

Lurupa
1.0

Generated by Doxygen 1.5.6

Thu Jun 26 18:08:52 2008

Contents

1	Lurupa Documentation	1
1.1	Introduction	1
1.2	Theory	2
1.2.1	Interval Arithmetic	3
1.2.2	Rigorous lower and upper bounds	3
1.3	Usage	4
1.3.1	Stand-alone usage	4
1.3.1.1	lp_solve solver modules	6
1.3.2	API usage	7
1.4	License terms	8
2	Class Documentation	17
2.1	Bound Struct Reference	17
2.1.1	Detailed Description	17
2.2	Certificate Struct Reference	19
2.2.1	Detailed Description	19
2.3	Condition Class Reference	20
2.3.1	Detailed Description	21
2.3.2	Constructor & Destructor Documentation	21
2.3.2.1	Condition	21
2.3.3	Member Function Documentation	22
2.3.3.1	generate_lp_rho_d	22
2.3.3.2	generate_lp_rho_p	22
2.3.3.3	norm	23
2.3.3.4	norm_1	23
2.3.3.5	norm_11	24
2.3.3.6	norm_F	24
2.3.3.7	norm_L1	24

2.3.3.8	rho_d	25
2.3.3.9	rho_p	25
2.3.3.10	rot	26
2.3.3.11	swap	26
2.4	Enclosure Struct Reference	27
2.4.1	Detailed Description	27
2.5	Lp Class Reference	28
2.5.1	Detailed Description	30
2.6	Lp_lp_solve Class Reference	31
2.6.1	Detailed Description	31
2.7	Lp_solver Class Reference	33
2.7.1	Detailed Description	33
2.8	Lp_stats Struct Reference	34
2.8.1	Detailed Description	35
2.9	Lurupa Class Reference	36
2.9.1	Detailed Description	42
2.9.2	Member Function Documentation	42
2.9.2.1	add_column_to_basis	42
2.9.2.2	compute_dual_deflation	42
2.9.2.3	compute_lower	42
2.9.2.4	compute_primal_deflation	43
2.9.2.5	compute_upper	43
2.9.2.6	cond	44
2.9.2.7	decrease_dual_deflation	44
2.9.2.8	decrease_primal_deflation	44
2.9.2.9	do_pivot	45
2.9.2.10	dual_certificate	45
2.9.2.11	eliminate	45
2.9.2.12	establish_dual_feasibility	46
2.9.2.13	establish_equality_constraints	46
2.9.2.14	establish_ix_simple_bounds	47
2.9.2.15	establish_iy_simple_bounds	47
2.9.2.16	establish_lagrange_parameters	48
2.9.2.17	establish_primal_feasibility	48
2.9.2.18	find_basis	49
2.9.2.19	find_constraint_basis	50

2.9.2.20	find_equation_basis	50
2.9.2.21	get_alpha	51
2.9.2.22	get_core_version	51
2.9.2.23	get_eta	51
2.9.2.24	get_module_version	51
2.9.2.25	get_solver_eps	51
2.9.2.26	increase_dual_deflation	51
2.9.2.27	increase_primal_deflation	52
2.9.2.28	is_inflate	52
2.9.2.29	lower_bound	52
2.9.2.30	pivot_element	53
2.9.2.31	primal_certificate	53
2.9.2.32	primal_feasible	54
2.9.2.33	print_module_options	54
2.9.2.34	process_bounded_variables	54
2.9.2.35	process_free_variables	55
2.9.2.36	process_initial_dual_solution	55
2.9.2.37	process_initial_primal_solution	56
2.9.2.38	process_perturbed_dual_solution	56
2.9.2.39	process_perturbed_primal_solution	57
2.9.2.40	read_lp	58
2.9.2.41	restore_dual_phase1	58
2.9.2.42	restore_primal_phase1	58
2.9.2.43	rho_d	59
2.9.2.44	rho_p	59
2.9.2.45	set_alpha	59
2.9.2.46	set_dual_phase1	60
2.9.2.47	set_eta	60
2.9.2.48	set_inflate	60
2.9.2.49	set_interface	61
2.9.2.50	set_lp	61
2.9.2.51	set_module	61
2.9.2.52	set_module_options	62
2.9.2.53	set_primal_phase1	62
2.9.2.54	shorten_basis_indices	62
2.9.2.55	solve_lp	63

2.9.2.56	upper_bound	63
2.10	Primal_deflation Struct Reference	64
2.10.1	Detailed Description	64
2.11	Report Class Reference	65
2.11.1	Detailed Description	66
2.12	Sm_lps3_2 Class Reference	67
2.12.1	Detailed Description	69
2.12.2	Member Function Documentation	69
2.12.2.1	adjust_eta	69
2.12.2.2	build_constraint_maps	69
2.12.2.3	dual_perturb	70
2.12.2.4	find_free_variables	70
2.12.2.5	get_accuracy	71
2.12.2.6	get_version	71
2.12.2.7	inflate_lp	71
2.12.2.8	init	71
2.12.2.9	primal_perturb	72
2.12.2.10	print_options	72
2.12.2.11	read_duals	72
2.12.2.12	read_general_data	73
2.12.2.13	read_lp	73
2.12.2.14	read_lp_mat	73
2.12.2.15	read_primals	74
2.12.2.16	read_right_hand_sides	74
2.12.2.17	read_simple_bounds	74
2.12.2.18	resize_lp	75
2.12.2.19	restore_dual	75
2.12.2.20	restore_primal	75
2.12.2.21	solve_dual_perturbed	76
2.12.2.22	solve_lp	76
2.12.2.23	solve_original	76
2.12.2.24	solve_primal_perturbed	77
2.12.2.25	transform_lp	77
2.13	Sm_lps4_0_1_0 Class Reference	79
2.13.1	Detailed Description	81
2.13.2	Member Function Documentation	81

2.13.2.1	adjust_eta	81
2.13.2.2	build_constraint_maps	82
2.13.2.3	dual_perturb	82
2.13.2.4	find_free_variables	82
2.13.2.5	get_accuracy	83
2.13.2.6	get_version	83
2.13.2.7	inflate_lp	83
2.13.2.8	init	83
2.13.2.9	lps_log	84
2.13.2.10	postprocess	84
2.13.2.11	primal_perturb	84
2.13.2.12	print_options	85
2.13.2.13	read_duals	85
2.13.2.14	read_general_data	85
2.13.2.15	read_lp	86
2.13.2.16	read_lp_mat	86
2.13.2.17	read_primals	87
2.13.2.18	read_right_hand_sides	87
2.13.2.19	read_simple_bounds	87
2.13.2.20	resize_lp	88
2.13.2.21	restore_dual	88
2.13.2.22	restore_primal	88
2.13.2.23	set_module_options	89
2.13.2.24	solve_dual_perturbed	89
2.13.2.25	solve_lp	90
2.13.2.26	solve_original	90
2.13.2.27	solve_primal_perturbed	91
2.13.2.28	transform_lp	91
2.14	Sm_lps4_0_1_11 Class Reference	93
2.14.1	Detailed Description	95
2.14.2	Member Function Documentation	95
2.14.2.1	adjust_eta	95
2.14.2.2	build_constraint_maps	96
2.14.2.3	dual_perturb	96
2.14.2.4	find_free_variables	96
2.14.2.5	get_accuracy	97

2.14.2.6	get_version	97
2.14.2.7	inflate_lp	97
2.14.2.8	init	97
2.14.2.9	lps_log	98
2.14.2.10	postprocess	98
2.14.2.11	primal_perturb	98
2.14.2.12	print_options	99
2.14.2.13	read_duals	99
2.14.2.14	read_general_data	99
2.14.2.15	read_lp	100
2.14.2.16	read_lp_mat	100
2.14.2.17	read_primals	101
2.14.2.18	read_right_hand_sides	101
2.14.2.19	read_simple_bounds	101
2.14.2.20	resize_lp	102
2.14.2.21	restore_dual	102
2.14.2.22	restore_primal	102
2.14.2.23	set_module_options	103
2.14.2.24	solve_dual_perturbed	103
2.14.2.25	solve_lp	104
2.14.2.26	solve_original	104
2.14.2.27	solve_primal_perturbed	105
2.14.2.28	transform_lp	105
2.15	Sm_lps5_5 Class Reference	107
2.15.1	Detailed Description	110
2.15.2	Member Function Documentation	110
2.15.2.1	adjust_eta	110
2.15.2.2	build_constraint_maps	110
2.15.2.3	dual_perturb	111
2.15.2.4	find_free_variables	111
2.15.2.5	get_accuracy	112
2.15.2.6	get_version	112
2.15.2.7	inflate_lp	112
2.15.2.8	init	112
2.15.2.9	lps_log	113
2.15.2.10	primal_perturb	113

2.15.2.11	print_options	113
2.15.2.12	read_duals	113
2.15.2.13	read_general_data	114
2.15.2.14	read_lp	114
2.15.2.15	read_lp_mat	115
2.15.2.16	read_primals	115
2.15.2.17	read_right_hand_sides	115
2.15.2.18	read_simple_bounds	116
2.15.2.19	resize_lp	116
2.15.2.20	restore_dual	116
2.15.2.21	restore_primal	117
2.15.2.22	set_module_options	117
2.15.2.23	solve_dual_perturbed	118
2.15.2.24	solve_lp	118
2.15.2.25	solve_original	119
2.15.2.26	solve_primal_perturbed	119
2.15.2.27	transform_lp	120
2.15.3	Member Data Documentation	120
2.15.3.1	timeout	120
2.15.3.2	trace	120
2.15.3.3	verbosity	120
2.16	Solver_module Class Reference	122
2.16.1	Detailed Description	124
2.17	Solver_module_interface Struct Reference	125
2.17.1	Detailed Description	126
3	File Documentation	127
3.1	cLurupa.cpp File Reference	127
3.1.1	Detailed Description	129
3.1.2	Function Documentation	129
3.1.2.1	clu_cleanup	129
3.1.2.2	clu_get_alpha	129
3.1.2.3	clu_get_core_version	129
3.1.2.4	clu_get_eta	130
3.1.2.5	clu_get_lp_name	130
3.1.2.6	clu_get_module_version	130
3.1.2.7	clu_get_solver_eps	130

3.1.2.8	clu_init	131
3.1.2.9	clu_is_inflate	131
3.1.2.10	clu_is_lp_maximize	131
3.1.2.11	clu_lower_bound	131
3.1.2.12	clu_print_module_options	131
3.1.2.13	clu_read_lp	132
3.1.2.14	clu_set_alpha	132
3.1.2.15	clu_set_eta	132
3.1.2.16	clu_set_inflate	132
3.1.2.17	clu_set_interface	133
3.1.2.18	clu_set_lp	133
3.1.2.19	clu_set_module	133
3.1.2.20	clu_set_module_options	133
3.1.2.21	clu_solve_lp	133
3.1.2.22	clu_upper_bound	134
3.2	cLurupa.h File Reference	135
3.2.1	Detailed Description	137
3.2.2	Function Documentation	137
3.2.2.1	clu_cleanup	137
3.2.2.2	clu_get_alpha	137
3.2.2.3	clu_get_core_version	137
3.2.2.4	clu_get_eta	137
3.2.2.5	clu_get_lp_name	138
3.2.2.6	clu_get_module_version	138
3.2.2.7	clu_get_solver_eps	138
3.2.2.8	clu_init	138
3.2.2.9	clu_is_inflate	138
3.2.2.10	clu_is_lp_maximize	139
3.2.2.11	clu_lower_bound	139
3.2.2.12	clu_print_module_options	139
3.2.2.13	clu_read_lp	139
3.2.2.14	clu_set_alpha	140
3.2.2.15	clu_set_eta	140
3.2.2.16	clu_set_inflate	140
3.2.2.17	clu_set_module	140
3.2.2.18	clu_set_module_options	140

3.2.2.19	clu_upper_bound	141
3.3	Condition.cpp File Reference	142
3.3.1	Detailed Description	142
3.4	Condition.h File Reference	143
3.4.1	Detailed Description	143
3.5	globals.h File Reference	145
3.5.1	Detailed Description	146
3.5.2	Enumeration Type Documentation	146
3.5.2.1	Bound_status	146
3.5.2.2	Csv_style	147
3.5.2.3	Solver_status	147
3.5.2.4	Write_mps	147
3.5.3	Variable Documentation	148
3.5.3.1	bound_status_string	148
3.5.3.2	solver_status_string	148
3.6	Lp.cpp File Reference	149
3.6.1	Detailed Description	149
3.7	Lp.h File Reference	150
3.7.1	Detailed Description	150
3.8	Lurupa.cpp File Reference	152
3.8.1	Detailed Description	152
3.9	Lurupa.h File Reference	154
3.9.1	Detailed Description	154
3.10	lurupa_cmd.cpp File Reference	156
3.10.1	Detailed Description	158
3.10.2	Function Documentation	159
3.10.2.1	compute_dual	159
3.10.2.2	compute_lower	159
3.10.2.3	compute_primal	159
3.10.2.4	compute_upper	160
3.10.2.5	main	160
3.10.2.6	print_bound_stats	160
3.10.2.7	print_brief_version	161
3.10.2.8	print_model_data	161
3.10.2.9	print_usage	161
3.10.2.10	print_version	162

3.10.2.11 process_solving_status	162
3.10.2.12 report_bound_quality	162
3.10.2.13 report_dual	163
3.10.2.14 report_lower	163
3.10.2.15 report_primal	163
3.10.2.16 report_upper	163
3.10.2.17 rounded_string	164
3.10.2.18 start_timer	164
3.10.2.19 stop_timer	164
3.10.2.20 timer_diff	165
3.10.2.21 write_csv_table	165
3.10.2.22 write_latex_table	166
3.10.2.23 write_tables	167
3.11 Report.h File Reference	168
3.11.1 Detailed Description	168
3.12 Sm_lps3_2.cpp File Reference	169
3.12.1 Detailed Description	169
3.12.2 Function Documentation	170
3.12.2.1 get_func_pointers	170
3.13 Sm_lps3_2.h File Reference	171
3.13.1 Detailed Description	171
3.13.2 Function Documentation	172
3.13.2.1 get_func_pointers	172
3.14 Sm_lps4_0_1_0.cpp File Reference	173
3.14.1 Detailed Description	173
3.14.2 Function Documentation	174
3.14.2.1 get_func_pointers	174
3.15 Sm_lps4_0_1_0.h File Reference	175
3.15.1 Detailed Description	175
3.15.2 Function Documentation	176
3.15.2.1 get_func_pointers	176
3.16 Sm_lps4_0_1_11.cpp File Reference	177
3.16.1 Detailed Description	177
3.16.2 Function Documentation	178
3.16.2.1 get_func_pointers	178
3.17 Sm_lps4_0_1_11.h File Reference	179

3.17.1 Detailed Description	179
3.17.2 Function Documentation	180
3.17.2.1 get_func_pointers	180
3.18 Sm_lps5_5.cpp File Reference	181
3.18.1 Detailed Description	181
3.18.2 Function Documentation	182
3.18.2.1 get_func_pointers	182
3.19 Sm_lps5_5.h File Reference	183
3.19.1 Detailed Description	183
3.19.2 Function Documentation	184
3.19.2.1 get_func_pointers	184
3.20 Solver_module.h File Reference	186
3.20.1 Detailed Description	187

Chapter 1

Lurupa Documentation

Lurupa is a tool for computing rigorous error bounds in linear programming.

This documentation consists of four parts.

- **Introduction** (p. 1)

We will start with the question why at all do we need such tools. What can go wrong when we solve a linear program? What can a verification tool do for us in these cases?

- **Theory** (p. 2)

After that we look at the mathematics behind the computations done in Lurupa. What happens in the algorithms and how can we make sure that the computed bounds are rigorously, including rounding errors.

- **Usage** (p. 4)

The usage of Lurupa has to be divided into two parts.

- **Stand-alone usage** (p. 4)

In this section we will investigate the usage as a stand-alone command line tool.

- **API usage** (p. 7)

Here we look at the usage through the API.

- **License terms** (p. 8)

Finally the terms under which you are allowed to use Lurupa.

1.1 Introduction

The usefulness of verification tools for linear programming can be demonstrated with simple examples. Let's consider the following simple linear program

$$\begin{array}{ll}\min & x \\ \text{s.t.} & 1 \cdot x < 0.999999999 \\ & 1 < x < \infty.\end{array}$$

While the infeasibility of this problem can be easily seen, many linear programming solvers like CPLEX 9.000, lp_solve 5.5, and MATLAB R14SP3's linprog will regard this problem as feasible.

With the default accuracy they will happily return with an optimal solution not even displaying a warning.

A similar construction for an unbounded problem,

$$\begin{array}{ll} \min & 0.999999999x_1 - x_2 \\ \text{s.t.} & x_1 - x_2 = 0 \\ & x_1 \geq 0, x_2 \geq 0, \end{array}$$

fools CPLEX and `lp_solve` into returning 0 as the optimal solution for the variables x_i and thus as the optimal function value.

Given, these examples are very simple. They are just used to illustrate that even in these dimensions problems may arise. Adjusting the accuracy of the solvers would enable them to return the correct results. One may ask if 0.999 is different from 1? Is 0.999999 different? What about 0.999999999999? If not the solvers are correct. The values used above, however, are recognized by the solvers as being different from 1.

Going to higher dimensions the problem becomes more subtle. The dependencies between the constraints cannot be seen easily. Let us look at a problem from the Netlib lp library [7], a free collection of industrial and academical linear programs. With 56 constraints and 97 variables *adlittle* is one of its smaller members. While being in fact feasible, *adlittle* suffers from ill-posedness. Perturbing the right hand side of the equality constraints by subtracting a tiny multiple of the 96th column of the equation matrix renders the linear program infeasible. Running this problem through CPLEX and `lp_solve` does again return a solution without any warnings.

Now we will have a look at what Lurupa returns when confronted with these problems. Using `lp_solve` to compute approximate solutions as a start off, Lurupa returns for the primal infeasible problem the optimal value interval $[1, \infty]$. While the lower bound does not help us much here, the upper bound tells us, that the problem is at least very close to infeasibility as Lurupa could not find verified feasible points. What this means is explained in the next section. Computing bounds for the second, unbounded problem Lurupa returns the interval $[-\infty, 0]$. Here we can pinpoint the unboundedness in the infinite lower bound. Using the final example as input, Lurupa computes the bounds $[2.2549495685e + 05, \infty]$ (we need the `resetbas` option of the solver module in this case to deal with the bad numerics). Again the upper bound indicates the problem being at least very close to infeasibility.

On the other hand let us have a look at what Lurupa does when confronted with a feasible lp. Taking *adlittle* again, now with the original coefficients, Lurupa returns the interval $[2.2549495675e + 05, 2.2549496497e + 05]$ for the optimal value. This does not only give a verified interval containing the optimal value, but also verifies that feasible solutions in fact exist. This problem cannot be infeasible or unbounded, as the lower and upper bounds prove the existence of dual and primal feasible points, respectively.

Results of Lurupa being applied to the whole selection of Netlib problems is contained in [5].

1.2 Theory

The idea behind the computations done in Lurupa is to use a combination of iteratively perturbing the linear program and using interval arithmetic to verify feasible points of the original linear program. Once such a point is verified a rigorous bound on the optimal value can be derived. The details can be found in the work of Jansson [3]. Here only an overview is given.

1.2.1 Interval Arithmetic

Before we look at the actual algorithms, we need a tool to ensure computations to be correct even in the presence of rounding errors. This can be achieved with interval arithmetic. The basic idea is to not represent the value of an expression by a number but by an interval that is guaranteed to contain the exact value.

Let us have a look at a simple example. We use a decimal floating point arithmetic with only 2 digits mantissa and compute

$$fl(10 + 0.1) = fl(10.1) = 10.$$

Here $fl(x)$ means the floating point evaluation of x . The 0.1 is completely annihilated by rounding the result to 2 digits. Repeating this using an interval arithmetic gives us the following

$$flint(10 + 0.1) = [fl_{\nabla}(10 + 0.1), fl_{\Delta}(10 + 0.1)] = [10, 11].$$

We arrive at a guaranteed enclosure, 10.1 is certainly between 10 and 11.

But how do we get there? The equation gives a hint how to arrive at this result. The interval enclosure $flint(x)$ of the expression x is formed by the lower bound $fl_{\nabla}(x)$ and the upper bound $fl_{\Delta}(x)$. These are computed using directed rounding towards minus infinity and plus infinity, respectively.

Having arrived at intervals we now have to continue our computations using these. The objective stays just the same. Compute an interval that is guaranteed to contain the correct result using directed rounding. As we do not know which number in the intervals is the correct one, we have to take all into account. If we have to divide the result of the previous example by $[2, 3]$, this means

$$flint\left(\frac{[10, 11]}{[2, 3]}\right) = [3.3, 5.5].$$

The lower bound is the result of rounding 10 over 3 downwards, and the upper bound stems from rounding 11 over 2 upwards.

For a more thorough introduction to interval arithmetic have a look at Kearfott [4] and Hayes [2] and the references therein. The interval computations homepage [6] is also a good starting point.

1.2.2 Rigorous lower and upper bounds

Looking at a linear program in standard form

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & Ax \leq a \\ & Bx = b \\ & \underline{x} \leq x \leq \bar{x}, \end{aligned}$$

we use a standard linear programming solver to compute an approximate solution x . We make no assumptions about the accuracy of the computed approximations. A good approximation, however, usually results in a sharper enclosure of the optimal value.

The next step is to enforce the simple bounds \underline{x}, \bar{x} . We achieve this by setting the components of x that violate their corresponding bound to the violated bound. If $x_i > \bar{x}_i$ or $x_i < \underline{x}_i$, we set $x_i = \bar{x}_i$ or $x_i = \underline{x}_i$, respectively.

Now we use an interval linear system solver to compute a rigorous enclosure of the solution set of $Bx = b$. This step transforms the point vector x into an interval vector \mathbf{x} , which is verified to contain at least one point satisfying the equations.

Finally we just have to check if the inequalities $Ax \leq a$ and the simple bounds \underline{x}, \bar{x} are satisfied by \mathbf{x} and thus by all points contained in the interval vector. If this is the case, \mathbf{x} contains at least one point that satisfies all constraints. Taking the maximum of the objective function over the box \mathbf{x} , that is over the objective value of all points in the interval vector, we arrive at an objective value that is verified to be larger than the one of a feasible point. This in turn is larger than the optimal value, which leaves us with an upper bound on the optimal value.

If \mathbf{x} does not satisfy the constraints, the iteration starts. We perturb the linear program by lowering the right hand sides of the inequalities a and tightening the simple bounds \underline{x}, \bar{x} . Using this perturbed linear program as an input to the linear programming solver, we compute a new approximate solution x . With the above procedure we try again to derive an interval vector \mathbf{x} verified to contain a feasible point of the *original* linear program.

If the linear programming solver does not return an approximate solution in any stage of the iteration, due to numerical failure for example, the only upper bound we can derive is plus infinity. The same holds if the interval linear system solver cannot compute an enclosure of the solution set of the equations. As the equations do not change during the iteration, this can be ruled out after the first iteration.

The lower bound on the optimal value can be derived in a similar way from the dual linear program

$$\begin{aligned} \max \quad & a^T y + b^T z + \underline{x}^T u + \bar{x}^T v \\ \text{s.t.} \quad & A^T y + B^T z + u + v = c \\ & y \leq 0, u \geq 0, v \leq 0. \end{aligned}$$

Here, however, we can make use of the special structure of the dual, thus reducing the computational complexity. The details are contained in [3].

1.3 Usage

Lurupa can be used in two ways.

- **Stand-alone usage** (p. 4) using Lurupa over the command line
- **API usage** (p. 7) including Lurupa as a library into a larger framework

In each case an lp solver specific module is needed to translate the generic calls from Lurupa into calls to the solver API and to transfer data between Lurupa and the solver.

1.3.1 Stand-alone usage

The command line client `lurupa` is controlled through various options. Mandatory are the options to select an lp and a solver module.

- `-lp file` This option specifies the file containing the lp to be processed. The file has to be in a format that can be handled by the solver module.
- `-sm file` This option specifies the solver module and thus the lp solver that is used to compute the approximate solutions.

The most basic call, using a solver module `Module`, assumend to be located in the local directory, would thus be

```
lurupa -sm Module -lp lp.mps
```

Here we assumed the module knows how to parse `lp.mps`. This call, however, would just call the solver through the module and solve the lp. To compute the bounds we have to specify which bounds we want. This is done via the bound parameters

-lb This option specifies, that the lower bound for the optimal value shall be computed.

-ub This option specifies, that the upper bound for the optimal value shall be computed.

Computing both bounds for `lp.mps` is therefor done with

```
lurupa -sm Module -lp lp.mps -lb -ub
```

It's important to note that while the computed bounds are rigorous the command line client does not provide for rigorous printing of the bounds. This means that the values printed for the bounds are rounded to display precision.

Several more options are available to alter Lurupa's behaviour. These can be divided into options affecting the input, the algorithm, and the output.

Options affecting the input

-i d This option specifies, that the cost vector and the constraint matrices and right hand sides are inflated to intervals with the given midpoint and a radius of `d`. The parameters get multiplied by the interval $[1 - d, 1 + d]$.

Options affecting the algorithm

-alpha d This option changes the algorithm parameter α to `d`. For details on the parameter refer to [3] .

-eta d This option changes the algorithm parameter η to `d`. For details on the parameter refer to [3] .

-inflate This option tells Lurupa to inflate perturbed lps in case of infeasibility. Due to the exponential deflation used in the algorithm the gap between lps that can not be verified and lps that are considered infeasible by the solver might be missed. If this option is specified Lurupa tries to inflate the feasible region of perturbed, infeasible lps to reach this gap. Using this option increases the number of algorithm iterations.

Options affecting the output

-csv file This option tells Lurupa to append a line containing the comma seperated results to `file`. If not present, the extension `.csv` is appended to the filename.

-latex file This option tells Lurupa to append a line containing the results to `file`, formatted to be used in a LaTeX table. If not present, the extension `.tex` is appended to the filename.

-t This option causes all messages printed by Lurupa to be prepended with the current time.

-vn This option selects the verbosity level of the messages printed by Lurupa. The verbosity increases with `n`. Possible values are:

-v0: No messages Print no messages from Lurupa. The command line client just prints the final results.

- v1: Errors** Add error messages from Lurupa to -v0.
- v2: Warnings** Add warnings.
- v3: Brief** Add brief program flow messages like the iteration number and whether we are currently approximately solving a perturbed problem or verifying the feasibility of a solution.
- v4: Normal** Add more program flow messages like whether we are perturbing or approximately solving a perturbed lp, and intermediate results like the approximate optimal value of perturbed lps.
- v5: Verbose** Add detailed information like changes of the deflation parameters.

The default verbosity level is **-v2**.

-write_vm style This option tells Lurupa to save intermediate vectors and matrices to disk. The format depends on **style**. Specifying **octave** stores all these in readable by Octave, this means all in one file with meta information about the dimension and the name of the vector or matrix. The file is named after the lp. Specifying **matlab** stores these in one file per vector or matrix. These files are put into a directory named after the model.

Remain the options that print informational messages.

- h, -?, -help** These options tell Lurupa to print usage information about the command line client containing the supported parameters along with all parameters supported by the solver module if specified.
- V** This option tells Lurupa to print version information about the command line client, the core, and the solver module if specified. This also contains values like the set algorithm parameters.
- Vb** This option tells Lurupa to print a brief version of the information printed by **-V**.

As mentioned the solver module might support additional options. These are specific to the solver module. They have the common prefix **-sm**, with additional parameters appended with commas. A typical option would be **-sm,timeout,3600**, which might tell the solver to time out after 3600 seconds.

1.3.1.1 lp_solve solver modules

There are solver modules for different versions of lp_solve [1]. Development is done on the latest version of lp_solve with features backported as appropriate. Thus the modules support different options

lp_solve 3.2 No further options

lp_solve 4.0.1.0 and 4.0.1.11

- sm,timeout,i** This option sets the timeout in seconds to **i**. Setting this calls lp_solve's **set_timeout**, causing lp_solve to stop solving an lp if it takes longer than **i** seconds.
- sm,vn** This option selects the verbosity level of the messages printed by lp_solve. The verbosity increases with **n**. Possible values are (taken from the lp_solve documentation):
 - v0: NEUTRAL** Only some specific debug messages in debug print routines are reported.
 - v1: CRITICAL** Only critical messages are reported. Hard errors like instability, out of memory, ...

- v2:** SEVERE Only severe messages are reported. Errors.
- v3:** IMPORTANT Only important messages are reported. Warnings and Errors.
- v4:** NORMAL Normal messages are reported.
- v5:** DETAILED Detailed messages are reported. Like model size, continuing B&B improvements, ...
- v6:** FULL All messages are reported. Useful for debugging purposes and small models.

The default verbosity level is **v3**.

lp_solve 5.5

-sm,resetbas This option causes the solver module to re-solve lps in the case of numerical failure with an all-slack basis. While this increases the runtime it may help with numerically difficult lps.

-sm,s This option enables the default scaling of lp_solve (Numerical range-based scaling).

-sm,timeout,i This option sets the timeout in seconds to **i**. Setting this calls lp_solve's **set_timeout**, causing lp_solve to stop solving an lp if it takes longer than **i** seconds.

-sm,vn This option selects the verbosity level of the messages printed by lp_solve. The verbosity increases with **n**. Possible values are (taken from the lp_solve documentation):

- v0:** NEUTRAL Only some specific debug messages in debug print routines are reported.
- v1:** CRITICAL Only critical messages are reported. Hard errors like instability, out of memory, ...
- v2:** SEVERE Only severe messages are reported. Errors.
- v3:** IMPORTANT Only important messages are reported. Warnings and Errors.
- v4:** NORMAL Normal messages are reported.
- v5:** DETAILED Detailed messages are reported. Like model size, continuing B&B improvements, ...
- v6:** FULL All messages are reported. Useful for debugging purposes and small models.

The default verbosity level is **v3**.

-sm,wmps This option causes the solver module to save the perturbed lps in mps format. The filename is constructed of the lp name and the current iteration number.

1.3.2 API usage

Using Lurupa through the API means greater flexibility and allows to embed Lurupa in a larger framework.

C++

To access the API we need to include the two header files **globals.h** (p.145) and **Lurupa.h** (p.154). Now we need to create an instance of the **Lurupa** (p.36) class. All interaction with Lurupa is done via this object. As in the stand-alone case we need a solver module, which selects the lp solver to be used. Selecting the lp to be solved, we arrive at the following code that will appear more or less like this in every API usage of Lurupa.

```
#include <lurupa/globals.h>
#include <lurupa/Lurupa.h>

[...]

Lurupa lurupa;
bool module_set = lurupa.set_module("path/to/module");
bool lp_read = lurupa.read_lp("path/to/lp", 0);
```

The second parameter to `read_lp` reads the lp as it is and does not inflate it to an interval problem. Now we can compute an approximate solution and after that the rigorous bounds on the optimal value.

```
double optimal_value;
Solver_status status;
bool lp_solved = lurupa.solve_lp(optimal_value, status);

double lower_bound, upper_bound;
int lower_iterations, upper_iterations;
Bound_status st_lower = lurupa.lower_bound(lower_bound, lower_iterations);
Bound_status st_upper = lurupa.upper_bound(upper_bound, upper_iterations);
```

This is all that is necessary to compute rigorous bounds on the optimal value. For details on how to set the algorithm parameters, how to change output and the like refer to the reference of the class **Lurupa** (p.36).

C

There is a C wrapper for the library, which maps all calls to an internal **Lurupa** (p.36) object. To use it we have to include the **cLurupa.h** (p.135) header. The names of the routines are chosen after the names of the corresponding member routines of **Lurupa** (p.36) with a prefix of `clu`. The previous C++ example looks like the following using the C wrapper.

```
#include <lurupa/cLurupa.h>

[...]

BOOL module_set, lp_read, lp_solved;
double optimal_value, lower_bound, upper_bound;
BOUND_STATUS st_lower, st_upper;
int lower_iterations, upper_iterations;

clu_init();
module_set = clu_set_module("path/to/module");
lp_read = clu_read_lp("path/to/lp", 0);
lp_solved = clu_solve_lp(&optimal_value, &status);
st_lower = clu_lower_bound(&lower_bound, &lower_iterations);
st_upper = clu_upper_bound(&upper_bound, &upper_iterations);
```

For a complete list of the wrapper functions refer to the reference of the **cLurupa.h** (p.135) header.

1.4 License terms

Copyright (C) 2006 by Christian Keil

Lurupa is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

GNU LESSER GENERAL PUBLIC LICENSE
Version 2.1, February 1999

Copyright (C) 1991, 1999 Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL. It also counts
as the successor of the GNU Library Public License, version 2, hence
the version number 2.1.]

Preamble

The licenses for most software are designed to take away your
freedom to share and change it. By contrast, the GNU General Public
Licenses are intended to guarantee your freedom to share and change
free software--to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some
specially designated software packages--typically libraries--of the
Free Software Foundation and other authors who decide to use it. You
can use it too, but we suggest you first think carefully about whether
this license or the ordinary General Public License is the better
strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use,
not price. Our General Public Licenses are designed to make sure that
you have the freedom to distribute copies of free software (and charge
for this service if you wish); that you receive source code or can get
it if you want it; that you can change the software and use pieces of
it in new free programs; and that you are informed that you can do
these things.

To protect your rights, we need to make restrictions that forbid
distributors to deny you these rights or to ask you to surrender these
rights. These restrictions translate to certain responsibilities for
you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis
or for a fee, you must give the recipients all the rights that we gave
you. You must make sure that they, too, receive or can get the source
code. If you link other code with the library, you must provide
complete object files to the recipients, so that they can relink them
with the library after making changes to the library and recompiling
it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the
library, and (2) we offer you this license, which gives you legal
permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that
there is no warranty for the free library. Also, if the library is
modified by someone else and passed on, the recipients should know
that what they have is not the original version, so that the original
author's reputation will not be affected by problems that might be
introduced by others.

Finally, software patents pose a constant threat to the existence of
any free program. We wish to make sure that a company cannot
effectively restrict the users of a free program by obtaining a
restrictive license from a patent holder. Therefore, we insist that
any patent license obtained for a version of the library must be

consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

GNU LESSER GENERAL PUBLIC LICENSE TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under

copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- a) The modified work must itself be a software library.
- b) You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
- c) You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
- d) If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

(For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the

entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

- a) Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
- b) Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.
- c) Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
- d) If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
- e) Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

- a) Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
- b) Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing

to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the

ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the library's name and a brief idea of what it does.>
Copyright (C) <year> <name of author>
```

```
This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.
```

```
This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
Lesser General Public License for more details.
```

```
You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
```

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the
library 'Frob' (a library for tweaking knobs) written by James Random Hacker.
```

```
<signature of Ty Coon>, 1 April 1990
Ty Coon, President of Vice
```

That's all there is to it!

Chapter 2

Class Documentation

2.1 Bound Struct Reference

Information about a Bound.

Public Attributes

- **bool compute**
Whether the bound shall be computed.
- **int iterations**
Iteration count.
- **double proc_time**
Process time needed to compute bound in seconds.
- **double realtime**
Wall clock time needed to compute bound in seconds.
- **Bound_status status**
Status of the bound computation (i.e., successfull or failed due to ...).
- **const char * time_status**
Status of the time taken (i.e., if any wrap around is possible / occurred).
- **double value**
Value of the bound.

2.1.1 Detailed Description

Information about a Bound.

This structure aggregates the value of a single (lower or upper) bound and corresponding data, that is the status of the bound and the iterations and the time necessary to compute it.

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

- **lurupa_cmd.cpp**

2.2 Certificate Struct Reference

A certificate.

Public Attributes

- **bool compute**
Whether the certificate shall be computed.
- **double proc_time**
Process time needed to compute certificate in seconds.
- **double realtime**
Wall clock time needed to compute certificate in seconds.
- **Bound_status status**
Status of the certificate computation.
- **const char * time_status**
Status of the time taken (i.e., if any wrap around is possible / occurred).

2.2.1 Detailed Description

A certificate.

This structure stores information about a certificate.

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

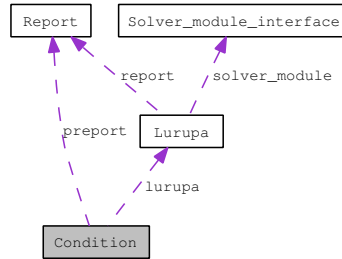
- **lurupa_cmd.cpp**

2.3 Condition Class Reference

Condition numbers

```
#include <lurupa/Condition.h>
```

Collaboration diagram for Condition:



Public Member Functions

- void **cond** (const **Lp** *lp, INTERVAL &enclosure)
Compute verified condition number.
- **Condition** (Lurupa *lurupa, Report *report)
Constructor.
- void **rho_d** (const **Lp** *lp, INTERVAL &enclosure)
Compute distance to dual infeasibility.
- void **rho_p** (const **Lp** *lp, INTERVAL &enclosure)
Compute distance to primal infeasibility.
- **~Condition** ()
Destructor.

Private Member Functions

- **Lp** * **generate_lp_rho_d** (const **Lp** *lp)
Generate lp to compute distance to dual infeasibility.
- **Lp** * **generate_lp_rho_p** (const **Lp** *lp)
Generate lp to compute distance to primal infeasibility.
- int **norm** (const **Lp** *lp, INTERVAL &enclosure)
Enclosure of the norm of an lp.
- INTERVAL **norm_1** (const INTERVAL_VECTOR &ix)
Enclosure of the 1 norm of a vector.

- **INTERVAL norm_11** (const INTERVAL_MATRIX &IA, const INTERVAL_MATRIX &IB)
Enclosure of the 1,1 operator norm of a matrix.
- **INTERVAL norm_F** (const INTERVAL_MATRIX &IA, const INTERVAL_MATRIX &IB)
Enclosure of the Frobenius norm of a matrix.
- **INTERVAL norm_L1** (const INTERVAL_MATRIX &IA, const INTERVAL_MATRIX &IB)
Enclosure of the L1 norm of a matrix.
- void **rot** (int *rg, int i1, int i2, int i3)
Rotate elements of array.
- void **swap** (int *rg, int i, int iT)
Swap elements of array.

Private Attributes

- **Lurupa * lurupa**
***Lurupa** (p. 36) reference for bound computation.*
- **Report * preport**
Reporting and debugging.

2.3.1 Detailed Description

Condition numbers

This class implements the computation of verified condition numbers. It is based on Ordonez and Freund's [8] linear programming characterization of the condition number. This in turn can be bounded with **Lurupa** (p. 36).

2.3.2 Constructor & Destructor Documentation

2.3.2.1 Condition::Condition (Lurupa * *lurupa*, Report * *preport*)

Constructor.

Parameters:

- ← *lurupa* **Lurupa** (p. 36) reference for bound computations.
- ← *preport* **Report** (p. 65) reference for reporting and debugging.

2.3.3 Member Function Documentation

2.3.3.1 `Lp * Condition::generate_lp_rho_d (const Lp * lp) [private]`

Generate lp to compute distance to dual infeasibility.

Generate the lp to compute the distance to dual infeasibility. Ordonez and Freund's lp (3.15) translates to the following lp. Note the transformation from p_L to $-p_L$.

$$\begin{array}{ll}
 \min & (1 \ 0 \ 0 \ 0 \ 0 \ 0) \begin{pmatrix} y \\ h_L \\ h_E \\ x \\ p \\ g \end{pmatrix} \\
 s.t. & \begin{pmatrix} -1 & e & e & 0 & 0 & 0 \\ -1 & 0 & 0 & c^T & 0 & 1 \\ -1 & 0 & 0 & -c^T & 0 & -1 \\ 0 & -I & 0 & A & I & 0 \\ 0 & 0 & -I & B & 0 & 0 \\ 0 & -I & 0 & -A & -I & 0 \\ 0 & 0 & -I & -B & 0 & 0 \end{pmatrix} \begin{pmatrix} y \\ h_L \\ h_E \\ x \\ p_L \\ g \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \\
 & x_{LB} \geq 0, x_{UB} \leq 0 \\
 & p \geq 0, g \geq 0 \\
 & y, h_L, h_E \geq 0
 \end{array}$$

References `Lp::free_variables`, `Lp::free_variables_size`, `Lp::ia`, `Lp::IA`, `Lp::ib`, `Lp::IB`, `Lp::ic`, `Lp::infinite`, `Lp::ix`, `Lp::iy`, `Lp::iz`, `Lp::lurupa`, `Lp::maximize`, `Lp::name`, `Lp::non_fixed_vars`, `Lp::xl`, and `Lp::xu`.

Referenced by `rho_d()`.

2.3.3.2 `Lp * Condition::generate_lp_rho_p (const Lp * lp) [private]`

Generate lp to compute distance to primal infeasibility.

Generate the lp to compute the distance to primal infeasibility. Ordonez and Freund's lp (3.14)

translates to the following lp. Note the transformation from y_L to $-y_L$.

$$\begin{aligned}
 \min \quad & (1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 0) \begin{pmatrix} x \\ h \\ y_L \\ y_E \\ v \\ s^+ \\ s^- \end{pmatrix} \\
 s.t. \quad & \begin{pmatrix} -1 & e & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & -a^T & b^T & -1 & 0 & 0 \\ -1 & 0 & a^T & -b^T & 1 & 0 & 0 \\ 0 & -I & -A^T & B^T & 0 & I & -I \\ 0 & -I & A^T & -B^T & 0 & -I & I \\ 0 & 0 & 0 & 0 & -1 & -\underline{x}^T & +\overline{x}^T \end{pmatrix} \begin{pmatrix} x \\ h \\ y_L \\ y_E \\ v \\ s^+ \\ s^- \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \\
 & x, h, y_L, s_{L_B}^+, s_{U_B}^- \geq 0 \\
 & s_{N \setminus L_B}^+, s_{N \setminus U_B}^- = 0
 \end{aligned}$$

References Lp::free_variables, Lp::free_variables_size, Lp::ia, Lp::IA, Lp::ib, Lp::IB, Lp::ic, Lp::infinite, Lp::ix, Lp::iy, Lp::iz, Lp::lurupa, Lp::maximize, Lp::name, Lp::non_fixed_vars, Lp::xl, and Lp::xu.

Referenced by rho_p().

2.3.3.3 int Condition::norm (const Lp * lp, INTERVAL & enclosure) [private]

Enclosure of the norm of an lp.

Compute an enclosure of the norm of lp. This is bounded by

References Lp::IA, Lp::ia, Lp::IB, Lp::ib, Lp::ic, Lp::ix, norm_1(), norm_1l(), norm_F(), norm_L1(), preport, and Report::print().

Referenced by cond().

2.3.3.4 INTERVAL Condition::norm_1 (const INTERVAL_VECTOR & ix) [private]

Enclosure of the 1 norm of a vector.

Compute an enclosure of the 1 norm of ix.

Parameters:

$\leftarrow ix$ the vector

Returns:

the norm enclosure

References Lp::ix.

Referenced by norm(), norm_1l(), and norm_L1().

2.3.3.5 INTERVAL Condition::norm_11 (const INTERVAL_MATRIX & *IA*, const INTERVAL_MATRIX & *IB*) [private]

Enclosure of the 1,1 operator norm of a matrix.

Compute an enclosure of the 1,1 norm of the matrix

$$\left\| \begin{pmatrix} A \\ B \end{pmatrix} \right\| := \max_j \left\| \begin{pmatrix} A \\ B \end{pmatrix} : j \right\|_1.$$

Parameters:

← *IA* upper part of the matrix

← *IB* lower part of the matrix

Returns:

the norm enclosure

References norm_1().

Referenced by norm().

2.3.3.6 INTERVAL Condition::norm_F (const INTERVAL_MATRIX & *IA*, const INTERVAL_MATRIX & *IB*) [private]

Enclosure of the Frobenius norm of a matrix.

Compute an enclosure of the Frobenius norm of the matrix $\begin{pmatrix} A \\ B \end{pmatrix}$.

Parameters:

← *IA* upper part of the matrix

← *IB* lower part of the matrix

Returns:

the norm enclosure

Referenced by norm().

2.3.3.7 INTERVAL Condition::norm_L1 (const INTERVAL_MATRIX & *IA*, const INTERVAL_MATRIX & *IB*) [private]

Enclosure of the L1 norm of a matrix.

Compute an enclosure of the L1 norm of the matrix $\begin{pmatrix} A \\ B \end{pmatrix}$.

Parameters:

← *IA* upper part of the matrix

← *IB* lower part of the matrix

Returns:

the norm enclosure

References `norm_1()`.

Referenced by `norm()`.

2.3.3.8 void Condition::rho_d (const Lp * lp, INTERVAL & enclosure)

Compute distance to dual infeasibility.

Compute distance to dual infeasibility ρ_d . Consists of the following steps:

- generate new lp
- tell module to transform lp to solver representation
- iterate over i/j settings maintaining enclosure of optimal value, i.e.,
 - set `x_i = j`
 - compute lower/upper bound
 - update `enclosure`, set to smallest lower/upper bound

Parameters:

- ← **lp** Lp to compute distance to primal infeasibility for
- **enclosure** Enclosure of distance to primal infeasibility

References `Lp::free_variables`, `Lp::free_variables_size`, `generate_lp_rho_d()`, `Lp::IA`, `Lp::IB`, `Lp::infinite`, `Lp::ix`, `Lurupa::lower_bound()`, `Solver_module_interface::lp2solver`, `lurupa`, `Lp::non_fixed_vars`, `preport`, `Report::print()`, `Solver_module_interface::set_bounds`, `Lurupa::solve_lp()`, `Lurupa::solver_module`, `swap()`, `Lurupa::upper_bound()`, `Lp::xl`, and `Lp::xu`.

Referenced by `cond()`, and `Lurupa::rho_d()`.

2.3.3.9 void Condition::rho_p (const Lp * lp, INTERVAL & enclosure)

Compute distance to primal infeasibility.

Compute distance to primal infeasibility ρ_p . Consists of the following steps:

- generate new lp
- tell module to transform lp to solver representation
- iterate over i/j settings maintaining enclosure of optimal value, i.e.,
 - set `y_i = j`
 - compute lower/upper bound
 - update `enclosure`, set to smallest lower/upper bound

Parameters:

- ← **lp** Lp (p. 28) to compute distance to primal infeasibility for
- **enclosure** Enclosure of distance to primal infeasibility

References `Lp::free_variables`, `Lp::free_variables_size`, `generate_lp_rho_p()`, `Lp::IA`, `Lp::IB`, `Lp::infinite`, `Lp::ix`, `Lurupa::lower_bound()`, `Solver_module_interface::lp2solver`, `lurupa`, `Lp::non_fixed_vars`, `preport`, `Report::print()`, `rot()`, `Solver_module_interface::set_bounds`, `Lurupa::solve_lp()`, `Lurupa::solver_module`, `swap()`, `Lurupa::upper_bound()`, `Lp::xl`, and `Lp::xu`.

Referenced by `cond()`, and `Lurupa::rho_p()`.

2.3.3.10 void Condition::rot (int * *rg*, int *i1*, int *i2*, int *i3*) [private]

Rotate elements of array.

Rotate the elements with index *i1*, *i2* and *i3* of array *rg* clockwise.

Parameters:

- ↔ *rg* the array
- ← *i1* index to rotate
- ← *i2* index to rotate
- ← *i3* index to rotate

References swap().

Referenced by rho_p().

2.3.3.11 void Condition::swap (int * *rg*, int *i*, int *iT*) [private]

Swap elements of array.

Swap the elements with index *i* and *iT* of array *rg*.

Parameters:

- ↔ *rg* the array
- ← *i* index to swap
- ← *iT* index to swap

Referenced by rho_d(), rho_p(), and rot().

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

- **Condition.h**
- **Condition.cpp**

2.4 Enclosure Struct Reference

An enclosure.

Public Attributes

- bool **compute**
Whether the enclosure shall be computed.
- double **lower**
The lower bound.
- double **upper**
The upper bound.

2.4.1 Detailed Description

An enclosure.

This structure stores information about an enclosure of a value.

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

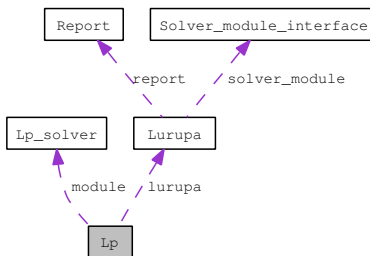
- **lurupa_cmd.cpp**

2.5 Lp Class Reference

A linear program.

```
#include <lurupa/Lp.h>
```

Collaboration diagram for Lp:



Public Member Functions

- **bool is_consistent () const**
Check lp for consistency.
- **Lp (Lurupa *lurupa)**
Constructor.
- **Lp ()**
Constructor.
- **void set_lurupa (Lurupa *lurupa)**
*Set **Lurupa** (p. 36) instance.*
- **~Lp ()**
Destructor.

Public Attributes

- **Solver_status feasibility**
Lp-solver's judgement of problem's feasibility.
- **int * free_variables**
The indices of the free variables.
- **int free_variables_size**
Number of free variables in model.
- **INTERVAL_VECTOR ia**
Right hand sides of the inequalities.
- **INTERVAL_MATRIX IA**

The inequality matrix.

- **INTERVAL_VECTOR ib**

Right hand sides of the equations.

- **INTERVAL_MATRIX IB**

The equation matrix.

- **INTERVAL_VECTOR ic**

The cost vector.

- **REAL infinite**

Used lp-solvers representation of infinity (e.g. as bound value).

- **INTERVAL_VECTOR ix**

Solution of the last solved problem (changes during algorithm as perturbed problems are solved).

- **INTERVAL_VECTOR iy**

Duals of the inequalities of the last solved problem (changes during algorithm as perturbed problems are solved).

- **INTERVAL_VECTOR iz**

Duals of the equations of the last solved problem (changes during algorithm as perturbed problems are solved).

- **Lurupa * lurupa**

*Reference to **Lurupa** (p. 36) instance.*

- **bool maximize**

True if objective function is to be maximized, false otherwise.

- **Lp_solver * module**

Additional storage for module data.

- **char * name**

Name of the lp model.

- **int non_fixed_vars**

Number of non fixed variables in model.

- **VECTOR xl**

The simple lower bound.

- **VECTOR xu**

The simple upper bound.

2.5.1 Detailed Description

A linear program.

This class is Lurupa's representation of an lp. The problem is stored in PROFILs interval types.

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

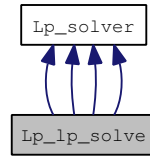
- **Lp.h**
- **Lp.cpp**

2.6 Lp_lp_solve Class Reference

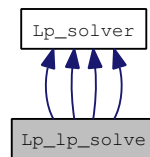
Information about lp_solve's problem.

```
#include <Sm_lps3_2.h>
```

Inheritance diagram for Lp_lp_solve:



Collaboration diagram for Lp_lp_solve:



Public Attributes

- **lprec * lp**
Reference to lp_solve's structure representing the lp.
- **int * mp_eq_con**
Map from equation to constraint.
- **int * mp_le_con**
Map from less-equal to constraint.
- **int * mp_parts**
Map connecting positive and negative parts of free variables.

2.6.1 Detailed Description

Information about lp_solve's problem.

This class stores additional information about lp_solve's representation of an lp.

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

- **Sm_lps3_2.h**
- **Sm_lps4_0_1_0.h**
- **Sm_lps4_0_1_11.h**
- **Sm_lps5_5.h**
- **Sm_lps3_2.cpp**

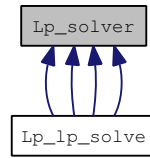
- `Sm_lps4_0_1_0.cpp`
- `Sm_lps4_0_1_11.cpp`
- `Sm_lps5_5.cpp`

2.7 Lp_solver Class Reference

Information about solver specific lp representation.

```
#include <Solver_module.h>
```

Inheritance diagram for Lp_solver:



2.7.1 Detailed Description

Information about solver specific lp representation.

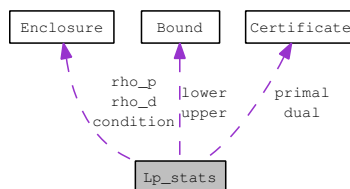
The documentation for this class was generated from the following file:

- **Solver_module.h**

2.8 Lp_stats Struct Reference

Information about an Lp.

Collaboration diagram for Lp_stats:



Public Attributes

- **bool automatic**
Whether bounds are automatically selected for computing based on approximate solver output.
- **double bound_quality**
Relative accuracy of the computed bounds.
- **Enclosure condition**
Data for enclosure of condition number.
- **Certificate dual**
The dual certificates data.
- **bool is_maximize**
Whether objective is to be maximized.
- **Bound lower**
The lower bound's data.
- **const char * name**
Name of the model.
- **double optimal_value**
Approximate optimal value from solver.
- **char * path**
Path to the lp file.
- **Certificate primal**
The primal certificates data.
- **Enclosure rho_d**
Data for enclosure of distance to dual infeasibility.
- **Enclosure rho_p**

Data for enclosure of distance to primal infeasibility.

- double **solve_proc_time**
Process time needed to solve the lp in seconds.
- double **solve_realtime**
Wall clock time needed to solve the lp in seconds.
- const char * **solve_time_status**
Status of the time taken (i.e., if any wrap around is possible / occurred).
- **Solver_status status**
Status of the solver (e.g., found optimal solution, model unbounded, failure while solving).
- double **total_proc_time**
Process time needed to solve the lp in seconds.
- double **total_realtime**
Wall clock time needed to solve the lp in seconds.
- const char * **total_time_status**
Status of the time taken (i.e., if any wrap around is possible / occurred).
- **Bound upper**
The upper bound's data.

2.8.1 Detailed Description

Information about an Lp.

This structure aggregates information about a linear program. The path to its file representation, information about the model itself, about the lp-solver solving the model and about the bounds for the optimal value.

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

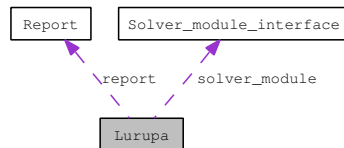
- **lurupa_cmd.cpp**

2.9 Lurupa Class Reference

Lurupa's core.

```
#include <lurupa/Lurupa.h>
```

Collaboration diagram for Lurupa:



Public Member Functions

- void **cond** (**Lp** *lp, double &lower, double &upper)
Compute verified condition number.
- **Bound_status** **dual_certificate** (**Lp** *lp)
Compute certificate for primal infeasibility.
- const char * **get_core_version** () const
Get core version string.
- char * **get_lp_name** () const
Get linear program's name.
- bool **is_lp_maximize** () const
Return whether linear program is maximized.
- **Bound_status** **lower_bound** (**Lp** *lp, double &bound, int &iterations)
Compute lower bound.
- **Lurupa** ()
Constructor.
- **Bound_status** **primal_certificate** (**Lp** *lp)
Compute certificate for dual infeasibility.
- void **print_core_brief_version** () const
Print brief core information.
- void **print_core_version** () const
Print core information.
- void **print_module_brief_version** () const
Print brief module information.
- void **print_module_version** () const

Print module information.

- void **rho_d** (**Lp** *lp, double &lower, double &upper)
Compute distance to dual infeasibility.
- void **rho_p** (**Lp** *lp, double &lower, double &upper)
Compute distance to primal infeasibility.
- **Bound_status upper_bound** (**Lp** *lp, double &bound, int &iterations)
Compute upper bound.
- **~Lurupa** ()
Destructor.
- double **get_alpha** () const
Get algorithm parameter alpha.
- double **get_eta** () const
Get algorithm parameter eta.
- bool **is_inflate** () const
Is inflation set.
- void **set_alpha** (double alpha)
Set algorithm parameter alpha.
- void **set_eta** (double eta)
Set algorithm parameter eta.
- void **set_inflate** (bool inflate)
Set inflation.
- const char * **get_module_version** () const
Get solver module version.
- double **get_solver_eps** () const
Get solver's accuracy.
- void **lp2solver** (**Lp** *lp)
*Set linear program from **Lurupa** (p. 36).*
- void **print_module_options** () const
Print options supported by module.
- **Lp** * **read_lp** (FILE *in, const double &relative_interval_radius)
Read linear program.
- void **set_interface** (**Solver_module_interface** module)
Set solver module.
- bool **set_lp** (**Lp** *lp, const double relative_interval_radius)

Set linear program from solver.

- **bool set_module** (char *module_path)
Set solver module.
- **bool set_module_options** (Lp *lp, int argc, char *argv[])
Set solver options.
- **bool set_solution** (Lp *lp, void *p)
Set linear program's solution.
- **Solver_status solve_lp** (Lp *lp, double &optimal_value)
Solve linear program.

Public Attributes

- **Report report**
Reporting and debugging.

Private Member Functions

- **void restore_dual_phase1** (Lp *lp, const INTERVAL_VECTOR &iaT, const INTERVAL_VECTOR &ibT, const VECTOR &xlT, const VECTOR &xuT, const int non_fixed_varsT, const int free_variables_sizeT)
Restore lp after dual phase 1 approach.
- **void restore_primal_phase1** (Lp *lp, const INTERVAL_VECTOR &icT, const bool maximize)
Restore lp after primal phase 1 approach.
- **void set_dual_phase1** (Lp *lp)
Set dual phase 1 lp.
- **void set_primal_phase1** (Lp *lp, bool *maximize)
Set primal phase 1 lp.

Find basis

Functions to find basis

These functions are used to find a regular submatrix to rigorously verify the primal or dual constraints.

- **void add_column_to_basis** (int step, INTERVAL_MATRIX &A, INTERVAL_MATRIX &IBasis, int *basis_indices)
Add column to basis.
- **void do_pivot** (MATRIX &LU, int pivot_index, int step, int *basis_indices)
Do the pivot.

- void **eliminate** (MATRIX &LU, int step)
Perform an elimination step.
- bool **find_basis** (INTERVAL_MATRIX &A, int *&basis_indices, int size_basis_indices, INTERVAL_MATRIX &IBasis, MATRIX &Basis_mid_inverse)
Find basis for linear system.
- double **pivot_element** (MATRIX LU, int step, int &pivot_index)
Find pivot element.
- void **shorten_basis_indices** (int *&basis_indices, int basis_size)
Shorten basis indices array.

Lower bound

Compute minimizing lower bound

These functions are used to compute a lower bound for the minimal objective value.

- void **compute_dual_deflation** (Lp *lp, VECTOR &d_c)
Compute dual deflation parameters.
- **Bound_status compute_lower** (Lp *lp, double &bound)
Compute minimizing lower bound.
- void **decrease_dual_deflation** (Lp *lp, VECTOR &d_c)
Decrease dual deflation parameters.
- bool **find_constraint_basis** (Lp *lp, INTERVAL_MATRIX &IResidual, int *&basis_indices, INTERVAL_MATRIX &IBasis, MATRIX &Basis_mid_inverse)
Find constraint basis.
- void **increase_dual_deflation** (Lp *lp, VECTOR &d_c, const INTERVAL_VECTOR &id_pos, const INTERVAL_VECTOR &id_neg)
Increase dual deflation parameters.
- void **process_initial_dual_solution** (Lp *lp, **Bound_status** &status, double &bound, const INTERVAL_MATRIX &IBasis, const MATRIX &Basis_mid_inverse, const INTERVAL_MATRIX &IResidual, const int *basis_indices)
Process initial dual solution.
- void **process_perturbed_dual_solution** (Lp *lp, **Bound_status** &status, double &bound, const INTERVAL_MATRIX &IBasis, const MATRIX &Basis_mid_inverse, const INTERVAL_MATRIX &IResidual, const int *basis_indices, VECTOR &deflation_c)
Process perturbed dual solution.

Upper bound

Compute minimizing upper bound

These functions are used to compute an upper bound for the minimal objective value.

- void **compute_primal_deflation** (Lp *lp, **Primal_deflation** &d)
Compute primal deflation parameters.

- **Bound_status compute_upper** (**Lp** *lp, double &bound)
Compute minimizing upper bound.
- **void decrease_primal_deflation** (**Lp** *lp, **Primal_deflation** &d)
Decrease primal deflation parameters.
- **bool find_equation_basis** (**Lp** *lp, **INTERVAL_MATRIX** &IResidual, int *&basis_indices, **INTERVAL_MATRIX** &IBasis, **MATRIX** &Basis_mid_inverse)
Find equation basis.
- **void increase_primal_deflation** (**Lp** *lp, **Primal_deflation** &d)
Increase primal deflation parameters.
- **void process_initial_primal_solution** (**Lp** *lp, **Bound_status** &status, double &bound, const **INTERVAL_MATRIX** &IBasis, const **MATRIX** &Basis_mid_inverse, const **INTERVAL_MATRIX** &IResidual, const int *basis_indices)
Process initial primal solution.
- **void process_perturbed_primal_solution** (**Lp** *lp, **Bound_status** &status, double &bound, const **INTERVAL_MATRIX** &IBasis, const **MATRIX** &Basis_mid_inverse, const **INTERVAL_MATRIX** &IResidual, const int *basis_indices, **Primal_deflation** &deflation)
Process perturbed primal solution.

Dual feasibility

Establish dual feasibility

These functions are used to establish a dual feasible solution starting from an approximate one delivered by the solver.

- **Bound_status establish_dual_feasibility** (**Lp** *lp, const **INTERVAL_MATRIX** &IBasis, const **MATRIX** &Basis_mid_inverse, const **INTERVAL_MATRIX** &IResidual, const int *basis_indices, **INTERVAL_VECTOR** &id_pos, **INTERVAL_VECTOR** &id_neg)
Establish dual feasibility.
- **void establish_ii_simple_bounds** (**Lp** *lp)
Establish simple bounds on lp.ii.
- **Bound_status establish_lagrange_parameters** (**Lp** *lp, const **INTERVAL_MATRIX** &IBasis, const **MATRIX** &Basis_mid_inverse, const **INTERVAL_MATRIX** &IResidual, const int *basis_indices, **INTERVAL_VECTOR** &id_pos, **INTERVAL_VECTOR** &id_neg)
Establish lagrange parameters.
- **Bound_status process_bounded_variables** (**Lp** *lp, **INTERVAL_VECTOR** &id_pos, **INTERVAL_VECTOR** &id_neg)
Process bounded variables.
- **Bound_status process_free_variables** (**Lp** *lp, const **INTERVAL_MATRIX** &IBasis, const **MATRIX** &Basis_mid_inverse, const **INTERVAL_MATRIX** &IResidual, const int *basis_indices)
Process free variables.

Primal feasibility

Establish primal feasibility

These functions are used to establish a primal feasible solution starting from an approximate one delivered by the solver.

- **bool establish_equality_constraints** (**Lp** *lp, const INTERVAL_MATRIX &IBasis, const MATRIX &Basis_mid_inverse, const INTERVAL_MATRIX &IResidual, const int *basis_indices)
Establish equality constraints.
- **void establish_ix_simple_bounds** (**Lp** *lp)
Establish simple bounds on lp.ix.
- **Bound_status establish_primal_feasibility** (**Lp** *lp, const INTERVAL_MATRIX &IBasis, const MATRIX &Basis_mid_inverse, const INTERVAL_MATRIX &IResidual, const int *basis_indices)
Establish primal feasibility.
- **bool primal_feasible** (**Lp** *lp)
Check for primal feasibility.

Private Attributes

- **double alpha**
Algorithm parameter.
- **double eps**
Algorithm parameter.
- **double eta**
Algorithm parameter.
- **bool inflate**
Try inflation to compute bound.
- **int iteration**
Current iteration count.
- **int max_iterations**
Upper limit for iterations.
- **lt_dlhandle mod_handle**
Handle to solver module object.
- **Solver_module_interface solver_module**
Interface to solver module.

Friends

- class **Condition**
- class **Lp**

2.9.1 Detailed Description

Lurupa's core.

This class implements Lurupa's core logic. It implements the interface visible to the user as well as the algorithm logic to compute the verified bounds using the interface to the solver modules.

2.9.2 Member Function Documentation

2.9.2.1 void Lurupa::add_column_to_basis (int *step*, INTERVAL_MATRIX & *A*, INTERVAL_MATRIX & *IBasis*, int * *basis_indices*) [private]

Add column to basis.

Set column *#step* of *IBasis* to the column of *A* designated by basis index *#step*. Zero out the column in *A*.

Parameters:

- ← *step* index into *IBasis* and *basis_indices*
- ↔ *A* contains the column to add / zero out
- *IBasis* receives the basis column
- ← *basis_indices* indices of basis columns

Referenced by `find_basis()`.

2.9.2.2 void Lurupa::compute_dual_deflation (Lp * *lp*, VECTOR & *d_c*) [private]

Compute dual deflation parameters.

Compute the initial dual deflation parameters for *lp*.

Parameters:

- ← *lp* the LP
- *d_c* stores the dual deflation paramters

References `alpha`, `eps`, `eta`, `Report::get_write_vm()`, `Lp::IA`, `Lp::IB`, `Lp::ic`, `Lp::infinite`, `Lp::iy`, `Lp::iz`, `Lp::name`, `report`, `Report::write_vector()`, `Lp::xl`, and `Lp::xu`.

Referenced by `compute_lower()`.

2.9.2.3 Bound_status Lurupa::compute_lower (Lp * *lp*, double & *bound*) [private]

Compute minimizing lower bound.

Compute the lower bound for the objective function of a minimizing problem. Try iteratively to transform the approximate dual solutions of perturbed problems into intervals verified to contain dual feasible points of the original problem.

Parameters:

- ← *lp* the LP
- *bound* the computed bound

Returns:

status of the bound computation

References `bs_rank`, `bs_running`, `compute_dual_deflation()`, `eta`, `Lp::feasibility`, `find_constraint_basis()`, `Lp::ic`, `iteration`, `Report::print()`, `process_initial_dual_solution()`, `process_perturbed_dual_solution()`, `report`, `Solver_module_interface::restore_dual`, `Solver_module_interface::solve_dual_perturbed`, `solver_module`, `ss_feasible`, and `ss_infeasible`.

Referenced by `lower_bound()`, and `upper_bound()`.

2.9.2.4 void Lurupa::compute_primal_deflation (Lp * *lp*, Primal_deflation & *d*) [private]

Compute primal deflation parameters.

Compute the initial primal deflation parameters for *lp*.

Parameters:

- ← *lp* the LP
- *d* stores the primal deflation paramters

References `Primal_deflation::a`, `alpha`, `eps`, `eta`, `Report::get_write_vm()`, `Lp::IA`, `Lp::ia`, `Lp::infinite`, `Lp::ix`, `Lp::name`, `report`, `Report::write_vector()`, `Lp::xl`, `Primal_deflation::xl`, `Lp::xu`, and `Primal_deflation::xu`.

Referenced by `compute_upper()`.

2.9.2.5 Bound_status Lurupa::compute_upper (Lp * *lp*, double & *bound*) [private]

Compute minimizing upper bound.

Compute the upper bound for the objective function of a minimizing problem. Try iteratively to transform the approximate primal solutions of perturbed problems into intervals verified to contain primal feasible points of the original problem.

Parameters:

- ↔ *lp* the LP
- *bound* the computed bound

Returns:

status of the bound computation

References `bs_rank`, `bs_running`, `compute_primal_deflation()`, `eta`, `Lp::feasibility`, `find_equation_basis()`, `iteration`, `Report::print()`, `process_initial_primal_solution()`, `process_perturbed_primal_solution()`, `report`, `Solver_module_interface::restore_primal`, `Solver_module_interface::solve_primal_perturbed`, `solver_module`, `ss_feasible`, and `ss_unbounded`.

Referenced by `lower_bound()`, and `upper_bound()`.

2.9.2.6 void Lurupa::cond (Lp * *lp*, double & *lower*, double & *upper*)

Compute verified condition number.

Compute an enclosure of the condition number of *lp*.

Parameters:

- ← *lp* the LP
- *lower* lower bound on the condition number
- *upper* upper bound on the condition number

References `Condition::cond()`, `Report::print()`, and `report`.

Referenced by `main()`.

2.9.2.7 void Lurupa::decrease_dual_deflation (Lp * *lp*, VECTOR & *d_c*) [private]

Decrease dual deflation parameters.

Decrease the dual deflation parameters of all constraints. Take the current value into account.

Parameters:

- ← *lp* the LP
- ↔ *d_c* the dual deflation parameters

References `alpha`, `eta`, `Report::get_write_vm()`, `iteration`, `Lp::name`, `Report::print()`, `report`, and `Report::write_vector()`.

Referenced by `process_perturbed_dual_solution()`.

2.9.2.8 void Lurupa::decrease_primal_deflation (Lp * *lp*, Primal_deflation & *d*) [private]

Decrease primal deflation parameters.

Decrease the primal deflation parameters of all constraints. Take the current value into account.

Parameters:

- ← *lp* the LP
- ↔ *d* the primal deflation parameters

References `Primal_deflation::a`, `alpha`, `eta`, `Report::get_write_vm()`, `iteration`, `Lp::name`, `Report::print()`, `report`, `Report::write_vector()`, `Primal_deflation::xl`, and `Primal_deflation::xu`.

Referenced by `process_perturbed_primal_solution()`.

2.9.2.9 void Lurupa::do_pivot (MATRIX & *LU*, int *pivot_index*, int *step*, int * *basis_indices*) [private]

Do the pivot.

Swap the columns of LU with indices *pivot_index* and *step* and the corresponding entries in *basis_indices*.

Parameters:

- ↔ *LU* matrix to perform pivot on
- ← *pivot_index* index of pivot column
- ← *step* index of current column
- ↔ *basis_indices* indices of basis columns

Referenced by find_basis().

2.9.2.10 Bound_status Lurupa::dual_certificate (Lp * *lp*)

Compute certificate for primal infeasibility.

Try to compute a dual certificate verifying primal infeasibility. A dual certificate comes with a dual improving ray, which is a solution to the homogeneous dual LP with a positive objective value (for a minimizing primal LP). Two approaches are tried.

- If the approximate solver offers a mean to extract an approximate dual improving ray, we try to verify a feasible point of the homogeneous dual from it.
- If that fails or the solver does not support it, a phase 1 approach is tried. We compute a lower bound for the primal phase 1 problem. If this is positive primal infeasibility is verified and the dual improving ray is derived from the dual enclosure.

Parameters:

- ← *lp* the LP

Return values:

- bs_priminf* (p. 146) if infeasibility is verified
- bs_failure* (p. 147) otherwise

References *bs_failure*, *bs_priminf*, *bs_verified*, *establish_dual_feasibility*(), *find_constraint_basis*(), *Solver_module_interface::get_dual_ray*, *Report::get_level*(), *Report::get_write_vm*(), *Lp::ia*, *Lp::ib*, *Lp::ic*, *Lp::infinite*, *iteration*, *Lp::ix*, *Lp::iy*, *Lp::iz*, *lower_bound*(), *Lp::maximize*, *Lp::name*, *Report::print*(), *report*, *restore_primal_phase1*(), *set_primal_phase1*(), *Solver_module_interface::set_primal_phase1*, *solve_lp*(), *solver_module*, *ss_infeasible*, *Report::write_vector*(), *Lp::xl*, and *Lp::xu*.

Referenced by *compute_dual*().

2.9.2.11 void Lurupa::eliminate (MATRIX & *LU*, int *step*) [private]

Perform an elimination step.

Perform an elimination step of the LU decomposition of the transpose of LU.

Parameters:

- ↔ *LU* matrix to decompose
- ← *step* elimination step number

Referenced by `find_basis()`.

2.9.2.12 `Bound_status Lurupa::establish_dual_feasibility (Lp * lp, const INTERVAL_MATRIX & IBasis, const MATRIX & Basis_mid_inverse, const INTERVAL_MATRIX & IResidual, const int * basis_indices, INTERVAL_VECTOR & id_pos, INTERVAL_VECTOR & id_neg) [private]`

Establish dual feasibility.

Try to establish a dual feasible solution in two steps.

1. Establish the dual simple bounds on the lagrange parameters of the primal inequality constraints (`establish_iy_simple_bounds` (p. 47)).
2. Compute lagrange parameters for the finite primal simple bounds, establish zero lagrange parameters for the infinite primal simple bounds (`establish_lagrange_parameters` (p. 48)).

Parameters:

- ← *lp* the LP
- ← *IBasis* dual constraint basis matrix
- ← *Basis_mid_inverse* approximate inverse of (midpoint of) *IBasis*
- ← *IResidual* residual of dual constraint matrix
- ← *basis_indices* indices of dual constraint matrix basis columns
- *id_pos* lagrange parameters for finite simple lower bounds
- *id_neg* lagrange parameters for finite simple upper bounds

Returns:

status of the bound computation

References `establish_iy_simple_bounds()`, `establish_lagrange_parameters()`, `Report::get_write_vm()`, `iteration`, `Lp::iy`, `Lp::iz`, `Lp::name`, `Report::print()`, `report`, and `Report::write_vector()`.

Referenced by `dual_certificate()`, `process_initial_dual_solution()`, and `process_perturbed_dual_solution()`.

2.9.2.13 `bool Lurupa::establish_equality_constraints (Lp * lp, const INTERVAL_MATRIX & IBasis, const MATRIX & Basis_mid_inverse, const INTERVAL_MATRIX & IResidual, const int * basis_indices) [private]`

Establish equality constraints.

Establish the equality constraints on the approximate primal solution `lp.ix`. Assume the columns of the constraint matrix to be partitioned into two sets. The columns indexed by `basis_indices`

form the regular (interval) matrix **IBasis**. The matrix **IResidual**, which is a copy of the constraint matrix with the former columns set to zero, contains the remaining columns.

Use the equivalent form of the equality constraints

$$Bx = b \quad \Leftrightarrow \quad IBasis \cdot x_{basis_indices} = b - IResidual \cdot x.$$

Try to compute an enclosure for the solution of the new system, fixing the values of the variables not corresponding to the columns in **IBasis**. If such an enclosure is found, store the enclosure in **lp.ix** and return true. Otherwise return false.

Parameters:

- ← **lp** the LP
- ← **IBasis** equality constraint basis matrix
- ← **Basis_mid_inverse** approximate inverse of (midpoint of) **IBasis**
- ← **IResidual** residual of equality constraint matrix
- ← **basis_indices** indices of basis columns

Return values:

- true** if equality constraints could be established,
- false** otherwise

References **Lp::ib**, and **Lp::ix**.

Referenced by **establish_primal_feasibility()**.

2.9.2.14 void Lurupa::establish_ix_simple_bounds (Lp * lp) [private]

Establish simple bounds on **lp.ix**.

Establish the simple bounds **lp.xl** and **lp.xu** on the approximate primal solution **lp.ix**. Assume the elements of the primal solution are point intervals on input. Set the bound violating elements to the violated bound,

$$\begin{aligned} lp.ix_i < lp.xl_i &\Rightarrow lp.ix_i := lp.xl_i \\ lp.ix_i > lp.xu_i &\Rightarrow lp.ix_i := lp.xu_i \end{aligned}$$

References **Lp::ix**, **Lp::xl**, and **Lp::xu**.

Referenced by **establish_primal_feasibility()**.

2.9.2.15 void Lurupa::establish_iy_simple_bounds (Lp * lp) [private]

Establish simple bounds on **lp.iy**.

Establish the simple upper bound 0 on the approximate dual solution part **lp.iy**. Assume the elements of the dual solution are point intervals on input. Set the positive elements to zero.

References **Lp::iy**.

Referenced by **establish_dual_feasibility()**.

2.9.2.16 `Bound_status Lurupa::establish_lagrange_parameters (Lp * lp, const INTERVAL_MATRIX & IBasis, const MATRIX & Basis_mid_inverse, const INTERVAL_MATRIX & IResidual, const int * basis_indices, INTERVAL_VECTOR & id_pos, INTERVAL_VECTOR & id_neg)`
`[private]`

Establish lagrange parameters.

Compute lagrange parameters for the primal simple bounds. Check for the computed parameters to be zero for infinite bounds.

Parameters:

- ← *lp* the LP
- ← *IBasis* dual constraint basis matrix
- ← *Basis_mid_inverse* approximate inverse of (midpoint of) *IBasis*
- ← *IResidual* residual of dual constraint matrix
- ← *basis_indices* indices of dual constraint matrix basis columns
- *id_pos* lagrange parameters for finite simple lower bounds
- *id_neg* lagrange parameters for finite simple upper bounds

Returns:

status of the bound computation

References `bs_noenc`, `bs_running`, `bs_verified`, `Lp::free_variables_size`, `process_bounded_variables()`, and `process_free_variables()`.

Referenced by `establish_dual_feasibility()`.

2.9.2.17 `Bound_status Lurupa::establish_primal_feasibility (Lp * lp, const INTERVAL_MATRIX & IBasis, const MATRIX & Basis_mid_inverse, const INTERVAL_MATRIX & IResidual, const int * basis_indices)`
`[private]`

Establish primal feasibility.

Try to establish primal feasibility on the approximate primal solution `lp.ix` in three steps.

1. Establish the simple bounds (`establish_ix_simple_bounds` (p. 47)).
2. Verify the equality constraints (`establish_equality_constraints` (p. 46)).
3. Check if the simple bounds still hold (the approximate solution is changed in step 2) and the inequality constraints are satisfied (`primal_feasible` (p. 54)).

Parameters:

- ← *lp* the LP
- ← *IBasis* equality constraint basis matrix
- ← *Basis_mid_inverse* approximate inverse of (midpoint of) *IBasis*
- ← *IResidual* residual of equality constraint matrix
- ← *basis_indices* indices of basis columns

Return values:

bs_verified (p. 146) if `lp.ix` is verified to be primal feasible

bs_noenc (p. 146) if `lp.ix` violates the equality constraints

bs_running (p. 147) if `lp.ix` meets the equality constraints but violates the inequality constraints or simple bounds

References `bs_noenc`, `bs_running`, `bs_verified`, `establish_equality_constraints()`, `establish_ix_simple_bounds()`, `Report::get_write_vm()`, `iteration`, `Lp::ix`, `Lp::name`, `primal_feasible()`, `Report::print()`, `report`, and `Report::write_vector()`.

Referenced by `primal_certificate()`, `process_initial_primal_solution()`, and `process_perturbed_primal_solution()`.

2.9.2.18 `bool Lurupa::find_basis (INTERVAL_MATRIX & A, int *& basis_indices, int size basis_indices, INTERVAL_MATRIX & IBasis, MATRIX & Basis_mid_inverse)` [private]

Find basis for linear system.

Find a (interval) submatrix of **A** whose midpoint matrix is regular. Assume that **A** has more columns than rows. Select linear independent columns from the column candidates indexed by `basis_indices`. Move the selected columns to the matrix **IBasis** and zero them out in **A**. Record the selected columns in the list of candidates.

If a proper (interval) submatrix is found compute its (midpoint's) approximate inverse. Remove the discarded columns from the candidate list and return true. Otherwise return false.

Note:

As **A** is changed during this method this has to be a copy if it is needed after the call.

Parameters:

↔ **A** matrix to be processed

↔ *basis_indices* indices of basis candidates on input,
indices of basis columns on output

← *size_basis_indices* number of basis candidates

→ **IBasis** matrix of basis columns

→ *Basis_mid_inverse* approximate inverse of (midpoint of) **IBasis**

Return values:

true if basis could be found,

false otherwise

References `add_column_to_basis()`, `do_pivot()`, `eliminate()`, `pivot_element()`, `Report::print()`, `report`, and `shorten_basis_indices()`.

Referenced by `find_constraint_basis()`, and `find_equation_basis()`.

2.9.2.19 `bool Lurupa::find_constraint_basis (Lp * lp, INTERVAL_MATRIX & IResidual, int *& basis_indices, INTERVAL_MATRIX & IBasis, MATRIX & Basis_mid_inverse) [private]`

Find constraint basis.

Find a basis of the dual constraints. Select a regular submatrix of the constraint matrix preferring columns corresponding to equality constraints.

Parameters:

- ← *lp* the LP
- ↔ *IResidual* constraint matrix on input, selected columns set to zero on output
- *basis_indices* indices of selected columns
- *IBasis* matrix of selected columns
- *Basis_mid_inverse* approximate inverse of (midpoint of) *IBasis*

Return values:

- true* if basis could be computed,
- false* otherwise

References `find_basis()`, `Lp::free_variables`, `Lp::free_variables_size`, `Lp::IA`, `Lp::IB`, `Report::print()`, and `report`.

Referenced by `compute_lower()`, and `dual_certificate()`.

2.9.2.20 `bool Lurupa::find_equation_basis (Lp * lp, INTERVAL_MATRIX & IResidual, int *& basis_indices, INTERVAL_MATRIX & IBasis, MATRIX & Basis_mid_inverse) [private]`

Find equation basis.

Find a basis of the equality constraints. Select a regular submatrix of the constraint matrix using only columns corresponding to non-fixed variables, that is variables with different lower and upper simple bound.

Parameters:

- ← *lp* the LP
- ↔ *IResidual* constraint matrix on input, selected columns set to zero on output
- *basis_indices* indices of selected columns
- *IBasis* matrix of selected columns
- *Basis_mid_inverse* approximate inverse of (midpoint of) *IBasis*

Return values:

- true* if basis could be computed
- false* otherwise

References `find_basis()`, `Lp::IB`, `Lp::non_fixed_vars`, `Report::print()`, `report`, `Lp::xl`, and `Lp::xu`.

Referenced by `compute_upper()`, and `primal_certificate()`.

2.9.2.21 double Lurupa::get_alpha () const

Get algorithm parameter alpha.

Get the algorithm parameter alpha.

References alpha.

Referenced by clu_get_alpha().

2.9.2.22 const char * Lurupa::get_core_version () const

Get core version string.

Get the core version string.

References version.

Referenced by clu_get_core_version().

2.9.2.23 double Lurupa::get_eta () const

Get algorithm parameter eta.

Get the algorithm parameter eta.

References eta.

Referenced by clu_get_eta().

2.9.2.24 const char * Lurupa::get_module_version () const

Get solver module version.

Get the solver module version string

References Solver_module_interface::get_version, mod_handle, Report::print(), report, and solver_module.

Referenced by clu_get_module_version().

2.9.2.25 double Lurupa::get_solver_eps () const

Get solver's accuracy.

Get solver accuracy epsilon.

References eps, mod_handle, Report::print(), and report.

Referenced by clu_get_solver_eps().

2.9.2.26 void Lurupa::increase_dual_deflation (Lp * lp, VECTOR & d_c, const INTERVAL_VECTOR & id_pos, const INTERVAL_VECTOR & id_neg) [private]

Increase dual deflation parameters.

Increase the dual deflation parameters of the violated constraints. Take the current value into account.

Parameters:

- ← *lp* the LP
- ↔ *d_c* the deflation parameters
- ← *id_pos* lagrange parameter of simple lower bound
- ← *id_neg* lagrange parameter of simple upper bound

References `alpha`, `eta`, `Report::get_write_vm()`, `Lp::infinite`, `iteration`, `Lp::ix`, `Lp::name`, `Report::print()`, `report`, `Report::write_vector()`, `Lp::xl`, and `Lp::xu`.

Referenced by `process_perturbed_dual_solution()`.

2.9.2.27 void Lurupa::increase_primal_deflation (Lp * *lp*, Primal_deflation & *d*) [private]

Increase primal deflation parameters.

Increase the primal deflation parameters of the violated constraints. Take the current value into account.

Parameters:

- ← *lp* the LP
- ↔ *d* the deflation parameters

References `Primal_deflation::a`, `alpha`, `eta`, `Report::get_write_vm()`, `Lp::IA`, `Lp::ia`, `iteration`, `Lp::ix`, `Lp::name`, `Report::print()`, `report`, `Report::write_vector()`, `Primal_deflation::xl`, `Lp::xl`, `Primal_deflation::xu`, and `Lp::xu`.

Referenced by `process_perturbed_primal_solution()`.

2.9.2.28 bool Lurupa::is_inflate () const

Is inflation set.

Get the algorithm flag inflate.

References `inflate`.

Referenced by `clu_is_inflate()`.

2.9.2.29 Bound_status Lurupa::lower_bound (Lp * *lp*, double & *bound*, int & *iterations*)

Compute lower bound.

Compute the lower bound of the linear program `lp` by calling **compute_lower** (p.42) or **compute_upper** (p.43) for the minimizing problem as appropriate. Return a status value indicating that the computation succeeded or why it failed.

Parameters:

- ↔ *lp* the LP
- *bound* computed lower bound
- *iterations* iterations needed to compute bound

Returns:

the value returned by **compute_lower** (p. 42) or **compute_upper** (p. 43)

References `compute_lower()`, `compute_upper()`, `iteration`, and `Lp::maximize`.

Referenced by `clu_lower_bound()`, `compute_lower()`, `dual_certificate()`, `Condition::rho_d()`, and `Condition::rho_p()`.

2.9.2.30 **double Lurupa::pivot_element** (MATRIX *LU*, int *i*, int & *j_pivot*) [private]

Find pivot element.

Find the pivot element, which is the element with largest absolute value in row *i* of *LU* and return its value and column index.

Parameters:

- ← *LU* matrix to find pivot in
- ← *i* row number to search pivot in
- *j_pivot* column index of pivot element

Returns:

the value of the pivot element

Referenced by `find_basis()`.

2.9.2.31 **Bound_status Lurupa::primal_certificate** (Lp * *lp*)

Compute certificate for dual infeasibility.

Try to compute a primal certificate verifying dual infeasibility. A primal certificate comes with a primal improving ray, which is a solution to the homogeneous primal LP with a negative objective value (for a minimizing primal LP). Two approaches are tried.

- If the approximate solver offers a mean to extract an approximate primal improving ray, we try to verify a feasible point of the homogeneous primal from it.
- If that fails or the solver does not support it, a phase 1 approach is tried. We compute an upper bound for the dual phase 1 problem. If this is negative dual infeasibility is verified and the primal improving ray is derived from the primal enclosure.

Parameters:

- ← *lp* the LP

Return values:

- bs_dualinf* (p. 146) if infeasibility is verified
- bs_failure* (p. 147) otherwise

References `bs_dualinf`, `bs_failure`, `bs_verified`, `establish_primal_feasibility()`, `find_equation_basis()`, `Lp::free_variables_size`, `Report::get_level()`, `Solver_module_interface::get_primal_ray`, `Report::get_write_vm()`, `Lp::ia`, `Lp::ib`, `Lp::ic`, `iteration`, `Lp::ix`, `Lp::maximize`, `Lp::name`,

Lp::non_fixed_vars, Report::print(), report, restore_dual_phase1(), set_dual_phase1(), Solver_module_interface::set_dual_phase1, solve_lp(), solver_module, ss_unbounded, upper_bound(), Report::write_vector(), Lp::xl, and Lp::xu.

Referenced by compute_primal().

2.9.2.32 bool Lurupa::primal_feasible (Lp * lp) [private]

Check for primal feasibility.

Test if the approximate primal solution lp.ix satisfies the inequality constraints. Assuming the approximate solution satisfies the simple bounds and contains points satisfying the equality constraints this means lp.ix contains verified primal feasible points.

Return values:

true if approximate solution is primal feasible,
false otherwise

References Report::get_level(), Lp::ia, Lp::IA, Lp::ix, Report::print(), report, Lp::xl, and Lp::xu.

Referenced by establish_primal_feasibility().

2.9.2.33 void Lurupa::print_module_options () const

Print options supported by module.

Print the options supported by solver module.

References mod_handle, Report::print(), Solver_module_interface::print_options, report, and solver_module.

Referenced by clu_print_module_options(), and print_usage().

2.9.2.34 Bound_status Lurupa::process_bounded_variables (Lp * lp, INTERVAL_VECTOR & id_pos, INTERVAL_VECTOR & id_neg) [private]

Process bounded variables.

Compute lagrange parameters for the primal simple bounds of the bounded variables (i.e., variables with at least one finite bound) satisfying the dual constraints. Check for the computed parameters to be zero for infinite simple bounds. Return **bs_verified** (p. 146) if this is the case, **bs_running** (p. 147) otherwise.

Parameters:

← *lp* the LP
 → *id_pos* lagrange parameter for finite simple lower bound
 → *id_neg* lagrange parameter for finite simple upper bound

Return values:

bs_verified (p. 146) if the infinite simple bounds' parameters are zero,
bs_running (p. 147) otherwise

References `bs_running`, `bs_verified`, `Report::get_level()`, `Lp::IA`, `Lp::IB`, `Lp::ic`, `Lp::infinite`, `Lp::iy`, `Lp::iz`, `Report::print()`, `report`, `Lp::xl`, and `Lp::xu`.

Referenced by `establish_lagrange_parameters()`.

2.9.2.35 `Bound_status Lurupa::process_free_variables (Lp * lp, const INTERVAL_MATRIX & IBasis, const MATRIX & Basis_mid_inverse, const INTERVAL_MATRIX & IResidual, const int * basis_indices)` [private]

Process free variables.

Establish zero lagrange parameters for the infinite simple bounds of the free variables. Set the lagrange parameters to zero. Try to find an enclosure of the solution set of the dual constraints corresponding to the free variables. If this fails return `bs_noenc` (p. 146), otherwise return `bs_running` (p. 147).

Parameters:

- ← *lp* the LP
- ← *IBasis* dual constraint basis matrix
- ← *Basis_mid_inverse* approximate inverse of (midpoint of) *IBasis*
- ← *IResidual* residual matrix of dual constraint matrix
- ← *basis_indices* indices of dual constraint matrix basis columns

Return values:

- bs_running* (p. 147) if zero lagrange parameters could be established for free variables,
- bs_noenc* (p. 146) otherwise

References `bs_noenc`, `bs_running`, `Lp::free_variables`, `Lp::free_variables_size`, `Lp::ic`, `Lp::iy`, and `Lp::iz`.

Referenced by `establish_lagrange_parameters()`.

2.9.2.36 `void Lurupa::process_initial_dual_solution (Lp * lp, Bound_status & status, double & bound, const INTERVAL_MATRIX & IBasis, const MATRIX & Basis_mid_inverse, const INTERVAL_MATRIX & IResidual, const int * basis_indices)` [private]

Process initial dual solution.

Process the approximate dual solution of the original problem. Based on the solver's judgement of the problem's feasibility try to verify a feasible solution, start iterating, or set an infinite bound if no verification is possible.

Parameters:

- ← *lp* the LP
- *status* status of bound computation
- *bound* rigorous bound
- ← *IBasis* dual constraint basis matrix
- ← *Basis_mid_inverse* approximate inverse of (midpoint of) *IBasis*

- ← ***IResidual*** residual of dual constraint matrix
- ← ***basis_indices*** indices of dual constraint matrix basis columns

References `bs_failure`, `bs_noenc`, `bs_organb`, `bs_timeout`, `bs_verified`, `establish_dual_feasibility()`, `Lp::feasibility`, `Lp::ia`, `Lp::ib`, `Lp::infinite`, `Lp::ix`, `Lp::iy`, `Lp::iz`, `Report::print()`, `report`, `ss_failure`, `ss_feasible`, `ss_infeasible`, `ss_timeout`, `ss_unbounded`, `Lp::xl`, and `Lp::xu`.

Referenced by `compute_lower()`.

2.9.2.37 `void Lurupa::process_initial_primal_solution (Lp * lp, Bound_status & status, double & bound, const INTERVAL_MATRIX & IBasis, const MATRIX & Basis_mid_inverse, const INTERVAL_MATRIX & IResidual, const int * basis_indices) [private]`

Process initial primal solution.

Process the approximate primal solution to the original problem. Based on the solver's judgement of the problem's feasibility try to verify a feasible solution, start iterating, or set an infinite bound if no verification is possible.

Parameters:

- ← *lp* the LP
- ***status*** status of bound computation
- ***bound*** rigorous bound
- ← ***IBasis*** equality constraint basis matrix
- ← ***Basis_mid_inverse*** approximate inverse of (midpoint of) *IBasis*
- ← ***IResidual*** residual of equality constraint matrix
- ← ***basis_indices*** indices of equality constraint matrix basis columns

References `bs_failure`, `bs_noenc`, `bs_organf`, `bs_timeout`, `bs_verified`, `establish_primal_feasibility()`, `Lp::feasibility`, `Lp::ic`, `Lp::ix`, `Report::print()`, `report`, `ss_failure`, `ss_feasible`, `ss_infeasible`, `ss_timeout`, and `ss_unbounded`.

Referenced by `compute_upper()`.

2.9.2.38 `void Lurupa::process_perturbed_dual_solution (Lp * lp, Bound_status & status, double & bound, const INTERVAL_MATRIX & IBasis, const MATRIX & Basis_mid_inverse, const INTERVAL_MATRIX & IResidual, const int * basis_indices, VECTOR & deflation_c) [private]`

Process perturbed dual solution.

Process the approximate dual solution to a perturbed problem. Based on the solver's judgement of the problem's feasibility try to verify the feasibility and modify the deflation parameters as appropriate. If the iteration count exceeds the maximum **`max_iterations`** (p. 41) set an infinite bound.

Parameters:

- ← *lp* the LP
- ***status*** status of bound computation

- **bound** rigorous bound
- ← **IBasis** equality constraint basis matrix
- ← **Basis_mid_inverse** approximate inverse of (midpoint of) **IBasis**
- ← **IResidual** residual of equality constraint matrix
- ← **basis_indices** indices of equality constraint matrix basis columns
- ↔ **deflation_c** the current deflation parameters on input
the new deflation parameters on output

References `bs_failure`, `bs_iter`, `bs_noenc`, `bs_pertinf`, `bs_timeout`, `bs_verified`, `decrease_dual_deflation()`, `establish_dual_feasibility()`, `Lp::feasibility`, `Lp::ia`, `Lp::ib`, `increase_dual_deflation()`, `Lp::infinite`, `inflate`, `iteration`, `Lp::ix`, `Lp::iy`, `Lp::iz`, `max_iterations`, `Report::print()`, `report`, `ss_failure`, `ss_feasible`, `ss_infeasible`, `ss_timeout`, `ss_unbounded`, `Lp::xl`, and `Lp::xu`.

Referenced by `compute_lower()`.

2.9.2.39 void Lurupa::process_perturbed_primal_solution (Lp * *lp*, Bound_status & *status*, double & *bound*, const INTERVAL_MATRIX & *IBasis*, const MATRIX & *Basis_mid_inverse*, const INTERVAL_MATRIX & *IResidual*, const int * *basis_indices*, Primal_deflation & *deflation*) [private]

Process perturbed primal solution.

Process the approximate primal solution to a perturbed problem. Based on the solver's judgement of the problem's feasibility try to verify a feasible solution and modify the deflation parameters as appropriate. If the iteration count exceeds the maximum **max_iterations** (p. 41) set an infinite bound.

Parameters:

- ← **lp** the LP
- **status** status of bound computation
- **bound** rigorous bound
- ← **IBasis** equality constraint basis matrix
- ← **Basis_mid_inverse** approximate inverse of (midpoint of) **IBasis**
- ← **IResidual** residual of equality constraint matrix
- ← **basis_indices** indices of equality constraint matrix basis columns
- ↔ **deflation** the current deflation parameters on input
the new deflation parameters on output

References `bs_failure`, `bs_iter`, `bs_noenc`, `bs_pertinf`, `bs_timeout`, `bs_verified`, `decrease_primal_deflation()`, `establish_primal_feasibility()`, `Lp::feasibility`, `Lp::ic`, `increase_primal_deflation()`, `inflate`, `iteration`, `Lp::ix`, `max_iterations`, `Report::print()`, `report`, `ss_failure`, `ss_feasible`, `ss_infeasible`, `ss_timeout`, and `ss_unbounded`.

Referenced by `compute_upper()`.

2.9.2.40 `Lp * Lurupa::read_lp (FILE * in, const double & relative_interval_radius)`

Read linear program.

Instruct the solver module to read a linear program.

Parameters:

- ← *in* pointer to linear program
- ← *relative_interval_radius* interval radius to inflate lp parameters to

Return values:

- true* if file read successfully,
- false* otherwise

References eta, Solver_module_interface::read_lp, Lp::set_lurupa(), and solver_module.

Referenced by clu_read_lp(), and main().

2.9.2.41 `void Lurupa::restore_dual_phase1 (Lp * lp, const INTERVAL_VECTOR & iaT, const INTERVAL_VECTOR & ibT, const VECTOR & xlT, const VECTOR & xuT, const int non_fixed_varsT, const int free_variables_sizeT) [private]`

Restore lp after dual phase 1 approach.

Reverse the changes from `set_dual_phase1` (p. 60) and `primal_certificate` (p. 53) applied to lp.

Parameters:

- ↔ *lp* the LP
- ← *iaT* backup of the original right hand sides of inequalities
- ← *ibT* backup of the original right hand sides of equations
- ← *xlT* backup of the original simple lower bounds
- ← *xuT* backup of the original simple upper bounds
- ← *non_fixed_varsT* backup of original `Lp::non_fixed_vars` (p. 29)
- ← *free_variables_sizeT* backup of original `Lp::free_variables_size` (p. 28)

References Lp::feasibility, Lp::free_variables_size, Lp::ia, Lp::ib, Lp::non_fixed_vars, Solver_module_interface::restore_dual_phase1, solver_module, ss_unbounded, Lp::xl, and Lp::xu.

Referenced by primal_certificate().

2.9.2.42 `void Lurupa::restore_primal_phase1 (Lp * lp, const INTERVAL_VECTOR & icT, const bool maximize) [private]`

Restore lp after primal phase 1 approach.

Reverse the changes from `set_primal_phase1` (p. 62) applied to lp.

Parameters:

- ↔ *lp* the LP
- ← *icT* backup of the original objective function
- ← *maximize* is the original lp maximized.

References Lp::feasibility, Lp::IA, Lp::IB, Lp::ic, Lp::ix, Lp::maximize, Lp::non_fixed_vars, Solver_module_interface::restore_primal_phase1, solver_module, ss_infeasible, Lp::xl, and Lp::xu.

Referenced by dual_certificate().

2.9.2.43 void Lurupa::rho_d (Lp * *lp*, double & *lower*, double & *upper*)

Compute distance to dual infeasibility.

Compute an enclosure of the distance to dual infeasibility ρ_d for lp.

Parameters:

- ↔ *lp* the LP
- *lower* lower bound on ρ_d
- *upper* upper bound on ρ_d

References Report::print(), report, and Condition::rho_d().

Referenced by main().

2.9.2.44 void Lurupa::rho_p (Lp * *lp*, double & *lower*, double & *upper*)

Compute distance to primal infeasibility.

Compute an enclosure of the distance to primal infeasibility ρ_p for lp.

Parameters:

- ↔ *lp* the LP
- *lower* lower bound on ρ_p
- *upper* upper bound on ρ_p

References Report::print(), report, and Condition::rho_p().

Referenced by main().

2.9.2.45 void Lurupa::set_alpha (double *alpha*)

Set algorithm parameter alpha.

Set the algorithm parameter alpha.

Referenced by clu_set_alpha(), and main().

2.9.2.46 void Lurupa::set_dual_phase1 (Lp * *lp*) [private]

Set dual phase 1 lp.

Modify *lp* to be a dual phase 1 problem. Add slacks to the dual constraints that may be violated and put just these with negative coefficients into the objective (positive ones when minimizing the dual). For the primal this procedure boils down to

- setting all infinite simple bounds to (for example) 1 and all finite ones to 0
- making all constraints homogeneous.

This allows nonzero lagrange parameters for all simple bounds and removes all terms in the dual objective that do not correspond to originally infinite simple bounds.

Before		After
$\begin{array}{ll} \min & c^T x \\ \text{subject to} & Ax \leq a \\ & Bx = b \\ & \underline{x} \leq x \leq \bar{x} \end{array}$	\longrightarrow	$\begin{array}{ll} \min & 0 \\ \text{subject to} & Ax \leq 0 \\ & Bx = 0 \\ & x \geq \begin{cases} -1 & \text{if } \underline{x} = -\infty \\ 0 & \text{otherwise} \end{cases} \\ & x \leq \begin{cases} 1 & \text{if } \bar{x} = \infty \\ 0 & \text{otherwise} \end{cases} \end{array}$

The objective vector c was at this point already cleared by **primal_certificate** (p. 53).

Parameters:

\leftrightarrow *lp* the LP

References Lp::free_variables_size, Lp::infinite, Lp::ix, Lp::non_fixed_vars, Solver_module_interface::set_dual_phase1, solver_module, Lp::xl, and Lp::xu.

Referenced by primal_certificate().

2.9.2.47 void Lurupa::set_eta (double *eta*)

Set algorithm parameter eta.

Set the algorithm parameter eta.

Referenced by clu_set_eta(), and main().

2.9.2.48 void Lurupa::set_inflate (bool *inflate*)

Set inflation.

Set the algorithm flag inflate.

Referenced by clu_set_inflate(), and main().

2.9.2.49 void Lurupa::set_interface (Solver_module_interface *module*)

Set solver module.

Directly set the already initialized interface to a solver module.

Parameters:

← *module* the module interface

References eps, Solver_module_interface::get_accuracy, Solver_module_interface::init, report, and solver_module.

Referenced by clu_set_interface().

2.9.2.50 bool Lurupa::set_lp (Lp * *lp*, const double *relative_interval_radius*)

Set linear program from solver.

Use the solver module to transform the linear program *lp*.

Parameters:

← *lp* the linear program

← *relative_interval_radius* interval radius to inflate lp parameters to

Return values:

true if the linear program is transformed successfully,

false otherwise

References eta, Solver_module_interface::set_lp, and solver_module.

Referenced by clu_set_lp().

2.9.2.51 bool Lurupa::set_module (char * *module_path*)

Set solver module.

Open the solver module pointed to by *module_path* using the ltdl API. Set up the interface to the solver module by initializing the function pointers in **solver_module** (p. 41) and get solver specific parameters.

Parameters:

← *module_path* path to solver module

Return values:

true if the module could be opened,

false otherwise

References eps, Solver_module_interface::get_accuracy, get_func_pointers(), Solver_module_interface::init, mod_handle, report, and solver_module.

Referenced by clu_set_module(), and main().

2.9.2.52 `bool Lurupa::set_module_options (Lp * lp, int argc, char * argv[])`

Set solver options.

Pass the solver options to the solver module.

Parameters:

- \leftrightarrow *lp* the LP
- \leftarrow *argc* count of solver options
- \leftarrow *argv* array of solver option strings

Return values:

- true* if options set successfully,
- false* otherwise

References Report::print(), report, Solver_module_interface::set_module_options, and solver_module.

Referenced by clu_set_module_options(), and main().

2.9.2.53 `void Lurupa::set_primal_phase1 (Lp * lp, bool * maximize)` [private]

Set primal phase 1 lp.

Modify *lp* to be a primal phase 1 problem. Add bounded slacks for all constraints and minimize the slack sum.

Before		After	
min $c^T x$		min $1^T s + 1^T t^+ + 1^T t^-$	
subject to $Ax \leq a$		subject to $Ax - s \leq a$	
$Bx = b$	\longrightarrow	$Bx + t^+ - t^- = b$	
$\underline{x} \leq x \leq \bar{x}$		$\underline{x} \leq x \leq \bar{x}$	
		$0 \leq s \leq \bar{s}, 0 \leq t^+ \leq \bar{t}^+, 0 \leq t^- \leq \bar{t}^-$	

The upper bounds on the slacks are chosen in a way that $\hat{x} := \min\{\max\{0, \underline{x}\}, \bar{x}\}$, $s = \bar{s}$, $t^+ - t^- = b - B\hat{x}$ is feasible.

Parameters:

- \leftrightarrow *lp* the LP
- \rightarrow *maximize* is *lp* maximized

References Lp::ia, Lp::IA, Lp::ib, Lp::IB, Lp::ic, Lp::infinite, Lp::ix, Lp::maximize, Lp::non_fixed_vars, Solver_module_interface::set_primal_phase1, solver_module, Lp::xl, and Lp::xu.

Referenced by dual_certificate().

2.9.2.54 `void Lurupa::shorten_basis_indices (int *& basis_indices, int basis_size)` [private]

Shorten basis indices array.

Shorten the *basis_indices* array to the first *basis_size* elements.

Parameters:

- ↔ *basis_indices* array to shorten
- ← *basis_size* size to achieve

Referenced by `find_basis()`.

2.9.2.55 Solver_status Lurupa::solve_lp (Lp * *lp*, double & *optimal_value*)

Solve linear program.

Instruct the solver module to approximately solve lp.

Parameters:

- ← *lp* the LP
- *optimal_value* the approximate optimal value computed by the solver

Returns:

the status of the solver

References `Lp::feasibility`, `Solver_module_interface::solve_original`, and `solver_module`.

Referenced by `clu_solve_lp()`, `dual_certificate()`, `main()`, `primal_certificate()`, `Condition::rho_d()`, and `Condition::rho_p()`.

2.9.2.56 Bound_status Lurupa::upper_bound (Lp * *lp*, double & *bound*, int & *iterations*)

Compute upper bound.

Compute the upper bound of a linear program by calling **compute_lower** (p. 42) or **compute_upper** (p. 43) for the minimizing problem as appropriate. Return a status value indicating that the computation succeeded or why it failed.

Parameters:

- ↔ *lp* the LP
- *bound* computed upper bound
- *iterations* iterations needed to compute bound

Returns:

the value returned by **compute_lower** (p. 42) or **compute_upper** (p. 43)

References `compute_lower()`, `compute_upper()`, `iteration`, and `Lp::maximize`.

Referenced by `clu_upper_bound()`, `compute_upper()`, `primal_certificate()`, `Condition::rho_d()`, and `Condition::rho_p()`.

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

- **Lurupa.h**
- **Lurupa.cpp**

2.10 Primal_deflation Struct Reference

Primal deflation parameters.

```
#include <lurupa/Lp.h>
```

Public Attributes

- VECTOR **a**
Deflation parameter for the right hand side of the inequalities.
- VECTOR **xl**
Deflation parameter for the simple lower bounds.
- VECTOR **xu**
Deflation parameter for the simple upper bounds.

2.10.1 Detailed Description

Primal deflation parameters.

This structure stores the primal deflation parameters for the lp.

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

- **Lp.h**

2.11 Report Class Reference

Functions for reporting and debugging.

```
#include <lurupa/Report.h>
```

Public Member Functions

- **Csv_style** **get_csv_style** () const
Get csv style.
- short **get_level** () const
Get verbosity level.
- bool **get_print_time** () const
Get time prepending.
- bool **get_write_vm** () const
Get vector and matrix writing.
- void **print** (short type, const char *format,...) const
Print message.
- **Report** ()
Constructor.
- void **set_csv_style** (Csv_style style)
Set csv style.
- void **set_error** (FILE *error)
Set destination of error messages.
- void **set_level** (short level)
Set verbosity level.
- void **set_normal** (FILE *normal)
Set destination of normal messages.
- void **set_print_time** (bool print_time)
Set time prepending.
- void **set_verbosity** (short level, bool print_time, bool write_vm)
Set verbosity.
- void **set_warning** (FILE *warning)
Set destination of warning messages.
- void **set_write_vm** (bool write_vm)
Set vector and matrix writing.

- void **write_matrix** (const char *file, const MATRIX &X, const char *name) const
Write matrix to disk.
- void **write_vector** (const char *file, const VECTOR &x, const char *name) const
Write vector to disk.

Private Attributes

- **Csv_style csv_style**
Use which csv style.
- **FILE * error**
Destination of error messages.
- **short level**
Maximum level of messages to be printed.
- **FILE * normal**
Destination of normal messages.
- **bool print_time**
Prepend time to messages.
- **FILE * warning**
Destination of warning messages.
- **bool write_vm**
Write vectors and matrices to disk.

2.11.1 Detailed Description

Functions for reporting and debugging.

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

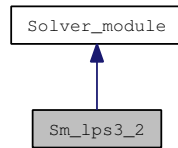
- **Report.h**

2.12 Sm_lps3_2 Class Reference

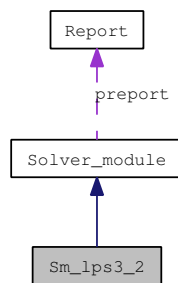
Solver module for lp_solve 3.2.

```
#include "Sm_lps3_2.h"
```

Inheritance diagram for Sm_lps3_2:



Collaboration diagram for Sm_lps3_2:



Static Public Member Functions

- static double **get_accuracy** ()
Get accuracy used by lp_solve.
- static const char * **get_version** ()
Get module version string.
- static bool **init** (**Report** *report)
Initialize the module.
- static void **print_brief_version** ()
Print brief module version.
- static void **print_options** ()
Print supported options.
- static void **print_version** ()
Print module version.
- static bool **read_lp** (**Lp** *lp, FILE *in, const double relative_interval_radius, double &eta)
Read lp.

- static void **restore__dual** (**Lp** *lp)
Restore lp after dual perturbation.
- static void **restore__primal** (**Lp** *lp)
Restore lp after primal perturbation.
- static bool **set__lp** (**Lp** *lp, const double relative__interval__radius, double &eta)
Set lp.
- static bool **solve__dual__perturbed** (const VECTOR &deflation__c, **Lp** *lp, const int iteration)
Compute solution of dual perturbed lp.
- static bool **solve__original** (**Lp** *lp, double &optimal__value)
Compute solution of unperturbed lp.
- static bool **solve__primal__perturbed** (const **Primal__deflation** &deflation, **Lp** *lp, const int iteration)
Compute solution of primal perturbed lp.

Static Private Member Functions

- static void **dual__perturb** (const VECTOR &d__c, const **Lp** *lp)
Dual perturb lp.
- static void **primal__perturb** (const **Primal__deflation** &d, const **Lp** *lp)
Primal perturb lp.
- static void **read__duals** (**Lp** *lp)
Read dual solution.
- static void **read__primals** (**Lp** *lp, const int rows)
Read primal solution.
- static bool **solve__lp** (**Lp** *lp)
Solve lp with lp_solve.

Lp transforming routines

The following routines read the lp from the data structures of lp_solve 3.2 and transform it into Lurupa's representation.

- static void **adjust__eta** (double &eta, const **Lp** *lp)
Adjust eta to the lp.
- static bool **build__constraint__maps** (int &c__eq, int &c__le, const int rows, const int cols, **Lp** *lp)
Build mapping between lp_solve's and Lurupa's constraint structures.
- static void **find__free__variables** (int &vars, **Lp** *lp)

Find free variables.

- static void **inflate_lp** (**Lp** *lp, const double r)
Inflate lp parameters.
- static void **read_general_data** (**Lp** *lp)
Read general data from lp_solve's structures.
- static void **read_lp_mat** (const int rows, **Lp** *lp)
Read lp_solve's lp_mat structure.
- static void **read_right_hand_sides** (**Lp** *lp)
Read right hand sides of the constraints.
- static void **read_simple_bounds** (**Lp** *lp, const int rows)
Read simple bounds.
- static void **resize_lp** (**Lp** *lp, const int cols, const int c_eq, const int c_le)
Resize Lurupa's data structures according to the lp dimensions.
- static bool **transform_lp** (**Lp** *lp, const double relative_interval_radius, double &eta)
Transform lp from lp_solve to Lurupa.

2.12.1 Detailed Description

Solver module for lp_solve 3.2.

2.12.2 Member Function Documentation

2.12.2.1 void Sm_lps3_2::adjust_eta (double & *eta*, const **Lp** * *lp*) [static, private]

Adjust eta to the lp.

Adjust the algorithm parameter **eta** to be problem dependent. It becomes the maximum of its previous value and 10^{-20} times the maximum norm of the right hand sides of the inequality constraints,

$$\eta = \max\{\eta, 10^{-20}\|a\|_{\infty}\}.$$

Parameters:

- **eta** parameter to be adjusted
- ← **lp** lp **eta** is adjusted to

References **Lp::ia**, **Solver_module::preport**, and **Report::print()**.

Referenced by **transform_lp()**.

2.12.2.2 bool Sm_lps3_2::build_constraint_maps (int & *c_eq*, int & *c_le*, const int *rows*, const int *cols*, **Lp** * *lp*) [static, private]

Build mapping between lp_solve's and Lurupa's constraint structures.

Build a mapping from Lurupa's equality and inequality constraints to `lp_solve`'s constraints, which is storing all constraints in one big matrix. Store the mapping in `lp->module->mp_eq_con` and `lp->module->mp_le_con`. Check if the number of constraints is less than the number of variables.

Parameters:

- `c_eq` number of equations in `lp`
- `c_le` number of inequalities in `lp`
- ← `rows` number of rows of constraint matrix
- ← `cols` number of columns of constraint matrix
- ← `lp` the `lp`

Return values:

- `true` if number of constraints is less than number of variables,
- `false` otherwise

References `Lp_lp_solve::lp`, `Lp::module`, `Lp_lp_solve::mp_eq_con`, `Lp_lp_solve::mp_le_con`, `Solver_module::preport`, and `Report::print()`.

Referenced by `transform_lp()`.

2.12.2.3 void Sm_lps3_2::dual_perturb (const VECTOR & d_c, const Lp * lp) [static, private]

Dual perturb `lp`.

Perturb the `lp` in the cost vector.

Parameters:

- ← `d_c` deflation parameters
- `lp` contains the base for the deflation

References `Lp::ic`, `Lp_lp_solve::lp`, `Lp::maximize`, `Lp::module`, and `Lp_lp_solve::mp_parts`.

Referenced by `solve_dual_perturbed()`.

2.12.2.4 void Sm_lps3_2::find_free_variables (int & vars, Lp * lp) [static, private]

Find free variables.

Find free variables in the `lp`. Store the indices of the free variables in `lp->free_variables` and their number in `lp->free_variables_size`.

Adjust the number of variables `vars` to reflect the actual count of variables and not include the split variables used in `lp_solve` to represent free variables.

Parameters:

- ↔ `vars` number of variables in `lp`
- `lp` stores number and indices of the free variables

References `Lp::free_variables`, `Lp::free_variables_size`, `Lp_lp_solve::lp`, `Lp::module`, and `Lp_lp_solve::mp_parts`.

Referenced by `transform_lp()`.

2.12.2.5 double Sm_lps3_2::get_accuracy () [static]

Get accuracy used by lp_solve.

Get lp_solve's set accuracy

Returns:

lp_solve's accuracy

Reimplemented from **Solver_module** (p. 123).

Referenced by get_func_pointers().

2.12.2.6 const char * Sm_lps3_2::get_version () [static]

Get module version string.

Get the version information of the solver module.

Reimplemented from **Solver_module** (p. 122).

References version.

Referenced by get_func_pointers().

2.12.2.7 void Sm_lps3_2::inflate_lp (Lp * lp, const double r) [static, private]

Inflate lp parameters.

Inflate all lp parameters to intervals with relative radius r ,

$$x \rightarrow \mathbf{x} \triangleq x \cdot [1 - r, 1 + r]$$

Parameters:

\rightarrow **lp** lp to be inflated

\leftarrow **r** relative interval radius

References Lp::ia, Lp::IA, Lp::ib, Lp::IB, and Lp::ic.

Referenced by transform_lp().

2.12.2.8 bool Sm_lps3_2::init (Report * report) [static]

Initialize the module.

Set **preport** (p. 124) to *report.

Parameters:

\leftarrow **report** the **Report** (p. 65) instance to use

Return values:

true

Reimplemented from **Solver_module** (p.122).

References Solver_module::preport, Report::print(), and version.

Referenced by get_func_pointers().

2.12.2.9 void Sm_lps3_2::primal_perturb (const Primal_deflation & d, const Lp * lp) [static, private]

Primal perturb lp.

Perturb the lp in the right hand sides of the inequalities and the simple bounds.

Parameters:

← **d** deflation parameters

→ **lp** contains the base for the deflation

References Primal_deflation::a, Lp::ia, Lp_lp_solve::lp, Lp::module, Lp_lp_solve::mp_le_con, Lp_lp_solve::mp_parts, Lp::xl, Primal_deflation::xl, Lp::xu, and Primal_deflation::xu.

Referenced by solve_primal_perturbed().

2.12.2.10 void Sm_lps3_2::print_options () [static]

Print supported options.

Print the command line options supported by the solver module.

Reimplemented from **Solver_module** (p.122).

References version.

Referenced by get_func_pointers().

2.12.2.11 void Sm_lps3_2::read_duals (Lp * lp) [static, private]

Read dual solution.

Read the dual solution from lp_solve and store it in lp->iy and lp->iz. Lp_solve's convention to determine the sign of the duals is, that the costs must be a linear combination of the gradients of the constraints.

The duals returned by lp_solve, however, are the duals for the original lp, that is the duals for greater equal constraints are multiplied by -1 , and all duals are again multiplied by -1 if the lp is a maximizing one. As Lurupa stores all linear programs as minimizing and all inequality constraints as less equal undo these multiplications.

Parameters:

→ **lp** receives the dual solution

References Lp::feasibility, Lp::iy, Lp::iz, Lp_lp_solve::lp, Lp::maximize, Lp::module, Lp_lp_solve::mp_eq_con, Lp_lp_solve::mp_le_con, and ss_infeasible.

Referenced by solve_dual_perturbed(), and solve_original().

2.12.2.12 void Sm_lps3_2::read_general_data (Lp * *lp*) [static, private]

Read general data from lp_solve's structures.

Read general lp properties from lp_solve's structures and store them in *lp*. This includes the value representing infinity, the direction of optimization, and the name of the lp.

Parameters:

→ *lp* stores the read data

References Lp::infinite, Lp::maximize, Lp::module, and Lp::name.

Referenced by transform_lp().

2.12.2.13 bool Sm_lps3_2::read_lp (Lp * *lp*, FILE * *in*, const double *relative_interval_radius*, double & *eta*) [static]

Read lp.

Read an lp from a file into lp_solve and *lp*. Inflate the lp to an interval valued one if *relative_interval_radius* is greater than 0, and adjust *eta* according to the lp's parameters.

Parameters:

→ *lp* receives the lp

← *in* file pointer to lp

← *relative_interval_radius* interval radius to inflate lp parameters to

→ *eta* algorithm parameter to be adjusted

Return values:

true if reading and transforming the lp succeeds,

false otherwise, transform_lp returns false

See also:

transform_lp (p. 77)

Reimplemented from **Solver_module** (p. 123).

References Lp_lp_solve::lp, Lp::module, and transform_lp().

Referenced by get_func_pointers().

2.12.2.14 void Sm_lps3_2::read_lp_mat (const int *rows*, Lp * *lp*) [static, private]

Read lp_solve's lp_mat structure.

Read lp_solve's lp_mat structure. This stores the left hand sides of the constraints together with the objective function. Split the read coefficients into equality and inequality constraints, transform inequality constraints to all less equal, and the direction of optimization to minimize.

Parameters:

← *rows* number of rows in constraint matrix

→ *lp* stores the constraints and objective function

References Lp::IA, Lp::IB, Lp::ic, Lp_lp_solve::lp, Lp::maximize, Lp::module, Lp_lp_solve::mp_eq_con, Lp_lp_solve::mp_le_con, and Lp_lp_solve::mp_parts.

Referenced by transform_lp().

2.12.2.15 void Sm_lps3_2::read_primals (Lp * *lp*, const int *rows*) [static, private]

Read primal solution.

Read the primal solution from lp_solve and store it in *lp*->*ix*. If the lp seems to be unbounded construct the last primal feasible point and read this. Join the solution parts of split variables.

Parameters:

→ *lp* receives the primal solution

← *rows* number of rows in constraint matrix

References Lp::feasibility, Lp::ic, Lp::ix, Lp_lp_solve::lp, Lp::module, Lp_lp_solve::mp_parts, Solver_module::preport, Report::print(), and ss_unbounded.

Referenced by solve_original(), and solve_primal_perturbed().

2.12.2.16 void Sm_lps3_2::read_right_hand_sides (Lp * *lp*) [static, private]

Read right hand sides of the constraints.

Read the right hand sides of the constraints. Split the read right hand sides into equality and inequality constraints, transform inequality constraints to all less equal. Store the values in *lp*->*ia* and *lp*->*ib*.

Parameters:

→ *lp* stores the right hand sides

References Lp::ia, Lp::ib, Lp_lp_solve::lp, Lp::module, Lp_lp_solve::mp_eq_con, Lp_lp_solve::mp_le_con, Solver_module::preport, and Report::print().

Referenced by transform_lp().

2.12.2.17 void Sm_lps3_2::read_simple_bounds (Lp * *lp*, const int *rows*) [static, private]

Read simple bounds.

Read the simple bounds of the variables and store them in *lp*->*xl* and *lp*->*xu*.

Parameters:

→ *lp* stores the simple bounds

← *rows* number of rows in lp_solve's constraint matrix

References Lp::infinite, Lp_lp_solve::lp, Lp::module, Lp_lp_solve::mp_parts, Lp::non_fixed_vars, Lp::xl, and Lp::xu.

Referenced by transform_lp().

2.12.2.18 `void Sm_lps3_2::resize_lp (Lp * lp, const int cols, const int c_eq, const int c_le)` [static, private]

Resize Lurupa's data structures according to the lp dimensions.

Resize the vectors and matrices in Lurupa's data structure to the sizes of the lp.

Parameters:

- *lp* Lurupa's structure with the vectors and matrices to be resized
- ← *cols* number of variables in the lp
- ← *c_eq* number of equality constraints in the lp
- ← *c_le* number of inequality constraints in the lp

References Lp::ia, Lp::IA, Lp::ib, Lp::IB, Lp::ic, Lp::ix, Lp::iy, Lp::iz, Lp::xl, and Lp::xu.

Referenced by transform_lp().

2.12.2.19 `void Sm_lps3_2::restore_dual (Lp * lp)` [static]

Restore lp after dual perturbation.

Take the model parameters for the costs from lp and write to lp->module->lp restoring the original model after computation of the lower bound.

Parameters:

- ← *lp* contains the original parameters

Reimplemented from **Solver_module** (p.123).

References Lp::ic, Lp_lp_solve::lp, Lp::maximize, Lp::module, Lp_lp_solve::mp_parts, Solver_module::preport, and Report::print().

Referenced by get_func_pointers().

2.12.2.20 `void Sm_lps3_2::restore_primal (Lp * lp)` [static]

Restore lp after primal perturbation.

Take the model parameters for the right hand sides and the simple bounds from lp and write to lp->module->lp restoring the original model after computation of the upper bound.

Parameters:

- ← *lp* contains the original parameters

Reimplemented from **Solver_module** (p.123).

References Lp::ia, Lp::IA, Lp_lp_solve::lp, Lp::module, Lp_lp_solve::mp_le_con, Lp_lp_solve::mp_parts, Solver_module::preport, Report::print(), Lp::xl, and Lp::xu.

Referenced by get_func_pointers().

2.12.2.21 `bool Sm_lps3_2::solve_dual_perturbed (const VECTOR & d_c, Lp * lp, const int iteration) [static]`

Compute solution of dual perturbed lp.

Perturb the lp and solve the new one. Read the dual solution from `lp_solve` and store it in `lp->iy` and `lp->iz`.

Parameters:

- ← *d_c* deflation parameter
- *lp* receives the solver status and the dual solution
- ← *iteration* number of the current iteration

Return values:

- true* if `solve_lp` returns true and the solution is read
- false* otherwise, `solve_lp` returns false

See also:

`solve_lp` (p. 76)

Reimplemented from `Solver_module` (p. 123).

References `dual_perturb()`, `Lp::feasibility`, `Solver_module::preport`, `Report::print()`, `read_duals()`, `solve_lp()`, `ss_feasible`, and `ss_infeasible`.

Referenced by `get_func_pointers()`.

2.12.2.22 `bool Sm_lps3_2::solve_lp (Lp * lp) [static, private]`

Solve lp with `lp_solve`.

Solve an lp with `lp_solve`. Store `lp_solve`'s status in `lp->feasibility`.

Parameters:

- *lp* receives `lp_solve`'s status

Return values:

- true* if `lp_solve` returns a known status code (i.e., `OPTIMAL`, `INFEASIBLE`, `UNBOUNDED`, or `FAILURE`)
- false* otherwise

References `Lp::feasibility`, `Lp::module`, `Solver_module::preport`, `Report::print()`, `ss_failure`, `ss_feasible`, `ss_infeasible`, and `ss_unbounded`.

Referenced by `solve_dual_perturbed()`, `solve_original()`, and `solve_primal_perturbed()`.

2.12.2.23 `bool Sm_lps3_2::solve_original (Lp * lp, double & optimal_value) [static]`

Compute solution of unperturbed lp.

Solve the original unperturbed lp with `lp_solve` and read the primal and dual solution.

Parameters:

- *lp* receives the solver status and the solution
- *optimal_value* receives the optimal value

Return values:

- true* if `solve_lp` returns true and the solution is read
- false* otherwise, `solve_lp` returns false

See also:

`read_duals` (p. 72), `read_primals` (p. 74), `solve_lp` (p. 76)

Reimplemented from `Solver_module` (p. 123).

References `Lp::module`, `Solver_module::preport`, `Report::print()`, `read_duals()`, `read_primals()`, and `solve_lp()`.

Referenced by `get_func_pointers()`.

2.12.2.24 `bool Sm_lps3_2::solve_primal_perturbed(const Primal_deflation & d, Lp * lp, const int iteration)` [static]

Compute solution of primal perturbed lp.

Perturb the lp and solve the new one. Read the primal solution and store it in `lp->ix`.

Parameters:

- ← *d* deflation parameters
- *lp* receives the solver status and the primal solution
- ← *iteration* number of the current iteration

Return values:

- true* if `solve_lp` returns true and the solution is read
- false* otherwise, `solve_lp` returns false

See also:

`solve_lp` (p. 76)

Reimplemented from `Solver_module` (p. 123).

References `Lp::feasibility`, `Lp::module`, `Solver_module::preport`, `primal_perturb()`, `Report::print()`, `read_primals()`, `solve_lp()`, `ss_feasible`, and `ss_unbounded`.

Referenced by `get_func_pointers()`.

2.12.2.25 `bool Sm_lps3_2::transform_lp(Lp * lp, const double relative_interval_radius, double & eta)` [static, private]

Transform lp from `lp_solve` to Lurupa.

Transform lp from the data structure of `lp_solve` 3.2 into the representation used in **Lurupa** (p. 36). Adjust algorithm parameters *eta* according to the lp, and inflate the lp to an interval valued one if `relative_interval_radius` is greater than 0.

Write the (midpoint) problem with `preport` (p. 124) if `Report::write_vm` (p. 66) is set.

Parameters:

- *lp* receives the transformed data
- ← *relative_interval_radius* relative interval radius to inflate lp parameters to
- *eta* gets adjusted to the model

Return values:

- true* if the transformation succeeds,
- false* otherwise, `lp->module->lp` is not set or `build_constraint_maps` returns false

See also:

`adjust_eta` (p. 69), `build_constraint_maps` (p. 69), `find_free_variables` (p. 70), `inflate_lp` (p. 71), `read_general_data` (p. 73), `read_lp_mat` (p. 73), `read_right_hand_sides` (p. 74), `read_simple_bounds` (p. 74), `resize_lp` (p. 75)

References `adjust_eta()`, `build_constraint_maps()`, `find_free_variables()`, `Lp::free_variables_size`, `Report::get_write_vm()`, `Lp::ia`, `Lp::IA`, `Lp::ib`, `Lp::IB`, `Lp::ic`, `inflate_lp()`, `Lp::module`, `Lp::name`, `Solver_module::preport`, `Report::print()`, `read_general_data()`, `read_lp_mat()`, `read_right_hand_sides()`, `read_simple_bounds()`, `resize_lp()`, `Report::write_matrix()`, `Report::write_vector()`, `Lp::xl`, and `Lp::xu`.

Referenced by `read_lp()`, and `set_lp()`.

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

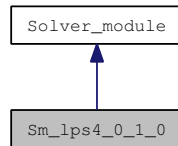
- **Sm_lps3_2.h**
- **Sm_lps3_2.cpp**

2.13 Sm_lps4_0_1_0 Class Reference

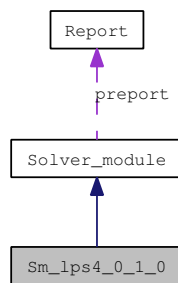
Solver module for lp_solve 4.0.1.0.

```
#include "Sm_lps4_0_1_0.h"
```

Inheritance diagram for Sm_lps4_0_1_0:



Collaboration diagram for Sm_lps4_0_1_0:



Static Public Member Functions

- static double **get_accuracy** ()
Get accuracy used by lp_solve.
- static const char * **get_version** ()
Get module version string.
- static bool **init** (**Report** *report)
Initialize the module.
- static void **print_brief_version** ()
Print brief module version.
- static void **print_options** ()
Print supported options.
- static void **print_version** ()
Print module version.
- static bool **read_lp** (**Lp** *lp, FILE *in, const double relative_interval_radius, double &eta)
Read lp.

- static void **restore__dual** (**Lp** *lp)
Restore lp after dual perturbation.
- static void **restore__primal** (**Lp** *lp)
Restore lp after primal perturbation.
- static bool **set__lp** (**Lp** *lp, const double relative__interval__radius, double &eta)
Set lp.
- static bool **set__module__options** (**Lp** *lp, int argc, char *argv[])
Set module options.
- static bool **solve__dual__perturbed** (const VECTOR &deflation__c, **Lp** *lp, const int iteration)
Compute solution of dual perturbed lp.
- static bool **solve__original** (**Lp** *lp, double &optimal__value)
Compute solution of unperturbed lp.
- static bool **solve__primal__perturbed** (const **Primal__deflation** &deflation, **Lp** *lp, const int iteration)
Compute solution of primal perturbed lp.

Static Private Member Functions

- static void **dual__perturb** (const VECTOR &d__c, const **Lp** *lp)
Dual perturb lp.
- static void WINAPI **lps__log** (void *lp, void *userhandle, char *buf)
Callback function for lp_solve's log messages.
- static void **postprocess** (**Lp** *lp, const int status)
Postprocess lp.
- static void **primal__perturb** (const **Primal__deflation** &d, const **Lp** *lp)
Primal perturb lp.
- static bool **read__duals** (**Lp** *lp)
Read dual solution.
- static bool **read__primals** (**Lp** *lp, const int cols)
Read primal solution.
- static bool **solve__lp** (**Lp** *lp)
Solve lp with lp_solve.

Lp transforming routines

The following routines read the lp from the data structures of lp_solve 4.0.1.0 and transform it into Lurupa's representation.

- static void **adjust_eta** (double &eta, const **Lp** *lp)
Adjust eta to the lp.
- static bool **build_constraint_maps** (int &c_eq, int &c_le, const int rows, const int cols, **Lp** *lp)
Build mapping between lp_solve's and Lurupa's constraint structures.
- static void **find_free_variables** (const int vars, **Lp** *lp)
Find free variables.
- static void **inflate_lp** (**Lp** *lp, const double r)
Inflate lp parameters.
- static void **read_general_data** (**Lp** *lp)
Read general data from lp_solve's structures.
- static void **read_lp_mat** (const int rows, **Lp** *lp)
Read lp_solve's lp_mat structure.
- static void **read_right_hand_sides** (**Lp** *lp)
Read right hand sides of the constraints.
- static void **read_simple_bounds** (**Lp** *lp)
Read simple bounds.
- static void **resize_lp** (**Lp** *lp, const int cols, const int c_eq, const int c_le)
Resize Lurupa's data structures according to the lp dimensions.
- static bool **transform_lp** (**Lp** *lp, const double relative_interval_radius, double &eta)
Transform lp from lp_solve to Lurupa.

2.13.1 Detailed Description

Solver module for lp_solve 4.0.1.0.

2.13.2 Member Function Documentation

2.13.2.1 void Sm_lps4_0_1_0::adjust_eta (double & eta, const Lp * lp) [static, private]

Adjust eta to the lp.

Adjust the algorithm parameter **eta** to be problem dependent. It becomes the maximum of its previous value and 10^{-20} times the maximum norm of the right hand sides of the inequality constraints,

$$\eta = \max\{\eta, 10^{-20}\|a\|_{\infty}\}.$$

Parameters:

- **eta** parameter to be adjusted
- ← **lp** lp eta is adjusted to

References Lp::ia, Solver_module::preport, and Report::print().

Referenced by transform_lp().

2.13.2.2 `bool Sm_lps4_0_1_0::build_constraint_maps (int & c_eq, int & c_le, const int rows, const int cols, Lp * lp) [static, private]`

Build mapping between lp_solve's and Lurupa's constraint structures.

Build a mapping from Lurupa's equality and inequality constraints to lp_solve's constraints, which is storing all constraints in one big matrix. Store the mapping in lp->module->mp_eq_con and lp->module->mp_le_con. Check if the number of constraints is less than the number of variables.

Parameters:

- *c_eq* number of equations in lp
- *c_le* number of inequalities in lp
- ← *rows* number of rows of constraint matrix
- ← *cols* number of columns of constraint matrix
- ← *lp* the lp

Return values:

- true* if number of constraints is less than number of variables,
- false* otherwise

References Lp_lp_solve::lp, Lp::module, Lp_lp_solve::mp_eq_con, Lp_lp_solve::mp_le_con, Solver_module::preport, and Report::print().

Referenced by transform_lp().

2.13.2.3 `void Sm_lps4_0_1_0::dual_perturb (const VECTOR & d_c, const Lp * lp) [static, private]`

Dual perturb lp.

Perturb the lp in the cost vector.

Parameters:

- ← *d_c* deflation parameters
- *lp* contains the base for the deflation

References Lp::ic, Lp_lp_solve::lp, Lp::maximize, and Lp::module.

Referenced by solve_dual_perturbed().

2.13.2.4 `void Sm_lps4_0_1_0::find_free_variables (const int vars, Lp * lp) [static, private]`

Find free variables.

Find free variables in the lp. Store the indices of the free variables in lp->free_variables and their number in lp->free_variables_size.

Parameters:

- ← *vars* number of variables in lp
- *lp* stores number and indices of the free variables

References Lp::free_variables, Lp::free_variables_size, Lp::infinite, and Lp::module.

Referenced by transform_lp().

2.13.2.5 double Sm_lps4_0_1_0::get_accuracy () [static]

Get accuracy used by lp_solve.

Get lp_solve's set accuracy

Returns:

lp_solve's accuracy

Reimplemented from **Solver_module** (p. 123).

Referenced by get_func_pointers().

2.13.2.6 const char * Sm_lps4_0_1_0::get_version () [static]

Get module version string.

Get the version information of the solver module.

Reimplemented from **Solver_module** (p. 122).

References version.

Referenced by get_func_pointers().

2.13.2.7 void Sm_lps4_0_1_0::inflate_lp (Lp * lp, const double r) [static, private]

Inflate lp parameters.

Inflate all lp parameters to intervals with relative radius *r*,

$$x \rightarrow \mathbf{x} \triangleq x \cdot [1 - r, 1 + r]$$

Parameters:

- *lp* lp to be inflated
- ← *r* relative interval radius

References Lp::ia, Lp::IA, Lp::ib, Lp::IB, and Lp::ic.

Referenced by transform_lp().

2.13.2.8 bool Sm_lps4_0_1_0::init (Report * report) [static]

Initialize the module.

Set **preport** (p. 124) to *report. Check the version of lp_solve.

Parameters:

← *report* the **Report** (p. 65) instance to use

Return values:

true

Reimplemented from **Solver_module** (p. 122).

References Solver_module::preport, Report::print(), and version.

Referenced by get_func_pointers().

2.13.2.9 void WINAPI Sm_lps4_0_1_0::lps_log (void * *lp*, void * *userhandle*, char * *buf*) [static, private]

Callback function for lp_solve's log messages.

Callback function for lp_solve's log functionality. Print the supplied log message marked as coming from lp_solve.

References Solver_module::preport, and Report::print().

Referenced by read_lp().

2.13.2.10 void Sm_lps4_0_1_0::postprocess (Lp * *lp*, const int *status*) [static, private]

Postprocess lp.

Construct 'last feasible points' if *status* indicates that lp_solve judges lp->module->lp to be infeasible or unbounded. Postprocess the lp.

When lp_solve determines an lp to be infeasible or unbounded it does not store any information gathered in the simplex algorithm but discards it during the postprocessing. Deriving 'last feasible points' from this information may deliver rigorous bounds to support lp_solve's claim. Calling lp_solve's preprocessing routine explicitly, which does not alter the lp in a harmful way regarding the verification, causes lp_solve not to postprocess the lp after solving. This way the information in the simplex tableau is preserved.

Parameters:

← *lp* the lp

← *status* return value of lp_solve solving the lp

References Lp::module, Solver_module::preport, and Report::print().

Referenced by solve_lp().

2.13.2.11 void Sm_lps4_0_1_0::primal_perturb (const Primal_deflation & *d*, const Lp * *lp*) [static, private]

Primal perturb lp.

Perturb the lp in the right hand sides of the inequalities and the simple bounds.

Parameters:

- ← *d* deflation parameters
- *lp* contains the base for the deflation

References Primal_deflation::a, Lp::ia, Lp_lp_solve::lp, Lp::module, Lp_lp_solve::mp_le_con, Lp::xl, Primal_deflation::xl, Lp::xu, and Primal_deflation::xu.

Referenced by solve_primal_perturbed().

2.13.2.12 void Sm_lps4_0_1_0::print_options () [static]

Print supported options.

Print the command line options supported by the solver module.

Reimplemented from **Solver_module** (p.122).

References version.

Referenced by get_func_pointers().

2.13.2.13 bool Sm_lps4_0_1_0::read_duals (Lp * lp) [static, private]

Read dual solution.

Read the dual solution from lp_solve and store it in *lp*->*iy* and *lp*->*iz*. Lp_solve's convention to determine the sign of the duals is, that the costs must be a linear combination of the gradients of the constraints.

The duals returned by lp_solve, however, are the duals for the original lp, that is the duals for greater equal constraints are multiplied by -1 , and all duals are again multiplied by -1 if the lp is a maximizing one. As Lurupa stores all linear programs as minimizing and all inequality constraints as less equal undo these multiplications.

Parameters:

- *lp* receives the dual solution

Return values:

- true* if reading the dual solution succeeded,
- false* otherwise, calling lp_solve's `get_sensitivity_rhs` returns false

References Lp::iy, Lp::iz, Lp_lp_solve::lp, Lp::maximize, Lp::module, Lp_lp_solve::mp_eq_con, Lp_lp_solve::mp_le_con, Solver_module::preport, and Report::print().

Referenced by solve_dual_perturbed(), and solve_original().

2.13.2.14 void Sm_lps4_0_1_0::read_general_data (Lp * lp) [static, private]

Read general data from lp_solve's structures.

Read general lp properties from lp_solve's structures and store them in *lp*. This includes the value representing infinity, the direction of optimization, and the name of the lp.

Parameters:

- *lp* stores the read data

References `Lp::infinite`, `Lp::maximize`, `Lp::module`, and `Lp::name`.

Referenced by `transform_lp()`.

2.13.2.15 `bool Sm_lps4_0_1_0::read_lp (Lp * lp, FILE * in, const double relative_interval_radius, double & eta) [static]`

Read `lp`.

Read an `lp` from a file into `lp_solve` and `lp`. Inflate the `lp` to an interval valued one if `relative_interval_radius` is greater than 0, and adjust `eta` according to the `lp`'s parameters.

Parameters:

- *lp* receives the `lp`
- ← *in* file pointer to `lp`
- ← *relative_interval_radius* interval radius to inflate `lp` parameters to
- *eta* algorithm parameter to be adjusted

Return values:

- true* if reading and transforming the `lp` succeeds,
- false* otherwise, reading the `lp` fails or `transform_lp` returns false

See also:

`transform_lp` (p. 91)

Reimplemented from `Solver_module` (p. 123).

References `Lp_lp_solve::lp`, `lps_log()`, `Lp::module`, and `transform_lp()`.

Referenced by `get_func_pointers()`.

2.13.2.16 `void Sm_lps4_0_1_0::read_lp_mat (const int rows, Lp * lp) [static, private]`

Read `lp_solve`'s `lp_mat` structure.

Read `lp_solve`'s `lp_mat` structure. This stores the left hand sides of the constraints together with the objective function. Split the read coefficients into equality and inequality constraints, transform inequality constraints to all less equal, and the direction of optimization to minimize.

Parameters:

- ← *rows* number of rows in constraint matrix
- *lp* stores the constraints and objective function

References `Lp::IA`, `Lp::IB`, `Lp::ic`, `Lp_lp_solve::lp`, `Lp::maximize`, `Lp::module`, `Lp_lp_solve::mp_eq_con`, and `Lp_lp_solve::mp_le_con`.

Referenced by `transform_lp()`.

2.13.2.17 `bool Sm_lps4_0_1_0::read_primals (Lp * lp, const int cols)` [static, private]

Read primal solution.

Read the primal solution from `lp_solve` and store it in `lp->ix`.

Parameters:

- *lp* receives the primal solution
- ← *cols* number of variables in the lp

Return values:

- true* if reading the primal solution succeeds,
- false* otherwise, calling `lp_solve`'s `get_variables` returns false

References `Lp::ic`, `Lp::ix`, `Lp_lp_solve::lp`, `Lp::module`, `Solver_module::preport`, and `Report::print()`.

Referenced by `solve_original()`, and `solve_primal_perturbed()`.

2.13.2.18 `void Sm_lps4_0_1_0::read_right_hand_sides (Lp * lp)` [static, private]

Read right hand sides of the constraints.

Read the right hand sides of the constraints. Split the read right hand sides into equality and inequality constraints, transform inequality constraints to all less equal. Store the values in `lp->ia` and `lp->ib`.

Parameters:

- *lp* stores the right hand sides

References `Lp::ia`, `Lp::ib`, `Lp_lp_solve::lp`, `Lp::module`, `Lp_lp_solve::mp_eq_con`, and `Lp_lp_solve::mp_le_con`.

Referenced by `transform_lp()`.

2.13.2.19 `void Sm_lps4_0_1_0::read_simple_bounds (Lp * lp)` [static, private]

Read simple bounds.

Read the simple bounds of the variables and store them in `lp->xl` and `lp->xu`.

Parameters:

- *lp* stores the simple bounds

References `Lp::module`, `Lp::non_fixed_vars`, `Lp::xl`, and `Lp::xu`.

Referenced by `transform_lp()`.

2.13.2.20 `void Sm_lps4_0_1_0::resize_lp (Lp * lp, const int cols, const int c_eq, const int c_le)` [static, private]

Resize Lurupa's data structures according to the lp dimensions.

Resize the vectors and matrices in Lurupa's data structure to the sizes of the lp.

Parameters:

- *lp* Lurupa's structure with the vectors and matrices to be resized
- ← *cols* number of variables in the lp
- ← *c_eq* number of equality constraints in the lp
- ← *c_le* number of inequality constraints in the lp

References Lp::ia, Lp::IA, Lp::ib, Lp::IB, Lp::ic, Lp::ix, Lp::iy, Lp::iz, Lp::xl, and Lp::xu.

Referenced by transform_lp().

2.13.2.21 `void Sm_lps4_0_1_0::restore_dual (Lp * lp)` [static]

Restore lp after dual perturbation.

Take the model parameters for the costs from lp and write to lp->module->lp restoring the original model after computation of the lower bound.

Parameters:

- ← *lp* contains the original parameters

Reimplemented from **Solver_module** (p.123).

References Lp::ic, Lp::maximize, Lp::module, Solver_module::preport, and Report::print().

Referenced by get_func_pointers().

2.13.2.22 `void Sm_lps4_0_1_0::restore_primal (Lp * lp)` [static]

Restore lp after primal perturbation.

Take the model parameters for the right hand sides and the simple bounds from lp and write to lp->module->lp restoring the original model after computation of the upper bound.

Parameters:

- ← *lp* contains the original parameters

Reimplemented from **Solver_module** (p.123).

References Lp::ia, Lp::IA, Lp_lp_solve::lp, Lp::module, Lp_lp_solve::mp_le_con, Solver_module::preport, Report::print(), Lp::xl, and Lp::xu.

Referenced by get_func_pointers().

2.13.2.23 `bool Sm_lps4_0_1_0::set_module_options (Lp * lp, int argc, char * argv[]) [static]`

Set module options.

Set the specified module options. These are supplied in `argc` and `argv` in a command line version. Each option is supplied as one string entry in the string array `argv`, `argc` specifies the number of options.

The module supports the following options

- `'-sm,timeout,n'`: Set `lp_solve`'s timeout to `n` seconds.
- `'-sm,vn'`: Set `lp_solve`'s verbosity level to `n`. The available levels are
 - `v0` NEUTRAL
 - `v1` CRITICAL
 - `v2` SEVERE
 - `v3` IMPORTANT (default)
 - `v4` NORMAL
 - `v5` DETAILED
 - `v6` FULL

Parameters:

- `lp` `lp` to set the options for
- ← `argc` number of arguments
- ← `argv` array of arguments

Return values:

- `true` if options are set
- `false` otherwise

References `Lp::module`, `Solver_module::preport`, and `Report::print()`.

Referenced by `get_func_pointers()`.

2.13.2.24 `bool Sm_lps4_0_1_0::solve_dual_perturbed (const VECTOR & d_c, Lp * lp, const int iteration) [static]`

Compute solution of dual perturbed `lp`.

Perturb the `lp` and solve the new one. Read the dual solution from `lp_solve` and store it in `lp->iy` and `lp->iz`.

Parameters:

- ← `d_c` deflation parameter
- `lp` receives the solver status and the dual solution
- ← `iteration` number of the current iteration

Return values:

- `true` if `solve_lp` returns true and the solution is read

false otherwise, `solve_lp` or `read_duals` returns false

See also:

`read_duals` (p. 85), `solve_lp` (p. 90)

Reimplemented from `Solver_module` (p. 123).

References `dual_perturb()`, `Lp::feasibility`, `Solver_module::preport`, `Report::print()`, `read_duals()`, `solve_lp()`, `ss_feasible`, and `ss_infeasible`.

Referenced by `get_func_pointers()`.

2.13.2.25 `bool Sm_lps4_0_1_0::solve_lp (Lp * lp) [static, private]`

Solve lp with `lp_solve`.

Solve an lp with `lp_solve`. Store `lp_solve`'s status in `lp->feasibility`.

Parameters:

→ *lp* receives `lp_solve`'s status

Return values:

true if `lp_solve` returns a known status code (i.e., `OPTIMAL`, `INFEASIBLE`, `UNBOUNDED`, `TIMEOUT`, or `FAILURE`)

false otherwise

References `Lp::feasibility`, `Lp::module`, `postprocess()`, `Solver_module::preport`, `Report::print()`, `ss_failure`, `ss_feasible`, `ss_infeasible`, `ss_timeout`, and `ss_unbounded`.

Referenced by `solve_dual_perturbed()`, `solve_original()`, and `solve_primal_perturbed()`.

2.13.2.26 `bool Sm_lps4_0_1_0::solve_original (Lp * lp, double & optimal_value) [static]`

Compute solution of unperturbed lp.

Solve the original unperturbed lp with `lp_solve` and read the primal and dual solution.

Parameters:

→ *lp* receives the solver status and the solution

→ *optimal_value* receives the optimal value

Return values:

true if `solve_lp` returns true and the solution is read

false otherwise, `solve_lp`, `read_primals`, or `read_duals` returns false

See also:

`read_duals` (p. 85), `read_primals` (p. 87), `solve_lp` (p. 90)

Reimplemented from **Solver_module** (p.123).

References `Lp::module`, `Solver_module::preport`, `Report::print()`, `read_duals()`, `read_primals()`, and `solve_lp()`.

Referenced by `get_func_pointers()`.

2.13.2.27 `bool Sm_lps4_0_1_0::solve_primal_perturbed (const Primal_deflation & d, Lp * lp, const int iteration)` [static]

Compute solution of primal perturbed lp.

Perturb the lp and solve the new one. Read the primal solution and store it in `lp->ix`.

Parameters:

- ← *d* deflation parameters
- *lp* receives the solver status and the primal solution
- ← *iteration* number of the current iteration

Return values:

- true* if `solve_lp` returns true and the solution is read
- false* otherwise, `solve_lp` or `read_primals` returns false

See also:

`read_primals` (p.87), `solve_lp` (p.90)

Reimplemented from **Solver_module** (p.123).

References `Lp::feasibility`, `Lp::module`, `Solver_module::preport`, `primal_perturb()`, `Report::print()`, `read_primals()`, `solve_lp()`, `ss_feasible`, and `ss_unbounded`.

Referenced by `get_func_pointers()`.

2.13.2.28 `bool Sm_lps4_0_1_0::transform_lp (Lp * lp, const double relative_interval_radius, double & eta)` [static, private]

Transform lp from `lp_solve` to **Lurupa**.

Transform lp from the data structure of `lp_solve` 4.0.1.0 into the representation used in **Lurupa** (p.36). Adjust algorithm parameters *eta* according to the lp, and inflate the lp to an interval valued one if `relative_interval_radius` is greater than 0.

Write the (midpoint) problem with **preport** (p.124) if **Report::write_vm** (p.66) is set.

Parameters:

- *lp* receives the transformed data
- ← *relative_interval_radius* relative interval radius to inflate lp parameters to
- *eta* gets adjusted to the model

Return values:

- true* if the transformation succeeds,

false otherwise, `lp->module->lp` is not set or `build_constraint_maps` returns false

See also:

`adjust_eta` (p. 81), `build_constraint_maps` (p. 82), `find_free_variables` (p. 82), `inflate_lp` (p. 83), `read_general_data` (p. 85), `read_lp_mat` (p. 86), `read_right_hand_sides` (p. 87), `read_simple_bounds` (p. 87), `resize_lp` (p. 88)

References `adjust_eta()`, `build_constraint_maps()`, `find_free_variables()`, `Lp::free_variables_size`, `Report::get_write_vm()`, `Lp::ia`, `Lp::IA`, `Lp::ib`, `Lp::IB`, `Lp::ic`, `inflate_lp()`, `Lp::module`, `Lp::name`, `Solver_module::preport`, `Report::print()`, `read_general_data()`, `read_lp_mat()`, `read_right_hand_sides()`, `read_simple_bounds()`, `resize_lp()`, `Report::write_matrix()`, `Report::write_vector()`, `Lp::xl`, and `Lp::xu`.

Referenced by `read_lp()`, and `set_lp()`.

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

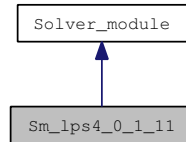
- **Sm_lps4_0_1_0.h**
- **Sm_lps4_0_1_0.cpp**

2.14 Sm_lps4_0_1_11 Class Reference

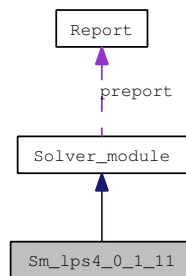
Solver module for lp_solve 4.0.1.11.

```
#include "Sm_lps4_0_1_11.h"
```

Inheritance diagram for Sm_lps4_0_1_11:



Collaboration diagram for Sm_lps4_0_1_11:



Static Public Member Functions

- static double **get__accuracy** ()
Get accuracy used by lp_solve.
- static const char * **get__version** ()
Get module version string.
- static bool **init** (**Report** *report)
Initialize the module.
- static void **print__brief__version** ()
Print brief module version.
- static void **print__options** ()
Print supported options.
- static void **print__version** ()
Print module version.
- static bool **read__lp** (**Lp** *lp, FILE *in, const double relative_interval_radius, double &eta)
Read lp.

- static void **restore__dual** (**Lp** *lp)
Restore lp after dual perturbation.
- static void **restore__primal** (**Lp** *lp)
Restore lp after primal perturbation.
- static bool **set__lp** (**Lp** *lp, const double relative__interval__radius, double &eta)
Set lp.
- static bool **set__module__options** (**Lp** *lp, int argc, char *argv[])
Set module options.
- static bool **solve__dual__perturbed** (const VECTOR &deflation__c, **Lp** *lp, const int iteration)
Compute solution of dual perturbed lp.
- static bool **solve__original** (**Lp** *lp, double &optimal__value)
Compute solution of unperturbed lp.
- static bool **solve__primal__perturbed** (const **Primal__deflation** &deflation, **Lp** *lp, const int iteration)
Compute solution of primal perturbed lp.

Static Private Member Functions

- static void **dual__perturb** (const VECTOR &d__c, const **Lp** *lp)
Dual perturb lp.
- static void WINAPI **lps__log** (void *lp, void *userhandle, char *buf)
Callback function for lp_solve's log messages.
- static void **postprocess** (**Lp** *lp, const int status)
Postprocess lp.
- static void **primal__perturb** (const **Primal__deflation** &d, const **Lp** *lp)
Primal perturb lp.
- static bool **read__duals** (**Lp** *lp)
Read dual solution.
- static bool **read__primals** (**Lp** *lp, const int cols)
Read primal solution.
- static bool **solve__lp** (**Lp** *lp)
Solve lp with lp_solve.

Lp transforming routines

The following routines read the lp from the data structures of lp_solve 4.0.1.11 and transform it into Lurupa's representation.

- static void **adjust_eta** (double &eta, const **Lp** *lp)
Adjust eta to the lp.
- static bool **build_constraint_maps** (int &c_eq, int &c_le, const int rows, const int cols, **Lp** *lp)
Build mapping between lp_solve's and Lurupa's constraint structures.
- static void **find_free_variables** (const int vars, **Lp** *lp)
Find free variables.
- static void **inflate_lp** (**Lp** *lp, const double r)
Inflate lp parameters.
- static void **read_general_data** (**Lp** *lp)
Read general data from lp_solve's structures.
- static void **read_lp_mat** (const int rows, **Lp** *lp)
Read lp_solve's lp_mat structure.
- static void **read_right_hand_sides** (**Lp** *lp)
Read right hand sides of the constraints.
- static void **read_simple_bounds** (**Lp** *lp)
Read simple bounds.
- static void **resize_lp** (**Lp** *lp, const int cols, const int c_eq, const int c_le)
Resize Lurupa's data structures according to the lp dimensions.
- static bool **transform_lp** (**Lp** *lp, const double relative_interval_radius, double &eta)
Transform lp from lp_solve to Lurupa.

2.14.1 Detailed Description

Solver module for lp_solve 4.0.1.11.

2.14.2 Member Function Documentation

2.14.2.1 void Sm_lps4_0_1_11::adjust_eta (double & eta, const Lp * lp) [static, private]

Adjust eta to the lp.

Adjust the algorithm parameter **eta** to be problem dependent. It becomes the maximum of its previous value and 10^{-20} times the maximum norm of the right hand sides of the inequality constraints,

$$\eta = \max\{\eta, 10^{-20}\|a\|_{\infty}\}.$$

Parameters:

→ **eta** parameter to be adjusted

← **lp** lp **eta** is adjusted to

References Lp::ia, Solver_module::preport, and Report::print().

Referenced by transform_lp().

2.14.2.2 `bool Sm_lps4_0_1_11::build_constraint_maps (int & c_eq, int & c_le, const int rows, const int cols, Lp * lp) [static, private]`

Build mapping between lp_solve's and Lurupa's constraint structures.

Build a mapping from Lurupa's equality and inequality constraints to lp_solve's constraints, which is storing all constraints in one big matrix. Store the mapping in lp->module->mp_eq_con and lp->module->mp_le_con. Check if the number of constraints is less than the number of variables.

Parameters:

- *c_eq* number of equations in lp
- *c_le* number of inequalities in lp
- ← *rows* number of rows of constraint matrix
- ← *cols* number of columns of constraint matrix
- ← *lp* the lp

Return values:

- true* if number of constraints is less than number of variables,
- false* otherwise

References Lp_lp_solve::lp, Lp::module, Lp_lp_solve::mp_eq_con, Lp_lp_solve::mp_le_con, Solver_module::preport, and Report::print().

Referenced by transform_lp().

2.14.2.3 `void Sm_lps4_0_1_11::dual_perturb (const VECTOR & d_c, const Lp * lp) [static, private]`

Dual perturb lp.

Perturb the lp in the cost vector.

Parameters:

- ← *d_c* deflation parameters
- *lp* contains the base for the deflation

References Lp::ic, Lp_lp_solve::lp, Lp::maximize, and Lp::module.

Referenced by solve_dual_perturbed().

2.14.2.4 `void Sm_lps4_0_1_11::find_free_variables (const int vars, Lp * lp) [static, private]`

Find free variables.

Find free variables in the lp. Store the indices of the free variables in lp->free_variables and their number in lp->free_variables_size.

Parameters:

- ← *vars* number of variables in lp
- *lp* stores number and indices of the free variables

References Lp::free_variables, Lp::free_variables_size, Lp::infinite, and Lp::module.

Referenced by transform_lp().

2.14.2.5 double Sm_lps4_0_1_11::get_accuracy () [static]

Get accuracy used by lp_solve.

Get lp_solve's set accuracy

Returns:

lp_solve's accuracy

Reimplemented from **Solver_module** (p. 123).

Referenced by get_func_pointers().

2.14.2.6 const char * Sm_lps4_0_1_11::get_version () [static]

Get module version string.

Get the version information of the solver module.

Reimplemented from **Solver_module** (p. 122).

References version.

Referenced by get_func_pointers().

2.14.2.7 void Sm_lps4_0_1_11::inflate_lp (Lp * lp, const double r) [static, private]

Inflate lp parameters.

Inflate all lp parameters to intervals with relative radius *r*,

$$x \rightarrow \mathbf{x} \triangleq x \cdot [1 - r, 1 + r]$$

Parameters:

- *lp* lp to be inflated
- ← *r* relative interval radius

References Lp::ia, Lp::IA, Lp::ib, Lp::IB, and Lp::ic.

Referenced by transform_lp().

2.14.2.8 bool Sm_lps4_0_1_11::init (Report * report) [static]

Initialize the module.

Set **preport** (p. 124) to *report. Check the version of lp_solve.

Parameters:

← *report* the **Report** (p. 65) instance to use

Return values:

true

Reimplemented from **Solver_module** (p. 122).

References Solver_module::preport, Report::print(), and version.

Referenced by get_func_pointers().

2.14.2.9 void WINAPI Sm_lps4_0_1_11::lps_log (void * *lp*, void * *userhandle*, char * *buf*) [static, private]

Callback function for lp_solve's log messages.

Callback function for lp_solve's log functionality. Print the supplied log message marked as coming from lp_solve.

References Solver_module::preport, and Report::print().

Referenced by read_lp().

2.14.2.10 void Sm_lps4_0_1_11::postprocess (Lp * *lp*, const int *status*) [static, private]

Postprocess lp.

Construct 'last feasible points' if *status* indicates that lp_solve judges lp->module->lp to be infeasible or unbounded. Postprocess the lp.

When lp_solve determines an lp to be infeasible or unbounded it does not store any information gathered in the simplex algorithm but discards it during the postprocessing. Deriving 'last feasible points' from this information may deliver rigorous bounds to support lp_solve's claim. Calling lp_solve's preprocessing routine explicitly, which does not alter the lp in a harmful way regarding the verification, causes lp_solve not to postprocess the lp after solving. This way the information in the simplex tableau is preserved.

Parameters:

← *lp* the lp

← *status* return value of lp_solve solving the lp

References Lp::module, Solver_module::preport, and Report::print().

Referenced by solve_lp().

2.14.2.11 void Sm_lps4_0_1_11::primal_perturb (const Primal_deflation & *d*, const Lp * *lp*) [static, private]

Primal perturb lp.

Perturb the lp in the right hand sides of the inequalities and the simple bounds.

Parameters:

- ← *d* deflation parameters
- *lp* contains the base for the deflation

References Primal_deflation::a, Lp::ia, Lp_lp_solve::lp, Lp::module, Lp_lp_solve::mp_le_con, Lp::xl, Primal_deflation::xl, Lp::xu, and Primal_deflation::xu.

Referenced by solve_primal_perturbed().

2.14.2.12 void Sm_lps4_0_1_11::print_options () [static]

Print supported options.

Print the command line options supported by the solver module.

Reimplemented from **Solver_module** (p.122).

References version.

Referenced by get_func_pointers().

2.14.2.13 bool Sm_lps4_0_1_11::read_duals (Lp * lp) [static, private]

Read dual solution.

Read the dual solution from lp_solve and store it in *lp->iy* and *lp->iz*. Lp_solve's convention to determine the sign of the duals is, that the costs must be a linear combination of the gradients of the constraints.

The duals returned by lp_solve, however, are the duals for the original lp, that is the duals for greater equal constraints are multiplied by -1 , and all duals are again multiplied by -1 if the lp is a maximizing one. As Lurupa stores all linear programs as minimizing and all inequality constraints as less equal undo these multiplications.

Parameters:

- *lp* receives the dual solution

Return values:

- true* if reading the dual solution succeeded,
- false* otherwise, calling lp_solve's `get_sensitivity_rhs` returns false

References Lp::iy, Lp::iz, Lp_lp_solve::lp, Lp::maximize, Lp::module, Lp_lp_solve::mp_eq_con, Lp_lp_solve::mp_le_con, Solver_module::preport, and Report::print().

Referenced by solve_dual_perturbed(), and solve_original().

2.14.2.14 void Sm_lps4_0_1_11::read_general_data (Lp * lp) [static, private]

Read general data from lp_solve's structures.

Read general lp properties from lp_solve's structures and store them in *lp*. This includes the value representing infinity, the direction of optimization, and the name of the lp.

Parameters:

- *lp* stores the read data

References `Lp::infinite`, `Lp::maximize`, `Lp::module`, and `Lp::name`.

Referenced by `transform_lp()`.

2.14.2.15 `bool Sm_lps4_0_1_11::read_lp (Lp * lp, FILE * in, const double relative_interval_radius, double & eta) [static]`

Read `lp`.

Read an `lp` from a file into `lp_solve` and `lp`. Inflate the `lp` to an interval valued one if `relative_interval_radius` is greater than 0, and adjust `eta` according to the `lp`'s parameters.

Parameters:

- *lp* receives the `lp`
- ← *in* file pointer to `lp`
- ← *relative_interval_radius* interval radius to inflate `lp` parameters to
- *eta* algorithm parameter to be adjusted

Return values:

- true* if reading and transforming the `lp` succeeds,
- false* otherwise, reading the `lp` fails or `transform_lp` returns false

See also:

`transform_lp` (p. 105)

Reimplemented from `Solver_module` (p. 123).

References `Lp_lp_solve::lp`, `lps_log()`, `Lp::module`, and `transform_lp()`.

Referenced by `get_func_pointers()`.

2.14.2.16 `void Sm_lps4_0_1_11::read_lp_mat (const int rows, Lp * lp) [static, private]`

Read `lp_solve`'s `lp_mat` structure.

Read `lp_solve`'s `lp_mat` structure. This stores the left hand sides of the constraints together with the objective function. Split the read coefficients into equality and inequality constraints, transform inequality constraints to all less equal, and the direction of optimization to minimize.

Parameters:

- ← *rows* number of rows in constraint matrix
- *lp* stores the constraints and objective function

References `Lp::IA`, `Lp::IB`, `Lp::ic`, `Lp_lp_solve::lp`, `Lp::maximize`, `Lp::module`, `Lp_lp_solve::mp_eq_con`, and `Lp_lp_solve::mp_le_con`.

Referenced by `transform_lp()`.

2.14.2.17 `bool Sm_lps4_0_1_11::read_primals (Lp * lp, const int cols)` [static, private]

Read primal solution.

Read the primal solution from `lp_solve` and store it in `lp->ix`.

Parameters:

- *lp* receives the primal solution
- ← *cols* number of variables in the lp

Return values:

- true* if reading the primal solution succeeds,
- false* otherwise, calling `lp_solve`'s `get_variables` returns false

References `Lp::ic`, `Lp::ix`, `Lp_lp_solve::lp`, `Lp::module`, `Solver_module::preport`, and `Report::print()`.

Referenced by `solve_original()`, and `solve_primal_perturbed()`.

2.14.2.18 `void Sm_lps4_0_1_11::read_right_hand_sides (Lp * lp)` [static, private]

Read right hand sides of the constraints.

Read the right hand sides of the constraints. Split the read right hand sides into equality and inequality constraints, transform inequality constraints to all less equal. Store the values in `lp->ia` and `lp->ib`.

Parameters:

- *lp* stores the right hand sides

References `Lp::ia`, `Lp::ib`, `Lp_lp_solve::lp`, `Lp::module`, `Lp_lp_solve::mp_eq_con`, and `Lp_lp_solve::mp_le_con`.

Referenced by `transform_lp()`.

2.14.2.19 `void Sm_lps4_0_1_11::read_simple_bounds (Lp * lp)` [static, private]

Read simple bounds.

Read the simple bounds of the variables and store them in `lp->xl` and `lp->xu`.

Parameters:

- *lp* stores the simple bounds

References `Lp::module`, `Lp::non_fixed_vars`, `Lp::xl`, and `Lp::xu`.

Referenced by `transform_lp()`.

2.14.2.20 `void Sm_lps4_0_1_11::resize_lp (Lp * lp, const int cols, const int c_eq, const int c_le)` [static, private]

Resize Lurupa's data structures according to the lp dimensions.

Resize the vectors and matrices in Lurupa's data structure to the sizes of the lp.

Parameters:

- *lp* Lurupa's structure with the vectors and matrices to be resized
- ← *cols* number of variables in the lp
- ← *c_eq* number of equality constraints in the lp
- ← *c_le* number of inequality constraints in the lp

References Lp::ia, Lp::IA, Lp::ib, Lp::IB, Lp::ic, Lp::ix, Lp::iy, Lp::iz, Lp::xl, and Lp::xu.

Referenced by transform_lp().

2.14.2.21 `void Sm_lps4_0_1_11::restore_dual (Lp * lp)` [static]

Restore lp after dual perturbation.

Take the model parameters for the costs from lp and write to lp->module->lp restoring the original model after computation of the lower bound.

Parameters:

- ← *lp* contains the original parameters

Reimplemented from **Solver_module** (p.123).

References Lp::ic, Lp::maximize, Lp::module, Solver_module::preport, and Report::print().

Referenced by get_func_pointers().

2.14.2.22 `void Sm_lps4_0_1_11::restore_primal (Lp * lp)` [static]

Restore lp after primal perturbation.

Take the model parameters for the right hand sides and the simple bounds from lp and write to lp->module->lp restoring the original model after computation of the upper bound.

Parameters:

- ← *lp* contains the original parameters

Reimplemented from **Solver_module** (p.123).

References Lp::ia, Lp::IA, Lp_lp_solve::lp, Lp::module, Lp_lp_solve::mp_le_con, Solver_module::preport, Report::print(), Lp::xl, and Lp::xu.

Referenced by get_func_pointers().

2.14.2.23 `bool Sm_lps4_0_1_11::set_module_options (Lp * lp, int argc, char * argv[]) [static]`

Set module options.

Set the specified module options. These are supplied in `argc` and `argv` in a command line version. Each option is supplied as one string entry in the string array `argv`, `argc` specifies the number of options.

The module supports the following options

- `'-sm,timeout,n'`: Set `lp_solve`'s timeout to `n` seconds.
- `'-sm,vn'`: Set `lp_solve`'s verbosity level to `n`. The available levels are
 - `v0` NEUTRAL
 - `v1` CRITICAL
 - `v2` SEVERE
 - `v3` IMPORTANT (default)
 - `v4` NORMAL
 - `v5` DETAILED
 - `v6` FULL

Parameters:

- `lp` `lp` to set the options for
- ← `argc` number of arguments
- ← `argv` array of arguments

Return values:

- `true` if options are set
- `false` otherwise

References `Lp::module`, `Solver_module::preport`, and `Report::print()`.

Referenced by `get_func_pointers()`.

2.14.2.24 `bool Sm_lps4_0_1_11::solve_dual_perturbed (const VECTOR & d_c, Lp * lp, const int iteration) [static]`

Compute solution of dual perturbed `lp`.

Perturb the `lp` and solve the new one. Read the dual solution from `lp_solve` and store it in `lp->iy` and `lp->iz`.

Parameters:

- ← `d_c` deflation parameter
- `lp` receives the solver status and the dual solution
- ← `iteration` number of the current iteration

Return values:

- `true` if `solve_lp` returns true and the solution is read

false otherwise, `solve_lp` or `read_duals` returns false

See also:

`read_duals` (p. 99), `solve_lp` (p. 104)

Reimplemented from `Solver_module` (p. 123).

References `dual_perturb()`, `Lp::feasibility`, `Solver_module::preport`, `Report::print()`, `read_duals()`, `solve_lp()`, `ss_feasible`, and `ss_infeasible`.

Referenced by `get_func_pointers()`.

2.14.2.25 `bool Sm_lps4_0_1_11::solve_lp (Lp * lp) [static, private]`

Solve lp with `lp_solve`.

Solve an lp with `lp_solve`. Store `lp_solve`'s status in `lp->feasibility`.

Parameters:

→ *lp* receives `lp_solve`'s status

Return values:

true if `lp_solve` returns a known status code (i.e., `OPTIMAL`, `INFEASIBLE`, `UNBOUNDED`, `TIMEOUT`, or `FAILURE`)

false otherwise

References `Lp::feasibility`, `Lp::module`, `postprocess()`, `Solver_module::preport`, `Report::print()`, `ss_failure`, `ss_feasible`, `ss_infeasible`, `ss_timeout`, and `ss_unbounded`.

Referenced by `solve_dual_perturbed()`, `solve_original()`, and `solve_primal_perturbed()`.

2.14.2.26 `bool Sm_lps4_0_1_11::solve_original (Lp * lp, double & optimal_value) [static]`

Compute solution of unperturbed lp.

Solve the original unperturbed lp with `lp_solve` and read the primal and dual solution.

Parameters:

→ *lp* receives the solver status and the solution

→ *optimal_value* receives the optimal value

Return values:

true if `solve_lp` returns true and the solution is read

false otherwise, `solve_lp`, `read_primals`, or `read_duals` returns false

See also:

`read_duals` (p. 99), `read_primals` (p. 101), `solve_lp` (p. 104)

Reimplemented from **Solver_module** (p.123).

References `Lp::module`, `Solver_module::preport`, `Report::print()`, `read_duals()`, `read_primals()`, and `solve_lp()`.

Referenced by `get_func_pointers()`.

2.14.2.27 `bool Sm_lps4_0_1_11::solve_primal_perturbed (const Primal_deflation & d, Lp * lp, const int iteration)` [static]

Compute solution of primal perturbed lp.

Perturb the lp and solve the new one. Read the primal solution and store it in `lp->ix`.

Parameters:

- ← *d* deflation parameters
- *lp* receives the solver status and the primal solution
- ← *iteration* number of the current iteration

Return values:

- true* if `solve_lp` returns true and the solution is read
- false* otherwise, `solve_lp` or `read_primals` returns false

See also:

`read_primals` (p.101), `solve_lp` (p.104)

Reimplemented from **Solver_module** (p.123).

References `Lp::feasibility`, `Lp::module`, `Solver_module::preport`, `primal_perturb()`, `Report::print()`, `read_primals()`, `solve_lp()`, `ss_feasible`, and `ss_unbounded`.

Referenced by `get_func_pointers()`.

2.14.2.28 `bool Sm_lps4_0_1_11::transform_lp (Lp * lp, const double relative_interval_radius, double & eta)` [static, private]

Transform lp from `lp_solve` to **Lurupa**.

Transform lp from the data structure of `lp_solve 4.0.1.11` into the representation used in **Lurupa** (p.36). Adjust algorithm parameters *eta* according to the lp, and inflate the lp to an interval valued one if `relative_interval_radius` is greater than 0.

Write the (midpoint) problem with **preport** (p.124) if **Report::write_vm** (p.66) is set.

Parameters:

- *lp* receives the transformed data
- ← *relative_interval_radius* relative interval radius to inflate lp parameters to
- *eta* gets adjusted to the model

Return values:

- true* if the transformation succeeds,

false otherwise, `lp->module->lp` is not set or `build_constraint_maps` returns false

See also:

`adjust_eta` (p. 95), `build_constraint_maps` (p. 96), `find_free_variables` (p. 96), `inflate_lp` (p. 97), `read_general_data` (p. 99), `read_lp_mat` (p. 100), `read_right_hand_sides` (p. 101), `read_simple_bounds` (p. 101), `resize_lp` (p. 102)

References `adjust_eta()`, `build_constraint_maps()`, `find_free_variables()`, `Lp::free_variables_size`, `Report::get_write_vm()`, `Lp::ia`, `Lp::IA`, `Lp::ib`, `Lp::IB`, `Lp::ic`, `inflate_lp()`, `Lp::module`, `Lp::name`, `Solver_module::preport`, `Report::print()`, `read_general_data()`, `read_lp_mat()`, `read_right_hand_sides()`, `read_simple_bounds()`, `resize_lp()`, `Report::write_matrix()`, `Report::write_vector()`, `Lp::xl`, and `Lp::xu`.

Referenced by `read_lp()`, and `set_lp()`.

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

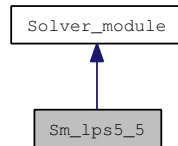
- `Sm_lps4_0_1_11.h`
- `Sm_lps4_0_1_11.cpp`

2.15 Sm_lps5_5 Class Reference

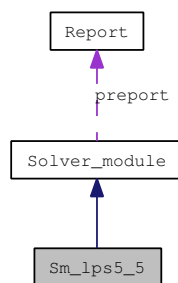
Solver module for lp_solve 5.5.

```
#include "Sm_lps5_5.h"
```

Inheritance diagram for Sm_lps5_5:



Collaboration diagram for Sm_lps5_5:



Static Public Member Functions

- static double **get_accuracy** ()
Get accuracy used by lp_solve.
- static bool **get_dual_ray** (Lp *lp, int &iyzray)
Get dual improving ray.
- static bool **get_primal_ray** (Lp *lp, int &ixray)
Get primal improving ray.
- static const char * **get_version** ()
Get module version string.
- static bool **init** (Report *report)
Initialize the module.
- static void **lp2solver** (Lp *lp)
Transform an lp into lp_solve's representation.
- static void **print_brief_version** ()
Print brief module version.
- static void **print_options** ()

Print supported options.

- static void **print_version** ()
Print module version.
- static bool **read_lp** (**Lp** *lp, FILE *in, const double relative_interval_radius, double &eta)
Read lp.
- static void **restore_dual** (**Lp** *lp)
Restore lp after dual perturbation.
- static void **restore_dual_phase1** (**Lp** *)
Restore lp after dual phase 1 perturbation.
- static void **restore_primal** (**Lp** *lp)
Restore lp after primal perturbation.
- static void **restore_primal_phase1** (**Lp** *)
Restore lp after primal phase 1 perturbation.
- static void **set_bounds** (**Lp** *lp, int var, double lower, double upper)
Set simple bounds on a variable.
- static void **set_dual_phase1** (**Lp** *lp)
Set dual phase 1 lp.
- static bool **set_lp** (**Lp** *lp, const double relative_interval_radius, double &eta)
Set lp.
- static bool **set_module_options** (**Lp** *lp, int argc, char *argv[])
Set module options.
- static void **set_primal_phase1** (**Lp** *lp)
Set primal phase 1 lp.
- static bool **solve_dual_perturbed** (const VECTOR &deflation_c, **Lp** *lp, const int iteration)
Compute solution of dual perturbed lp.
- static bool **solve_original** (**Lp** *lp, double &optimal_value)
Compute solution of unperturbed lp.
- static bool **solve_primal_perturbed** (const **Primal_deflation** &deflation, **Lp** *lp, const int iteration)
Compute solution of primal perturbed lp.

Static Private Member Functions

- static void **dual_perturb** (const VECTOR &d_c, const **Lp** *lp)
Dual perturb lp.
- static void __WINAPI **lps_log** (lprec *lp_lps, void *userhandle, char *buf)
Callback function for lp_solve's log messages.
- static void **primal_perturb** (const **Primal_deflation** &d, const **Lp** *lp)
Primal perturb lp.
- static bool **read_duals** (**Lp** *lp)
Read dual solution.
- static bool **read_primals** (**Lp** *lp)
Read primal solution.
- static bool **solve_lp** (**Lp** *lp)
Solve lp with lp_solve.
- static void **write_lp** (const char *sz_description, lprec *lp_lps, char *sz_file)
Write an LP to file.

Lp transforming routines

The following routines read the lp from the data structures of lp_solve 5.5 and transform it into Lurupa's representation.

- static void **adjust_eta** (double &eta, const **Lp** *lp)
Adjust eta to the lp.
- static bool **build_constraint_maps** (int &c_eq, int &c_le, const int rows, const int cols, **Lp** *lp)
Build mapping between lp_solve's and Lurupa's constraint structures.
- static void **find_free_variables** (const int vars, **Lp** *lp, int *c_inf_bnd)
Find free variables.
- static void **inflate_lp** (**Lp** *lp, const double r)
Inflate lp parameters.
- static void **read_general_data** (**Lp** *lp)
Read general data from lp_solve's structures.
- static bool **read_lp_mat** (const int rows, **Lp** *lp)
Read lp_solve's lp_mat structure.
- static void **read_right_hand_sides** (**Lp** *lp)
Read right hand sides of the constraints.
- static void **read_simple_bounds** (**Lp** *lp)
Read simple bounds.

- static void **resize_lp** (**Lp** *lp, const int cols, const int c_eq, const int c_le)
Resize Lurupa's data structures according to the lp dimensions.
- static bool **transform_lp** (**Lp** *lp, const double relative_interval_radius, double &eta)
Transform lp from lp_solve to Lurupa.

Static Private Attributes

- static bool **freset_bas** = false
Whether models are resolved with an all slack basis in case of numerical failure.
- static long **timeout** = 0
- static MYBOOL **trace** = FALSE
- static int **verbosity** = 3
- static **Write_mps** **write_mps** = wm_none
Whether intermediate models are saved.

2.15.1 Detailed Description

Solver module for lp_solve 5.5.

2.15.2 Member Function Documentation

2.15.2.1 void Sm_lps5_5::adjust_eta (double & eta, const Lp * lp) [static, private]

Adjust eta to the lp.

Adjust the algorithm parameter **eta** to be problem dependent. It becomes the maximum of its previous value and 10^{-20} times the maximum norm of the right hand sides of the inequality constraints,

$$\eta = \max\{\eta, 10^{-20}\|a\|_{\infty}\}.$$

Parameters:

- **eta** parameter to be adjusted
- ← **lp** lp **eta** is adjusted to

References Lp::ia, Solver_module::preport, and Report::print().

Referenced by transform_lp().

2.15.2.2 bool Sm_lps5_5::build_constraint_maps (int & c_eq, int & c_le, const int rows, const int cols, Lp * lp) [static, private]

Build mapping between lp_solve's and Lurupa's constraint structures.

Build a mapping from Lurupa's equality and inequality constraints to lp_solve's constraints, which is storing all constraints in one big matrix. Store the mapping in lp->module->mp_eq_con and lp->module->mp_le_con. Check if the number of constraints is less than the number of variables.

Parameters:

- *c_eq* number of equations in lp
- *c_le* number of inequalities in lp
- ← *rows* number of rows of constraint matrix
- ← *cols* number of columns of constraint matrix
- ← *lp* the lp

Return values:

- true* if number of constraints is less than number of variables,
- false* otherwise

References Lp_lp_solve::lp, Lp::module, Lp_lp_solve::mp_eq_con, Lp_lp_solve::mp_le_con, Solver_module::preport, and Report::print().

Referenced by transform_lp().

2.15.2.3 void Sm_lps5_5::dual_perturb (const VECTOR & d_c, const Lp * lp)
[static, private]

Dual perturb lp.

Perturb the lp in the cost vector.

Parameters:

- ← *d_c* deflation parameters
- *lp* contains the base for the deflation

References Lp::ic, Lp_lp_solve::lp, Lp::maximize, and Lp::module.

Referenced by solve_dual_perturbed().

2.15.2.4 void Sm_lps5_5::find_free_variables (const int vars, Lp * lp, int * c_inf_bnd) [static, private]

Find free variables.

Find free variables in the lp. Store the indices of the free variables in lp->free_variables and their number in lp->free_variables_size.

Parameters:

- ← *vars* number of variables in lp
- *lp* stores number and indices of the free variables
- *c_inf_bnd* count of variables with infinite bounds

References Lp::free_variables, Lp::free_variables_size, and Lp::module.

Referenced by transform_lp().

2.15.2.5 double Sm_lps5_5::get_accuracy () [static]

Get accuracy used by lp_solve.

Get lp_solve's set accuracy

Returns:

lp_solve's accuracy

Reimplemented from **Solver_module** (p. 123).

Referenced by get_func_pointers().

2.15.2.6 const char * Sm_lps5_5::get_version () [static]

Get module version string.

Get the version information of the solver module.

Reimplemented from **Solver_module** (p. 122).

References version.

Referenced by get_func_pointers().

2.15.2.7 void Sm_lps5_5::inflate_lp (Lp * lp, const double r) [static, private]

Inflate lp parameters.

Inflate all lp parameters to intervals with relative radius r ,

$$x \rightarrow \mathbf{x} \triangleq x \cdot [1 - r, 1 + r]$$

Parameters:

\rightarrow **lp** lp to be inflated

\leftarrow **r** relative interval radius

References Lp::ia, Lp::IA, Lp::ib, Lp::IB, and Lp::ic.

Referenced by transform_lp().

2.15.2.8 bool Sm_lps5_5::init (Report * report) [static]

Initialize the module.

Set **preport** (p. 124) to ***report**. Check the version of lp_solve.

Parameters:

\leftarrow **report** the **Report** (p. 65) instance to use

Return values:

true

Reimplemented from **Solver_module** (p.122).

References Solver_module::preport, Report::print(), and version.

Referenced by get_func_pointers().

2.15.2.9 void __WINAPI Sm_lps5_5::lps_log (lprec * *lp_lps*, void * *userhandle*, char * *buf*) [static, private]

Callback function for lp_solve's log messages.

Callback function for lp_solve's log functionality. Print the supplied log message marked as coming from lp_solve.

References Solver_module::preport, and Report::print().

Referenced by lp2solver(), and read_lp().

2.15.2.10 void Sm_lps5_5::primal_perturb (const Primal_deflation & *d*, const Lp * *lp*) [static, private]

Primal perturb lp.

Perturb the lp in the right hand sides of the inequalities and the simple bounds.

Parameters:

← *d* deflation parameters

→ *lp* contains the base for the deflation

References Primal_deflation::a, Lp::ia, Lp_lp_solve::lp, Lp::module, Lp_lp_solve::mp_le_con, Lp::xl, Primal_deflation::xl, Lp::xu, and Primal_deflation::xu.

Referenced by solve_primal_perturbed().

2.15.2.11 void Sm_lps5_5::print_options () [static]

Print supported options.

Print the command line options supported by the solver module.

Reimplemented from **Solver_module** (p.122).

References version.

Referenced by get_func_pointers().

2.15.2.12 bool Sm_lps5_5::read_duals (Lp * *lp*) [static, private]

Read dual solution.

Read the dual solution from lp_solve and store it in *lp->iy* and *lp->iz*. Lp_solve's convention to determine the sign of the duals is, that the costs must be a linear combination of the gradients of the constraints.

The duals returned by lp_solve, however, are the duals for the original lp, that is the duals for greater equal constraints are multiplied by -1 , and all duals are again multiplied by -1 if the lp is a maximizing one. As Lurupa stores all linear programs as minimizing and all inequality constraints as less equal undo these multiplications.

Parameters:

→ *lp* receives the dual solution

Return values:

true if reading the dual solution succeeded,
false otherwise, calling `lp_solve's get_sensitivity_rhs` returns false

References `Lp::iy`, `Lp::iz`, `Lp_lp_solve::lp`, `Lp::maximize`, `Lp::module`, `Lp_lp_solve::mp_eq_con`, `Lp_lp_solve::mp_le_con`, `Solver_module::preport`, and `Report::print()`.

Referenced by `solve_dual_perturbed()`, and `solve_original()`.

2.15.2.13 void Sm_lps5_5::read_general_data (Lp * lp) [static, private]

Read general data from `lp_solve's` structures.

Read general `lp` properties from `lp_solve's` structures and store them in `lp`. This includes the value representing infinity, the direction of optimization, and the name of the `lp`.

Parameters:

→ *lp* stores the read data

References `Lp::infinite`, `Lp::maximize`, `Lp::module`, and `Lp::name`.

Referenced by `transform_lp()`.

2.15.2.14 bool Sm_lps5_5::read_lp (Lp * lp, FILE * in, const double relative_interval_radius, double & eta) [static]

Read `lp`.

Read an `lp` from a file into `lp_solve` and `lp`. Inflate the `lp` to an interval valued one if `relative_interval_radius` is greater than 0, and adjust `eta` according to the `lp's` parameters.

Parameters:

→ *lp* receives the `lp`
 ← *in* file pointer to `lp`
 ← *relative_interval_radius* interval radius to inflate `lp` parameters to
 → *eta* algorithm parameter to be adjusted

Return values:

true if reading and transforming the `lp` succeeds,
false otherwise, reading the `lp` fails or `transform_lp` returns false

See also:

`transform_lp` (p.120)

Reimplemented from `Solver_module` (p.123).

References `Lp_lp_solve::lp`, `lps_log()`, `Lp::module`, and `transform_lp()`.

Referenced by `get_func_pointers()`.

2.15.2.15 `bool Sm_lps5_5::read_lp_mat (const int rows, Lp * lp)` [static, private]

Read lp_solve's lp_mat structure.

Read lp_solve's lp_mat structure. This stores the left hand sides of the constraints together with the objective function. Split the read coefficients into equality and inequality constraints, transform inequality constraints to all less equal, and the direction of optimization to minimize.

Parameters:

- ← *rows* number of rows in constraint matrix
- *lp* stores the constraints and objective function

Return values:

- true* if reading lp_mat succeeds,
- false* otherwise, lp_solve's get_column returns false

References Lp::IA, Lp::IB, Lp::ic, Lp_lp_solve::lp, Lp::module, Lp_lp_solve::mp_eq_con, and Lp_lp_solve::mp_le_con.

Referenced by transform_lp().

2.15.2.16 `bool Sm_lps5_5::read_primals (Lp * lp)` [static, private]

Read primal solution.

Read the primal solution from lp_solve and store it in lp->ix.

Parameters:

- *lp* receives the primal solution

Return values:

- true* if reading the primal solution succeeds,
- false* otherwise, calling lp_solve's get_variables returns false

References Lp::ic, Lp::ix, Lp_lp_solve::lp, Lp::module, Solver_module::preport, and Report::print().

Referenced by solve_original(), and solve_primal_perturbed().

2.15.2.17 `void Sm_lps5_5::read_right_hand_sides (Lp * lp)` [static, private]

Read right hand sides of the constraints.

Read the right hand sides of the constraints. Split the read right hand sides into equality and inequality constraints, transform inequality constraints to all less equal. Store the values in lp->ia and lp->ib.

Parameters:

- *lp* stores the right hand sides

References `Lp::ia`, `Lp::ib`, `Lp_lp_solve::lp`, `Lp::module`, `Lp_lp_solve::mp_eq_con`, and `Lp_lp_solve::mp_le_con`.

Referenced by `transform_lp()`.

2.15.2.18 `void Sm_lps5_5::read_simple_bounds (Lp * lp) [static, private]`

Read simple bounds.

Read the simple bounds of the variables and store them in `lp->xl` and `lp->xu`.

Parameters:

→ *lp* stores the simple bounds

References `Lp::module`, `Lp::non_fixed_vars`, `Lp::xl`, and `Lp::xu`.

Referenced by `transform_lp()`.

2.15.2.19 `void Sm_lps5_5::resize_lp (Lp * lp, const int cols, const int c_eq, const int c_le) [static, private]`

Resize Lurupa's data structures according to the lp dimensions.

Resize the vectors and matrices in Lurupa's data structure to the sizes of the lp.

Parameters:

→ *lp* Lurupa's structure with the vectors and matrices to be resized

← *cols* number of variables in the lp

← *c_eq* number of equality constraints in the lp

← *c_le* number of inequality constraints in the lp

References `Lp::ia`, `Lp::IA`, `Lp::ib`, `Lp::IB`, `Lp::ic`, `Lp::ix`, `Lp::iy`, `Lp::iz`, `Lp::xl`, and `Lp::xu`.

Referenced by `restore_primal_phase1()`, `set_primal_phase1()`, and `transform_lp()`.

2.15.2.20 `void Sm_lps5_5::restore_dual (Lp * lp) [static]`

Restore lp after dual perturbation.

Take the model parameters for the costs from `lp` and write to `lp->module->lp` restoring the original model after computation of the lower bound.

Parameters:

← *lp* contains the original parameters

Reimplemented from **Solver_module** (p.123).

References `Lp::ic`, `Lp::maximize`, `Lp::module`, `Solver_module::preport`, and `Report::print()`.

Referenced by `get_func_pointers()`.

2.15.2.21 void Sm_lps5_5::restore_primal (Lp * *lp*) [static]

Restore *lp* after primal perturbation.

Take the model parameters for the right hand sides and the simple bounds from *lp* and write to *lp->module->lp* restoring the original model after computation of the upper bound.

Parameters:

← *lp* contains the original parameters

Reimplemented from **Solver_module** (p.123).

References Lp::ia, Lp::IA, Lp_lp_solve::lp, Lp::module, Lp_lp_solve::mp_le_con, Solver_module::preport, Report::print(), Lp::xl, and Lp::xu.

Referenced by get_func_pointers().

2.15.2.22 bool Sm_lps5_5::set_module_options (Lp * *lp*, int *argc*, char * *argv*[]) [static]

Set module options.

Set the specified module options. These are supplied in *argc* and *argv* in a command line version. Each option is supplied as one string entry in the string array *argv*, *argc* specifies the number of options.

The module supports the following options

- '-sm,resetbas': Resolve lps in the case of numerical failure with a default basis of all slacks. This may enable *lp_solve* to find a solution at the expense of increased runtime.
- '-sm,s': Use *lp_solve*'s default scaling (Numerical range-based scaling).
- '-sm,timeout,n': Set *lp_solve*'s timeout to *n* seconds.
- '-sm,vn': Set *lp_solve*'s verbosity level to *n*. The available levels are
 - v0 NEUTRAL
 - v1 CRITICAL
 - v2 SEVERE
 - v3 IMPORTANT (default)
 - v4 NORMAL
 - v5 DETAILED
 - v6 FULL

Parameters:

→ *lp* *lp* to set the options for

← *argc* number of arguments

← *argv* array of arguments

Return values:

true if options are set

false otherwise

References `freset_bas`, `Lp::module`, `Solver_module::preport`, `Report::print()`, `timeout`, `trace`, `verbosity`, `wm_all`, `wm_ask`, `wm_none`, and `write_mps`.

Referenced by `get_func_pointers()`.

2.15.2.23 `bool Sm_lps5_5::solve_dual_perturbed (const VECTOR & d_c, Lp * lp, const int iteration) [static]`

Compute solution of dual perturbed lp.

Perturb the lp and solve the new one. Read the dual solution from `lp_solve` and store it in `lp->iy` and `lp->iz`. Save the perturbed lp as `lpname1-lbii.mps` according to the setting of `write_mps` (p. 110). Substitute the name of the lp for `lpname` and `iteration` for `ii`.

Parameters:

- ← `d_c` deflation parameter
- `lp` receives the solver status and the dual solution
- ← `iteration` number of the current iteration

Return values:

- true* if `solve_lp` returns true and the solution is read
- false* otherwise, `solve_lp` or `read_duals` returns false

See also:

`read_duals` (p. 113), `solve_lp` (p. 118)

Reimplemented from `Solver_module` (p. 123).

References `dual_perturb()`, `Lp::feasibility`, `Lp::module`, `Lp::name`, `Solver_module::preport`, `Report::print()`, `read_duals()`, `solve_lp()`, `ss_feasible`, and `write_lp()`.

Referenced by `get_func_pointers()`.

2.15.2.24 `bool Sm_lps5_5::solve_lp (Lp * lp) [static, private]`

Solve lp with `lp_solve`.

Solve an lp with `lp_solve`. Store `lp_solve`'s status in `lp->feasibility`.

Parameters:

- `lp` receives `lp_solve`'s status

Return values:

- true* if `lp_solve` returns a known status code (i.e., `OPTIMAL`, `INFEASIBLE`, `UNBOUNDED`, `TIMEOUT`, `NUMFAILURE`, `DEGENERATE`, or `SUBOPTIMAL`)
- false* otherwise

References `Lp::feasibility`, `freset_bas`, `Lp::module`, `Solver_module::preport`, `Report::print()`, `ss_failure`, `ss_feasible`, `ss_infeasible`, `ss_timeout`, and `ss_unbounded`.

Referenced by `solve_dual_perturbed()`, `solve_original()`, and `solve_primal_perturbed()`.

2.15.2.25 `bool Sm_lps5_5::solve_original (Lp * lp, double & optimal_value)`
`[static]`

Compute solution of unperturbed lp.

Solve the original unperturbed lp with `lp_solve` and read the primal and dual solution.

Parameters:

- *lp* receives the solver status and the solution
- *optimal_value* receives the optimal value

Return values:

- true* if `solve_lp` returns true and the solution is read
- false* otherwise, `solve_lp`, `read_primals`, or `read_duals` returns false

See also:

`read_duals` (p. 113), `read_primals` (p. 115), `solve_lp` (p. 118)

Reimplemented from `Solver_module` (p. 123).

References `Lp::module`, `Solver_module::preport`, `Report::print()`, `read_duals()`, `read_primals()`, and `solve_lp()`.

Referenced by `get_func_pointers()`.

2.15.2.26 `bool Sm_lps5_5::solve_primal_perturbed (const Primal_deflation & d, Lp * lp, const int iteration)` `[static]`

Compute solution of primal perturbed lp.

Perturb the lp and solve the new one. Read the primal solution and store it in `lp->ix`. Save the perturbed lp to `lpname-ubii.mps` according to setting of `write_mps` (p. 110). Substitute the name of the lp for `lpname` and `iteration` for `ii`.

Parameters:

- ← *d* deflation parameters
- *lp* receives the solver status and the primal solution
- ← *iteration* number of the current iteration

Return values:

- true* if `solve_lp` returns true and the solution is read
- false* otherwise, `solve_lp` or `read_primals` returns false

See also:

`read_primals` (p. 115), `solve_lp` (p. 118)

Reimplemented from `Solver_module` (p. 123).

References `Lp::feasibility`, `Lp::module`, `Lp::name`, `Solver_module::preport`, `primal_perturb()`, `Report::print()`, `read_primals()`, `solve_lp()`, `ss_feasible`, and `write_lp()`.

Referenced by `get_func_pointers()`.

2.15.2.27 `bool Sm_lps5_5::transform_lp (Lp * lp, const double
relative_interval_radius, double & eta)` [static, private]

Transform lp from lp_solve to Lurupa.

Transform lp from the data structure of lp_solve 5.5 into the representation used in **Lurupa** (p. 36). Adjust algorithm parameters **eta** according to the lp, and inflate the lp to an interval valued one if **relative_interval_radius** is greater than 0.

Write the (midpoint) problem with **preport** (p. 124) if **Report::write_vm** (p. 66) is set.

Parameters:

- **lp** receives the transformed data
- ← **relative_interval_radius** relative interval radius to inflate lp parameters to
- **eta** gets adjusted to the model

Return values:

- true** if the transformation succeeds,
- false** otherwise, lp->module->lp is not set or build_constraint_maps returns false

See also:

adjust_eta (p. 110), **build_constraint_maps** (p. 110), **find_free_variables** (p. 111), **inflate_lp** (p. 112), **read_general_data** (p. 114), **read_lp_mat** (p. 115), **read_right_hand_sides** (p. 115), **read_simple_bounds** (p. 116), **resize_lp** (p. 116)

References **adjust_eta()**, **build_constraint_maps()**, **find_free_variables()**, **Lp::free_variables_size**, **Report::get_write_vm()**, **Lp::ia**, **Lp::IA**, **Lp::ib**, **Lp::IB**, **Lp::ic**, **inflate_lp()**, **Lp::module**, **Lp::name**, **Solver_module::preport**, **Report::print()**, **read_general_data()**, **read_lp_mat()**, **read_right_hand_sides()**, **read_simple_bounds()**, **resize_lp()**, **Report::write_matrix()**, **Report::write_vector()**, **Lp::xl**, and **Lp::xu**.

Referenced by **read_lp()**, and **set_lp()**.

2.15.3 Member Data Documentation

2.15.3.1 `long Sm_lps5_5::timeout = 0` [static, private]

Timeout setting

Referenced by **lp2solver()**, and **set_module_options()**.

2.15.3.2 `MYBOOL Sm_lps5_5::trace = FALSE` [static, private]

trace setting

Referenced by **lp2solver()**, and **set_module_options()**.

2.15.3.3 `int Sm_lps5_5::verbosity = 3` [static, private]

Verbosity setting

Referenced by **lp2solver()**, and **set_module_options()**.

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

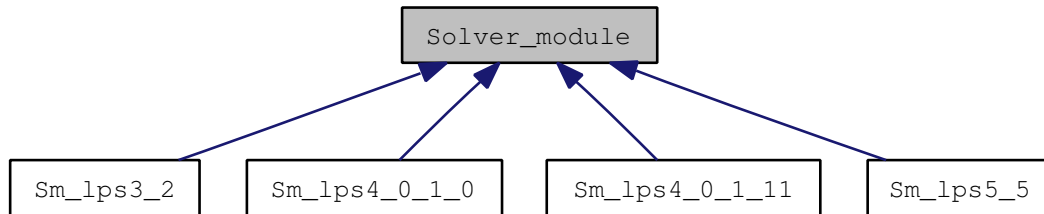
- Sm_lps5_5.h
- Sm_lps5_5.cpp

2.16 Solver_module Class Reference

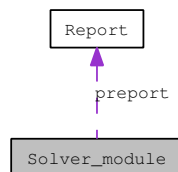
Interface for an lp-solver module.

```
#include <lurupa/Solver_module.h>
```

Inheritance diagram for Solver_module:



Collaboration diagram for Solver_module:



Static Public Member Functions

- static bool **get_dual_ray** (Lp *lp, int &iyzray)
Get dual improving ray.
- static bool **get_primal_ray** (Lp *lp, int &ixray)
Get primal improving ray.
- static const char * **get_version** ()
Get module version string.
- static bool **init** (Report *preport)
Initialize the module.
- static void **lp2solver** (Lp *lp_lurupa)
Transform an lp into solver's representation.
- static void **print_brief_version** ()
Print brief version information.
- static void **print_options** ()
Print options supported by module.
- static void **print_version** ()

Print version information.

- static void **restore__dual** (**Lp** *lp)
Restore lp after dual perturbation.
- static void **restore__dual__phase1** (**Lp** *)
Restore lp after dual phase 1 perturbation.
- static void **restore__primal** (**Lp** *lp)
Restore lp after primal perturbation.
- static void **restore__primal__phase1** (**Lp** *)
Restore lp after primal phase 1 perturbation.
- static void **set__bounds** (**Lp** *lp, int var, double lower, double upper)
Set simple bounds on a variable.
- static void **set__dual__phase1** (**Lp** *lp)
Set dual phase 1 lp.
- static bool **set__lp** (**Lp** *lp, const double relative__interval__radius, double &eta)
Set an lp.
- static void **set__primal__phase1** (**Lp** *lp)
Set primal phase 1 lp.
- static bool **solve__dual__perturbed** (const VECTOR &deflation_c, **Lp** *lp, const int iteration)
Compute solution of dual perturbed lp.
- static bool **solve__original** (**Lp** *lp, double &optimal__value)
Compute solution of unperturbed lp.
- static bool **solve__primal__perturbed** (const **Primal_deflation** &deflation, **Lp** *lp, const int iteration)
Compute solution of primal perturbed lp.
- static double **get__accuracy** ()
Get accuracy used by solver.
- static bool **read__lp** (**Lp** *lp, FILE *in, const double relative__interval__radius, double &eta)
Read an lp.
- static bool **set__module__options** (int argc, char *argv[])
Set module options.

Static Protected Attributes

- static **Report** * **preport** = NULL
pointer to report class instance

2.16.1 Detailed Description

Interface for an lp-solver module.

This class is the prototype for the lp-solver modules. The interface between **Lurupa** (p. 36) and the lp-solver is specified.

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

- **Solver_module.h**
- **Sm_lps3_2.cpp**
- **Sm_lps4_0_1_0.cpp**
- **Sm_lps4_0_1_11.cpp**
- **Sm_lps5_5.cpp**

2.17 Solver_module_interface Struct Reference

Function pointer version of Solver_module.

```
#include <lurupa/Solver_module.h>
```

Public Attributes

- `double(* get__accuracy)()`
Get accuracy used by solver.
- `bool(* get__dual_ray)(Lp *, int &)`
Get dual improving ray.
- `bool(* get__primal_ray)(Lp *, int &)`
Get primal improving ray.
- `const char *(* get__version)()`
Get module version string.
- `bool(* init)(Report *)`
Initialize the module.
- `void(* lp2solver)(Lp *)`
Transform an lp into solver's representation.
- `void(* print__brief__version)()`
Print brief version information.
- `void(* print__options)()`
Print options supported by module.
- `void(* print__version)()`
Print version information.
- `bool(* read__lp)(Lp *, FILE *, const double, double &)`
Read an lp.
- `void(* restore__dual)(Lp *)`
Restore lp after dual perturbation.
- `void(* restore__dual__phase1)(Lp *)`
Restore lp after dual phase 1 perturbation.
- `void(* restore__primal)(Lp *)`
Restore lp after primal perturbation.
- `void(* restore__primal__phase1)(Lp *)`
Restore lp after primal phase 1 perturbation.

- `void(* set_bounds)(Lp *, int, double, double)`
Set simple bounds on a variable.
- `void(* set_dual_phase1)(Lp *)`
Set dual phase 1 lp.
- `bool(* set_lp)(Lp *, const double, double &)`
Set an lp.
- `bool(* set_module_options)(Lp *, int, char *[])`
Set module options.
- `void(* set_primal_phase1)(Lp *)`
Set primal phase 1 lp.
- `bool(* solve_dual_perturbed)(const VECTOR &, Lp *, const int)`
Compute solution of dual perturbed lp.
- `bool(* solve_original)(Lp *, double &)`
Compute solution of unperturbed lp.
- `bool(* solve_primal_perturbed)(const Primal_deflation &, Lp *, const int)`
Compute solution of primal perturbed lp.

2.17.1 Detailed Description

Function pointer version of Solver_module.

This struct is a function pointer version of **Solver_module** (p. 122).

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

- **Solver_module.h**

Chapter 3

File Documentation

3.1 cLurupa.cpp File Reference

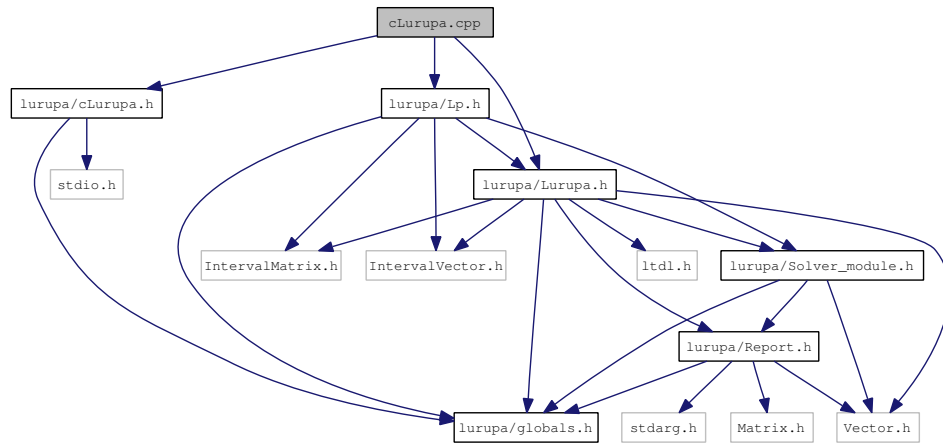
Definition of C wrapper for Lurupa.

```
#include <lurupa/cLurupa.h>
```

```
#include <lurupa/Lurupa.h>
```

```
#include <lurupa/Lp.h>
```

Include dependency graph for cLurupa.cpp:



Functions

- void **clu_cleanup** ()
Cleanup library wrapper.
- double **clu_get_alpha** ()
Get algorithm parameter alpha.
- const char * **clu_get_core_version** ()
Get core version string.

- double **clu_get_eta** ()
Get algorithm parameter eta.
- char * **clu_get_lp_name** (PLP lp)
Get linear program's name.
- const char * **clu_get_module_version** ()
Get solver module version.
- double **clu_get_solver_eps** ()
Get solver's accuracy.
- void **clu_init** ()
Initialize library wrapper.
- BOOL **clu_is_inflate** ()
Is inflation set.
- BOOL **clu_is_lp_maximize** (PLP lp)
Return whether linear program is maximized.
- **Bound_status clu_lower_bound** (PLP lp, double *bound, int *iterations)
Compute lower bound.
- void **clu_print_module_options** ()
Print options supported by module.
- PLP **clu_read_lp** (FILE *in, const double relative_interval_radius)
Read linear program.
- void **clu_set_alpha** (double alpha)
Set algorithm parameter alpha.
- void **clu_set_eta** (double eta)
Set algorithm parameter eta.
- void **clu_set_inflate** (BOOL inflate)
Set inflation.
- void **clu_set_interface** (Solver_module_interface interface)
- BOOL **clu_set_lp** (PLP lp, const double relative_interval_radius)
- BOOL **clu_set_module** (char *module_path)
Set solver module.
- BOOL **clu_set_module_options** (PLP lp, int argc, char *argv[])
Set solver options.
- BOOL **clu_set_solution** (PLP lp, void *p)
Set a linear program's feasibility.

- **BOOL clu_solve_lp** (PLP lp, double *optimal_value)
- **Bound_status clu_upper_bound** (PLP lp, double *bound, int *iterations)
Compute upper bound.

Variables

- static **Lurupa l**
Lurupa (p. 36) instance used by C wrapper.

3.1.1 Detailed Description

Definition of C wrapper for Lurupa.

This is work in progress.

Author:

Christian Keil

Id

cLurupa.cpp (p. 127) 533 2008-06-12 20:08:12Z keil

3.1.2 Function Documentation

3.1.2.1 void clu_cleanup ()

Cleanup library wrapper.

Cleanup the library wrapper.

3.1.2.2 double clu_get_alpha ()

Get algorithm parameter alpha.

Get the algorithm parameter alpha.

See also:

Lurupa::get_alpha (p. 51)

References `clu_init()`, and `Lurupa::get_alpha()`.

3.1.2.3 const char* clu_get_core_version ()

Get core version string.

Get the core version string.

See also:

Lurupa::get_core_version (p. 51)

References `clu_init()`, and `Lurupa::get_core_version()`.

3.1.2.4 double clu_get_eta ()

Get algorithm parameter eta.

Get the algorithm parameter eta.

See also:

Lurupa::get_eta (p. 51)

References `clu_init()`, and `Lurupa::get_eta()`.

3.1.2.5 char* clu_get_lp_name (PLP *lp*)

Get linear program's name.

Get the linear program's name.

See also:

Lurupa::get_lp_name (p. 36)

References `clu_init()`.

3.1.2.6 const char* clu_get_module_version ()

Get solver module version.

Get the solver module version.

See also:

Lurupa::get_module_version (p. 51)

References `clu_init()`, and `Lurupa::get_module_version()`.

3.1.2.7 double clu_get_solver_eps ()

Get solver's accuracy.

Get solver accuracy.

See also:

Lurupa::get_solver_eps (p. 51)

References `clu_init()`, and `Lurupa::get_solver_eps()`.

3.1.2.8 void clu_init ()

Initialize library wrapper.

Initialize the library wrapper.

Referenced by clu_get_alpha(), clu_get_core_version(), clu_get_eta(), clu_get_lp_name(), clu_get_module_version(), clu_get_solver_eps(), clu_is_inflate(), clu_is_lp_maximize(), clu_lower_bound(), clu_print_module_options(), clu_read_lp(), clu_set_alpha(), clu_set_eta(), clu_set_inflate(), clu_set_interface(), clu_set_lp(), clu_set_module(), clu_set_module_options(), clu_set_solution(), clu_solve_lp(), and clu_upper_bound().

3.1.2.9 BOOL clu_is_inflate ()

Is inflation set.

Get the algorithm flag inflate.

See also:

Lurupa::is_inflate (p. 52)

References clu_init(), and Lurupa::is_inflate().

3.1.2.10 BOOL clu_is_lp_maximize (PLP lp)

Return whether linear program is maximized.

Return whether the linear program is maximized.

See also:

Lurupa::is_lp_maximize (p. 36)

References clu_init().

3.1.2.11 Bound_status clu_lower_bound (PLP lp, double * bound, int * iterations)

Compute lower bound.

Compute the lower bound.

See also:

Lurupa::lower_bound (p. 52)

References clu_init(), and Lurupa::lower_bound().

3.1.2.12 void clu_print_module_options ()

Print options supported by module.

Print the options supported by the module.

See also:

Lurupa::print_module_options (p. 54)

References `clu_init()`, and `Lurupa::print_module_options()`.

3.1.2.13 PLP `clu_read_lp` (FILE * *in*, const double *relative_interval_radius*)

Read linear program.

Read a linear program.

See also:

Lurupa::read_lp (p. 58)

References `clu_init()`, and `Lurupa::read_lp()`.

3.1.2.14 void `clu_set_alpha` (double *alpha*)

Set algorithm parameter alpha.

Set the algorithm parameter alpha.

See also:

Lurupa::set_alpha (p. 59)

References `clu_init()`, and `Lurupa::set_alpha()`.

3.1.2.15 void `clu_set_eta` (double *eta*)

Set algorithm parameter eta.

Set the algorithm parameter eta.

See also:

Lurupa::set_eta (p. 60)

References `clu_init()`, and `Lurupa::set_eta()`.

3.1.2.16 void `clu_set_inflate` (BOOL *inflate*)

Set inflation.

Set the algorithm flag inflation.

See also:

Lurupa::set_inflate (p. 60)

References `clu_init()`, and `Lurupa::set_inflate()`.

3.1.2.17 void clu_set_interface (Solver_module_interface *interface*)

Set the solver module.

See also:

Lurupa::set_interface (p. 61)

References `clu_init()`, and `Lurupa::set_interface()`.

3.1.2.18 BOOL clu_set_lp (PLP *lp*, const double *relative_interval_radius*)

Set a linear program.

See also:

Lurupa::set_lp (p. 61)

References `clu_init()`, and `Lurupa::set_lp()`.

3.1.2.19 BOOL clu_set_module (char * *module_path*)

Set solver module.

Set the solver module.

See also:

Lurupa::set_module (p. 61)

References `clu_init()`, and `Lurupa::set_module()`.

3.1.2.20 BOOL clu_set_module_options (PLP *lp*, int *argc*, char * *argv*[])

Set solver options.

Set module options.

See also:

Lurupa::set_module_options (p. 62)

References `clu_init()`, and `Lurupa::set_module_options()`.

3.1.2.21 BOOL clu_solve_lp (PLP *lp*, double * *optimal_value*)

Solve the linear program.

See also:

Lurupa::solve_lp (p. 63)

References `clu_init()`, and `Lurupa::solve_lp()`.

3.1.2.22 `Bound_status clu_upper_bound` (PLP *lp*, double * *bound*, int * *iterations*)

Compute upper bound.

Compute the upper bound.

See also:

`Lurupa::upper_bound` (p. 63)

References `clu_init()`, and `Lurupa::upper_bound()`.

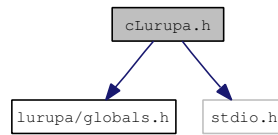
3.2 cLurupa.h File Reference

Declaration of C wrapper for Lurupa.

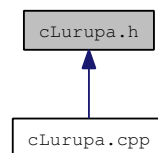
```
#include <lurupa/globals.h>
```

```
#include <stdio.h>
```

Include dependency graph for cLurupa.h:



This graph shows which files directly or indirectly include this file:



Typedefs

C version of variable types

*These definitions are used when including **cLurupa.h** (p. 135) in a C source.*

- typedef char **BOOL**
- typedef enum **Bound_status** **BOUND_STATUS**
- typedef void * **PLP**
- typedef enum **Solver_status** **SOLVER_STATUS**

Functions

- void **clu_cleanup** ()
Cleanup library wrapper.
- double **clu_get_alpha** ()
Get algorithm parameter alpha.
- const char * **clu_get_core_version** ()
Get core version string.
- double **clu_get_eta** ()
Get algorithm parameter eta.
- char * **clu_get_lp_name** (PLP Lp)
Get linear program's name.

- `const char * clu_get_module_version ()`
Get solver module version.
- `double clu_get_solver_eps ()`
Get solver's accuracy.
- `void clu_init ()`
Initialize library wrapper.
- `BOOL clu_is_inflate ()`
Is inflation set.
- `BOOL clu_is_lp_maximize (PLP Lp)`
Return whether linear program is maximized.
- `BOUND_STATUS clu_lower_bound (PLP Lp, double *bound, int *iterations)`
Compute lower bound.
- `void clu_print_module_options ()`
Print options supported by module.
- `PLP clu_read_lp (FILE *in, const double relative_interval_radius)`
Read linear program.
- `void clu_set_alpha (double alpha)`
Set algorithm parameter alpha.
- `void clu_set_eta (double eta)`
Set algorithm parameter eta.
- `void clu_set_inflate (BOOL inflate)`
Set inflation.
- `BOOL clu_set_lp (PLP *lp, const double relative_interval_radius)`
Set linear program.
- `BOOL clu_set_module (char *module_path)`
Set solver module.
- `BOOL clu_set_module_options (PLP Lp, int argc, char *argv[])`
Set solver options.
- `BOOL clu_set_solution (PLP Lp, void *p)`
Set a linear program's feasibility.
- `BOOL clu_solve_lp (double *optimal_value, SOLVER_STATUS *status)`
Solve linear program.
- `BOUND_STATUS clu_upper_bound (PLP Lp, double *bound, int *iterations)`
Compute upper bound.

3.2.1 Detailed Description

Declaration of C wrapper for Lurupa.

This is work in progress.

Author:

Christian Keil

Id

cLurupa.h (p. 135) 542 2008-06-26 12:44:51Z keil

3.2.2 Function Documentation

3.2.2.1 void clu_cleanup ()

Cleanup library wrapper.

Cleanup the library wrapper.

3.2.2.2 double clu_get_alpha ()

Get algorithm parameter alpha.

Get the algorithm parameter alpha.

See also:

Lurupa::get_alpha (p. 51)

References `clu_init()`, and `Lurupa::get_alpha()`.

3.2.2.3 const char* clu_get_core_version ()

Get core version string.

Get the core version string.

See also:

Lurupa::get_core_version (p. 51)

References `clu_init()`, and `Lurupa::get_core_version()`.

3.2.2.4 double clu_get_eta ()

Get algorithm parameter eta.

Get the algorithm parameter eta.

See also:

Lurupa::get_eta (p. 51)

References `clu_init()`, and `Lurupa::get_eta()`.

3.2.2.5 char* clu_get_lp_name (PLP lp)

Get linear program's name.

Get the linear program's name.

See also:

Lurupa::get_lp_name (p. 36)

References `clu_init()`.

3.2.2.6 const char* clu_get_module_version ()

Get solver module version.

Get the solver module version.

See also:

Lurupa::get_module_version (p. 51)

References `clu_init()`, and `Lurupa::get_module_version()`.

3.2.2.7 double clu_get_solver_eps ()

Get solver's accuracy.

Get solver accuracy.

See also:

Lurupa::get_solver_eps (p. 51)

References `clu_init()`, and `Lurupa::get_solver_eps()`.

3.2.2.8 void clu_init ()

Initialize library wrapper.

Initialize the library wrapper.

Referenced by `clu_get_alpha()`, `clu_get_core_version()`, `clu_get_eta()`, `clu_get_lp_name()`, `clu_get_module_version()`, `clu_get_solver_eps()`, `clu_is_inflate()`, `clu_is_lp_maximize()`, `clu_lower_bound()`, `clu_print_module_options()`, `clu_read_lp()`, `clu_set_alpha()`, `clu_set_eta()`, `clu_set_inflate()`, `clu_set_interface()`, `clu_set_lp()`, `clu_set_module()`, `clu_set_module_options()`, `clu_set_solution()`, `clu_solve_lp()`, and `clu_upper_bound()`.

3.2.2.9 BOOL clu_is_inflate ()

Is inflation set.

Get the algorithm flag inflate.

See also:

Lurupa::is_inflate (p. 52)

References `clu_init()`, and `Lurupa::is_inflate()`.

3.2.2.10 **BOOL** `clu_is_lp_maximize (PLP lp)`

Return whether linear program is maximized.

Return whether the linear program is maximized.

See also:

Lurupa::is_lp_maximize (p. 36)

References `clu_init()`.

3.2.2.11 **BOUND_STATUS** `clu_lower_bound (PLP lp, double * bound, int * iterations)`

Compute lower bound.

Compute the lower bound.

See also:

Lurupa::lower_bound (p. 52)

References `clu_init()`, and `Lurupa::lower_bound()`.

3.2.2.12 **void** `clu_print_module_options ()`

Print options supported by module.

Print the options supported by the module.

See also:

Lurupa::print_module_options (p. 54)

References `clu_init()`, and `Lurupa::print_module_options()`.

3.2.2.13 **PLP** `clu_read_lp (FILE * in, const double relative_interval_radius)`

Read linear program.

Read a linear program.

See also:

Lurupa::read_lp (p. 58)

References `clu_init()`, and `Lurupa::read_lp()`.

3.2.2.14 void clu_set_alpha (double *alpha*)

Set algorithm parameter alpha.

Set the algorithm parameter alpha.

See also:

Lurupa::set_alpha (p. 59)

References `clu_init()`, and `Lurupa::set_alpha()`.

3.2.2.15 void clu_set_eta (double *eta*)

Set algorithm parameter eta.

Set the algorithm parameter eta.

See also:

Lurupa::set_eta (p. 60)

References `clu_init()`, and `Lurupa::set_eta()`.

3.2.2.16 void clu_set_inflate (BOOL *inflate*)

Set inflation.

Set the algorithm flag inflation.

See also:

Lurupa::set_inflate (p. 60)

References `clu_init()`, and `Lurupa::set_inflate()`.

3.2.2.17 BOOL clu_set_module (char * *module_path*)

Set solver module.

Set the solver module.

See also:

Lurupa::set_module (p. 61)

References `clu_init()`, and `Lurupa::set_module()`.

3.2.2.18 BOOL clu_set_module_options (PLP *lp*, int *argc*, char * *argv*[])

Set solver options.

Set module options.

See also:

Lurupa::set_module_options (p. 62)

References `clu_init()`, and `Lurupa::set_module_options()`.

3.2.2.19 BOUND_STATUS `clu_upper_bound` (PLP *lp*, double * *bound*, int * *iterations*)

Compute upper bound.

Compute the upper bound.

See also:

Lurupa::upper_bound (p. 63)

References `clu_init()`, and `Lurupa::upper_bound()`.

3.3 Condition.cpp File Reference

Implementation of **Condition** (p. 20).

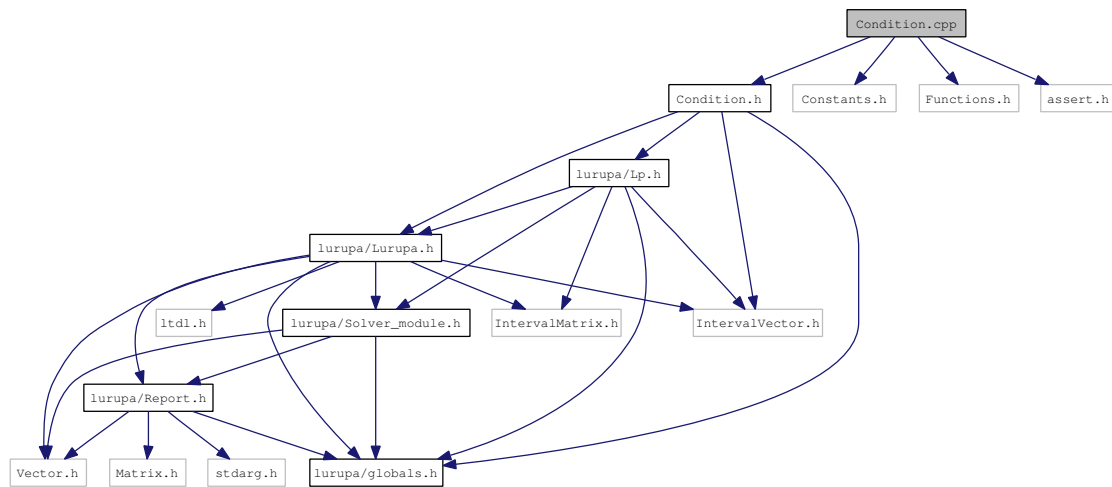
```
#include <Condition.h>
```

```
#include <Constants.h>
```

```
#include <Functions.h>
```

```
#include <assert.h>
```

Include dependency graph for Condition.cpp:



3.3.1 Detailed Description

Implementation of **Condition** (p. 20).

This file contains the implementation of **Condition** (p. 20).

Author:

Christian Keil

Id

Condition.cpp (p. 142) 533 2008-06-12 20:08:12Z keil

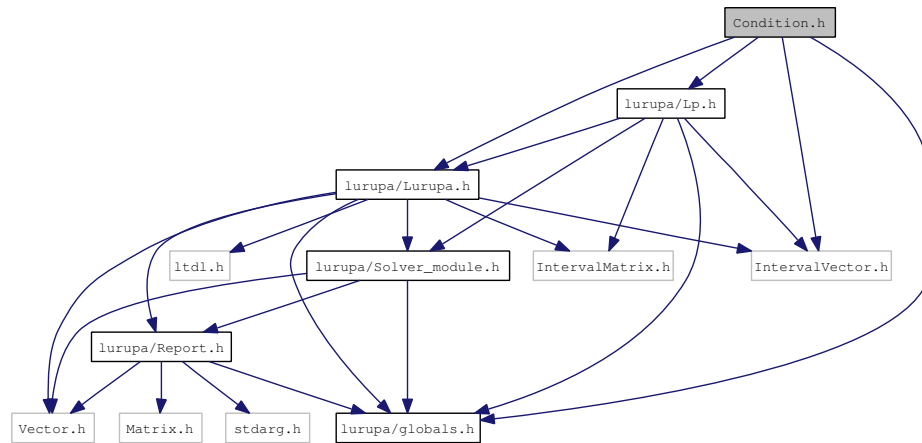
Name

3.4 Condition.h File Reference

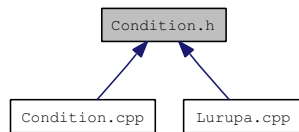
Definition of **Condition** (p. 20).

```
#include <lurupa/Lurupa.h>
#include <lurupa/Lp.h>
#include <lurupa/globals.h>
#include <IntervalVector.h>
```

Include dependency graph for Condition.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **Condition**
Condition numbers

3.4.1 Detailed Description

Definition of **Condition** (p. 20).

This file contains the definition of **Condition** (p. 20).

Author:

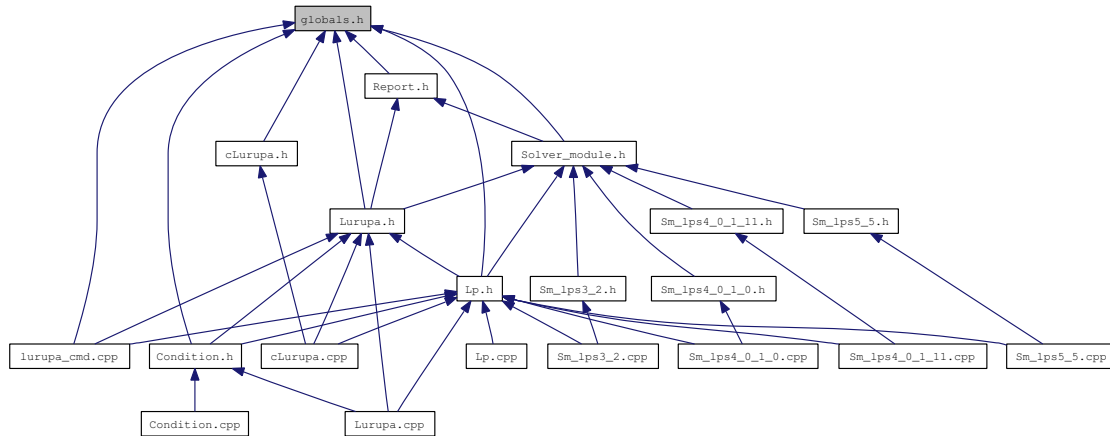
Christian Keil

Id**Condition.h** (p. 143) 533 2008-06-12 20:08:12Z keil**Name**

3.5 globals.h File Reference

Global properties.

This graph shows which files directly or indirectly include this file:



Bound status

- enum **Bound_status** {
 bs_verified, bs_priminf, bs_dualinf, bs_orginf,
 bs_orgunb, bs_pertinf, bs_noenc, bs_warn,
 bs_rank, bs_iter, bs_failure, bs_timeout,
 bs_running }
Bound status
- static const char * **bound_status_string** []
Bound status strings

Lp-solver status

- enum **Solver_status** {
 ss_feasible, ss_unbounded, ss_infeasible, ss_failure,
 ss_timeout, ss_unknown }
Lp-solver status.
- static const char * **solver_status_string** []
Lp-solver status strings.

Defines

Tables

The kind of tables that are written to disk by the command line client.

- `#define LU__T__CSV 2`
CSV table.
- `#define LU__T__LATEX 1`
LaTeX table.

Enumerations

- `enum Csv__style { octave, matlab }`
CSV style.
- `enum Write__mps { wm__all, wm__ask, wm__none }`
*Write *mps* as *MPS* file.*

3.5.1 Detailed Description

Global properties.

This file contains global properties shared by all components of Lurupa. It contains possible return values and parameters of methods as enums or defines for enhanced readability as well as string arrays with textual descriptions of some of these for displaying.

Author:

Christian Keil

Id

`globals.h` (p. 145) 533 2008-06-12 20:08:12Z keil

3.5.2 Enumeration Type Documentation

3.5.2.1 `enum Bound__status`

Bound status

The possible status values of a bound computation.

Enumerator:

- `bs__verified` Bound successfully computed
- `bs__priminf` Bound successfully computed; problem is verified to be infeasible
- `bs__dualinf` Bound successfully computed; problem is verified to be unbounded
- `bs__orginf` Bound not computed; the lp-solver claims infeasibility
- `bs__orgunb` Bound not computed; the lp-solver claims unboundedness
- `bs__pertinf` Bound not computed; a perturbed problem seems to be infeasible and inflation is not set
- `bs__noenc` Bound not computed; no enclosure of the solution set of linear system could be found

- bs_warn* Bound not computed; the lp-solver gave a warning while solving a perturbed problem
- bs_rank* Bound not computed; the equation matrix has not full rank
- bs_iter* Bound not computed; the maximum number of iterations is exceeded
- bs_failure* Bound not computed; an unexpected error occurred
- bs_timeout* Bound not computed; the lp-solver timed out solving a perturbed problem
- bs_running* Bound computation in progress

3.5.2.2 enum Csv_style

CSV style.

The style which is used to save csv files in **Report::write_vector** (p. 66) and **Report::write_matrix** (p. 66)

Enumerator:

- octave* Octave style; one file (*problem_name.csv*) with dimension and name as meta information
- matlab* Matlab style; all variables in different files without dimension in one directory (*problem_name*)

3.5.2.3 enum Solver_status

Lp-solver status.

The possible status values returned by lp-solvers.

Enumerator:

- ss_feasible* Lp-solver judges model to be feasible
- ss_unbounded* Lp-solver judges model to be unbounded
- ss_infeasible* Lp-solver judges model to be infeasible
- ss_failure* Lp-solver returned from solving with a failure
- ss_timeout* Lp-solver timed out solving
- ss_unknown* Lp-solver status not yet known

3.5.2.4 enum Write_mps

Write lps as MPS file.

Which intermediate lps are written as MPS files to disk.

Enumerator:

- wm_all* All intermediate lps.
- wm_ask* Ask for all intermediate lps.
- wm_none* No intermediate lp.

3.5.3 Variable Documentation

3.5.3.1 `const char* bound_status_string[]` [static]

Initial value:

```
{
    "ok",
    "primal infeasible",
    "dual infeasible",
    "lp-solver infeasible",
    "lp-solver unbounded",
    "perturbed lp infeasible",
    "no ls enclosure",
    "solver warning",
    "rank deficiency",
    "iteration limit reached",
    "solver failure",
    "solver timeout",
    "running"
}
```

Bound status strings

Status string array converting `Bound_status` values to real text for displaying.

Referenced by `print_bound_stats()`, `report_dual()`, `report_primal()`, `write_csv_table()`, and `write_latex_table()`.

3.5.3.2 `const char* solver_status_string[]` [static]

Initial value:

```
{
    "feasible",
    "unbounded",
    "infeasible",
    "failure",
    "timeout",
    "unknown"
}
```

Lp-solver status strings.

Status string array converting `Solver_status` values to real text for displaying.

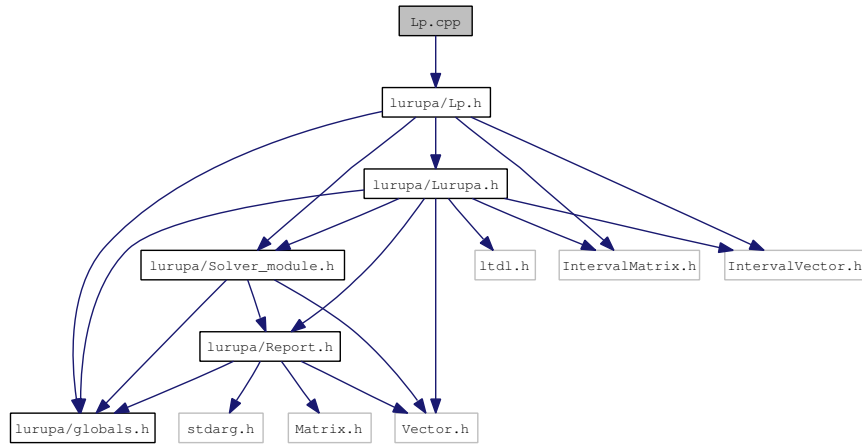
Referenced by `process_solving_status()`, `write_csv_table()`, and `write_latex_table()`.

3.6 Lp.cpp File Reference

Definition of Lp storage class.

```
#include <lurupa/Lp.h>
```

Include dependency graph for Lp.cpp:



3.6.1 Detailed Description

Definition of Lp storage class.

This storage class aggregates all the model data and the lp-solver's judgement of its feasibility.

Author:

Christian Keil

Id

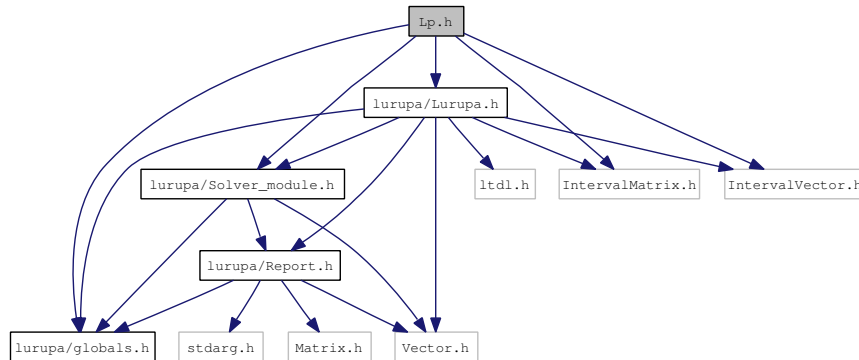
Lp.cpp (p. 149) 530 2008-06-12 08:57:52Z keil

3.7 Lp.h File Reference

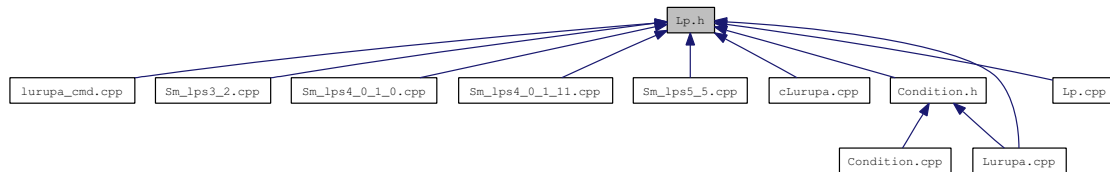
Declaration of Lp storage class.

```
#include <lurupa/globals.h>
#include <lurupa/Lurupa.h>
#include <lurupa/Solver_module.h>
#include <IntervalMatrix.h>
#include <IntervalVector.h>
```

Include dependency graph for Lp.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **Lp**
A linear program.
- struct **Primal_deflation**
Primal deflation parameters.

3.7.1 Detailed Description

Declaration of Lp storage class.

This storage class aggregates all the model data and the lp-solver's judgement of its feasibility.

Author:

Christian Keil

Id

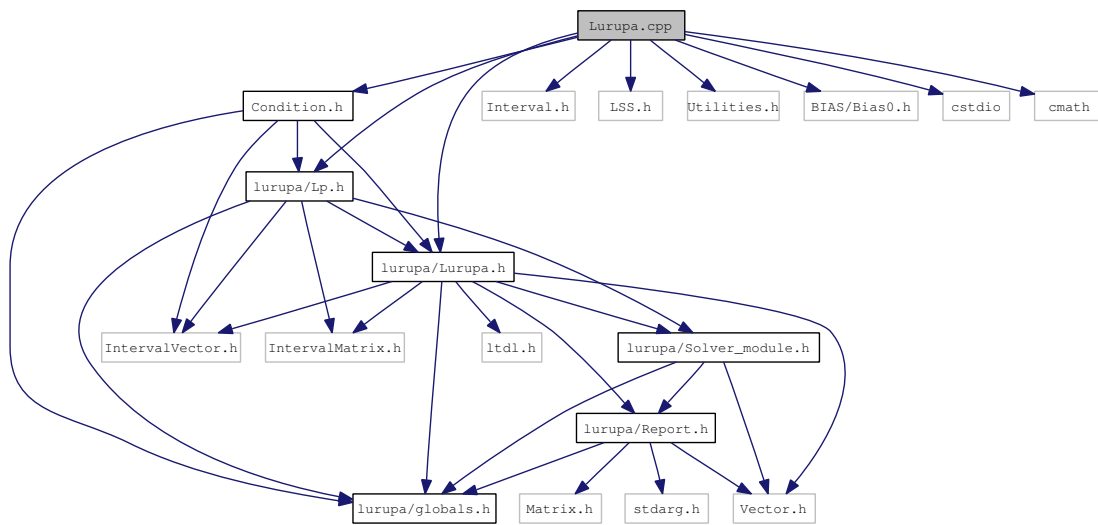
Lp.h (p. 150) 533 2008-06-12 20:08:12Z keil

3.8 Lurupa.cpp File Reference

Definition of Lurupa's core.

```
#include <lurupa/Lurupa.h>
#include <lurupa/Lp.h>
#include "Condition.h"
#include <Interval.h>
#include <LSS.h>
#include <Utilities.h>
#include <BIAS/Bias0.h>
#include <cstdio>
#include <cmath>
```

Include dependency graph for Lurupa.cpp:



Variables

- static const char * **version** = "Lurupa core 1.1"
Version string.

3.8.1 Detailed Description

Definition of Lurupa's core.

This file contains the definition of **Lurupa** (p. 36)'s core.

Author:

Christian Keil

Id

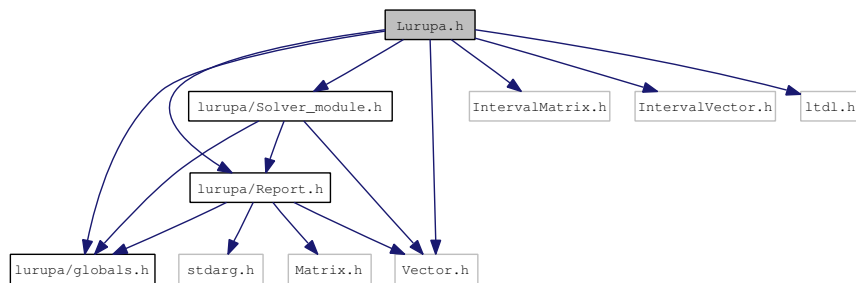
Lurupa.cpp.in 530 2008-06-12 08:57:52Z keil

3.9 Lurupa.h File Reference

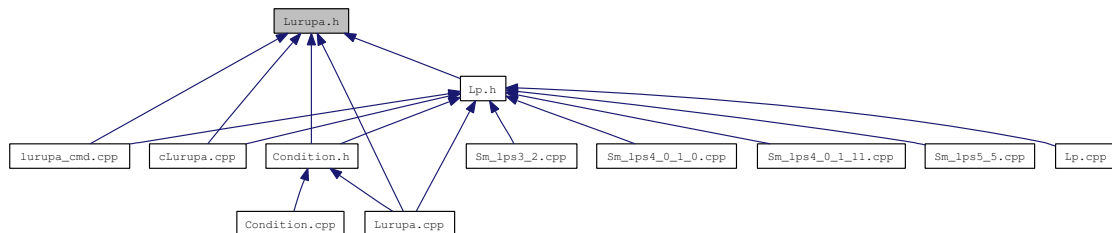
Declaration of Lurupa's core.

```
#include <lurupa/Solver_module.h>
#include <lurupa/Report.h>
#include <lurupa/globals.h>
#include <IntervalMatrix.h>
#include <IntervalVector.h>
#include <Vector.h>
#include <ltdl.h>
```

Include dependency graph for Lurupa.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **Lurupa**
Lurupa's core.

3.9.1 Detailed Description

Declaration of Lurupa's core.

This file contains the declaration of **Lurupa** (p.36).

Author:

Christian Keil

Id

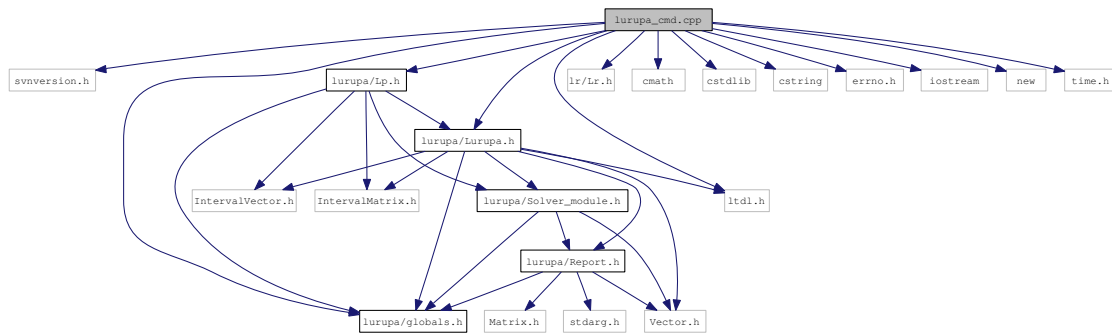
Lurupa.h (p. 154) 533 2008-06-12 20:08:12Z keil

3.10 lurupa_cmd.cpp File Reference

Lurupa's command line client.

```
#include "svnversion.h"
#include <lurupa/globals.h>
#include <lurupa/Lurupa.h>
#include <lurupa/Lp.h>
#include <lr/Lr.h>
#include <ltdl.h>
#include <cmath>
#include <cstdlib>
#include <cstring>
#include <errno.h>
#include <iostream>
#include <new>
#include <time.h>
```

Include dependency graph for lurupa_cmd.cpp:



Classes

- struct **Bound**
Information about a Bound.
- struct **Certificate**
A certificate.
- struct **Enclosure**
An enclosure.
- struct **Lp_stats**
Information about an Lp.

Functions

- void **compute_dual** (**Lurupa** *lurupa, **Lp** *lp, **Lp_stats** *lp_stats)
Compute dual certificate.
- void **compute_lower** (**Lurupa** *lurupa, **Lp** *lp, **Lp_stats** *lp_stats)
Compute lower bound.
- void **compute_primal** (**Lurupa** *lurupa, **Lp** *lp, **Lp_stats** *lp_stats)
Compute primal certificate.
- void **compute_upper** (**Lurupa** *lurupa, **Lp** *lp, **Lp_stats** *lp_stats)
Compute upper bound.
- int **main** (int argc, char *argv[])
Command line main.
- void **out_of_memory** ()
Out of memory handler.
- void **print_bound_stats** (const **Bound** &bound, const **Lp_stats** &lp_stats)
Print bound statistics.
- void **print_brief_version** (const **Lurupa** &lurupa, bool solver_module_set)
Print brief version information.
- void **print_model_data** (const **Lp_stats** &lp_stats)
Print model data.
- void **print_usage** (char *self, const **Lurupa** &lurupa, bool solver_module_set)
Print usage information.
- void **print_version** (const **Lurupa** &lurupa, bool solver_module_set)
Print version information.
- void **process_solving_status** (**Lp_stats** &lp_stats)
Process solving statistics.
- void **report_bound_quality** (**Lp_stats** &lp_stats)
Compute bound quality.
- void **report_dual** (**Lp_stats** &lp_stats)
***Report** (p. 65) dual certificate.*
- void **report_lower** (**Lp** *lp, **Lp_stats** &lp_stats)
***Report** (p. 65) lower bound.*
- void **report_primal** (**Lp_stats** &lp_stats)
***Report** (p. 65) primal certificate.*

- void **report_upper** (**Lp** *lp, **Lp_stats** &lp_stats)
Report (p. 65) upper bound.
- const char * **rounded_string** (double x, int direction)
Correctly round double to string.
- void **start_timer** (clock_t &pt, int &rt)
Start timer.
- void **stop_timer** (clock_t &pt, int &rt)
Stop timer.
- const char * **timer_diff** (clock_t start_pt, clock_t end_pt, int start_rt, int end_rt, double &pt_diff, double &rt_diff)
Compute timer difference.
- void **write_csv_table** (const **Lp_stats** &lp_stats, FILE *csv)
Write csv table.
- void **write_latex_table** (const **Lp_stats** &lp_stats, FILE *latex)
Write LaTeX table.
- void **write_tables** (short tables, const **Lp_stats** &lp_stats, FILE *latex, FILE *csv)
Write tables.

Variables

- static const char * **copyright** = "Copyright (C) 2006 Christian Keil"
Copyright string.
- static const char * **version** = "Lurupa command line client 1.2"
Version string.

3.10.1 Detailed Description

Lurupa's command line client.

This is Lurupa's command line client. It uses the core logic and a user specified solver module to compute verified bounds for a linear program. The results and the time needed to compute them are displayed and written to disk in csv or LaTeX format.

Author:

Christian Keil

Id

lurupa_cmd.cpp (p. 156) 533 2008-06-12 20:08:12Z keil

3.10.2 Function Documentation

3.10.2.1 void compute_dual (Lurupa * *lurupa*, Lp * *lp*, Lp_stats * *lp_stats*)

Compute dual certificate.

Compute and time a dual certificate for primal infeasibility *lp*.

Parameters:

- ← ***lurupa*** **Lurupa** (p. 36) reference used to compute certificate
- ← ***lp*** the lp
- ***lp_stats*** stores certificate status and time

References Lp_stats::dual, Lurupa::dual_certificate(), Certificate::proc_time, Certificate::realtime, start_timer(), Certificate::status, stop_timer(), Certificate::time_status, and timer_diff().

Referenced by main().

3.10.2.2 void compute_lower (Lurupa * *lurupa*, Lp * *lp*, Lp_stats * *lp_stats*)

Compute lower bound.

Compute and time a lower bound for the optimal value of *lp*.

Parameters:

- ← ***lurupa*** **Lurupa** (p. 36) reference used to compute bound
- ← ***lp*** the lp
- ***lp_stats*** stores bound and time

References Bound::iterations, Lp_stats::lower, Lurupa::lower_bound(), Bound::proc_time, Bound::realtime, start_timer(), Bound::status, stop_timer(), Bound::time_status, timer_diff(), and Bound::value.

Referenced by main().

3.10.2.3 void compute_primal (Lurupa * *lurupa*, Lp * *lp*, Lp_stats * *lp_stats*)

Compute primal certificate.

Compute and time a primal certificate for dual infeasibility of *lp*.

Parameters:

- ← ***lurupa*** **Lurupa** (p. 36) reference used to compute certificate
- ← ***lp*** the lp
- ***lp_stats*** stores certificate status and time

References Lp_stats::primal, Lurupa::primal_certificate(), Certificate::proc_time, Certificate::realtime, start_timer(), Certificate::status, stop_timer(), Certificate::time_status, and timer_diff().

Referenced by main().

3.10.2.4 void compute_upper (Lurupa * *lurupa*, Lp * *lp*, Lp_stats * *lp_stats*)

Compute upper bound.

Compute and time an upper bound for the optimal value of *lp*.

Parameters:

- ← *lurupa* **Lurupa** (p. 36) reference used to compute bound
- ← *lp* the *lp*
- *lp_stats* stores bound and time

References Bound::iterations, Bound::proc_time, Bound::realtime, start_timer(), Bound::status, stop_timer(), Bound::time_status, timer_diff(), Lp_stats::upper, Lurupa::upper_bound(), and Bound::value.

Referenced by main().

3.10.2.5 int main (int *argc*, char * *argv*[])

Command line main.

Parse the command line arguments and call **Lurupa** (p. 36) appropriately. Print the results and write them to disk.

References Lp_stats::automatic, Enclosure::compute, Certificate::compute, Bound::compute, compute_dual(), compute_lower(), compute_primal(), compute_upper(), Lurupa::cond(), Lp_stats::condition, Lp_stats::dual, Lp_stats::is_maximize, l, Enclosure::lower, Lp_stats::lower, LU_T_CSV, LU_T_LATEX, matlab, Lp::maximize, Lp::name, Lp_stats::name, octave, Lp_stats::optimal_value, out_of_memory(), Lp_stats::path, Lp_stats::primal, print_brief_version(), print_model_data(), print_usage(), print_version(), process_solving_status(), Lurupa::read_lp(), Lurupa::report, report_bound_quality(), report_dual(), report_lower(), report_primal(), report_upper(), Lurupa::rho_d(), Lp_stats::rho_d, Lurupa::rho_p(), Lp_stats::rho_p, rounded_string(), Lurupa::set_alpha(), Report::set_csv_style(), Lurupa::set_eta(), Lurupa::set_inflate(), Lurupa::set_module(), Lurupa::set_module_options(), Report::set_verbosity(), Lurupa::solve_lp(), Lp_stats::solve_proc_time, Lp_stats::solve_realtime, Lp_stats::solve_time_status, ss_failure, ss_feasible, ss_infeasible, ss_unbounded, start_timer(), Lp_stats::status, stop_timer(), timer_diff(), Lp_stats::total_proc_time, Lp_stats::total_realtime, Lp_stats::total_time_status, Enclosure::upper, Lp_stats::upper, and write_tables().

3.10.2.6 void print_bound_stats (const Bound & *bound*, const Lp_stats & *lp_stats*)

Print bound statistics.

Print statistics about a computed bound. These include of course the bound itself along with the count of algorithm iterations needed, the time needed, the ratio of the time needed to compute the bounds and the time needed to compute the approximate optimal value, the status of the computed bound (e.g. bound successfully computed, no bound computed because of rank deficiency).

Parameters:

- ← *bound* the bound's statistics
- ← *lp_stats* contains the time to solve the model

References `bound_status_string`, `Bound::iterations`, `Bound::proc_time`, `Bound::realtime`, `Lp_stats::solve_proc_time`, `Lp_stats::solve_realtime`, `Bound::status`, and `Bound::time_status`.

Referenced by `report_lower()`, and `report_upper()`.

3.10.2.7 void print_brief_version (const Lurupa & *lurupa*, bool *solver_module_set*)

Print brief version information.

Print brief version information suitable for log output.

Parameters:

← *lurupa* **Lurupa** (p. 36) instance to print information about

← *solver_module_set* whether a solver module is loaded

References `copyright`, `Lurupa::print_core_brief_version()`, `Lurupa::print_module_brief_version()`, and `version`.

Referenced by `main()`.

3.10.2.8 void print_model_data (const Lp_stats & *lp_stats*)

Print model data.

Print the name and the optimization direction of the model to be processed.

Parameters:

← *lp_stats* contains the model's data

References `Lp_stats::is_maximize`, and `Lp_stats::name`.

Referenced by `main()`.

3.10.2.9 void print_usage (char * *self*, const Lurupa & *lurupa*, bool *solver_module_set*)

Print usage information.

Print usage information for the command line client; which parameters with which values are supported.

Parameters:

← *self* path to running program

← *lurupa* **Lurupa** (p. 36) instance to access solver module

← *solver_module_set* whether a solver module is loaded

References `Lurupa::print_module_options()`, and `version`.

Referenced by `main()`.

3.10.2.10 void print_version (const Lurupa & *lurupa*, bool *solver_module_set*)

Print version information.

Print version information for the application parts.

Parameters:

- ← *lurupa* **Lurupa** (p. 36) instance to print information about
- ← *solver_module_set* whether a solver module is loaded

References `copyright`, `Lurupa::print_core_version()`, `Lurupa::print_module_version()`, and `version`.

Referenced by `main()`.

3.10.2.11 void process_solving_status (Lp_stats & *lp_stats*)

Process solving statistics.

Print statistics regarding the solving of the lp like the approximate optimal value, the time needed to compute it, the status of the lp-solver (e.g., the solver's judgement of the model's feasibility or timeout while solving the model). Set `lp.lower.compute` and `lp.upper.compute` to false if the lp-solver did not judge the model to be either feasible, infeasible, or unbounded, that is returned with any kind of error.

Parameters:

- ↔ *lp_stats* contains the solving statistics

References `Bound::compute`, `Lp_stats::lower`, `Lp_stats::optimal_value`, `Lp_stats::solve_proc_time`, `Lp_stats::solve_realtime`, `Lp_stats::solve_time_status`, `solver_status_string`, `ss_feasible`, `ss_infeasible`, `ss_unbounded`, `Lp_stats::status`, and `Lp_stats::upper`.

Referenced by `main()`.

3.10.2.12 void report_bound_quality (Lp_stats & *lp_stats*)

Compute bound quality.

Compute and print the accuracy of the verified bounds. It is computed by

$$\mu(a, b) = \frac{|a - b|}{\max\{1, |a + b|/2\}},$$

where a and b are the computed bounds or the approximate optimal value if one of the bounds could not be computed. Note that μ is computed from the bound values as they are computed and not from the printed ones. The printed values of the bounds are rounded to display a rigorous enclosure of the optimal value.

Parameters:

- ↔ *lp_stats* contains the verified bounds

References `Lp_stats::bound_quality`, `Bound::compute`, `Lp_stats::lower`, `Lp_stats::optimal_value`, `Lp_stats::upper`, and `Bound::value`.

Referenced by `main()`.

3.10.2.13 void report_dual (Lp_stats & lp_stats)

Report (p. 65) dual certificate.

Report (p. 65) success and time of computing a dual certificate for an lp.

Parameters:

← *lp_stats* contains the certificate to report

References bound_status_string, Lp_stats::dual, Certificate::proc_time, Certificate::realtime, Certificate::status, and Certificate::time_status.

Referenced by main().

3.10.2.14 void report_lower (Lp * lp, Lp_stats & lp_stats)

Report (p. 65) lower bound.

Report (p. 65) value and stats of the lower bound on the optimal value of lp.

Parameters:

← *lp* the lp

← *lp_stats* the stats

References Lp_stats::lower, Lp_stats::optimal_value, print_bound_stats(), rounded_string(), ss_feasible, Lp_stats::status, and Bound::value.

Referenced by main().

3.10.2.15 void report_primal (Lp_stats & lp_stats)

Report (p. 65) primal certificate.

Report (p. 65) success and time of computing a primal certificate for an lp.

Parameters:

← *lp_stats* contains the certificate to report

References bound_status_string, Lp_stats::primal, Certificate::proc_time, Certificate::realtime, Certificate::status, and Certificate::time_status.

Referenced by main().

3.10.2.16 void report_upper (Lp * lp, Lp_stats & lp_stats)

Report (p. 65) upper bound.

Report (p. 65) value and stats of the upper bound on the optimal value of lp.

Parameters:

← *lp* the lp

← *lp_stats* the stats

References `Lp_stats::optimal_value`, `print_bound_stats()`, `rounded_string()`, `ss_feasible`, `Lp_stats::status`, `Lp_stats::upper`, and `Bound::value`.

Referenced by `main()`.

3.10.2.17 `const char* rounded_string (double x, int direction)`

Correctly round double to string.

Generate a correctly rounded string representation of `x`. The direction of rounding can be selected via `direction`. Possible values are `LR_RND_DOWN`, `LR_RND_UP`, `LR_RND_NEAR`, `LR_RND_CHOP`.

Parameters:

- ← `x` the double value to convert
- ← `direction` direction of rounding

Returns:

rounded string representation

Referenced by `main()`, `report_lower()`, and `report_upper()`.

3.10.2.18 `void start_timer (clock_t & pt, int & rt)`

Start timer.

Get realtime and process time counter values. These can be compared to the ones returned by `stop_timer` (p. 164) to compute the elapsed time.

Parameters:

- `pt` process time counter value
- `rt` realtime counter value

See also:

`stop_timer` (p. 164)

Referenced by `compute_dual()`, `compute_lower()`, `compute_primal()`, `compute_upper()`, and `main()`.

3.10.2.19 `void stop_timer (clock_t & pt, int & rt)`

Stop timer.

Get realtime and process time counter values. These can be compared to the ones returned by `start_timer` (p. 164) to compute the elapsed time.

Parameters:

- `pt` process time counter value
- `rt` realtime counter value

See also:

`start_timer` (p. 164)

Referenced by `compute_dual()`, `compute_lower()`, `compute_primal()`, `compute_upper()`, and `main()`.

3.10.2.20 `const char* timer_diff (clock_t start_pt, clock_t end_pt, int start_rt, int end_rt, double & pt_diff, double & rt_diff)`

Compute timer difference.

Compute the elapsed realtime and process time verifying that no wrap around in the process time counter occurred. If a wrap around occurred or cannot be ruled out or the calls to the time routines failed, return a short string explaining what happened.

Parameters:

- ← *start_pt* process time counter value at start
- ← *end_pt* process time counter value at end
- ← *start_rt* realtime counter value at start
- ← *end_rt* realtime counter value at end
- *pt_diff* elapsed process time in seconds
- *rt_diff* elapsed realtime in seconds

Returns:

time status string

Referenced by `compute_dual()`, `compute_lower()`, `compute_primal()`, `compute_upper()`, and `main()`.

3.10.2.21 `void write_csv_table (const Lp_stats & lp_stats, FILE * csv)`

Write csv table.

Write a line containing the gathered results with maximal precision to disk in csv format. This includes

- name of the model
- status of the solver solving the original model
- approximate optimal value
- time needed to compute the approximate value
- verified lower bound's data
 - bound status
 - bound value
 - number of iterations to compute the bound
 - time needed to compute the bound

- verified upper bound's data
 - bound status
 - bound value
 - number of iterations to compute the bound
 - time needed to compute the bound
- relative accuracy of the bounds

Parameters:

- ← *lp_stats* contains the model's and bound's statistics
- ← *csv* file to be written to

References Lp_stats::bound_quality, bound_status_string, Bound::compute, Bound::iterations, Lp_stats::lower, Lp_stats::name, Lp_stats::optimal_value, Bound::proc_time, Bound::realtime, Lp_stats::solve_proc_time, Lp_stats::solve_realtime, solver_status_string, Bound::status, Lp_stats::status, Lp_stats::upper, and Bound::value.

Referenced by write_tables().

3.10.2.22 void write_latex_table (const Lp_stats & lp_stats, FILE * latex)

Write LaTeX table.

Write a line containing the gathered results to disk in LaTeX table format with precision tailored to viewing. This consists of

- name of the model
- approximate optimal solution or the solver status if solving failed
- time needed to solve the model
- verified lower bound's data
 - bound value or the bound status if its computation failed
 - number of algorithm iterations to compute the bound
 - ratio of the time needed to compute the bound and the time needed to solve the model
- verified upper bound's data
 - bound value or the bound status if its computation failed
 - number of algorithm iterations to compute the bound
 - ratio of the time needed to compute the bound and the time needed to solve the model
- relative accuracy of the bounds

Parameters:

- ← *lp_stats* contains the model's and bound's statistics
- ← *latex* file to be written to

References Lp_stats::bound_quality, bound_status_string, bs_verified, Bound::compute, Bound::iterations, Lp_stats::lower, Lp_stats::name, Lp_stats::optimal_value, Bound::proc_time, Bound::realtime, Lp_stats::solve_proc_time, Lp_stats::solve_realtime, solver_status_string, ss_feasible, Bound::status, Lp_stats::status, Lp_stats::upper, and Bound::value.

Referenced by write_tables().

3.10.2.23 void write_tables (short *tables*, const Lp_stats & *lp_stats*, FILE * *latex*, FILE * *csv*)

Write tables.

Calls the routines to write LaTeX and csv tables as specified by tables.

Parameters:

- ← *tables* specifies the tables to be written (c.f. **globals.h** (p. 145))
- ← *lp_stats* contains the model's and bound's statistics
- ← *latex* file to write LaTeX table to
- ← *csv* file to write csv table to

References LU_T_CSV, LU_T_LATEX, write_csv_table(), and write_latex_table().

Referenced by main().

3.11 Report.h File Reference

Declaration of Report class.

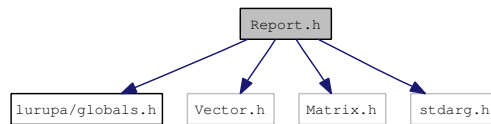
```
#include <lurupa/globals.h>
```

```
#include <Vector.h>
```

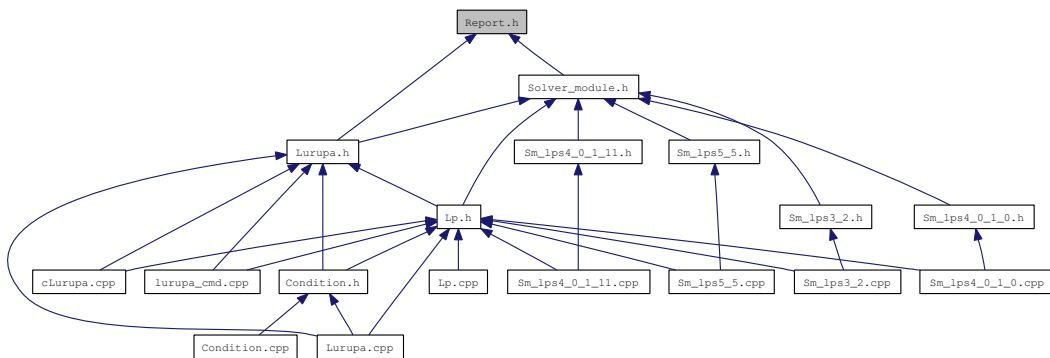
```
#include <Matrix.h>
```

```
#include <stdarg.h>
```

Include dependency graph for Report.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **Report**

Functions for reporting and debugging.

3.11.1 Detailed Description

Declaration of Report class.

This file contains the declaration of the **Report** (p. 65) class.

Author:

Christian Keil

Id

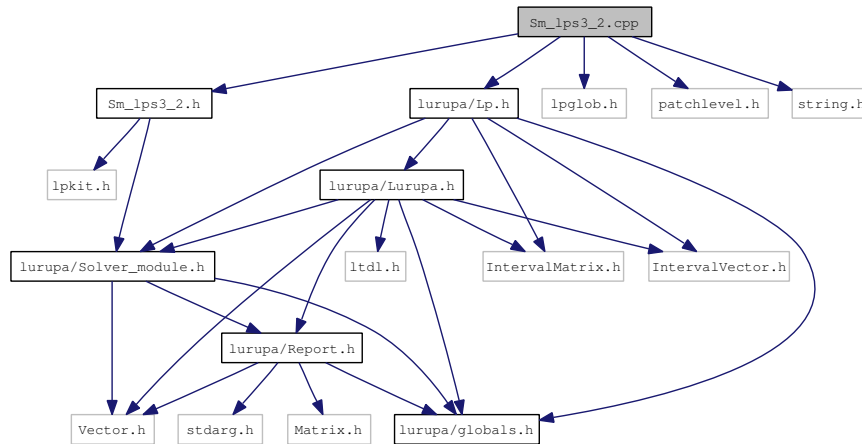
Report.h (p. 168) 530 2008-06-12 08:57:52Z keil

3.12 Sm_lps3_2.cpp File Reference

Definition of lp_solve 3.2 solver module.

```
#include "Sm_lps3_2.h"
#include <lurupa/Lp.h>
#include <lpglob.h>
#include <patchlevel.h>
#include <string.h>
```

Include dependency graph for Sm_lps3_2.cpp:



Functions

- void **get_func_pointers** (Solver_module_interface &i)
Get function pointers.

Variables

- static const char * **version** = "Lp_solve 3.2 module 1.0"
Version string.

3.12.1 Detailed Description

Definition of lp_solve 3.2 solver module.

This file defines the solver module for lp_solve 3.2, **Sm_lps3_2** (p. 67).

Author:

Christian Keil

Id

Sm_lps3_2.cpp (p. 169) 533 2008-06-12 20:08:12Z keil

3.12.2 Function Documentation

3.12.2.1 void get_func_pointers (Solver_module_interface & *i*)

Get function pointers.

Set the pointers of *i* to point to the member functions of **Sm_lps3_2** (p. 67).

Parameters:

→ *i* receives the pointers to member functions

Referenced by Lurupa::set_module().

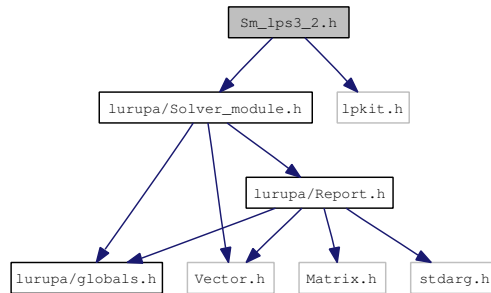
3.13 Sm_lps3_2.h File Reference

Declaration of lp_solve 3.2 solver module.

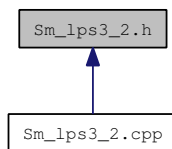
```
#include <lurupa/Solver_module.h>
```

```
#include <lpkit.h>
```

Include dependency graph for Sm_lps3_2.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **Lp_lp_solve**
Information about lp_solve's problem.
- class **Sm_lps3_2**
Solver module for lp_solve 3.2.

Functions

- void **get_func_pointers** (Solver_module_interface &i)
Get function pointers.

3.13.1 Detailed Description

Declaration of lp_solve 3.2 solver module.

This file declares the solver module for lp_solve 3.2, **Sm_lps3_2** (p. 67).

Author:

Christian Keil

Id

Sm_lps3_2.h (p. 171) 533 2008-06-12 20:08:12Z keil

3.13.2 Function Documentation

3.13.2.1 void get_func_pointers (Solver_module_interface & i)

Get function pointers.

Set the pointers of **i** to point to the member functions of **Sm_lps3_2** (p. 67).

Parameters:

→ **i** receives the pointers to member functions

Set the pointers of **i** to point to the member functions of **Sm_lps4_0_1_0** (p. 79).

Parameters:

→ **i** receives the pointers to member functions

Set the pointers of **i** to point to the member functions of **Sm_lps4_0_1_11** (p. 93).

Parameters:

→ **i** receives the pointers to member functions

Set the pointers of **i** to point to the member functions of **Sm_lps5_5** (p. 107).

Parameters:

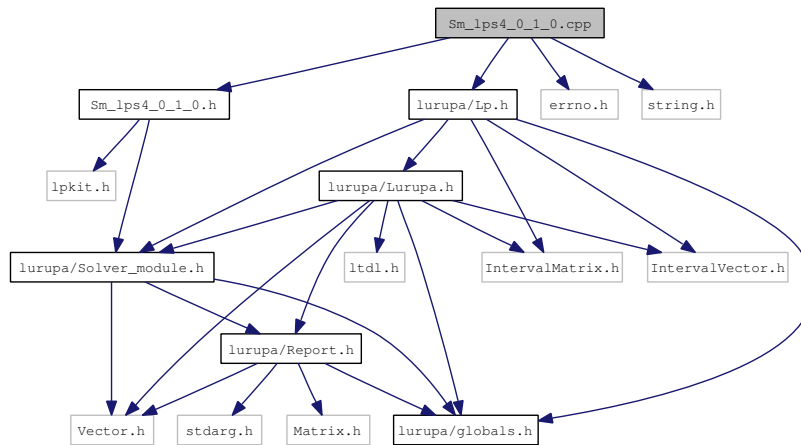
→ **i** receives the pointers to member functions

3.14 Sm_lps4_0_1_0.cpp File Reference

Definition of lp_solve 4.0.1.0 solver module.

```
#include "Sm_lps4_0_1_0.h"
#include <lurupa/Lp.h>
#include <errno.h>
#include <string.h>
```

Include dependency graph for Sm_lps4_0_1_0.cpp:



Functions

- void **get_func_pointers** (Solver_module_interface &i)
Get function pointers.

Variables

- static const char * **version** = "Lp_solve 4.0.1.0 module 1.0"
Version string.

3.14.1 Detailed Description

Definition of lp_solve 4.0.1.0 solver module.

This file defines the solver module for lp_solve 4.0.1.0, **Sm_lps4_0_1_0** (p. 79).

Author:

Christian Keil

Id

Sm_lps4_0_1_0.cpp (p. 173) 542 2008-06-26 12:44:51Z keil

3.14.2 Function Documentation

3.14.2.1 void get_func_pointers (Solver_module_interface & i)

Get function pointers.

Set the pointers of *i* to point to the member functions of **Sm_lps4_0_1_0** (p. 79).

Parameters:

→ *i* receives the pointers to member functions

References Sm_lps4_0_1_0::get_accuracy(), Solver_module_interface::get_accuracy,
 Solver_module_interface::get_dual_ray, Solver_module_interface::get_primal_ray, Sm_lps4_0_1_0::get_version(), Solver_module_interface::get_version, Sm_lps4_0_1_0::init(), Solver_module_interface::init, Sm_lps4_0_1_0::print_brief_version(), Solver_module_interface::print_brief_version, Sm_lps4_0_1_0::print_options(), Solver_module_interface::print_options, Sm_lps4_0_1_0::print_version(), Solver_module_interface::print_version, Sm_lps4_0_1_0::read_lp(), Solver_module_interface::read_lp, Sm_lps4_0_1_0::restore_dual(), Solver_module_interface::restore_dual, Solver_module_interface::restore_dual_phase1, Sm_lps4_0_1_0::restore_primal(), Solver_module_interface::restore_primal, Solver_module_interface::restore_primal_phase1, Solver_module_interface::set_dual_phase1, Sm_lps4_0_1_0::set_lp(), Solver_module_interface::set_lp, Sm_lps4_0_1_0::set_module_options(), Solver_module_interface::set_module_options, Solver_module_interface::set_primal_phase1, Sm_lps4_0_1_0::solve_dual_perturbed(), Solver_module_interface::solve_dual_perturbed, Sm_lps4_0_1_0::solve_original(), Solver_module_interface::solve_original, Sm_lps4_0_1_0::solve_primal_perturbed(), and Solver_module_interface::solve_primal_perturbed.

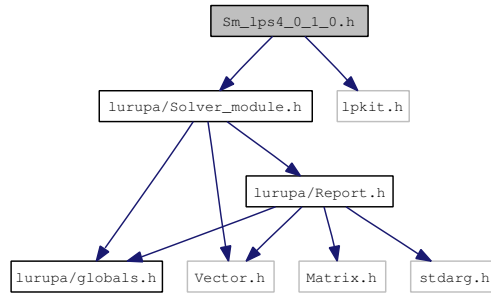
3.15 Sm_lps4_0_1_0.h File Reference

Declaration of lp_solve 4.0.1.0 solver module.

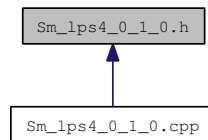
```
#include <lurupa/Solver_module.h>
```

```
#include <lpkit.h>
```

Include dependency graph for Sm_lps4_0_1_0.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **Lp_lp_solve**
Information about lp_solve's problem.
- class **Sm_lps4_0_1_0**
Solver module for lp_solve 4.0.1.0.

Functions

- void **get_func_pointers** (Solver_module_interface &i)
Get function pointers.

3.15.1 Detailed Description

Declaration of lp_solve 4.0.1.0 solver module.

This file declares the solver module for lp_solve 4.0.1.0, **Sm_lps4_0_1_0** (p. 79).

Author:

Christian Keil

Id

Sm_lps4_0_1_0.h (p. 175) 533 2008-06-12 20:08:12Z keil

3.15.2 Function Documentation

3.15.2.1 void get_func_pointers (Solver_module_interface & i)

Get function pointers.

Set the pointers of **i** to point to the member functions of **Sm_lps3_2** (p. 67).

Parameters:

→ **i** receives the pointers to member functions

Set the pointers of **i** to point to the member functions of **Sm_lps4_0_1_0** (p. 79).

Parameters:

→ **i** receives the pointers to member functions

Set the pointers of **i** to point to the member functions of **Sm_lps4_0_1_11** (p. 93).

Parameters:

→ **i** receives the pointers to member functions

Set the pointers of **i** to point to the member functions of **Sm_lps5_5** (p. 107).

Parameters:

→ **i** receives the pointers to member functions

3.16 Sm_lps4_0_1_11.cpp File Reference

Definition of lp_solve 4.0.1.11 solver module.

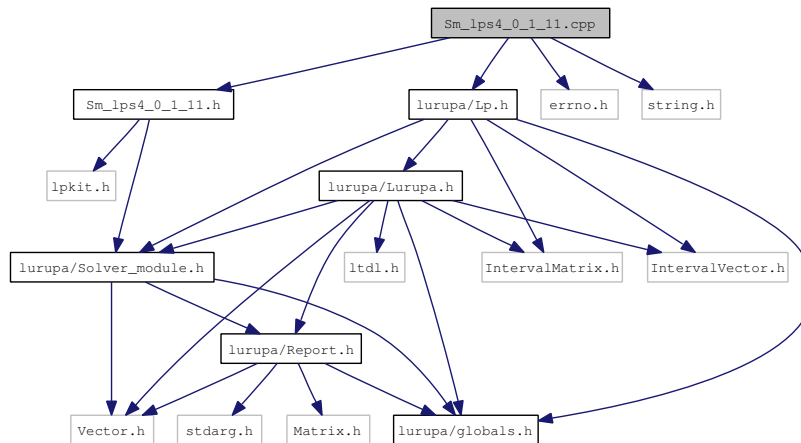
```
#include "Sm_lps4_0_1_11.h"
```

```
#include <lurupa/Lp.h>
```

```
#include <errno.h>
```

```
#include <string.h>
```

Include dependency graph for Sm_lps4_0_1_11.cpp:



Functions

- void **get_func_pointers** (Solver_module_interface &i)
Get function pointers.

Variables

- static const char * **version** = "Lp_solve 4.0.1.11 module 1.0"
Version string.

3.16.1 Detailed Description

Definition of lp_solve 4.0.1.11 solver module.

This file defines the solver module for lp_solve 4.0.1.11, **Sm_lps4_0_1_11** (p. 93).

Author:

Christian Keil

Id

Sm_lps4_0_1_11.cpp (p. 177) 542 2008-06-26 12:44:51Z keil

3.16.2 Function Documentation

3.16.2.1 void get_func_pointers (Solver_module_interface & i)

Get function pointers.

Set the pointers of *i* to point to the member functions of **Sm_lps4_0_1_11** (p.93).

Parameters:

→ *i* receives the pointers to member functions

References Sm_lps4_0_1_11::get_accuracy(), Solver_module_interface::get_accuracy, Solver_module_interface::get_dual_ray, Solver_module_interface::get_primal_ray, Sm_lps4_0_1_11::get_version(), Solver_module_interface::get_version, Sm_lps4_0_1_11::init(), Solver_module_interface::init, Sm_lps4_0_1_11::print_brief_version(), Solver_module_interface::print_brief_version, Sm_lps4_0_1_11::print_options(), Solver_module_interface::print_options, Sm_lps4_0_1_11::print_version(), Solver_module_interface::print_version, Sm_lps4_0_1_11::read_lp(), Solver_module_interface::read_lp, Sm_lps4_0_1_11::restore_dual(), Solver_module_interface::restore_dual, Solver_module_interface::restore_dual_phase1, Sm_lps4_0_1_11::restore_primal(), Solver_module_interface::restore_primal, Solver_module_interface::restore_primal_phase1, Solver_module_interface::set_dual_phase1, Sm_lps4_0_1_11::set_lp(), Solver_module_interface::set_lp, Sm_lps4_0_1_11::set_module_options(), Solver_module_interface::set_module_options, Solver_module_interface::set_primal_phase1, Sm_lps4_0_1_11::solve_dual_perturbed(), Solver_module_interface::solve_dual_perturbed, Sm_lps4_0_1_11::solve_original(), Solver_module_interface::solve_original, Sm_lps4_0_1_11::solve_primal_perturbed(), and Solver_module_interface::solve_primal_perturbed.

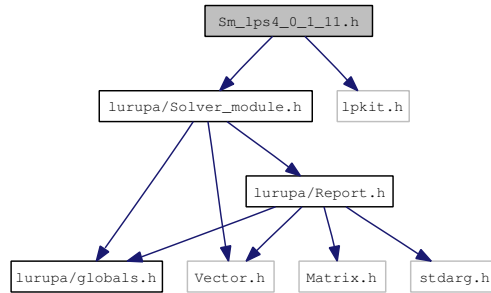
3.17 Sm_lps4_0_1_11.h File Reference

Declaration of lp_solve 4.0.1.11 solver module.

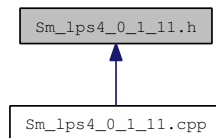
```
#include <lurupa/Solver_module.h>
```

```
#include <lpkit.h>
```

Include dependency graph for Sm_lps4_0_1_11.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **Lp_lp_solve**
Information about lp_solve's problem.
- class **Sm_lps4_0_1_11**
Solver module for lp_solve 4.0.1.11.

Functions

- void **get_func_pointers** (Solver_module_interface &i)
Get function pointers.

3.17.1 Detailed Description

Declaration of lp_solve 4.0.1.11 solver module.

This file declares the solver module for lp_solve 4.0.1.11, **Sm_lps4_0_1_11** (p. 93).

Author:

Christian Keil

Id

Sm_lps4_0_1_11.h (p. 179) 533 2008-06-12 20:08:12Z keil

3.17.2 Function Documentation**3.17.2.1 void get_func_pointers (Solver_module_interface & i)**

Get function pointers.

Set the pointers of **i** to point to the member functions of **Sm_lps3_2** (p. 67).

Parameters:

→ **i** receives the pointers to member functions

Set the pointers of **i** to point to the member functions of **Sm_lps4_0_1_0** (p. 79).

Parameters:

→ **i** receives the pointers to member functions

Set the pointers of **i** to point to the member functions of **Sm_lps4_0_1_11** (p. 93).

Parameters:

→ **i** receives the pointers to member functions

Set the pointers of **i** to point to the member functions of **Sm_lps5_5** (p. 107).

Parameters:

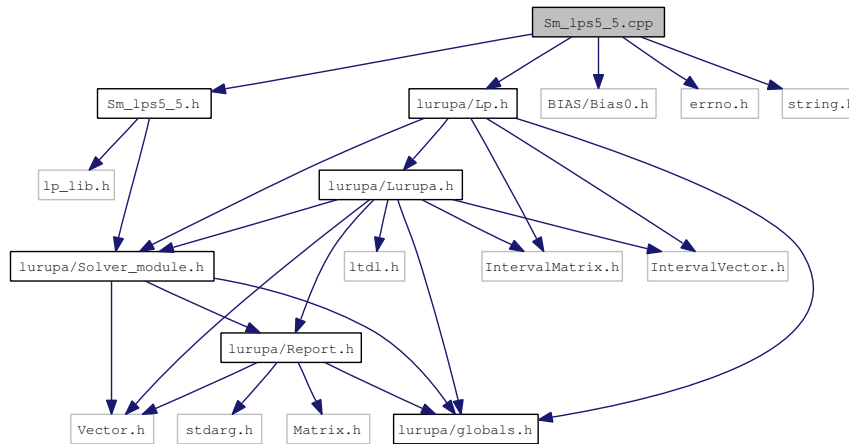
→ **i** receives the pointers to member functions

3.18 Sm_lps5_5.cpp File Reference

Definition of lp_solve 5.5 solver module.

```
#include "Sm_lps5_5.h"
#include <lurupa/Lp.h>
#include <BIAS/Bias0.h>
#include <errno.h>
#include <string.h>
```

Include dependency graph for Sm_lps5_5.cpp:



Functions

- void **get_func_pointers** (Solver_module_interface &i)
Get function pointers.

Variables

- static const char * **version** = "Lp_solve 5.5 module 1.1"
Version string.

3.18.1 Detailed Description

Definition of lp_solve 5.5 solver module.

This file defines the solver module for lp_solve 5.5, **Sm_lps5_5** (p. 107).

Author:

Christian Keil

Id

Sm_lps5_5.cpp (p. 181) 533 2008-06-12 20:08:12Z keil

3.18.2 Function Documentation**3.18.2.1 void get_func_pointers (Solver_module_interface & i)**

Get function pointers.

Set the pointers of **i** to point to the member functions of **Sm_lps5_5** (p. 107).

Parameters:

→ **i** receives the pointers to member functions

References Sm_lps5_5::get_accuracy(), Sm_lps3_2::get_accuracy(), Solver_module_interface::get_accuracy, Sm_lps5_5::get_dual_ray(), Solver_module_interface::get_dual_ray, Sm_lps5_5::get_primal_ray(), Solver_module_interface::get_primal_ray, Sm_lps5_5::get_version(), Sm_lps3_2::get_version(), Solver_module_interface::get_version, Sm_lps5_5::init(), Sm_lps3_2::init(), Solver_module_interface::init, Sm_lps5_5::lp2solver(), Solver_module_interface::lp2solver, Sm_lps5_5::print_brief_version(), Sm_lps3_2::print_brief_version(), Solver_module_interface::print_brief_version, Sm_lps5_5::print_options(), Sm_lps3_2::print_options(), Solver_module_interface::print_options, Sm_lps5_5::print_version(), Sm_lps3_2::print_version(), Solver_module_interface::print_version, Sm_lps5_5::read_lp(), Sm_lps3_2::read_lp(), Solver_module_interface::read_lp, Sm_lps5_5::restore_dual(), Sm_lps3_2::restore_dual(), Solver_module_interface::restore_dual, Sm_lps5_5::restore_dual_phase1(), Solver_module_interface::restore_dual_phase1, Sm_lps5_5::restore_primal(), Sm_lps3_2::restore_primal(), Solver_module_interface::restore_primal, Sm_lps5_5::restore_primal_phase1(), Solver_module_interface::restore_primal_phase1, Sm_lps5_5::set_bounds(), Solver_module_interface::set_bounds, Sm_lps5_5::set_dual_phase1(), Solver_module_interface::set_dual_phase1, Sm_lps5_5::set_lp(), Sm_lps3_2::set_lp(), Solver_module_interface::set_lp, Sm_lps5_5::set_module_options(), Solver_module_interface::set_module_options, Sm_lps5_5::set_primal_phase1(), Solver_module_interface::set_primal_phase1, Sm_lps5_5::solve_dual_perturbed(), Sm_lps3_2::solve_dual_perturbed(), Solver_module_interface::solve_dual_perturbed, Sm_lps5_5::solve_original(), Sm_lps3_2::solve_original(), Solver_module_interface::solve_original, Sm_lps5_5::solve_primal_perturbed(), Sm_lps3_2::solve_primal_perturbed(), and Solver_module_interface::solve_primal_perturbed.

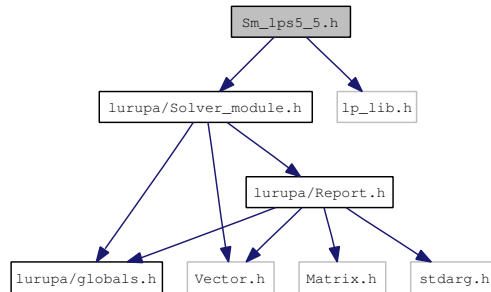
3.19 Sm_lps5_5.h File Reference

Declaration of lp_solve 5.5 solver module.

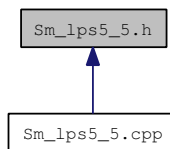
```
#include <lurupa/Solver_module.h>
```

```
#include <lp_lib.h>
```

Include dependency graph for Sm_lps5_5.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **Lp_lp_solve**
Information about lp_solve's problem.
- class **Sm_lps5_5**
Solver module for lp_solve 5.5.

Functions

- LU_SM_LPS5_5_SCOPE void **get_func_pointers** (Solver_module_interface &i)
Get function pointers.

3.19.1 Detailed Description

Declaration of lp_solve 5.5 solver module.

This file declares the solver module for lp_solve 5.5, **Sm_lps5_5** (p. 107).

Author:

Christian Keil

Id**Sm_lps5_5.h** (p. 183) 533 2008-06-12 20:08:12Z keil**3.19.2 Function Documentation****3.19.2.1 LU_SM_LPS5_5_SCOPE void get_func_pointers
(Solver_module_interface & i)**

Get function pointers.

Set the pointers of **i** to point to the member functions of **Sm_lps3_2** (p. 67).**Parameters:**→ **i** receives the pointers to member functionsSet the pointers of **i** to point to the member functions of **Sm_lps4_0_1_0** (p. 79).**Parameters:**→ **i** receives the pointers to member functionsSet the pointers of **i** to point to the member functions of **Sm_lps4_0_1_11** (p. 93).**Parameters:**→ **i** receives the pointers to member functionsSet the pointers of **i** to point to the member functions of **Sm_lps5_5** (p. 107).**Parameters:**→ **i** receives the pointers to member functions

References Sm_lps5_5::get_accuracy(), Sm_lps4_0_1_11::get_accuracy(), Sm_lps4_0_1_0::get_accuracy(), Sm_lps3_2::get_accuracy(), Solver_module_interface::get_accuracy, Sm_lps5_5::get_dual_ray(), Solver_module_interface::get_dual_ray, Sm_lps5_5::get_primal_ray(), Solver_module_interface::get_primal_ray, Sm_lps5_5::get_version(), Sm_lps4_0_1_11::get_version(), Sm_lps4_0_1_0::get_version(), Sm_lps3_2::get_version(), Solver_module_interface::get_version, Sm_lps5_5::init(), Sm_lps4_0_1_11::init(), Sm_lps4_0_1_0::init(), Sm_lps3_2::init(), Solver_module_interface::init, Sm_lps5_5::lp2solver(), Solver_module_interface::lp2solver, Sm_lps5_5::print_brief_version(), Sm_lps4_0_1_11::print_brief_version(), Sm_lps4_0_1_0::print_brief_version(), Sm_lps3_2::print_brief_version(), Solver_module_interface::print_brief_version, Sm_lps5_5::print_options(), Sm_lps4_0_1_11::print_options(), Sm_lps4_0_1_0::print_options(), Sm_lps3_2::print_options(), Solver_module_interface::print_options, Sm_lps5_5::print_version(), Sm_lps4_0_1_11::print_version(), Sm_lps4_0_1_0::print_version(), Sm_lps3_2::print_version(), Solver_module_interface::print_version, Sm_lps5_5::read_lp(), Sm_lps4_0_1_11::read_lp(), Sm_lps4_0_1_0::read_lp(), Sm_lps3_2::read_lp(), Solver_module_interface::read_lp, Sm_lps5_5::restore_dual(), Sm_lps4_0_1_11::restore_dual(), Sm_lps4_0_1_0::restore_dual(),

Sm_lps3_2::restore_dual(), Solver_module_interface::restore_dual, Sm_lps5_5::restore_dual_phase1(), Solver_module_interface::restore_dual_phase1, Sm_lps5_5::restore_primal(), Sm_lps4_0_1_11::restore_primal(), Sm_lps4_0_1_0::restore_primal(), Sm_lps3_2::restore_primal(), Solver_module_interface::restore_primal, Sm_lps5_5::restore_primal_phase1(), Solver_module_interface::restore_primal_phase1, Sm_lps5_5::set_bounds(), Solver_module_interface::set_bounds, Sm_lps5_5::set_dual_phase1(), Solver_module_interface::set_dual_phase1, Sm_lps5_5::set_lp(), Sm_lps4_0_1_11::set_lp(), Sm_lps4_0_1_0::set_lp(), Sm_lps3_2::set_lp(), Solver_module_interface::set_lp, Sm_lps5_5::set_module_options(), Sm_lps4_0_1_11::set_module_options(), Sm_lps4_0_1_0::set_module_options(), Solver_module_interface::set_module_options, Sm_lps5_5::set_primal_phase1(), Solver_module_interface::set_primal_phase1, Sm_lps5_5::solve_dual_perturbed(), Sm_lps4_0_1_11::solve_dual_perturbed(), Sm_lps4_0_1_0::solve_dual_perturbed(), Sm_lps3_2::solve_dual_perturbed(), Solver_module_interface::solve_dual_perturbed, Sm_lps5_5::solve_original(), Sm_lps4_0_1_11::solve_original(), Sm_lps4_0_1_0::solve_original(), Sm_lps3_2::solve_original(), Solver_module_interface::solve_original, Sm_lps5_5::solve_primal_perturbed(), Sm_lps4_0_1_11::solve_primal_perturbed(), Sm_lps4_0_1_0::solve_primal_perturbed(), Sm_lps3_2::solve_primal_perturbed(), and Solver_module_interface::solve_primal_perturbed.

Referenced by Lurupa::set_module().

3.20 Solver_module.h File Reference

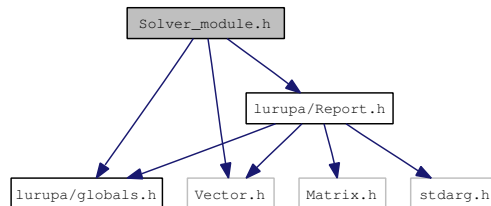
Declaration of Solver module interface.

```
#include <lurupa/globals.h>
```

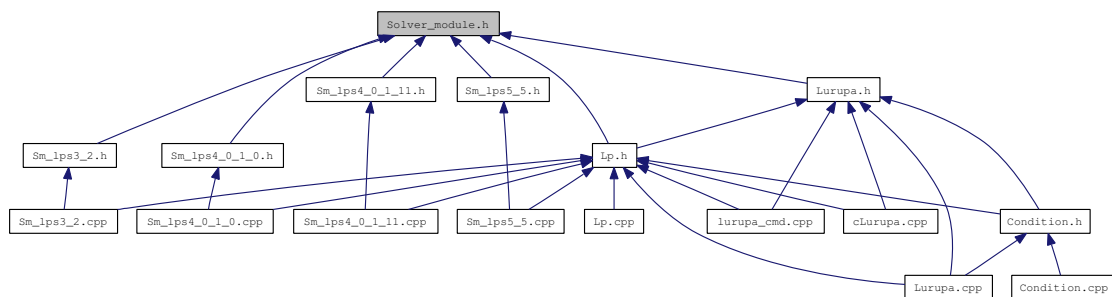
```
#include <lurupa/Report.h>
```

```
#include <Vector.h>
```

Include dependency graph for Solver_module.h:



This graph shows which files directly or indirectly include this file:



Classes

- class **Lp_solver**
Information about solver specific lp representation.
- class **Solver_module**
Interface for an lp-solver module.
- struct **Solver_module_interface**
Function pointer version of Solver_module.

Functions

- void **getFuncPointers** (Solver_module_interface &i)
Get pointers to member functions.

3.20.1 Detailed Description

Declaration of Solver module interface.

This file contains the interface declaration for solver modules.

Author:

Christian Keil

Id

Solver__module.h (p. 186) 533 2008-06-12 20:08:12Z keil

Index

add_column_to_basis
 Lurupa, 42

adjust_eta
 Sm_lps3_2, 69
 Sm_lps4_0_1_0, 81
 Sm_lps4_0_1_11, 95
 Sm_lps5_5, 110

Bound, 17

Bound_status
 globals.h, 146

bound_status_string
 globals.h, 148

bs_dualinf
 globals.h, 146

bs_failure
 globals.h, 147

bs_iter
 globals.h, 147

bs_noenc
 globals.h, 146

bs_orginf
 globals.h, 146

bs_orgunb
 globals.h, 146

bs_pertinf
 globals.h, 146

bs_priminf
 globals.h, 146

bs_rank
 globals.h, 147

bs_running
 globals.h, 147

bs_timeout
 globals.h, 147

bs_verified
 globals.h, 146

bs_warn
 globals.h, 146

build_constraint_maps
 Sm_lps3_2, 69
 Sm_lps4_0_1_0, 82
 Sm_lps4_0_1_11, 96
 Sm_lps5_5, 110

Certificate, 19

clu_cleanup
 cLurupa.cpp, 129
 cLurupa.h, 137

clu_get_alpha
 cLurupa.cpp, 129
 cLurupa.h, 137

clu_get_core_version
 cLurupa.cpp, 129
 cLurupa.h, 137

clu_get_eta
 cLurupa.cpp, 130
 cLurupa.h, 137

clu_get_lp_name
 cLurupa.cpp, 130
 cLurupa.h, 137

clu_get_module_version
 cLurupa.cpp, 130
 cLurupa.h, 138

clu_get_solver_eps
 cLurupa.cpp, 130
 cLurupa.h, 138

clu_init
 cLurupa.cpp, 130
 cLurupa.h, 138

clu_is_inflate
 cLurupa.cpp, 131
 cLurupa.h, 138

clu_is_lp_maximize
 cLurupa.cpp, 131
 cLurupa.h, 139

clu_lower_bound
 cLurupa.cpp, 131
 cLurupa.h, 139

clu_print_module_options
 cLurupa.cpp, 131
 cLurupa.h, 139

clu_read_lp
 cLurupa.cpp, 132
 cLurupa.h, 139

clu_set_alpha
 cLurupa.cpp, 132
 cLurupa.h, 139

clu_set_eta
 cLurupa.cpp, 132

- cLurupa.h, 140
- clu_set_inflate
 - cLurupa.cpp, 132
 - cLurupa.h, 140
- clu_set_interface
 - cLurupa.cpp, 132
- clu_set_lp
 - cLurupa.cpp, 133
- clu_set_module
 - cLurupa.cpp, 133
 - cLurupa.h, 140
- clu_set_module_options
 - cLurupa.cpp, 133
 - cLurupa.h, 140
- clu_solve_lp
 - cLurupa.cpp, 133
- clu_upper_bound
 - cLurupa.cpp, 133
 - cLurupa.h, 141
- cLurupa.cpp, 127
 - clu_cleanup, 129
 - clu_get_alpha, 129
 - clu_get_core_version, 129
 - clu_get_eta, 130
 - clu_get_lp_name, 130
 - clu_get_module_version, 130
 - clu_get_solver_eps, 130
 - clu_init, 130
 - clu_is_inflate, 131
 - clu_is_lp_maximize, 131
 - clu_lower_bound, 131
 - clu_print_module_options, 131
 - clu_read_lp, 132
 - clu_set_alpha, 132
 - clu_set_eta, 132
 - clu_set_inflate, 132
 - clu_set_interface, 132
 - clu_set_lp, 133
 - clu_set_module, 133
 - clu_set_module_options, 133
 - clu_solve_lp, 133
 - clu_upper_bound, 133
- cLurupa.h, 135
 - clu_cleanup, 137
 - clu_get_alpha, 137
 - clu_get_core_version, 137
 - clu_get_eta, 137
 - clu_get_lp_name, 137
 - clu_get_module_version, 138
 - clu_get_solver_eps, 138
 - clu_init, 138
 - clu_is_inflate, 138
 - clu_is_lp_maximize, 139
 - clu_lower_bound, 139
 - clu_print_module_options, 139
 - clu_read_lp, 139
 - clu_set_alpha, 139
 - clu_set_eta, 140
 - clu_set_inflate, 140
 - clu_set_module, 140
 - clu_set_module_options, 140
 - clu_upper_bound, 141
- compute_dual
 - lurupa_cmd.cpp, 159
- compute_dual_deflation
 - Lurupa, 42
- compute_lower
 - Lurupa, 42
 - lurupa_cmd.cpp, 159
- compute_primal
 - lurupa_cmd.cpp, 159
- compute_primal_deflation
 - Lurupa, 43
- compute_upper
 - Lurupa, 43
 - lurupa_cmd.cpp, 159
- cond
 - Lurupa, 44
- Condition, 20
 - Condition, 21
 - generate_lp_rho_d, 22
 - generate_lp_rho_p, 22
 - norm, 23
 - norm_1, 23
 - norm_11, 23
 - norm_F, 24
 - norm_L1, 24
 - rho_d, 25
 - rho_p, 25
 - rot, 25
 - swap, 26
- Condition.cpp, 142
- Condition.h, 143
- Csv_style
 - globals.h, 147
- decrease_dual_deflation
 - Lurupa, 44
- decrease_primal_deflation
 - Lurupa, 44
- do_pivot
 - Lurupa, 44
- dual_certificate
 - Lurupa, 45
- dual_perturb
 - Sm_lps3_2, 70
 - Sm_lps4_0_1_0, 82
 - Sm_lps4_0_1_11, 96

- Sm_lps5_5, 111
- eliminate
 - Lurupa, 45
- Enclosure, 27
- establish_dual_feasibility
 - Lurupa, 46
- establish_equality_constraints
 - Lurupa, 46
- establish_ix_simple_bounds
 - Lurupa, 47
- establish_iy_simple_bounds
 - Lurupa, 47
- establish_lagrange_parameters
 - Lurupa, 47
- establish_primal_feasibility
 - Lurupa, 48
- find_basis
 - Lurupa, 49
- find_constraint_basis
 - Lurupa, 49
- find_equation_basis
 - Lurupa, 50
- find_free_variables
 - Sm_lps3_2, 70
 - Sm_lps4_0_1_0, 82
 - Sm_lps4_0_1_11, 96
 - Sm_lps5_5, 111
- generate_lp_rho_d
 - Condition, 22
- generate_lp_rho_p
 - Condition, 22
- get_accuracy
 - Sm_lps3_2, 70
 - Sm_lps4_0_1_0, 83
 - Sm_lps4_0_1_11, 97
 - Sm_lps5_5, 111
- get_alpha
 - Lurupa, 50
- get_core_version
 - Lurupa, 51
- get_eta
 - Lurupa, 51
- get_func_pointers
 - Sm_lps3_2.cpp, 170
 - Sm_lps3_2.h, 172
 - Sm_lps4_0_1_0.cpp, 174
 - Sm_lps4_0_1_0.h, 176
 - Sm_lps4_0_1_11.cpp, 178
 - Sm_lps4_0_1_11.h, 180
 - Sm_lps5_5.cpp, 182
 - Sm_lps5_5.h, 184
- get_module_version
 - Lurupa, 51
- get_solver_eps
 - Lurupa, 51
- get_version
 - Sm_lps3_2, 71
 - Sm_lps4_0_1_0, 83
 - Sm_lps4_0_1_11, 97
 - Sm_lps5_5, 112
- globals.h, 145
 - Bound_status, 146
 - bound_status_string, 148
 - bs_dualinf, 146
 - bs_failure, 147
 - bs_iter, 147
 - bs_noenc, 146
 - bs_ordinf, 146
 - bs_organb, 146
 - bs_pertinf, 146
 - bs_priminf, 146
 - bs_rank, 147
 - bs_running, 147
 - bs_timeout, 147
 - bs_verified, 146
 - bs_warn, 146
 - Csv_style, 147
 - matlab, 147
 - octave, 147
 - Solver_status, 147
 - solver_status_string, 148
 - ss_failure, 147
 - ss_feasible, 147
 - ss_infeasible, 147
 - ss_timeout, 147
 - ss_unbounded, 147
 - ss_unknown, 147
 - wm_all, 147
 - wm_ask, 147
 - wm_none, 147
 - Write_mps, 147
- increase_dual_deflation
 - Lurupa, 51
- increase_primal_deflation
 - Lurupa, 52
- inflate_lp
 - Sm_lps3_2, 71
 - Sm_lps4_0_1_0, 83
 - Sm_lps4_0_1_11, 97
 - Sm_lps5_5, 112
- init
 - Sm_lps3_2, 71
 - Sm_lps4_0_1_0, 83
 - Sm_lps4_0_1_11, 97

- Sm_lps5_5, 112
- is_inflate
 - Lurupa, 52
- lower_bound
 - Lurupa, 52
- Lp, 28
- Lp.cpp, 149
- Lp.h, 150
- Lp_lp_solve, 31
- Lp_solver, 33
- Lp_stats, 34
- lps_log
 - Sm_lps4_0_1_0, 84
 - Sm_lps4_0_1_11, 98
 - Sm_lps5_5, 113
- Lurupa, 36
 - add_column_to_basis, 42
 - compute_dual_deflation, 42
 - compute_lower, 42
 - compute_primal_deflation, 43
 - compute_upper, 43
 - cond, 44
 - decrease_dual_deflation, 44
 - decrease_primal_deflation, 44
 - do_pivot, 44
 - dual_certificate, 45
 - eliminate, 45
 - establish_dual_feasibility, 46
 - establish_equality_constraints, 46
 - establish_ix_simple_bounds, 47
 - establish_iy_simple_bounds, 47
 - establish_lagrange_parameters, 47
 - establish_primal_feasibility, 48
 - find_basis, 49
 - find_constraint_basis, 49
 - find_equation_basis, 50
 - get_alpha, 50
 - get_core_version, 51
 - get_eta, 51
 - get_module_version, 51
 - get_solver_eps, 51
 - increase_dual_deflation, 51
 - increase_primal_deflation, 52
 - is_inflate, 52
 - lower_bound, 52
 - pivot_element, 53
 - primal_certificate, 53
 - primal_feasible, 54
 - print_module_options, 54
 - process_bounded_variables, 54
 - process_free_variables, 55
 - process_initial_dual_solution, 55
 - process_initial_primal_solution, 56
 - process_perturbed_dual_solution, 56
 - process_perturbed_primal_solution, 57
 - read_lp, 57
 - restore_dual_phase1, 58
 - restore_primal_phase1, 58
 - rho_d, 59
 - rho_p, 59
 - set_alpha, 59
 - set_dual_phase1, 59
 - set_eta, 60
 - set_inflate, 60
 - set_interface, 60
 - set_lp, 61
 - set_module, 61
 - set_module_options, 61
 - set_primal_phase1, 62
 - shorten_basis_indices, 62
 - solve_lp, 63
 - upper_bound, 63
- Lurupa.cpp, 152
- Lurupa.h, 154
- lurupa_cmd.cpp, 156
 - compute_dual, 159
 - compute_lower, 159
 - compute_primal, 159
 - compute_upper, 159
 - main, 160
 - print_bound_stats, 160
 - print_brief_version, 161
 - print_model_data, 161
 - print_usage, 161
 - print_version, 161
 - process_solving_status, 162
 - report_bound_quality, 162
 - report_dual, 162
 - report_lower, 163
 - report_primal, 163
 - report_upper, 163
 - rounded_string, 164
 - start_timer, 164
 - stop_timer, 164
 - timer_diff, 165
 - write_csv_table, 165
 - write_latex_table, 166
 - write_tables, 166
- main
 - lurupa_cmd.cpp, 160
- matlab
 - globals.h, 147
- norm
 - Condition, 23
- norm_1

- Condition, 23
- norm_11
 - Condition, 23
- norm_F
 - Condition, 24
- norm_L1
 - Condition, 24
- octave
 - globals.h, 147
- pivot_element
 - Lurupa, 53
- postprocess
 - Sm_lps4_0_1_0, 84
 - Sm_lps4_0_1_11, 98
- primal_certificate
 - Lurupa, 53
- Primal_deflation, 64
- primal_feasible
 - Lurupa, 54
- primal_perturb
 - Sm_lps3_2, 72
 - Sm_lps4_0_1_0, 84
 - Sm_lps4_0_1_11, 98
 - Sm_lps5_5, 113
- print_bound_stats
 - lurupa_cmd.cpp, 160
- print_brief_version
 - lurupa_cmd.cpp, 161
- print_model_data
 - lurupa_cmd.cpp, 161
- print_module_options
 - Lurupa, 54
- print_options
 - Sm_lps3_2, 72
 - Sm_lps4_0_1_0, 85
 - Sm_lps4_0_1_11, 99
 - Sm_lps5_5, 113
- print_usage
 - lurupa_cmd.cpp, 161
- print_version
 - lurupa_cmd.cpp, 161
- process_bounded_variables
 - Lurupa, 54
- process_free_variables
 - Lurupa, 55
- process_initial_dual_solution
 - Lurupa, 55
- process_initial_primal_solution
 - Lurupa, 56
- process_perturbed_dual_solution
 - Lurupa, 56
- process_perturbed_primal_solution

- Lurupa, 57
- process_solving_status
 - lurupa_cmd.cpp, 162
- read_duals
 - Sm_lps3_2, 72
 - Sm_lps4_0_1_0, 85
 - Sm_lps4_0_1_11, 99
 - Sm_lps5_5, 113
- read_general_data
 - Sm_lps3_2, 72
 - Sm_lps4_0_1_0, 85
 - Sm_lps4_0_1_11, 99
 - Sm_lps5_5, 114
- read_lp
 - Lurupa, 57
 - Sm_lps3_2, 73
 - Sm_lps4_0_1_0, 86
 - Sm_lps4_0_1_11, 100
 - Sm_lps5_5, 114
- read_lp_mat
 - Sm_lps3_2, 73
 - Sm_lps4_0_1_0, 86
 - Sm_lps4_0_1_11, 100
 - Sm_lps5_5, 114
- read_primals
 - Sm_lps3_2, 74
 - Sm_lps4_0_1_0, 86
 - Sm_lps4_0_1_11, 100
 - Sm_lps5_5, 115
- read_right_hand_sides
 - Sm_lps3_2, 74
 - Sm_lps4_0_1_0, 87
 - Sm_lps4_0_1_11, 101
 - Sm_lps5_5, 115
- read_simple_bounds
 - Sm_lps3_2, 74
 - Sm_lps4_0_1_0, 87
 - Sm_lps4_0_1_11, 101
 - Sm_lps5_5, 116
- Report, 65
- Report.h, 168
- report_bound_quality
 - lurupa_cmd.cpp, 162
- report_dual
 - lurupa_cmd.cpp, 162
- report_lower
 - lurupa_cmd.cpp, 163
- report_primal
 - lurupa_cmd.cpp, 163
- report_upper
 - lurupa_cmd.cpp, 163
- resize_lp
 - Sm_lps3_2, 74

- Sm_lps4_0_1_0, 87
- Sm_lps4_0_1_11, 101
- Sm_lps5_5, 116
- restore_dual
 - Sm_lps3_2, 75
 - Sm_lps4_0_1_0, 88
 - Sm_lps4_0_1_11, 102
 - Sm_lps5_5, 116
- restore_dual_phase1
 - Lurupa, 58
- restore_primal
 - Sm_lps3_2, 75
 - Sm_lps4_0_1_0, 88
 - Sm_lps4_0_1_11, 102
 - Sm_lps5_5, 116
- restore_primal_phase1
 - Lurupa, 58
- rho_d
 - Condition, 25
 - Lurupa, 59
- rho_p
 - Condition, 25
 - Lurupa, 59
- rot
 - Condition, 25
- rounded_string
 - lurupa_cmd.cpp, 164
- set_alpha
 - Lurupa, 59
- set_dual_phase1
 - Lurupa, 59
- set_eta
 - Lurupa, 60
- set_inflate
 - Lurupa, 60
- set_interface
 - Lurupa, 60
- set_lp
 - Lurupa, 61
- set_module
 - Lurupa, 61
- set_module_options
 - Lurupa, 61
 - Sm_lps4_0_1_0, 88
 - Sm_lps4_0_1_11, 102
 - Sm_lps5_5, 117
- set_primal_phase1
 - Lurupa, 62
- shorten_basis_indices
 - Lurupa, 62
- Sm_lps3_2, 67
 - adjust_eta, 69
 - build_constraint_maps, 69
 - dual_perturb, 70
 - find_free_variables, 70
 - get_accuracy, 70
 - get_version, 71
 - inflate_lp, 71
 - init, 71
 - primal_perturb, 72
 - print_options, 72
 - read_duals, 72
 - read_general_data, 72
 - read_lp, 73
 - read_lp_mat, 73
 - read_primals, 74
 - read_right_hand_sides, 74
 - read_simple_bounds, 74
 - resize_lp, 74
 - restore_dual, 75
 - restore_primal, 75
 - solve_dual_perturbed, 75
 - solve_lp, 76
 - solve_original, 76
 - solve_primal_perturbed, 77
 - transform_lp, 77
- Sm_lps3_2.cpp, 169
 - get_func_pointers, 170
- Sm_lps3_2.h, 171
 - get_func_pointers, 172
- Sm_lps4_0_1_0, 79
 - adjust_eta, 81
 - build_constraint_maps, 82
 - dual_perturb, 82
 - find_free_variables, 82
 - get_accuracy, 83
 - get_version, 83
 - inflate_lp, 83
 - init, 83
 - lps_log, 84
 - postprocess, 84
 - primal_perturb, 84
 - print_options, 85
 - read_duals, 85
 - read_general_data, 85
 - read_lp, 86
 - read_lp_mat, 86
 - read_primals, 86
 - read_right_hand_sides, 87
 - read_simple_bounds, 87
 - resize_lp, 87
 - restore_dual, 88
 - restore_primal, 88
 - set_module_options, 88
 - solve_dual_perturbed, 89
 - solve_lp, 90
 - solve_original, 90

- solve_primal_perturbed, 91
- transform_lp, 91
- Sm_lps4_0_1_0.cpp, 173
 - get_func_pointers, 174
- Sm_lps4_0_1_0.h, 175
 - get_func_pointers, 176
- Sm_lps4_0_1_11, 93
 - adjust_eta, 95
 - build_constraint_maps, 96
 - dual_perturb, 96
 - find_free_variables, 96
 - get_accuracy, 97
 - get_version, 97
 - inflate_lp, 97
 - init, 97
 - lps_log, 98
 - postprocess, 98
 - primal_perturb, 98
 - print_options, 99
 - read_duals, 99
 - read_general_data, 99
 - read_lp, 100
 - read_lp_mat, 100
 - read_primals, 100
 - read_right_hand_sides, 101
 - read_simple_bounds, 101
 - resize_lp, 101
 - restore_dual, 102
 - restore_primal, 102
 - set_module_options, 102
 - solve_dual_perturbed, 103
 - solve_lp, 104
 - solve_original, 104
 - solve_primal_perturbed, 105
 - transform_lp, 105
- Sm_lps4_0_1_11.cpp, 177
 - get_func_pointers, 178
- Sm_lps4_0_1_11.h, 179
 - get_func_pointers, 180
- Sm_lps5_5, 107
 - adjust_eta, 110
 - build_constraint_maps, 110
 - dual_perturb, 111
 - find_free_variables, 111
 - get_accuracy, 111
 - get_version, 112
 - inflate_lp, 112
 - init, 112
 - lps_log, 113
 - primal_perturb, 113
 - print_options, 113
 - read_duals, 113
 - read_general_data, 114
 - read_lp, 114
 - read_lp_mat, 114
 - read_primals, 115
 - read_right_hand_sides, 115
 - read_simple_bounds, 116
 - resize_lp, 116
 - restore_dual, 116
 - restore_primal, 116
 - set_module_options, 117
 - solve_dual_perturbed, 118
 - solve_lp, 118
 - solve_original, 118
 - solve_primal_perturbed, 119
 - timeout, 120
 - trace, 120
 - transform_lp, 119
 - verbosity, 120
- Sm_lps5_5.cpp, 181
 - get_func_pointers, 182
- Sm_lps5_5.h, 183
 - get_func_pointers, 184
- solve_dual_perturbed
 - Sm_lps3_2, 75
 - Sm_lps4_0_1_0, 89
 - Sm_lps4_0_1_11, 103
 - Sm_lps5_5, 118
- solve_lp
 - Lurupa, 63
 - Sm_lps3_2, 76
 - Sm_lps4_0_1_0, 90
 - Sm_lps4_0_1_11, 104
 - Sm_lps5_5, 118
- solve_original
 - Sm_lps3_2, 76
 - Sm_lps4_0_1_0, 90
 - Sm_lps4_0_1_11, 104
 - Sm_lps5_5, 118
- solve_primal_perturbed
 - Sm_lps3_2, 77
 - Sm_lps4_0_1_0, 91
 - Sm_lps4_0_1_11, 105
 - Sm_lps5_5, 119
- Solver_module, 122
- Solver_module.h, 186
- Solver_module_interface, 125
- Solver_status
 - globals.h, 147
- solver_status_string
 - globals.h, 148
- ss_failure
 - globals.h, 147
- ss_feasible
 - globals.h, 147
- ss_infeasible
 - globals.h, 147

- ss_timeout
 - globals.h, 147
- ss_unbounded
 - globals.h, 147
- ss_unknown
 - globals.h, 147
- start_timer
 - lurupa_cmd.cpp, 164
- stop_timer
 - lurupa_cmd.cpp, 164
- swap
 - Condition, 26

- timeout
 - Sm_lps5_5, 120
- timer_diff
 - lurupa_cmd.cpp, 165
- trace
 - Sm_lps5_5, 120
- transform_lp
 - Sm_lps3_2, 77
 - Sm_lps4_0_1_0, 91
 - Sm_lps4_0_1_11, 105
 - Sm_lps5_5, 119

- upper_bound
 - Lurupa, 63

- verbosity
 - Sm_lps5_5, 120

- wm_all
 - globals.h, 147
- wm_ask
 - globals.h, 147
- wm_none
 - globals.h, 147
- write_csv_table
 - lurupa_cmd.cpp, 165
- write_latex_table
 - lurupa_cmd.cpp, 166
- Write_mps
 - globals.h, 147
- write_tables
 - lurupa_cmd.cpp, 166

Bibliography

- [1] Berkelaar, M., K. Eikland, and P. Notebaert: 'lp_solve'. World Wide Web, http://groups.yahoo.com/group/lp_solve.
- [2] Hayes, B.: 2003, 'A Lucid Interval'. American Scientist, Volume 91, Number 6, Pages 484–488.
- [3] Jansson, C.: 2004, 'Rigorous Lower and Upper Bounds in Linear Programming'. SIAM J. Optim 14(3), 914–935.
- [4] Kearfott, B.: 'Interval Computations: Introduction, Uses, and Resources'. World Wide Web, ftp://interval.louisiana.edu/pub/interval_math/papers/papers-of-Kearfott/Euromath_bulletin_survey_article/survey.ps.
- [5] Keil, C. and C. Jansson: 2006, 'Computational Experience with Rigorous Error Bounds for the Netlib Linear Programming Library'. Reliable Computing, Volume 12, Issue 4, Pages 303–321.
- [6] Kreinovich, V., D. J. Berleant, and M. Koshelev: 'Interval Computations'. World Wide Web, <http://www.cs.utep.edu/interval-comp/main.html>.
- [7] Netlib, 'Netlib Linear Programming Library'. World Wide Web, <http://www.netlib.org/lp>.
- [8] Ordonez, F. and Freund, R. M.: 'Computational experience and the explanatory value of condition measures for linear programming'. SIAM J. Optim., Volume 14, Number 2, Pages 307–333.