**Returns**

A [lcg\\_para](#) type instance.

**4.32.2.5 lcg\_error\_str()**

```
void lcg_error_str (
    int er_index,
    bool er_throw = false )
```

Display or throw out a string explanation for the [lcg\\_solver\(\)](#) function's return values.



## Parameters

in	<i>er_index</i>	The error index returned by the <a href="#">lcg_solver()</a> function.
in	<i>er_throw</i>	throw out a char string of the explanation.

## Returns

A string explanation of the error.

#### 4.32.2.6 `lcg_select_solver()`

```
lcg_solver_enum lcg_select_solver (
    std::string slr_char )
```

Select a type of solver according to the name.

## Parameters

in	<i>slr_char</i>	Name of the solver
----	-----------------	--------------------

## Returns

The lcg solver enum.



# Index

- `_AxProduct`
    - `CLCG_CUDA_Solver`, [6](#)
    - `CLCG_CUDAF_Solver`, [13](#)
    - `CLCG_EIGEN_Solver`, [20](#)
    - `CLCG_Solver`, [27](#)
    - `LCG_CUDA_Solver`, [32](#)
    - `LCG_EIGEN_Solver`, [40](#)
    - `LCG_Solver`, [48](#)
  - `_MxProduct`
    - `CLCG_CUDA_Solver`, [7](#)
    - `CLCG_CUDAF_Solver`, [14](#)
    - `CLCG_EIGEN_Solver`, [21](#)
    - `LCG_CUDA_Solver`, [33](#)
    - `LCG_EIGEN_Solver`, [40](#)
    - `LCG_Solver`, [48](#)
  - `_Progress`
    - `CLCG_CUDA_Solver`, [7](#)
    - `CLCG_CUDAF_Solver`, [14](#)
    - `CLCG_EIGEN_Solver`, [21](#)
    - `CLCG_Solver`, [28](#)
    - `LCG_CUDA_Solver`, [33](#)
    - `LCG_EIGEN_Solver`, [41](#)
    - `LCG_Solver`, [49](#)
  - `~CLCG_CUDAF_Solver`
    - `CLCG_CUDAF_Solver`, [13](#)
  - `~CLCG_CUDA_Solver`
    - `CLCG_CUDA_Solver`, [6](#)
  - `~CLCG_EIGEN_Solver`
    - `CLCG_EIGEN_Solver`, [20](#)
  - `~CLCG_Solver`
    - `CLCG_Solver`, [27](#)
  - `~LCG_CUDA_Solver`
    - `LCG_CUDA_Solver`, [32](#)
  - `~LCG_EIGEN_Solver`
    - `LCG_EIGEN_Solver`, [40](#)
  - `~LCG_Solver`
    - `LCG_Solver`, [48](#)
- `abs_diff`
  - `clcg_para`, [25](#)
  - `lcg_para`, [45](#)
- `algebra.cpp`, [55](#)
  - `lcg_abs`, [56](#)
  - `lcg_dot`, [56](#)
  - `lcg_free`, [57](#)
  - `lcg_malloc`, [57](#), [58](#)
  - `lcg_matvec`, [58](#)
  - `lcg_matvec_coo`, [58](#)
  - `lcg_max`, [59](#)
  - `lcg_min`, [59](#)
  - `lcg_set2box`, [60](#)
  - `lcg_squaredl2norm`, [60](#)
  - `lcg_vecrnd`, [61](#)
  - `lcg_vecset`, [61](#), [62](#)
- `algebra.h`, [62](#)
  - `clcg_complex_e`, [64](#)
  - `Conjugate`, [64](#)
  - `lcg_abs`, [64](#)
  - `lcg_dot`, [65](#)
  - `lcg_float`, [63](#)
  - `lcg_free`, [65](#)
  - `lcg_malloc`, [66](#)
  - `lcg_matrix_e`, [64](#)
  - `lcg_matvec`, [66](#)
  - `lcg_matvec_coo`, [67](#)
  - `lcg_max`, [67](#)
  - `lcg_min`, [68](#)
  - `lcg_set2box`, [68](#)
  - `lcg_squaredl2norm`, [69](#)
  - `lcg_vecrnd`, [69](#)
  - `lcg_vecset`, [70](#)
  - `MatNormal`, [64](#)
  - `MatTranspose`, [64](#)
  - `NonConjugate`, [64](#)
- `algebra_cuda.h`, [71](#)
  - `lcg_set2box_cuda`, [71](#)
  - `lcg_smDcsr_get_diagonal`, [71](#)
  - `lcg_vecDvecD_element_wise`, [72](#)
  - `lcg_vecMvecD_element_wise`, [73](#)
- `algebra_eigen.cpp`, [73](#)
  - `lcg_set2box_eigen`, [73](#)
- `algebra_eigen.h`, [74](#)
  - `lcg_set2box_eigen`, [74](#)
- `AxProduct`
  - `CLCG_CUDA_Solver`, [8](#)
  - `CLCG_CUDAF_Solver`, [15](#)
  - `CLCG_EIGEN_Solver`, [22](#)
  - `CLCG_Solver`, [28](#)
  - `LCG_CUDA_Solver`, [34](#)
  - `LCG_EIGEN_Solver`, [41](#)
  - `LCG_Solver`, [49](#)
- `beta`
  - `lcg_para`, [45](#)
- `clbicg`
  - `clcg.cpp`, [75](#)
  - `clcg_eigen.cpp`, [86](#)
- `clbicg_symmetric`
  - `clcg.cpp`, [75](#)

- clcg\_eigen.cpp, 86
- clbicgstab
  - clcg.cpp, 76
- clcg.cpp, 74
  - clbicg, 75
  - clbicg\_symmetric, 75
  - clbicgstab, 76
  - clcg\_solver, 76
  - clcg\_solver\_ptr, 75
  - clcgs, 77
  - cltfqmr, 77
- clcg.h, 77
  - clcg\_axfunc\_ptr, 78
  - clcg\_progress\_ptr, 78
  - clcg\_solver, 78
- CLCG\_ALREADY\_OPTIMIZED
  - util.h, 175
- clcg\_axfunc\_cuda\_ptr
  - clcg\_cuda.h, 80
- clcg\_axfunc\_cudaf\_ptr
  - clcg\_cudaf.h, 83
- clcg\_axfunc\_eigen\_ptr
  - clcg\_eigen.h, 90
- clcg\_axfunc\_ptr
  - clcg.h, 78
- CLCG\_BICG
  - util.h, 176
- CLCG\_BICG\_SYM
  - util.h, 176
- CLCG\_BICGSTAB
  - util.h, 176
- clcg\_Cdiff
  - lcg\_complex\_cuda.h, 121
- CLCG\_CGS
  - util.h, 176
- clcg\_Cholesky
  - preconditioner\_eigen.cpp, 159
  - preconditioner\_eigen.h, 165
- clcg\_complex\_e
  - algebra.h, 64
- clcg\_conjugate
  - lcg\_complex.cpp, 108
  - lcg\_complex.h, 115
- CLCG\_CONVERGENCE
  - util.h, 175
- clcg\_Cscale
  - lcg\_complex\_cuda.h, 122
- clcg\_Csqrt
  - lcg\_complex\_cuda.h, 122
- clcg\_Csum
  - lcg\_complex\_cuda.h, 123
- clcg\_cuda.h, 79
  - clcg\_axfunc\_cuda\_ptr, 80
  - clcg\_progress\_cuda\_ptr, 80
  - clcg\_solver\_cuda, 81
  - clcg\_solver\_preconditioned\_cuda, 81
- CLCG\_CUDA\_Solver, 5
  - \_AxProduct, 6
  - \_MxProduct, 7
  - \_Progress, 7
  - ~CLCG\_CUDA\_Solver, 6
  - AxProduct, 8
  - CLCG\_CUDA\_Solver, 6
  - inter\_, 12
  - Minimize, 8
  - MinimizePreconditioned, 9
  - MxProduct, 10
  - param\_, 12
  - Progress, 10
  - set\_clcg\_parameter, 11
  - set\_report\_interval, 11
  - silent, 11
  - silent\_, 12
- clcg\_cudaf.h, 82
  - clcg\_axfunc\_cudaf\_ptr, 83
  - clcg\_progress\_cudaf\_ptr, 83
  - clcg\_solver\_cuda, 84
  - clcg\_solver\_preconditioned\_cuda, 84
- CLCG\_CUDAF\_Solver, 12
  - \_AxProduct, 13
  - \_MxProduct, 14
  - \_Progress, 14
  - ~CLCG\_CUDAF\_Solver, 13
  - AxProduct, 15
  - CLCG\_CUDAF\_Solver, 13
  - inter\_, 19
  - Minimize, 15
  - MinimizePreconditioned, 16
  - MxProduct, 17
  - param\_, 19
  - Progress, 17
  - set\_clcg\_parameter, 18
  - set\_report\_interval, 18
  - silent, 18
  - silent\_, 19
- clcg\_default\_parameters
  - util.cpp, 172
  - util.h, 177
- clcg\_dot
  - lcg\_complex.cpp, 108
  - lcg\_complex.h, 115
- clcg\_eigen.cpp, 85
  - clbicg, 86
  - clbicg\_symmetric, 86
  - clcg\_solver\_eigen, 87
  - clcg\_solver\_preconditioned\_eigen, 87
  - clcgs, 88
  - clpbicg, 88
  - clpcg, 88
  - cltfqmr, 89
  - eigen\_preconditioned\_solver\_ptr, 86
  - eigen\_solver\_ptr, 86
- clcg\_eigen.h, 89
  - clcg\_axfunc\_eigen\_ptr, 90
  - clcg\_progress\_eigen\_ptr, 90
  - clcg\_solver\_eigen, 90

- clcg\_solver\_preconditioned\_eigen, 91
- CLCG\_EIGEN\_Solver, 19
  - \_AxProduct, 20
  - \_MxProduct, 21
  - \_Progress, 21
  - ~CLCG\_EIGEN\_Solver, 20
  - AxProduct, 22
  - CLCG\_EIGEN\_Solver, 20
  - inter\_, 24
  - Minimize, 22
  - MinimizePreconditioned, 22
  - MxProduct, 23
  - param\_, 25
  - Progress, 23
  - set\_clcg\_parameter, 24
  - set\_report\_interval, 24
  - silent, 24
  - silent\_, 25
- clcg\_error\_str
  - util.cpp, 172
  - util.h, 177
- clcg\_free
  - lcg\_complex.cpp, 109
  - lcg\_complex.h, 115, 116
- clcg\_free\_cuda
  - lcg\_complex\_cuda.h, 123
- clcg\_incomplete\_Cholesky
  - preconditioner\_eigen.cpp, 159
  - preconditioner\_eigen.h, 165
- clcg\_incomplete\_Cholesky\_cuda\_full
  - preconditioner\_cuda.h, 155
- clcg\_incomplete\_Cholesky\_cuda\_half
  - preconditioner\_cuda.h, 156
- clcg\_incomplete\_Cholesky\_cuda\_half\_buffsize
  - preconditioner\_cuda.h, 157
- clcg\_incomplete\_LU
  - preconditioner\_eigen.cpp, 160
  - preconditioner\_eigen.h, 166
- clcg\_inner
  - lcg\_complex.cpp, 110
  - lcg\_complex.h, 116
- CLCG\_INVALID\_POINTER
  - util.h, 175
- clcg\_invert\_lower\_triangle
  - preconditioner\_eigen.cpp, 160
  - preconditioner\_eigen.h, 166
- clcg\_invert\_upper\_triangle
  - preconditioner\_eigen.cpp, 161
  - preconditioner\_eigen.h, 166
- CLCG\_INVILAD\_EPSILON
  - util.h, 175
- CLCG\_INVILAD\_MAX\_ITERATIONS
  - util.h, 175
- CLCG\_INVILAD\_VARIABLE\_SIZE
  - util.h, 175
- clcg\_malloc
  - lcg\_complex.cpp, 110
  - lcg\_complex.h, 116, 117
- clcg\_malloc\_cuda
  - lcg\_complex\_cuda.h, 123
- clcg\_matvec
  - lcg\_complex.cpp, 111
  - lcg\_complex.h, 117
- clcg\_module
  - lcg\_complex.cpp, 111
  - lcg\_complex.h, 118
- CLCG\_NAN\_VALUE
  - util.h, 175
- clcg\_para, 25
  - abs\_diff, 25
  - epsilon, 26
  - max\_iterations, 26
- CLCG\_PBICG
  - util.h, 176
- CLCG\_PCG
  - util.h, 176
- clcg\_progress\_cuda\_ptr
  - clcg\_cuda.h, 80
- clcg\_progress\_cudaf\_ptr
  - clcg\_cudaf.h, 83
- clcg\_progress\_eigen\_ptr
  - clcg\_eigen.h, 90
- clcg\_progress\_ptr
  - clcg.h, 78
- CLCG\_REACHED\_MAX\_ITERATIONS
  - util.h, 175
- clcg\_return\_enum
  - util.h, 175
- clcg\_select\_solver
  - util.cpp, 173
  - util.h, 178
- clcg\_set
  - lcg\_complex.cpp, 111
  - lcg\_complex.h, 118
- CLCG\_SIZE\_NOT\_MATCH
  - util.h, 175
- clcg\_smCcoo\_row2col
  - lcg\_complex\_cuda.h, 124
- clcg\_smCcsr\_get\_diagonal
  - lcg\_complex\_cuda.h, 124
- clcg\_smZcoo\_row2col
  - lcg\_complex\_cuda.h, 125
- clcg\_smZcsr\_get\_diagonal
  - lcg\_complex\_cuda.h, 125
- clcg\_solve\_lower\_triangle
  - preconditioner\_eigen.cpp, 161
  - preconditioner\_eigen.h, 167
- clcg\_solve\_upper\_triangle
  - preconditioner\_eigen.cpp, 161
  - preconditioner\_eigen.h, 167
- CLCG\_Solver, 26
  - \_AxProduct, 27
  - \_Progress, 28
  - ~CLCG\_Solver, 27
  - AxProduct, 28
  - CLCG\_Solver, 27

- inter\_, 30
- Minimize, 29
- param\_, 30
- Progress, 29
- set\_clcg\_parameter, 30
- set\_report\_interval, 30
- silent, 30
- silent\_, 31
- clcg\_solver
  - clcg.cpp, 76
  - clcg.h, 78
- clcg\_solver\_cuda
  - clcg\_cuda.h, 81
  - clcg\_cudaf.h, 84
- clcg\_solver\_eigen
  - clcg\_eigen.cpp, 87
  - clcg\_eigen.h, 90
- clcg\_solver\_enum
  - util.h, 176
- clcg\_solver\_preconditioned\_cuda
  - clcg\_cuda.h, 81
  - clcg\_cudaf.h, 84
- clcg\_solver\_preconditioned\_eigen
  - clcg\_eigen.cpp, 87
  - clcg\_eigen.h, 91
- clcg\_solver\_ptr
  - clcg.cpp, 75
- clcg\_square
  - lcg\_complex.cpp, 112
  - lcg\_complex.h, 118
- CLCG\_STOP
  - util.h, 175
- CLCG\_SUCCESS
  - util.h, 175
- CLCG\_TFQMR
  - util.h, 176
- CLCG\_UNKNOWN\_ERROR
  - util.h, 175
- CLCG\_UNKNOWN\_SOLVER
  - util.h, 175
- clcg\_vecC\_conjugate
  - lcg\_complex\_cuda.h, 126
- clcg\_vecDvecC\_element\_wise
  - lcg\_complex\_cuda.h, 126
- clcg\_vecDvecZ\_element\_wise
  - lcg\_complex\_cuda.h, 127
- clcg\_vecMvecC\_element\_wise
  - lcg\_complex\_cuda.h, 127
- clcg\_vecMvecZ\_element\_wise
  - lcg\_complex\_cuda.h, 128
- clcg\_vecrnd
  - lcg\_complex.cpp, 112
  - lcg\_complex.h, 119
- clcg\_vecset
  - lcg\_complex.cpp, 113
  - lcg\_complex.h, 119, 120
- clcg\_vecset\_cuda
  - lcg\_complex\_cuda.h, 128
- clcg\_vecZ\_conjugate
  - lcg\_complex\_cuda.h, 129
- clcg\_Zdiff
  - lcg\_complex\_cuda.h, 129
- clcg\_Zscale
  - lcg\_complex\_cuda.h, 129
- clcg\_Zsqrt
  - lcg\_complex\_cuda.h, 130
- clcg\_Zsum
  - lcg\_complex\_cuda.h, 130
- clcgs
  - clcg.cpp, 77
  - clcg\_eigen.cpp, 88
- clpbicg
  - clcg\_eigen.cpp, 88
- clpcg
  - clcg\_eigen.cpp, 88
- cltfqmr
  - clcg.cpp, 77
  - clcg\_eigen.cpp, 89
- config.h, 92
  - LibLCG\_CUDA, 92
  - LibLCG\_EIGEN, 92
  - LibLCG\_OPENMP, 92
  - LibLCG\_STD\_COMPLEX, 92
- Conjugate
  - algebra.h, 64
- cuda2lcg\_complex
  - lcg\_complex\_cuda.h, 131
- eigen\_preconditioned\_solver\_ptr
  - clcg\_eigen.cpp, 86
- eigen\_solver\_ptr
  - clcg\_eigen.cpp, 86
  - lcg\_eigen.cpp, 136
- eigen\_solver\_ptr2
  - lcg\_eigen.cpp, 137
- epsilon
  - clcg\_para, 26
  - lcg\_para, 45
- inter\_
  - CLCG\_CUDA\_Solver, 12
  - CLCG\_CUDAF\_Solver, 19
  - CLCG\_EIGEN\_Solver, 24
  - CLCG\_Solver, 30
  - LCG\_CUDA\_Solver, 38
  - LCG\_EIGEN\_Solver, 44
  - LCG\_Solver, 53
- lbicgstab
  - lcg.cpp, 95
  - lcg\_eigen.cpp, 137
- lbicgstab2
  - lcg.cpp, 95
  - lcg\_eigen.cpp, 138
- lcg
  - lcg.cpp, 96
  - lcg.h, 103

- lcg\_eigen.cpp, 139
- lcg.cpp, 93
  - lbicgstab, 95
  - lbicgstab2, 95
  - lcg, 96
  - lcg\_solver, 97
  - lcg\_solver\_constrained, 97
  - lcg\_solver\_preconditioned, 98
  - lcg\_solver\_ptr, 94
  - lcg\_solver\_ptr2, 94
  - lcgs, 99
  - lpcg, 100
  - lpg, 100
  - lspg, 101
- lcg.h, 102
  - lcg, 103
  - lcg\_axfunc\_ptr, 103
  - lcg\_progress\_ptr, 103
  - lcg\_solver, 104
  - lcg\_solver\_constrained, 105
  - lcg\_solver\_preconditioned, 106
  - lcgs, 106
- lcg2cuda\_complex
  - lcg\_complex\_cuda.h, 131
- lcg\_abs
  - algebra.cpp, 56
  - algebra.h, 64
- LCG\_ALREADY\_OPTIMIZED
  - util.h, 176
- lcg\_axfunc\_cuda\_ptr
  - lcg\_cuda.h, 132
- lcg\_axfunc\_eigen\_ptr
  - lcg\_eigen.h, 145
- lcg\_axfunc\_ptr
  - lcg.h, 103
- LCG\_BICGSTAB
  - util.h, 177
- LCG\_BICGSTAB2
  - util.h, 177
- LCG\_CG
  - util.h, 177
- LCG\_CGS
  - util.h, 177
- lcg\_Cholesky
  - preconditioner\_eigen.cpp, 162
  - preconditioner\_eigen.h, 167
- lcg\_complex
  - lcg\_complex.h, 114
- lcg\_complex.cpp, 107
  - clcg\_conjugate, 108
  - clcg\_dot, 108
  - clcg\_free, 109
  - clcg\_inner, 110
  - clcg\_malloc, 110
  - clcg\_matvec, 111
  - clcg\_module, 111
  - clcg\_set, 111
  - clcg\_square, 112
  - clcg\_vecrnd, 112
  - clcg\_vecset, 113
- lcg\_complex.h, 114
  - clcg\_conjugate, 115
  - clcg\_dot, 115
  - clcg\_free, 115, 116
  - clcg\_inner, 116
  - clcg\_malloc, 116, 117
  - clcg\_matvec, 117
  - clcg\_module, 118
  - clcg\_set, 118
  - clcg\_square, 118
  - clcg\_vecrnd, 119
  - clcg\_vecset, 119, 120
- lcg\_complex\_cuda.h, 120
  - clcg\_Cdiff, 121
  - clcg\_Cscale, 122
  - clcg\_Csqrt, 122
  - clcg\_Csum, 123
  - clcg\_free\_cuda, 123
  - clcg\_malloc\_cuda, 123
  - clcg\_smCcoo\_row2col, 124
  - clcg\_smCcsr\_get\_diagonal, 124
  - clcg\_smZcoo\_row2col, 125
  - clcg\_smZcsr\_get\_diagonal, 125
  - clcg\_vecC\_conjugate, 126
  - clcg\_vecDvecC\_element\_wise, 126
  - clcg\_vecDvecZ\_element\_wise, 127
  - clcg\_vecMvecC\_element\_wise, 127
  - clcg\_vecMvecZ\_element\_wise, 128
  - clcg\_vecset\_cuda, 128
  - clcg\_vecZ\_conjugate, 129
  - clcg\_Zdiff, 129
  - clcg\_Zscale, 129
  - clcg\_Zsqrt, 130
  - clcg\_Zsum, 130
  - cuda2lcg\_complex, 131
  - lcg2cuda\_complex, 131
- LCG\_CONVERGENCE
  - util.h, 176
- lcg\_cuda.h, 131
  - lcg\_axfunc\_cuda\_ptr, 132
  - lcg\_progress\_cuda\_ptr, 133
  - lcg\_solver\_constrained\_cuda, 133
  - lcg\_solver\_cuda, 134
  - lcg\_solver\_preconditioned\_cuda, 135
- LCG\_CUDA\_Solver, 31
  - \_AxProduct, 32
  - \_MxProduct, 33
  - \_Progress, 33
  - ~LCG\_CUDA\_Solver, 32
  - AxProduct, 34
  - inter\_, 38
  - LCG\_CUDA\_Solver, 32
  - Minimize, 34
  - MinimizeConstrained, 35
  - MinimizePreconditioned, 36

- MxProduct, 36
- param\_, 38
- Progress, 37
- set\_lcg\_parameter, 37
- set\_report\_interval, 37
- silent, 38
- silent\_, 38
- lcg\_default\_parameters
  - util.cpp, 173
  - util.h, 178
- lcg\_dot
  - algebra.cpp, 56
  - algebra.h, 65
- lcg\_eigen.cpp, 135
  - eigen\_solver\_ptr, 136
  - eigen\_solver\_ptr2, 137
  - lbicgstab, 137
  - lbicgstab2, 138
  - lcg, 139
  - lcg\_solver\_constrained\_eigen, 139
  - lcg\_solver\_eigen, 140
  - lcg\_solver\_preconditioned\_eigen, 140
  - lcgs, 141
  - lpcg, 142
  - lpg, 143
  - lspg, 143
- lcg\_eigen.h, 144
  - lcg\_axfunc\_eigen\_ptr, 145
  - lcg\_progress\_eigen\_ptr, 145
  - lcg\_solver\_constrained\_eigen, 145
  - lcg\_solver\_eigen, 146
  - lcg\_solver\_preconditioned\_eigen, 147
- LCG\_EIGEN\_Solver, 38
  - \_AxProduct, 40
  - \_MxProduct, 40
  - \_Progress, 41
  - ~LCG\_EIGEN\_Solver, 40
  - AxProduct, 41
  - inter\_, 44
  - LCG\_EIGEN\_Solver, 39
  - Minimize, 41
  - MinimizeConstrained, 42
  - MinimizePreconditioned, 42
  - MxProduct, 43
  - param\_, 44
  - Progress, 43
  - set\_lcg\_parameter, 43
  - set\_report\_interval, 44
  - silent, 44
  - silent\_, 44
- lcg\_error\_str
  - util.cpp, 173
  - util.h, 178
- lcg\_float
  - algebra.h, 63
- lcg\_free
  - algebra.cpp, 57
  - algebra.h, 65
- lcg\_full\_rank\_coo
  - preconditioner.cpp, 148
  - preconditioner.h, 152
- lcg\_incomplete\_Cholesky
  - preconditioner\_eigen.cpp, 162
  - preconditioner\_eigen.h, 168
- lcg\_incomplete\_Cholesky\_full\_coo
  - preconditioner.cpp, 148
  - preconditioner.h, 152
- lcg\_incomplete\_Cholesky\_half\_buffersize\_coo
  - preconditioner.cpp, 149
  - preconditioner.h, 153
- lcg\_incomplete\_Cholesky\_half\_coo
  - preconditioner.cpp, 149
  - preconditioner.h, 153
- lcg\_incomplete\_LU
  - preconditioner\_eigen.cpp, 162
  - preconditioner\_eigen.h, 168
- LCG\_INVALID\_BETA
  - util.h, 176
- LCG\_INVALID\_LAMBDA
  - util.h, 176
- LCG\_INVALID\_MAXIM
  - util.h, 176
- LCG\_INVALID\_POINTER
  - util.h, 176
- LCG\_INVALID\_SIGMA
  - util.h, 176
- lcg\_invert\_lower\_triangle
  - preconditioner\_eigen.cpp, 163
  - preconditioner\_eigen.h, 169
- lcg\_invert\_upper\_triangle
  - preconditioner\_eigen.cpp, 163
  - preconditioner\_eigen.h, 169
- LCG\_INVILAD\_EPSILON
  - util.h, 176
- LCG\_INVILAD\_MAX\_ITERATIONS
  - util.h, 176
- LCG\_INVILAD\_RESTART\_EPSILON
  - util.h, 176
- LCG\_INVILAD\_VARIABLE\_SIZE
  - util.h, 176
- lcg\_malloc
  - algebra.cpp, 57, 58
  - algebra.h, 66
- lcg\_matrix\_e
  - algebra.h, 64
- lcg\_matvec
  - algebra.cpp, 58
  - algebra.h, 66
- lcg\_matvec\_coo
  - algebra.cpp, 58
  - algebra.h, 67
- lcg\_max
  - algebra.cpp, 59
  - algebra.h, 67
- lcg\_min
  - algebra.cpp, 59



- algebra.h, 68
- LCG\_NAN\_VALUE
  - util.h, 176
- LCG\_NULL\_PRECONDITION\_MATRIX
  - util.h, 176
- lcg\_para, 45
  - abs\_diff, 45
  - beta, 45
  - epsilon, 45
  - max\_iterations, 46
  - maxi\_m, 46
  - restart\_epsilon, 46
  - sigma, 46
  - step, 46
- LCG\_PCG
  - util.h, 177
- LCG\_PG
  - util.h, 177
- lcg\_progress\_cuda\_ptr
  - lcg\_cuda.h, 133
- lcg\_progress\_eigen\_ptr
  - lcg\_eigen.h, 145
- lcg\_progress\_ptr
  - lcg.h, 103
- LCG\_REACHED\_MAX\_ITERATIONS
  - util.h, 176
- lcg\_return\_enum
  - util.h, 176
- lcg\_select\_solver
  - util.cpp, 174
  - util.h, 179
- lcg\_set2box
  - algebra.cpp, 60
  - algebra.h, 68
- lcg\_set2box\_cuda
  - algebra\_cuda.h, 71
- lcg\_set2box\_eigen
  - algebra\_eigen.cpp, 73
  - algebra\_eigen.h, 74
- LCG\_SIZE\_NOT\_MATCH
  - util.h, 176
- lcg\_smDcsr\_get\_diagonal
  - algebra\_cuda.h, 71
- lcg\_solve\_lower\_triangle
  - preconditioner\_eigen.cpp, 163
  - preconditioner\_eigen.h, 169
- lcg\_solve\_lower\_triangle\_coo
  - preconditioner.cpp, 150
  - preconditioner.h, 154
- lcg\_solve\_upper\_triangle
  - preconditioner\_eigen.cpp, 164
  - preconditioner\_eigen.h, 170
- lcg\_solve\_upper\_triangle\_coo
  - preconditioner.cpp, 151
  - preconditioner.h, 154
- LCG\_Solver, 47
  - \_AxProduct, 48
  - \_MxProduct, 48
  - \_Progress, 49
  - ~LCG\_Solver, 48
  - AxProduct, 49
  - inter\_, 53
  - LCG\_Solver, 48
  - Minimize, 50
  - MinimizeConstrained, 50
  - MinimizePreconditioned, 51
  - MxProduct, 51
  - param\_, 53
  - Progress, 52
  - set\_lcg\_parameter, 52
  - set\_report\_interval, 52
  - silent, 53
  - silent\_, 53
- lcg\_solver
  - lcg.cpp, 97
  - lcg.h, 104
- lcg\_solver\_constrained
  - lcg.cpp, 97
  - lcg.h, 105
- lcg\_solver\_constrained\_cuda
  - lcg\_cuda.h, 133
- lcg\_solver\_constrained\_eigen
  - lcg\_eigen.cpp, 139
  - lcg\_eigen.h, 145
- lcg\_solver\_cuda
  - lcg\_cuda.h, 134
- lcg\_solver\_eigen
  - lcg\_eigen.cpp, 140
  - lcg\_eigen.h, 146
- lcg\_solver\_enum
  - util.h, 177
- lcg\_solver\_preconditioned
  - lcg.cpp, 98
  - lcg.h, 106
- lcg\_solver\_preconditioned\_cuda
  - lcg\_cuda.h, 135
- lcg\_solver\_preconditioned\_eigen
  - lcg\_eigen.cpp, 140
  - lcg\_eigen.h, 147
- lcg\_solver\_ptr
  - lcg.cpp, 94
- lcg\_solver\_ptr2
  - lcg.cpp, 94
- LCG\_SPG
  - util.h, 177
- lcg\_squaredl2norm
  - algebra.cpp, 60
  - algebra.h, 69
- LCG\_STOP
  - util.h, 176
- LCG\_SUCCESS
  - util.h, 176
- LCG\_UNKNOWN\_ERROR
  - util.h, 176
- lcg\_vecDvecD\_element\_wise
  - algebra\_cuda.h, 72

- lcg\_vecMvecD\_element\_wise
  - algebra\_cuda.h, [73](#)
- lcg\_vecrnd
  - algebra.cpp, [61](#)
  - algebra.h, [69](#)
- lcg\_vecset
  - algebra.cpp, [61](#), [62](#)
  - algebra.h, [70](#)
- lcgs
  - lcg.cpp, [99](#)
  - lcg.h, [106](#)
  - lcg\_eigen.cpp, [141](#)
- LibLCG\_CUDA
  - config.h, [92](#)
- LibLCG\_EIGEN
  - config.h, [92](#)
- LibLCG\_OPENMP
  - config.h, [92](#)
- LibLCG\_STD\_COMPLEX
  - config.h, [92](#)
- lpcg
  - lcg.cpp, [100](#)
  - lcg\_eigen.cpp, [142](#)
- lpg
  - lcg.cpp, [100](#)
  - lcg\_eigen.cpp, [143](#)
- lspg
  - lcg.cpp, [101](#)
  - lcg\_eigen.cpp, [143](#)
- MatNormal
  - algebra.h, [64](#)
- MatTranspose
  - algebra.h, [64](#)
- max\_iterations
  - clcg\_para, [26](#)
  - lcg\_para, [46](#)
- maxi\_m
  - lcg\_para, [46](#)
- Minimize
  - CLCG\_CUDA\_Solver, [8](#)
  - CLCG\_CUDAF\_Solver, [15](#)
  - CLCG\_EIGEN\_Solver, [22](#)
  - CLCG\_Solver, [29](#)
  - LCG\_CUDA\_Solver, [34](#)
  - LCG\_EIGEN\_Solver, [41](#)
  - LCG\_Solver, [50](#)
- MinimizeConstrained
  - LCG\_CUDA\_Solver, [35](#)
  - LCG\_EIGEN\_Solver, [42](#)
  - LCG\_Solver, [50](#)
- MinimizePreconditioned
  - CLCG\_CUDA\_Solver, [9](#)
  - CLCG\_CUDAF\_Solver, [16](#)
  - CLCG\_EIGEN\_Solver, [22](#)
  - LCG\_CUDA\_Solver, [36](#)
  - LCG\_EIGEN\_Solver, [42](#)
  - LCG\_Solver, [51](#)
- MxProduct
  - CLCG\_CUDA\_Solver, [10](#)
  - CLCG\_CUDAF\_Solver, [17](#)
  - CLCG\_EIGEN\_Solver, [23](#)
  - LCG\_CUDA\_Solver, [36](#)
  - LCG\_EIGEN\_Solver, [43](#)
  - LCG\_Solver, [51](#)
- NonConjugate
  - algebra.h, [64](#)
- param\_
  - CLCG\_CUDA\_Solver, [12](#)
  - CLCG\_CUDAF\_Solver, [19](#)
  - CLCG\_EIGEN\_Solver, [25](#)
  - CLCG\_Solver, [30](#)
  - LCG\_CUDA\_Solver, [38](#)
  - LCG\_EIGEN\_Solver, [44](#)
  - LCG\_Solver, [53](#)
- preconditioner.cpp, [147](#)
  - lcg\_full\_rank\_coo, [148](#)
  - lcg\_incomplete\_Cholesky\_full\_coo, [148](#)
  - lcg\_incomplete\_Cholesky\_half\_buffersize\_coo, [149](#)
  - lcg\_incomplete\_Cholesky\_half\_coo, [149](#)
  - lcg\_solve\_lower\_triangle\_coo, [150](#)
  - lcg\_solve\_upper\_triangle\_coo, [151](#)
- preconditioner.h, [151](#)
  - lcg\_full\_rank\_coo, [152](#)
  - lcg\_incomplete\_Cholesky\_full\_coo, [152](#)
  - lcg\_incomplete\_Cholesky\_half\_buffersize\_coo, [153](#)
  - lcg\_incomplete\_Cholesky\_half\_coo, [153](#)
  - lcg\_solve\_lower\_triangle\_coo, [154](#)
  - lcg\_solve\_upper\_triangle\_coo, [154](#)
- preconditioner\_cuda.h, [155](#)
  - clcg\_incomplete\_Cholesky\_cuda\_full, [155](#)
  - clcg\_incomplete\_Cholesky\_cuda\_half, [156](#)
  - clcg\_incomplete\_Cholesky\_cuda\_half\_buffersize, [157](#)
- preconditioner\_eigen.cpp, [158](#)
  - clcg\_Cholesky, [159](#)
  - clcg\_incomplete\_Cholesky, [159](#)
  - clcg\_incomplete\_LU, [160](#)
  - clcg\_invert\_lower\_triangle, [160](#)
  - clcg\_invert\_upper\_triangle, [161](#)
  - clcg\_solve\_lower\_triangle, [161](#)
  - clcg\_solve\_upper\_triangle, [161](#)
  - lcg\_Cholesky, [162](#)
  - lcg\_incomplete\_Cholesky, [162](#)
  - lcg\_incomplete\_LU, [162](#)
  - lcg\_invert\_lower\_triangle, [163](#)
  - lcg\_invert\_upper\_triangle, [163](#)
  - lcg\_solve\_lower\_triangle, [163](#)
  - lcg\_solve\_upper\_triangle, [164](#)
  - triplet\_bl, [159](#)
  - triplet\_cd, [159](#)
  - triplet\_d, [159](#)
- preconditioner\_eigen.h, [164](#)
  - clcg\_Cholesky, [165](#)
  - clcg\_incomplete\_Cholesky, [165](#)
  - clcg\_incomplete\_LU, [166](#)

- clcg\_invert\_lower\_triangle, [166](#)
  - clcg\_invert\_upper\_triangle, [166](#)
  - clcg\_solve\_lower\_triangle, [167](#)
  - clcg\_solve\_upper\_triangle, [167](#)
  - lcg\_Cholesky, [167](#)
  - lcg\_incomplete\_Cholesky, [168](#)
  - lcg\_incomplete\_LU, [168](#)
  - lcg\_invert\_lower\_triangle, [169](#)
  - lcg\_invert\_upper\_triangle, [169](#)
  - lcg\_solve\_lower\_triangle, [169](#)
  - lcg\_solve\_upper\_triangle, [170](#)
- Progress
  - CLCG\_CUDA\_Solver, [10](#)
  - CLCG\_CUDAF\_Solver, [17](#)
  - CLCG\_EIGEN\_Solver, [23](#)
  - CLCG\_Solver, [29](#)
  - LCG\_CUDA\_Solver, [37](#)
  - LCG\_EIGEN\_Solver, [43](#)
  - LCG\_Solver, [52](#)
- restart\_epsilon
  - lcg\_para, [46](#)
- set\_clcg\_parameter
  - CLCG\_CUDA\_Solver, [11](#)
  - CLCG\_CUDAF\_Solver, [18](#)
  - CLCG\_EIGEN\_Solver, [24](#)
  - CLCG\_Solver, [30](#)
- set\_lcg\_parameter
  - LCG\_CUDA\_Solver, [37](#)
  - LCG\_EIGEN\_Solver, [43](#)
  - LCG\_Solver, [52](#)
- set\_report\_interval
  - CLCG\_CUDA\_Solver, [11](#)
  - CLCG\_CUDAF\_Solver, [18](#)
  - CLCG\_EIGEN\_Solver, [24](#)
  - CLCG\_Solver, [30](#)
  - LCG\_CUDA\_Solver, [37](#)
  - LCG\_EIGEN\_Solver, [44](#)
  - LCG\_Solver, [52](#)
- sigma
  - lcg\_para, [46](#)
- silent
  - CLCG\_CUDA\_Solver, [11](#)
  - CLCG\_CUDAF\_Solver, [18](#)
  - CLCG\_EIGEN\_Solver, [24](#)
  - CLCG\_Solver, [30](#)
  - LCG\_CUDA\_Solver, [38](#)
  - LCG\_EIGEN\_Solver, [44](#)
  - LCG\_Solver, [53](#)
- silent\_
  - CLCG\_CUDA\_Solver, [12](#)
  - CLCG\_CUDAF\_Solver, [19](#)
  - CLCG\_EIGEN\_Solver, [25](#)
  - CLCG\_Solver, [31](#)
  - LCG\_CUDA\_Solver, [38](#)
  - LCG\_EIGEN\_Solver, [44](#)
  - LCG\_Solver, [53](#)
- solver.cpp, [170](#)
- solver.h, [170](#)
- solver\_cuda.h, [171](#)
- solver\_eigen.cpp, [171](#)
- solver\_eigen.h, [171](#)
- step
  - lcg\_para, [46](#)
- triplet\_bl
  - preconditioner\_eigen.cpp, [159](#)
- triplet\_cd
  - preconditioner\_eigen.cpp, [159](#)
- triplet\_d
  - preconditioner\_eigen.cpp, [159](#)
- util.cpp, [171](#)
  - clcg\_default\_parameters, [172](#)
  - clcg\_error\_str, [172](#)
  - clcg\_select\_solver, [173](#)
  - lcg\_default\_parameters, [173](#)
  - lcg\_error\_str, [173](#)
  - lcg\_select\_solver, [174](#)
- util.h, [174](#)
  - CLCG\_ALREADY\_OPTIMIZED, [175](#)
  - CLCG\_BICG, [176](#)
  - CLCG\_BICG\_SYM, [176](#)
  - CLCG\_BICGSTAB, [176](#)
  - CLCG\_CGS, [176](#)
  - CLCG\_CONVERGENCE, [175](#)
  - clcg\_default\_parameters, [177](#)
  - clcg\_error\_str, [177](#)
  - CLCG\_INVALID\_POINTER, [175](#)
  - CLCG\_INVILAD\_EPSILON, [175](#)
  - CLCG\_INVILAD\_MAX\_ITERATIONS, [175](#)
  - CLCG\_INVILAD\_VARIABLE\_SIZE, [175](#)
  - CLCG\_NAN\_VALUE, [175](#)
  - CLCG\_PBICG, [176](#)
  - CLCG\_PCG, [176](#)
  - CLCG\_REACHED\_MAX\_ITERATIONS, [175](#)
  - clcg\_return\_enum, [175](#)
  - clcg\_select\_solver, [178](#)
  - CLCG\_SIZE\_NOT\_MATCH, [175](#)
  - clcg\_solver\_enum, [176](#)
  - CLCG\_STOP, [175](#)
  - CLCG\_SUCCESS, [175](#)
  - CLCG\_TFQMR, [176](#)
  - CLCG\_UNKNOWN\_ERROR, [175](#)
  - CLCG\_UNKNOWN\_SOLVER, [175](#)
  - LCG\_ALREADY\_OPTIMIZED, [176](#)
  - LCG\_BICGSTAB, [177](#)
  - LCG\_BICGSTAB2, [177](#)
  - LCG\_CG, [177](#)
  - LCG\_CGS, [177](#)
  - LCG\_CONVERGENCE, [176](#)
  - lcg\_default\_parameters, [178](#)
  - lcg\_error\_str, [178](#)
  - LCG\_INVALID\_BETA, [176](#)
  - LCG\_INVALID\_LAMBDA, [176](#)
  - LCG\_INVALID\_MAXIM, [176](#)
  - LCG\_INVALID\_POINTER, [176](#)

LCG\_INVALID\_SIGMA, [176](#)  
LCG\_INVILAD\_EPSILON, [176](#)  
LCG\_INVILAD\_MAX\_ITERATIONS, [176](#)  
LCG\_INVILAD\_RESTART\_EPSILON, [176](#)  
LCG\_INVILAD\_VARIABLE\_SIZE, [176](#)  
LCG\_NAN\_VALUE, [176](#)  
LCG\_NULL\_PRECONDITION\_MATRIX, [176](#)  
LCG\_PCG, [177](#)  
LCG\_PG, [177](#)  
LCG\_REACHED\_MAX\_ITERATIONS, [176](#)  
lcg\_return\_enum, [176](#)  
lcg\_select\_solver, [179](#)  
LCG\_SIZE\_NOT\_MATCH, [176](#)  
lcg\_solver\_enum, [177](#)  
LCG\_SPG, [177](#)  
LCG\_STOP, [176](#)  
LCG\_SUCCESS, [176](#)  
LCG\_UNKNOWN\_ERROR, [176](#)