C interfaces to GALAHAD PRESOLVE

Jari Fowkes and Nick Gould
STFC Rutherford Appleton Laboratory
Tue Apr 5 2022

Chapter 1

GALAHAD C package presolve

1.1 Introduction

1.1.1 Purpose

Presolving aims to **improve the formulation of a given optimization problem by applying a sequence of simple transformations**, and thereby to produce a *reduced* problem in a *standard form* that should be simpler to solve. This reduced problem may then be passed to an appropriate solver. Once the reduced problem has been solved, it is then *restored* to recover the solution for the original formulation.

This package applies presolving techniques to a **linear**

$$\text{minimize } \; l(x) = g^T x + f$$

or **quadratic program**

$$\text{minimize } \; q(x) = \frac{1}{2} x^T H x + g^T x + f$$

subject to the general linear constraints

$$c_i^l \leq a_i^T x \leq c_i^u, \quad i = 1, \ldots, m,$$

and the simple bound constraints

$$x_j^l \leq x_j \leq x_j^u, \quad j = 1, \ldots, n,$$

where the n by n symmetric matrix H, the vectors g, a_i, c^l, c^u, x^l, x^u and the scalar f are given. Any of the constraint bounds c_i^l, c_i^u, x_j^l and x_j^u may be infinite.

In addition, bounds on the Lagrange multipliers y associated with the general linear constraints and on the dual variables z associated with the simple bound constraints

$$y_i^l \leq y_i \leq y_i^u, \quad i = 1, \ldots, m,$$

and

$$z_i^l \leq z_i \leq z_i^u, \quad i = 1, \ldots, n,$$

are also provided, where the m-dimensional vectors y^l and y^u, as well as the n-dimensional vectors x^l and x^u are given. Any component of c^l, c^u, x^l, x^u, y^l, y^u, z^l or z^u may be infinite.

1.1.2 Authors

N. I. M. Gould, STFC-Rutherford Appleton Laboratory, England and Ph. L. Toint, University of Namur, Belgium

C interface, additionally J. Fowkes, STFC-Rutherford Appleton Laboratory.

1.1.3 Originally released

March 2002, C interface March 2022.

1.1.4 Terminology

The required solution x necessarily satisfies the primal optimality conditions

(1a) $$Ax = c$$

and

(1b) $$c^l \leq c \leq c^u, \;\; x^l \leq x \leq x^u,$$

the dual optimality conditions

(2a) $$Hx + g = A^T y + z$$

where

(2b) $$y = y^l + y^u, \;\; z = z^l + z^u, \;\; y^l \geq 0, \;\; y^u \leq 0, \;\; z^l \geq 0 \;\; \text{and} \;\; z^u \leq 0,$$

and the complementary slackness conditions

(3) $$(Ax - c^l)^T y^l = 0, \;\; (Ax - c^u)^T y^u = 0, \;\; (x - x^l)^T z^l = 0 \;\; \text{and} \;\; (x - x^u)^T z^u = 0,$$

where the vectors y and z are known as the Lagrange multipliers for2 the general linear constraints, and the dual variables for the bounds, respectively, and where the vector inequalities hold component-wise.

1.1.5 Method

The purpose of presolving is to exploit these equations in order to reduce the problem to the standard form defined as follows:

- The variables are ordered so that their bounds appear in the order

free			x		
non-negativity	0	$\leq$	x		
lower	x^l	$\leq$	x		
range	x^l	$\leq$	x	$\leq$	x^u
upper			x	$\leq$	x^u
non-positivity			x	$\leq$	0

  Fixed variables are removed. Within each category, the variables are further ordered so that those with non-zero diagonal Hessian entries occur before the remainder.

- The constraints are ordered so that their bounds appear in the order

non-negativity	0	$\leq$	Ax		
equality	c^l	$=$	Ax		
lower	c^l	$\leq$	Ax		
range	c^l	$\leq$	Ax	$\leq$	c^u
upper			Ax	$\leq$	c^u
non-positivity			Ax	$\leq$	0

  Free constraints are removed.

- In addition, constraints may be removed or bounds tightened, to reduce the size of the feasible region or simplify the problem if this is possible, and bounds may be tightened on the dual variables and the multipliers associated with the problem.

The presolving algorithm proceeds by applying a (potentially long) series of simple transformations to the problem, each transformation introducing a further simplification of the problem. These involve the removal of empty and singleton rows, the removal of redundant and forcing primal constraints, the tightening of primal and dual bounds, the exploitation of linear singleton, linear doubleton and linearly unconstrained columns, the merging dependent variables, row sparsification and split equalities. Transformations are applied in successive passes, each pass involving the following actions:

1. remove empty and singletons rows,
2. try to eliminate variables that are linearly unconstrained,
3. attempt to exploit the presence of linear singleton columns,
4. attempt to exploit the presence of linear doubleton columns,
5. complete the analysis of the dual constraints,
6. remove empty and singletons rows,
7. possibly remove dependent variables,
8. analyze the primal constraints,
9. try to make A sparser by combining its rows,
10. check the current status of the variables, dual variables and multipliers.

All these transformations are applied to the structure of the original problem, which is only permuted to standard form after all transformations are completed. *Note that the Hessian and Jacobian of the resulting reduced problem are always stored in sparse row-wise format.* The reduced problem is then solved by a quadratic or linear programming solver, thus ensuring sufficiently small primal-dual feasibility and complementarity. Finally, the solution of the simplified problem is re-translated in the variables/constraints/format of the original problem formulation by a *restoration* phase.

If the number of problem transformations exceeds `control.transf_buffer_size`, the transformation buffer size, then they are saved in a ``history'' file, whose name may be chosen by specifying the control.transf_file_name control parameter, When this is the case, this file is subsequently reread by `presolve_restore_solution`. It must not be altered by the user.

Overall, the presolving process follows one of the two sequences:

$$\boxed{\text{initialize}} \rightarrow \left[\boxed{\text{apply transformations}} \rightarrow \text{(solve problem)} \rightarrow \boxed{\text{restore}}\right] \rightarrow \boxed{\text{terminate}}$$

or

$$\boxed{\text{initialize}} \rightarrow \left[\boxed{\text{read specfile}} \rightarrow \boxed{\text{apply transformations}} \rightarrow \text{(solve problem)} \rightarrow \boxed{\text{restore}}\right] \rightarrow \boxed{\text{terminate}}$$

where the procedure's control parameter may be modified by reading the specfile, and where (solve problem) indicates that the reduced problem is solved. Each of the ``boxed'' steps in these sequences corresponds to calling a specific routine of the package In the diagrams above, brackated subsequence of steps means that they can be repeated with problem having the same structure. The value of the `problem.new_problem_structure` must be true on entry of `presolve_apply_to_problem` on the first time it is used in this repeated subsequence. Such a subsequence must be terminated by a call to `presolve_terminate` before presolving is applied to a problem with a different structure.

Note that the values of the multipliers and dual variables (and thus of their respective bounds) depend on the functional form assumed for the Lagrangian function associated with the problem. This form is given by

$$L(x, y, z) = qx) - y\_sign * y^T(Ax - c) - z\_sign * z,$$

(considering only active constraints $Ax = c$), where the parameters y_{sign} and z_{sign} are +1 or -1 and can be chosen by the user. Thus, if y_{sign} = +1, the multipliers associated to active constraints originally posed as inequalities are non-negative if the inequality is a lower bound and non-positive if it is an upper bound. Obvioulsy they are not constrained in sign for constraints originally posed as equalities. These sign conventions are reversed if y_{sign} = -1. Similarly, if z_{sign} = +1}, the dual variables associated to active bounds are non-negative if the original bound is an lower bound, non-positive if it is an upper bound, or unconstrained in sign if the variables is fixed; and this convention is reversed in z_sign = -1}. The values of z_{sign} and y_{sign} may be chosen by setting the corresponding components of the `control` structure to `1` or `-1`.

1.1.6 Reference

The algorithm is described in more detail in

N. I. M. Gould and Ph. L. Toint (2004). Presolving for quadratic programming. Mathematical Programming **100**(1), pp 95–132.

1.1.7 Call order

To solve a given problem, functions from the presolve package must be called in the following order:

- presolve_initialize - provide default control parameters and set up initial data structures
- presolve_read_specfile (optional) - override control values by reading replacement values from a file
- presolve_import_problem - import the problem data and report the dimensions of the transformed problem
- presolve_transform_problem - apply the presolve algorithm to transform the data
- presolve_restore_solution - restore the solution from that of the transformed problem
- presolve_information (optional) - recover information about the solution and solution process
- presolve_terminate - deallocate data structures

See Section 4.1 for examples of use.

1.1.8 Unsymmetric matrix storage formats

The unsymmetric m by n constraint matrix A may be presented and stored in a variety of convenient input formats.

Both C-style (0 based) and fortran-style (1-based) indexing is allowed. Choose `control.f_indexing` as `false` for C style and `true` for fortran style; the discussion below presumes C style, but add 1 to indices for the corresponding fortran version.

Wrappers will automatically convert between 0-based (C) and 1-based (fortran) array indexing, so may be used transparently from C. This conversion involves both time and memory overheads that may be avoided by supplying data that is already stored using 1-based indexing.

1.1.8.1 Dense storage format

The matrix A is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. In this case, component $n * i + j$ of the storage array A_val will hold the value A_{ij} for $0 \leq i \leq m - 1, 0 \leq j \leq n - 1$.

1.1.8.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry, $0 \leq l \leq ne - 1$, of A, its row index i, column index j and value A_{ij}, $0 \leq i \leq m - 1$, $0 \leq j \leq n - 1$, are stored as the l-th components of the integer arrays A_row and A_col and real array A_val, respectively, while the number of nonzeros is recorded as A_ne = ne.

1.1.8.3 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row i appear directly before those in row i+1. For the i-th row of A the i-th component of the integer array A_ptr holds the position of the first entry in this row, while A_ptr(m) holds the total number of entries plus one. The column indices j, $0 \leq j \leq n - 1$, and values A_{ij} of the nonzero entries in the i-th row are stored in components l = A_ptr(i), ..., A_ptr(i+1)-1, $0 \leq i \leq m - 1$, of the integer array A_col, and real array A_val, respectively. For sparse matrices, this scheme almost always requires less storage than its predecessor.

1.1.9 Symmetric matrix storage formats

Likewise, the symmetric n by n objective Hessian matrix H may be presented and stored in a variety of formats. But crucially symmetry is exploited by only storing values from the lower triangular part (i.e, those entries that lie on or below the leading diagonal).

1.1.9.1 Dense storage format

The matrix H is stored as a compact dense matrix by rows, that is, the values of the entries of each row in turn are stored in order within an appropriate real one-dimensional array. Since H is symmetric, only the lower triangular part (that is the part h_{ij} for $0 \leq j \leq i \leq n - 1$) need be held. In this case the lower triangle should be stored by rows, that is component $i * i/2 + j$ of the storage array H_val will hold the value h_{ij} (and, by symmetry, h_{ji}) for $0 \leq j \leq i \leq n - 1$.

1.1.9.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry, $0 \leq l \leq ne - 1$, of H, its row index i, column index j and value h_{ij}, $0 \leq j \leq i \leq n - 1$, are stored as the l-th components of the integer arrays H_row and H_col and real array H_val, respectively, while the number of nonzeros is recorded as H_ne = ne. Note that only the entries in the lower triangle should be stored.

1.1.9.3 Sparse row-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in row i appear directly before those in row i+1. For the i-th row of H the i-th component of the integer array H_ptr holds the position of the first entry in this row, while H_ptr(n) holds the total number of entries plus one. The column indices j, $0 \leq j \leq i$, and values h_{ij} of the entries in the i-th row are stored in components l = H_ptr(i), ..., H_ptr(i+1)-1 of the integer array H_col, and real array H_val, respectively. Note that as before only the entries in the lower triangle should be stored. For sparse matrices, this scheme almost always requires less storage than its predecessor.

1.1.9.4 Diagonal storage format

If H is diagonal (i.e., $H_{ij} = 0$ for all $0 \leq i \neq j \leq n - 1$) only the diagonals entries H_{ii}, $0 \leq i \leq n - 1$ need be stored, and the first n components of the array H_val may be used for the purpose.

1.1.9.5 Multiples of the identity storage format

If H is a multiple of the identity matrix, (i.e., $H = \alpha I$ where I is the n by n identity matrix and α is a scalar), it suffices to store α as the first component of H_val.

1.1.9.6 The identity matrix format

If H is the identity matrix, no values need be stored.

1.1.9.7 The zero matrix format

The same is true if H is the zero matrix.

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

File Documentation

3.1 galahad_presolve.h File Reference

```
#include <stdbool.h>
#include "galahad_precision.h"
```

Data Structures

- struct presolve_control_type
- struct presolve_inform_type

Functions

- void presolve_initialize (void **data, struct presolve_control_type *control, int *status)
- void presolve_read_specfile (struct presolve_control_type *control, const char specfile[ ])
- void presolve_import_problem (struct presolve_control_type *control, void **data, int *status, int n, int m, const char H_type[ ], int H_ne, const int H_row[ ], const int H_col[ ], const int H_ptr[ ], const real_wp_ H_val[ ], const real_wp_ g[ ], const real_wp_ f, const char A_type[ ], int A_ne, const int A_row[ ], const int A_col[ ], const int A_ptr[ ], const real_wp_ A_val[ ], const real_wp_ c_l[ ], const real_wp_ c_u[ ], const real_wp_ x_l[ ], const real_wp_ x_u[ ], int *n_out, int *m_out, int *H_ne_out, int *A_ne_out)
- void presolve_transform_problem (void **data, int *status, int n, int m, int H_ne, int H_col[ ], int H_ptr[ ], real_wp_ H_val[ ], real_wp_ g[ ], real_wp_ *f, int A_ne, int A_col[ ], int A_ptr[ ], real_wp_ A_val[ ], real_wp_ c_l[ ], real_wp_ c_u[ ], real_wp_ x_l[ ], real_wp_ x_u[ ], real_wp_ y_l[ ], real_wp_ y_u[ ], real_wp_ z_l[ ], real_wp_ z_u[ ])
- void presolve_restore_solution (void **data, int *status, int n_in, int m_in, const real_wp_ x_in[ ], const real_wp_ c_in[ ], const real_wp_ y_in[ ], const real_wp_ z_in[ ], int n, int m, real_wp_ x[ ], real_wp_ c[ ], real_wp_ y[ ], real_wp_ z[ ])
- void presolve_information (void **data, struct presolve_inform_type *inform, int *status)
- void presolve_terminate (void **data, struct presolve_control_type *control, struct presolve_inform_type *inform)

3.1.1 Data Structure Documentation

3.1.1.1 struct presolve_control_type

control derived type as a C struct

Examples

presolvet.c, and presolvetf.c.

Data Fields

bool	f_indexing	use C or Fortran sparse matrix indexing
int	termination	Determines the strategy for terminating the presolve analysis. Possible values are: • 1 presolving is continued as long as one of the sizes of the problem (n, m, a_ne, or h_ne) is being reduced; • 2 presolving is continued as long as problem transformations remain possible. NOTE: the maximum number of analysis passes (control.max_nbr_passes) and the maximum number of problem transformations (control.max_nbr_transforms) set an upper limit on the presolving effort irrespective of the choice of control.termination. The only effect of this latter parameter is to allow for early termination.
int	max_nbr_transforms	The maximum number of problem transformations, cumulated over all calls to `presolve`.
int	max_nbr_passes	The maximum number of analysis passes for problem analysis during a single call of `presolve_transform_problem`.
real_wp_	c_accuracy	The relative accuracy at which the general linear constraints are satisfied at the exit of the solver. Note that this value is not used before the restoration of the problem.
real_wp_	z_accuracy	The relative accuracy at which the dual feasibility constraints are satisfied at the exit of the solver. Note that this value is not used before the restoration of the problem.
real_wp_	infinity	The value beyond which a number is deemed equal to plus infinity (minus infinity being defined as its opposite)
int	out	The unit number associated with the device used for printout.
int	errout	The unit number associated with the device used for error ouput.
int	print_level	The level of printout requested by the user. Can take the values: • 0 no printout is produced • 1 only reports the major steps in the analysis • 2 reports the identity of each problem transformation • 3 reports more details • 4 reports lots of information. • 5 reports a completely silly amount of information
bool	dual_transformations	true if dual transformations of the problem are allowed. Note that this implies that the reduced problem is solved accurately (for the dual feasibility condition to hold) as to be able to restore the problem to the original constraints and variables. false prevents dual transformations to be applied, thus allowing for inexact solution of the reduced problem. The setting of this control parameter overides that of get_z, get_z_bounds, get_y, get_y_bounds, dual_constraints_freq, singleton_columns_freq, doubleton_columns_freq, z_accuracy, check_dual_feasibility.
bool	redundant_xc	true if the redundant variables and constraints (i.e. variables that do not appear in the objective function and appear with a consistent sign in the constraints) are to be removed with their associated constraints before other transformations are attempted.

Data Fields

int	primal_constraints_freq	The frequency of primal constraints analysis in terms of presolving passes. A value of j = 2 indicates that primal constraints are analyzed every 2 presolving passes. A zero value indicates that they are never analyzed.
int	dual_constraints_freq	The frequency of dual constraints analysis in terms of presolving passes. A value of j = 2 indicates that dual constraints are analyzed every 2 presolving passes. A zero value indicates that they are never analyzed.
int	singleton_columns_freq	The frequency of singleton column analysis in terms of presolving passes. A value of j = 2 indicates that singleton columns are analyzed every 2 presolving passes. A zero value indicates that they are never analyzed.
int	doubleton_columns_freq	The frequency of doubleton column analysis in terms of presolving passes. A value of j indicates that doubleton columns are analyzed every 2 presolving passes. A zero value indicates that they are never analyzed.
int	unc_variables_freq	The frequency of the attempts to fix linearly unconstrained variables, expressed in terms of presolving passes. A value of j = 2 indicates that attempts are made every 2 presolving passes. A zero value indicates that no attempt is ever made.
int	dependent_variables_freq	The frequency of search for dependent variables in terms of presolving passes. A value of j = 2 indicates that dependent variables are searched for every 2 presolving passes. A zero value indicates that they are never searched for.
int	sparsify_rows_freq	The frequency of the attempts to make A sparser in terms of presolving passes. A value of j = 2 indicates that attempts are made every 2 presolving passes. A zero value indicates that no attempt is ever made.
int	max_fill	The maximum percentage of fill in each row of A. Note that this is a row-wise measure: globally fill never exceeds the storage initially used for A, no matter how large control.max_fill is chosen. If max_fill is negative, no limit is put on row fill.
int	transf_file_nbr	The unit number to be associated with the file(s) used for saving problem transformations on a disk file.
int	transf_buffer_size	The number of transformations that can be kept in memory at once (that is without being saved on a disk file).
int	transf_file_status	The exit status of the file where problem transformations are saved: • 0 the file is not deleted after program termination • 1 the file is not deleted after program termination
char	transf_file_name[31]	The name of the file (to be) used for storing problem transformation on disk. NOTE: this parameter must be identical for all calls to `presolve` following `presolve_read_specfile`. It can then only be changed after calling presolve_terminate.

Data Fields

int	y_sign	Determines the convention of sign used for the multipliers associated with the general linear constraints. • 1 All multipliers corresponding to active inequality constraints are non-negative for lower bound constraints and non-positive for upper bounds constraints. • -1 All multipliers corresponding to active inequality constraints are non-positive for lower bound constraints and non-negative for upper bounds constraints.
int	inactive_y	Determines whether or not the multipliers corresponding to constraints that are inactive at the unreduced point corresponding to the reduced point on input to `presolve_restore_solution` must be set to zero. Possible values are: associated with the general linear constraints. • 0 All multipliers corresponding to inactive inequality constraints are forced to zero, possibly at the expense of deteriorating the dual feasibility condition. • 1 Multipliers corresponding to inactive inequality constraints are left unaltered.
int	z_sign	Determines the convention of sign used for the dual variables associated with the bound constraints. • 1 All dual variables corresponding to active lower bounds are non-negative, and non-positive for active upper bounds. • -1 All dual variables corresponding to active lower bounds are non-positive, and non-negative for active upper bounds.
int	inactive_z	Determines whether or not the dual variables corresponding to bounds that are inactive at the unreduced point corresponding to the reduced point on input to `presolve_restore_solution` must be set to zero. Possible values are: associated with the general linear constraints. • 0: All dual variables corresponding to inactive bounds are forced to zero, possibly at the expense of deteriorating the dual feasibility condition. • 1 Dual variables corresponding to inactive bounds are left unaltered.

Data Fields

int	final_x_bounds	The type of final bounds on the variables returned by the package. This parameter can take the values: • 0 the final bounds are the tightest bounds known on the variables (at the risk of being redundant with other constraints, which may cause degeneracy); • 1 the best known bounds that are known to be non-degenerate. This option implies that an additional real workspace of size 2 $*$ n must be allocated. • 2 the loosest bounds that are known to keep the problem equivalent to the original problem. This option also implies that an additional real workspace of size 2 $*$ n must be allocated. NOTE: this parameter must be identical for all calls to presolve (except presolve_initialize).
int	final_z_bounds	The type of final bounds on the dual variables returned by the package. This parameter can take the values: • 0 the final bounds are the tightest bounds known on the dual variables (at the risk of being redundant with other constraints, which may cause degeneracy); • 1 the best known bounds that are known to be non-degenerate. This option implies that an additional real workspace of size 2 $*$ n must be allocated. • 2 the loosest bounds that are known to keep the problem equivalent to the original problem. This option also implies that an additional real workspace of size 2 $*$ n must be allocated. NOTE: this parameter must be identical for all calls to presolve (except presolve_initialize).
int	final_c_bounds	The type of final bounds on the constraints returned by the package. This parameter can take the values: • 0 the final bounds are the tightest bounds known on the constraints (at the risk of being redundant with other constraints, which may cause degeneracy); • 1 the best known bounds that are known to be non-degenerate. This option implies that an additional real workspace of size 2 $*$ m must be allocated. • 2 the loosest bounds that are known to keep the problem equivalent to the original problem. This option also implies that an additional real workspace of size 2 $*$ n must be allocated. NOTES: 1) This parameter must be identical for all calls to presolve (except presolve_initialize). 2) If different from 0, its value must be identical to that of control.final_x_bounds.

Data Fields

int	final_y_bounds	The type of final bounds on the multipliers returned by the package. This parameter can take the values: • 0 the final bounds are the tightest bounds known on the multipliers (at the risk of being redundant with other constraints, which may cause degeneracy); • 1 the best known bounds that are known to be non-degenerate. This option implies that an additional real workspace of size 2 $*$ m must be allocated. • 2 the loosest bounds that are known to keep the problem equivalent to the original problem. This option also implies that an additional real workspace of size 2 $*$ n must be allocated. NOTE: this parameter must be identical for all calls to presolve (except presolve_initialize).
int	check_primal_feasibility	The level of feasibility check (on the values of x) at the start of the restoration phase. This parameter can take the values: • 0 no check at all; • 1 the primal constraints are recomputed at x and a message issued if the computed value does not match the input value, or if it is out of bounds (if control.print_level $>=$ 2); • 2 the same as for 1, but presolve is terminated if an incompatibilty is detected.
int	check_dual_feasibility	The level of dual feasibility check (on the values of x, y and z) at the start of the restoration phase. This parameter can take the values: • 0 no check at all; • 1 the dual feasibility condition is recomputed at (x, y, z) and a message issued if the computed value does not match the input value (if control.print_level $>=$ 2); • 2 the same as for 1, but presolve is terminated if an incompatibilty is detected. The last two values imply the allocation of an additional real workspace vector of size equal to the number of variables in the reduced problem.
real_wp_	pivot_tol	The relative pivot tolerance above which pivoting is considered as numerically stable in transforming the coefficient matrix A. A zero value corresponds to a totally unsafeguarded pivoting strategy (potentially unstable).
real_wp_	min_rel_improve	The minimum relative improvement in the bounds on x, y and z for a tighter bound on these quantities to be accepted in the course of the analysis. More formally, if lower is the current value of the lower bound on one of the x, y or z, and if new_lower is a tentative tighter lower bound on the same quantity, it is only accepted if. new_lower $>=$ lower + tol $*$ MAX(1, ABS(lower)), where tol = control.min_rel_improve. Similarly, a tentative tighter upper bound new_upper only replaces the current upper bound upper if new_upper $<=$ upper - tol $*$ MAX(1, ABS(upper)). Note that this parameter must exceed the machine precision significantly.

Data Fields

real_wp_	max_growth_factor	The maximum growth factor (in absolute value) that is accepted between the maximum data item in the original problem and any data item in the reduced problem. If a transformation results in this bound being exceeded, the transformation is skipped.

3.1.1.2 struct presolve_inform_type

inform derived type as a C struct

Examples

presolvet.c, and presolvetf.c.

Data Fields

int	status	The presolve exit condition. It can take the following values (symbol in parentheses is the related Fortran code): • (OK) successful exit; • 1 (MAX_NBR_TRANSF) the maximum number of problem transformation has been reached NOTE: this exit is not really an error, since the problem can nevertheless be permuted and solved. It merely signals that further problem reduction could possibly be obtained with a larger value of the parameter `control.max_nbr_transforms` • -21 (PRIMAL_INFEASIBLE) the problem is primal infeasible; • -22 (DUAL_INFEASIBLE) the problem is dual infeasible; • -23 (WRONG_G_DIMENSION) the dimension of the gradient is incompatible with the problem dimension; • -24 (WRONG_HVAL_DIMENSION) the dimension of the vector containing the entries of the Hessian is erroneously specified; • -25 (WRONG_HPTR_DIMENSION) the dimension of the vector containing the addresses of the first entry of each Hessian row is erroneously specified; • -26 (WRONG_HCOL_DIMENSION) the dimension of the vector containing the column indices of the nonzero Hessian entries is erroneously specified; • -27 (WRONG_HROW_DIMENSION) the dimension of the vector containing the row indices of the nonzero Hessian entries is erroneously specified; • -28 (WRONG_AVAL_DIMENSION) the dimension of the vector containing the entries of the Jacobian is erroneously specified; • -29 (WRONG_APTR_DIMENSION) the dimension of the vector containing the addresses of the first entry of each Jacobian row is erroneously specified; • -30 (WRONG_ACOL_DIMENSION) the dimension of the vector containing the column indices of the nonzero Jacobian entries is erroneously specified; • -31 (WRONG_AROW_DIMENSION) the dimension of the vector containing the row indices of the nonzero Jacobian entries is erroneously specified; • -32 (WRONG_X_DIMENSION) the dimension of the vector of variables is incompatible with the problem dimension; • -33 (WRONG_XL_DIMENSION) the dimension of the vector of lower bounds on the variables is incompatible with the problem dimension; • -34 (WRONG_XU_DIMENSION) the dimension of the vector of upper bounds on the variables is incompatible with the problem dimension; • -35 (WRONG_Z_DIMENSION) the dimension of the vector of dual variables is incompatible with the problem dimension; • -36 (WRONG_ZL_DIMENSION) the dimension of the vector of lower bounds on the dual variables is incompatible with the problem dimension; • -37 (WRONG_ZU_DIMENSION) the dimension of the vector of upper bounds on the dual variables is incompatible with the problem dimension;

Data Fields

int	status_continue	continuation of status (name in previous column should be status, doxygen issue): • -38 (WRONG_C_DIMENSION) the dimension of the vector of constraints values is incompatible with the problem dimension; • -39 (WRONG_CL_DIMENSION) the dimension of the vector of lower bounds on the constraints is incompatible with the problem dimension; • -40 (WRONG_CU_DIMENSION) the dimension of the vector of upper bounds on the constraints is incompatible with the problem dimension; • -41 (WRONG_Y_DIMENSION) the dimension of the vector of multipliers values is incompatible with the problem dimension; • -42 (WRONG_YL_DIMENSION) the dimension of the vector of lower bounds on the multipliers is incompatible with the problem dimension; • -43 (WRONG_YU_DIMENSION) the dimension of the vector of upper bounds on the multipliers is incompatible with the problem dimension; • -44 (STRUCTURE_NOT_SET) the problem structure has not been set or has been cleaned up before an attempt to analyze; • -45 (PROBLEM_NOT_ANALYZED) the problem has not been analyzed before an attempt to permute it; • -46 (PROBLEM_NOT_PERMUTED) the problem has not been permuted or fully reduced before an attempt to restore it • -47 (H_MISSPECIFIED) the column indices of a row of the sparse Hessian are not in increasing order, in that they specify an entry above the diagonal; • -48 (CORRUPTED_SAVE_FILE) one of the files containing saved problem transformations has been corrupted between writing and reading; • -49 (WRONG_XS_DIMENSION) the dimension of the vector of variables' status is incompatible with the problem dimension; • -50 (WRONG_CS_DIMENSION) the dimension of the vector of constraints' status is incompatible with the problem dimension; • -52 (WRONG_N) the problem does not contain any (active) variable; • -53 (WRONG_M) the problem contains a negative number of constraints; • -54 (SORT_TOO_LONG) the vectors are too long for the sorting routine; • -55 (X_OUT_OF_BOUNDS) the value of a variable that is obtained by substitution from a constraint is incoherent with the variable's bounds. This may be due to a relatively loose accuracy on the linear constraints. Try to increase control.c_accuracy. • -56 (X_NOT_FEASIBLE) the value of a constraint that is obtained by recomputing its value on input of `presolve_restore_solution` from the current x is incompatible with its declared value or its bounds. This may caused the restored problem to be infeasible. • -57 (Z_NOT_FEASIBLE) the value of a dual variable that is obtained by recomputing its value on input to `presolve_restore_solution` (assuming dual feasibility) from the current values of (x, y, z) is incompatible with its declared value. This may caused the restored problem to be infeasible or suboptimal.

Data Fields

int	status_continued	continuation of status (name in previous column should be status, doxygen issue): • -58 (Z_CANNOT_BE_ZEROED) a dual variable whose value is nonzero because the corresponding primal is at an artificial bound cannot be zeroed while maintaining dual feasibility (on restoration). This can happen when (x, y, z) on input of RESTORE are not (sufficiently) optimal. • -1 (MEMORY_FULL) memory allocation failed • -2 (FILE_NOT_OPENED) a file intended for saving problem transformations could not be opened; • -3 (COULD_NOT_WRITE) an IO error occurred while saving transformations on the relevant disk file; • -4 (TOO_FEW_BITS_PER_BYTE) an integer contains less than NBRH + 1 bits. • -60 (UNRECOGNIZED_KEYWORD) a keyword was not recognized in the analysis of the specification file • -61 (UNRECOGNIZED_VALUE) a value was not recognized in the analysis of the specification file • -63 (G_NOT_ALLOCATED) the vector G has not been allocated although it has general values • -64 (C_NOT_ALLOCATED) the vector C has not been allocated although m $>$ 0 • -65 (AVAL_NOT_ALLOCATED) the vector A.val has not been allocated although m $>$ 0 • -66 (APTR_NOT_ALLOCATED) the vector A.ptr has not been allocated although m $>$ 0 and A is stored in row-wise sparse format • -67 (ACOL_NOT_ALLOCATED) the vector A.col has not been allocated although m $>$ 0 and A is stored in row-wise sparse format or sparse coordinate format • -68 (AROW_NOT_ALLOCATED) the vector A.row has not been allocated although m $>$ 0 and A is stored in sparse coordinate format • -69 (HVAL_NOT_ALLOCATED) the vector H.val has not been allocated although H.ne $>$ 0 • -70 (HPTR_NOT_ALLOCATED) the vector H.ptr has not been allocated although H.ne $>$ 0 and H is stored in row-wise sparse format • -71 (HCOL_NOT_ALLOCATED) the vector H.col has not been allocated although H.ne $>$ 0 and H is stored in row-wise sparse format or sparse coordinate format • -72 (HROW_NOT_ALLOCATED) the vector H.row has not been allocated although H.ne $>$ 0 and A is stored in sparse coordinate format • -73 (WRONG_ANE) incompatible value of A_ne • -74 (WRONG_HNE) incompatible value of H_ne
int	nbr_transforms	The final number of problem transformations, as reported to the user at exit.

Data Fields

char	message[3][81]	A few lines containing a description of the exit condition on exit of PRESOLVE, typically including more information than indicated in the description of control.status above. It is printed out on device errout at the end of execution if control.print_level >= 1.

3.1.2 Function Documentation

3.1.2.1 presolve_initialize()

```
void presolve_initialize (
            void ** data,
            struct presolve_control_type * control,
            int * status )
```

Set default control values and initialize private data

Parameters

in,out	*data*	holds private internal data
out	*control*	is a struct containing control information (see presolve_control_type)
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are (currently): • 0. The import was succesful.

Examples

presolvet.c, and presolvetf.c.

3.1.2.2 presolve_read_specfile()

```
void presolve_read_specfile (
            struct presolve_control_type * control,
            const char specfile[] )
```

Read the content of a specification file, and assign values associated with given keywords to the corresponding control parameters

Parameters

in,out	*control*	is a struct containing control information (see presolve_control_type)
in	*specfile*	is a character string containing the name of the specification file

3.1.2.3 presolve_import_problem()

```
void presolve_import_problem (
             struct presolve_control_type * control,
             void ** data,
             int * status,
             int n,
             int m,
             const char H_type[],
             int H_ne,
             const int H_row[],
             const int H_col[],
             const int H_ptr[],
             const real_wp_ H_val[],
             const real_wp_ g[],
             const real_wp_ f,
             const char A_type[],
             int A_ne,
             const int A_row[],
             const int A_col[],
             const int A_ptr[],
             const real_wp_ A_val[],
             const real_wp_ c_l[],
             const real_wp_ c_u[],
             const real_wp_ x_l[],
             const real_wp_ x_u[],
             int * n_out,
             int * m_out,
             int * H_ne_out,
             int * A_ne_out )
```

Import the initial data, and apply the presolve algorithm to report crucial characteristics of the transformed variant

Parameters

in	*control*	is a struct whose members provide control paramters for the remaining prcedures (see presolve_control_type)
in,out	*data*	holds private internal data

Parameters

in,out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The import was succesful • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The restrictions n $>$ 0 or m $>$ 0 or requirement that a type contains its relevant string 'dense', 'coordinate', 'sparse_by_rows' or 'diagonal' has been violated. • -23. An entry from the strict upper triangle of H has been specified.
in	*n*	is a scalar variable of type int, that holds the number of variables.
in	*m*	is a scalar variable of type int, that holds the number of general linear constraints.
in	*H_type*	is a one-dimensional array of type char that specifies the symmetric storage scheme used for the Hessian, H. It should be one of 'coordinate', 'sparse_by_rows', 'dense', 'diagonal', 'scaled_identity', 'identity', 'zero' or 'none', the latter pair if $H = 0$; lower or upper case variants are allowed.
in	*H_ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of H in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes.
in	*H_row*	is a one-dimensional array of size H_ne and type int, that holds the row indices of the lower triangular part of H in the sparse co-ordinate storage scheme. It need not be set for any of the other three schemes, and in this case can be NULL.
in	*H_col*	is a one-dimensional array of size H_ne and type int, that holds the column indices of the lower triangular part of H in either the sparse co-ordinate, or the sparse row-wise storage scheme. It need not be set when the dense, diagonal or (scaled) identity storage schemes are used, and in this case can be NULL.
in	*H_ptr*	is a one-dimensional array of size n+1 and type int, that holds the starting position of each row of the lower triangular part of H, as well as the total number of entries plus one, in the sparse row-wise storage scheme. It need not be set when the other schemes are used, and in this case can be NULL.
in	*H_val*	is a one-dimensional array of size h_ne and type double, that holds the values of the entries of the lower triangular part of the Hessian matrix H in any of the available storage schemes.
in	*g*	is a one-dimensional array of size n and type double, that holds the linear term g of the objective function. The j-th component of g, j = 0, ... , n-1, contains g_j.
in	*f*	is a scalar of type double, that holds the constant term f of the objective function.
in	*A_type*	is a one-dimensional array of type char that specifies the unsymmetric storage scheme used for the constraint Jacobian, A. It should be one of 'coordinate', 'sparse_by_rows' or 'dense; lower or upper case variants are allowed.
in	*A_ne*	is a scalar variable of type int, that holds the number of entries in A in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes.
in	*A_row*	is a one-dimensional array of size A_ne and type int, that holds the row indices of A in the sparse co-ordinate storage scheme. It need not be set for any of the other schemes, and in this case can be NULL.

Parameters

in	*A_col*	is a one-dimensional array of size A_ne and type int, that holds the column indices of A in either the sparse co-ordinate, or the sparse row-wise storage scheme. It need not be set when the dense or diagonal storage schemes are used, and in this case can be NULL.
in	*A_ptr*	is a one-dimensional array of size n+1 and type int, that holds the starting position of each row of A, as well as the total number of entries plus one, in the sparse row-wise storage scheme. It need not be set when the other schemes are used, and in this case can be NULL.
in	*A_val*	is a one-dimensional array of size a_ne and type double, that holds the values of the entries of the constraint Jacobian matrix A in any of the available storage schemes.
in	*c_l*	is a one-dimensional array of size m and type double, that holds the lower bounds c^l on the constraints Ax. The i-th component of c_l, i = 0, ... , m-1, contains c_i^l.
in	*c_u*	is a one-dimensional array of size m and type double, that holds the upper bounds c^l on the constraints Ax. The i-th component of c_u, i = 0, ... , m-1, contains c_i^u.
in	*x_l*	is a one-dimensional array of size n and type double, that holds the lower bounds x^l on the variables x. The j-th component of x_l, j = 0, ... , n-1, contains x_j^l.
in	*x_u*	is a one-dimensional array of size n and type double, that holds the upper bounds x^l on the variables x. The j-th component of x_u, j = 0, ... , n-1, contains x_j^l.
out	*n_out*	is a scalar variable of type int, that holds the number of variables in the transformed problem.
out	*m_out*	is a scalar variable of type int, that holds the number of general linear constraints in the transformed problem.
out	*H_ne_out*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of H in the transformed problem.
out	*A_ne_out*	is a scalar variable of type int, that holds the number of entries in A in the transformed problem.

Examples

presolvet.c, and presolvetf.c.

3.1.2.4 presolve_transform_problem()

```
void presolve_transform_problem (
             void ** data,
             int * status,
             int n,
             int m,
             int H_ne,
             int H_col[],
             int H_ptr[],
             real_wp_ H_val[],
             real_wp_ g[],
             real_wp_ * f,
             int A_ne,
             int A_col[],
             int A_ptr[],
```

```
real_wp_ A_val[],
real_wp_ c_l[],
real_wp_ c_u[],
real_wp_ x_l[],
real_wp_ x_u[],
real_wp_ y_l[],
real_wp_ y_u[],
real_wp_ z_l[],
real_wp_ z_u[] )
```

Apply the presolve algorithm to simplify the input problem, and output the transformed variant

Parameters

in,out	*data*	holds private internal data
in,out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The import was succesful • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The input values n, m, A_ne or H_ne do not agree with those output as necessary from presolve_import_problem.
out	*n*	is a scalar variable of type int, that holds the number of variables in the transformed problem. This must match the value n_out from the last call to presolve_import_problem.
out	*m*	is a scalar variable of type int, that holds the number of general linear constraints. This must match the value m_out from the last call to presolve_import_problem.
out	*H_ne*	is a scalar variable of type int, that holds the number of entries in the lower triangular part of the transformed H. This must match the value H_ne_out from the last call to presolve_import_problem.
out	*H_col*	is a one-dimensional array of size H_ne and type int, that holds the column indices of the lower triangular part of the transformed H in the sparse row-wise storage scheme.
out	*H_ptr*	is a one-dimensional array of size n+1 and type int, that holds the starting position of each row of the lower triangular part of the transformed H in the sparse row-wise storage scheme.
out	*H_val*	is a one-dimensional array of size h_ne and type double, that holds the values of the entries of the lower triangular part of the the transformed Hessian matrix H in the sparse row-wise storage scheme.
out	*g*	is a one-dimensional array of size n and type double, that holds the the transformed linear term g of the objective function. The j-th component of g, j = 0, ... , n-1, contains g_j.
out	*f*	is a scalar of type double, that holds the transformed constant term f of the objective function.
out	*A_ne*	is a scalar variable of type int, that holds the number of entries in the transformed A. This must match the value A_ne_out from the last call to presolve_import_problem.
out	*A_col*	is a one-dimensional array of size A_ne and type int, that holds the column indices of the transformed A in the sparse row-wise storage scheme.

Parameters

out	*A_ptr*	is a one-dimensional array of size n+1 and type int, that holds the starting position of each row of the transformed A, as well as the total number of entries plus one, in the sparse row-wise storage scheme.
out	*A_val*	is a one-dimensional array of size a_ne and type double, that holds the values of the entries of the transformed constraint Jacobian matrix A in the sparse row-wise storage scheme.
out	*c_l*	is a one-dimensional array of size m and type double, that holds the transformed lower bounds c^l on the constraints Ax. The i-th component of c_l, i = 0, ... , m-1, contains c_i^l.
out	*c_u*	is a one-dimensional array of size m and type double, that holds the transformed upper bounds c^l on the constraints Ax. The i-th component of c_u, i = 0, ... , m-1, contains c_i^u.
out	*x_l*	is a one-dimensional array of size n and type double, that holds the transformed lower bounds x^l on the variables x. The j-th component of x_l, j = 0, ... , n-1, contains x_j^l.
out	*x_u*	is a one-dimensional array of size n and type double, that holds the transformed upper bounds x^l on the variables x. The j-th component of x_u, j = 0, ... , n-1, contains x_j^l.
out	*y_l*	is a one-dimensional array of size m and type double, that holds the implied lower bounds y^l on the transformed Lagrange multipliers y. The i-th component of y_l, i = 0, ... , m-1, contains y_i^l.
out	*y_u*	is a one-dimensional array of size m and type double, that holds the implied upper bounds y^u on the transformed Lagrange multipliers y. The i-th component of y_u, i = 0, ... , m-1, contains y_i^u.
out	*z_l*	is a one-dimensional array of size m and type double, that holds the implied lower bounds y^l on the transformed dual variables z. The j-th component of z_l, j = 0, ... , n-1, contains z_i^l.
out	*z_u*	is a one-dimensional array of size m and type double, that holds the implied upper bounds y^u on the transformed dual variables z. The j-th component of z_u, j = 0, ... , n-1, contains z_i^u.

Examples

presolvet.c, and presolvetf.c.

3.1.2.5 presolve_restore_solution()

```
void presolve_restore_solution (
             void ** data,
             int * status,
             int n_in,
             int m_in,
             const real_wp_ x_in[],
             const real_wp_ c_in[],
             const real_wp_ y_in[],
             const real_wp_ z_in[],
             int n,
             int m,
             real_wp_ x[],
             real_wp_ c[],
             real_wp_ y[],
             real_wp_ z[] )
```

Given the solution (x_in,c_in,y_in,z_in) to the transformed problem, restore to recover the solution (x,c,y,z) to the original

Parameters

in,out	*data*	holds private internal data
in,out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are: • 0. The import was succesful • -1. An allocation error occurred. A message indicating the offending array is written on unit control.error, and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -2. A deallocation error occurred. A message indicating the offending array is written on unit control.error and the returned allocation status and a string containing the name of the offending array are held in inform.alloc_status and inform.bad_alloc respectively. • -3. The input values n, m, n_in and m_in do not agree with those input to and output as necessary from presolve_import_problem.
out	*n_in*	is a scalar variable of type int, that holds the number of variables in the transformed problem. This must match the value n_out from the last call to presolve_import_problem.
out	*m_in*	is a scalar variable of type int, that holds the number of general linear constraints. This must match the value m_out from the last call to presolve_import_problem.
in	*x_in*	is a one-dimensional array of size n_in and type double, that holds the transformed values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
in	*c_in*	is a one-dimensional array of size m and type double, that holds the transformed residual $c(x)$. The i-th component of c, j = 0, ... , n-1, contains $c_j(x)$.
in	*y_in*	is a one-dimensional array of size n_in and type double, that holds the values y of the transformed Lagrange multipliers for the general linear constraints. The j-th component of y, j = 0, ... , n-1, contains y_j.
in	*z_in*	is a one-dimensional array of size n_in and type double, that holds the values z of the transformed dual variables. The j-th component of z, j = 0, ... , n-1, contains z_j.
in	*n*	is a scalar variable of type int, that holds the number of variables in the transformed problem. This must match the value n as input to presolve_import_problem.
in	*m*	is a scalar variable of type int, that holds the number of general linear constraints. This must match the value m as input to presolve_import_problem.
out	*x*	is a one-dimensional array of size n and type double, that holds the transformed values x of the optimization variables. The j-th component of x, j = 0, ... , n-1, contains x_j.
out	*c*	is a one-dimensional array of size m and type double, that holds the transformed residual $c(x)$. The i-th component of c, j = 0, ... , n-1, contains $c_j(x)$.
out	*y*	is a one-dimensional array of size n and type double, that holds the values y of the transformed Lagrange multipliers for the general linear constraints. The j-th component of y, j = 0, ... , n-1, contains y_j.
out	*z*	is a one-dimensional array of size n and type double, that holds the values z of the transformed dual variables. The j-th component of z, j = 0, ... , n-1, contains z_j.

Examples

presolvet.c, and presolvetf.c.

3.1.2.6 presolve_information()

```
void presolve_information (
            void ** data,
            struct presolve_inform_type * inform,
            int * status )
```

Provides output information

Parameters

in,out	*data*	holds private internal data
out	*inform*	is a struct containing output information (see presolve_inform_type)
out	*status*	is a scalar variable of type int, that gives the exit status from the package. Possible values are (currently): • 0. The values were recorded succesfully

Examples

presolvet.c, and presolvetf.c.

3.1.2.7 presolve_terminate()

```
void presolve_terminate (
            void ** data,
            struct presolve_control_type * control,
            struct presolve_inform_type * inform )
```

Deallocate all internal private storage

Parameters

in,out	*data*	holds private internal data
out	*control*	is a struct containing control information (see presolve_control_type)
out	*inform*	is a struct containing output information (see presolve_inform_type)

Examples

presolvet.c, and presolvetf.c.

Chapter 4

Example Documentation

4.1 presolvet.c

This is an example of how to use the package to solve a quadratic program. A variety of supported Hessian and constraint matrix storage formats are shown.

Notice that C-style indexing is used, and that this is flaggeed by setting `control.f_indexing` to `false`.

```
/* presolvet.c */
/* Full test for the PRESOLVE C interface using C sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "galahad_presolve.h"
int main(void) {
    // Derived types
    void *data;
    struct presolve_control_type control;
    struct presolve_inform_type inform;
    // Set problem data
    int n = 6; // dimension
    int m = 5; // number of general constraints
    int H_ne = 1; // Hesssian elements
    int H_row[] = {0};    // row indices, NB lower triangle
    int H_col[] = {0};     // column indices, NB lower triangle
    int H_ptr[] = {0, 1, 1, 1, 1, 1, 1}; // row pointers
    double H_val[] = {1.0};    // values
    double g[] = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0}; // linear term in the objective
    double f = 1.0;  // constant term in the objective
    int A_ne = 8; // Jacobian elements
    int A_row[] = {2, 2, 2, 3, 3, 4, 4, 4}; // row indices
    int A_col[] = {2, 3, 4, 2, 5, 3, 4, 5}; // column indices
    int A_ptr[] = {0, 0, 0, 3, 5, 8}; // row pointers
    double A_val[] = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0}; // values
    double c_l[] = { 0.0, 0.0, 2.0, 1.0, 3.0};    // constraint lower bound
    double c_u[] = {1.0, 1.0, 3.0, 3.0, 3.0};    // constraint upper bound
    double x_l[] = {-3.0, 0.0, 0.0, 0.0, 0.0, 0.0}; // variable lower bound
    double x_u[] = {3.0, 1.0, 1.0, 1.0, 1.0, 1.0}; // variable upper bound
    // Set output storage
    char st;
    int status;
    printf(" C sparse matrix indexing\n\n");
    printf(" basic tests of qp storage formats\n\n");
    for( int d=1; d <= 7; d++){
      int n_trans, m_trans, H_ne_trans, A_ne_trans;
        // Initialize PRESOLVE
        presolve_initialize( &data, &control, &status );
        // Set user-defined control options
        control.f_indexing = false; // C sparse matrix indexing
        switch(d){
            case 1: // sparse co-ordinate storage
                st = 'C';
                presolve_import_problem( &control, &data, &status, n, m,
                          "coordinate", H_ne, H_row, H_col, NULL, H_val, g, f,
                          "coordinate", A_ne, A_row, A_col, NULL, A_val,
                          c_l, c_u, x_l, x_u,
```

```
                    &n_trans, &m_trans, &H_ne_trans, &A_ne_trans );
        break;
    printf(" case %1i break\n",d);
    case 2: // sparse by rows
        st = 'R';
        presolve_import_problem( &control, &data, &status, n, m,
               "sparse_by_rows", H_ne, NULL, H_col, H_ptr, H_val, g, f,
               "sparse_by_rows", A_ne, NULL, A_col, A_ptr, A_val,
               c_l, c_u, x_l, x_u,
               &n_trans, &m_trans, &H_ne_trans, &A_ne_trans );
        break;
    case 3: // dense
        st = 'D';
        int H_dense_ne = n*(n+1)/2; // number of elements of H
        int A_dense_ne = m*n; // number of elements of A
        double H_dense[] = {1.0,
                            0.0, 0.0,
                            0.0, 0.0, 0.0,
                            0.0, 0.0, 0.0, 0.0,
                            0.0, 0.0, 0.0, 0.0, 0.0,
                            0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
        double A_dense[] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                            0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                            0.0, 0.0, 1.0, 1.0, 1.0, 0.0,
                            0.0, 0.0, 1.0, 0.0, 0.0, 1.0,
                            0.0, 0.0, 0.0, 1.0, 1.0, 1.0};
        presolve_import_problem( &control, &data, &status, n, m,
                    "dense", H_dense_ne, NULL, NULL, NULL, H_dense, g,
                    f, "dense", A_dense_ne, NULL, NULL, NULL, A_dense,
                    c_l, c_u, x_l, x_u,
                    &n_trans, &m_trans, &H_ne_trans, &A_ne_trans );
        break;
    case 4: // diagonal
        st = 'L';
        presolve_import_problem( &control, &data, &status, n, m,
                    "diagonal", n, NULL, NULL, NULL, H_val, g, f,
                    "sparse_by_rows", A_ne, NULL, A_col, A_ptr, A_val,
                    c_l, c_u, x_l, x_u,
                    &n_trans, &m_trans, &H_ne_trans, &A_ne_trans );
        break;
    case 5: // scaled identity
        st = 'S';
        presolve_import_problem( &control, &data, &status, n, m,
                "scaled_identity", 1, NULL, NULL, NULL, H_val, g, f,
                "sparse_by_rows", A_ne, NULL, A_col, A_ptr, A_val,
                c_l, c_u, x_l, x_u,
                &n_trans, &m_trans, &H_ne_trans, &A_ne_trans );
        break;
    case 6: // identity
        st = 'I';
        presolve_import_problem( &control, &data, &status, n, m,
                    "identity", 0, NULL, NULL, NULL, NULL, g, f,
                    "sparse_by_rows", A_ne, NULL, A_col, A_ptr, A_val,
                    c_l, c_u, x_l, x_u,
                    &n_trans, &m_trans, &H_ne_trans, &A_ne_trans );
        break;
    case 7: // zero
        st = 'Z';
        presolve_import_problem( &control, &data, &status, n, m,
                    "zero", 0, NULL, NULL, NULL, NULL, g, f,
                    "sparse_by_rows", A_ne, NULL, A_col, A_ptr, A_val,
                    c_l, c_u, x_l, x_u,
                    &n_trans, &m_trans, &H_ne_trans, &A_ne_trans );
        break;
    }
//printf("%c: n, m, h_ne, a_ne = %2i, %2i, %2i, %2i\n",
//           st, n_trans, m_trans, H_ne_trans, A_ne_trans);
double f_trans;  // transformed constant term in the objective
int H_ptr_trans[n_trans+1]; // transformed Hessian row pointers
int H_col_trans[H_ne_trans]; // transformed Hessian column indices
double H_val_trans[H_ne_trans]; // transformed Hessian values
double g_trans[n_trans]; // transformed gradient
int A_ptr_trans[m_trans+1]; // transformed Jacobian row pointers
int A_col_trans[A_ne_trans]; // transformed Jacobian column indices
double A_val_trans[A_ne_trans]; // transformed Jacobian values
double x_l_trans[n_trans]; // transformed lower variable bounds
double x_u_trans[n_trans]; // transformed upper variable bounds
double c_l_trans[m_trans]; // transformed lower constraint bounds
double c_u_trans[m_trans]; // transformed upper constraint bounds
double y_l_trans[m_trans]; // transformed lower multiplier bounds
double y_u_trans[m_trans]; // transformed upper multiplier bounds
double z_l_trans[n_trans]; // transformed lower dual variable bounds
double z_u_trans[n_trans]; // transformed upper dual variable bounds
presolve_transform_problem( &data, &status, n_trans, m_trans,
                    H_ne_trans, H_col_trans, H_ptr_trans,
                    H_val_trans, g_trans, &f_trans, A_ne_trans,
                    A_col_trans, A_ptr_trans, A_val_trans,
```

```
                            c_l_trans, c_u_trans, x_l_trans, x_u_trans,
                            y_l_trans, y_u_trans, z_l_trans, z_u_trans );
        double x_trans[n_trans]; // transformed variables
        for( int i = 0; i < n_trans; i++) x_trans[i] = 0.0;
        double c_trans[m_trans]; // transformed constraints
        for( int i = 0; i < m_trans; i++) c_trans[i] = 0.0;
        double y_trans[m_trans]; // transformed Lagrange multipliers
        for( int i = 0; i < m_trans; i++) y_trans[i] = 0.0;
        double z_trans[n_trans]; // transformed dual variables
        for( int i = 0; i < n_trans; i++) z_trans[i] = 0.0;
        double x[n]; // primal variables
        double c[m]; // constraint values
        double y[m]; // Lagrange multipliers
        double z[n]; // dual variables
        //printf("%c: n_trans, m_trans, n, m = %2i, %2i, %2i, %2i\n",
        //           st, n_trans, m_trans, n, m );
        presolve_restore_solution( &data, &status, n_trans, m_trans,
                  x_trans, c_trans, y_trans, z_trans, n, m, x, c, y, z );
        presolve_information( &data, &inform, &status );
        if(inform.status == 0){
            printf("%c:%6i transformations, n, m = %2i, %2i, status = %1i\n",
                   st, inform.nbr_transforms, n_trans, m_trans, inform.status);
        }else{
            printf("%c: PRESOLVE_solve exit status = %1i\n", st, inform.status);
        }
        //printf("x: ");
        //for( int i = 0; i < n; i++) printf("%f ", x[i]);
        //printf("\n");
        //printf("gradient: ");
        //for( int i = 0; i < n; i++) printf("%f ", g[i]);
        //printf("\n");
        // Delete internal workspace
        presolve_terminate( &data, &control, &inform );
    }
}
```

4.2 presolvetf.c

This is the same example, but now fortran-style indexing is used.

```
/* presolvetf.c */
/* Full test for the PRESOLVE C interface using Fortran sparse matrix indexing */
#include <stdio.h>
#include <math.h>
#include "galahad_presolve.h"
int main(void) {
    // Derived types
    void *data;
    struct presolve_control_type control;
    struct presolve_inform_type inform;
    // Set problem data
    int n = 6; // dimension
    int m = 5; // number of general constraints
    int H_ne = 1; // Hesssian elements
    int H_row[] = {1};    // row indices, NB lower triangle
    int H_col[] = {1};    // column indices, NB lower triangle
    int H_ptr[] = {1, 2, 2, 2, 2, 2, 2}; // row pointers
    double H_val[] = {1.0};   // values
    double g[] = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0}; // linear term in the objective
    double f = 1.0;  // constant term in the objective
    int A_ne = 8; // Jacobian elements
    int A_row[] = {3, 3, 3, 4, 4, 5, 5, 5}; // row indices
    int A_col[] = {3, 4, 5, 3, 6, 4, 5, 6}; // column indices
    int A_ptr[] = {1, 1, 1, 4, 6, 9}; // row pointers
    double A_val[] = {1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0}; // values
    double c_l[] = { 0.0, 0.0, 2.0, 1.0, 3.0};   // constraint lower bound
    double c_u[] = {1.0, 1.0, 3.0, 3.0, 3.0};   // constraint upper bound
    double x_l[] = {-3.0, 0.0, 0.0, 0.0, 0.0, 0.0}; // variable lower bound
    double x_u[] = {3.0, 1.0, 1.0, 1.0, 1.0, 1.0}; // variable upper bound
    // Set output storage
    char st;
    int status;
    printf(" Fortran sparse matrix indexing\n\n");
    printf(" basic tests of qp storage formats\n\n");
    for( int d=1; d <= 7; d++){
      int n_trans, m_trans, H_ne_trans, A_ne_trans;
        // Initialize PRESOLVE
        presolve_initialize( &data, &control, &status );
        // Set user-defined control options
        control.f_indexing = true; // Fortran sparse matrix indexing
        switch(d){
```

```
    case 1: // sparse co-ordinate storage
        st = 'C';
        presolve_import_problem( &control, &data, &status, n, m,
                  "coordinate", H_ne, H_row, H_col, NULL, H_val, g, f,
                  "coordinate", A_ne, A_row, A_col, NULL, A_val,
                  c_l, c_u, x_l, x_u,
                  &n_trans, &m_trans, &H_ne_trans, &A_ne_trans );
        break;
    printf(" case %1i break\n",d);
    case 2: // sparse by rows
        st = 'R';
        presolve_import_problem( &control, &data, &status, n, m,
               "sparse_by_rows", H_ne, NULL, H_col, H_ptr, H_val, g, f,
               "sparse_by_rows", A_ne, NULL, A_col, A_ptr, A_val,
               c_l, c_u, x_l, x_u,
               &n_trans, &m_trans, &H_ne_trans, &A_ne_trans );
        break;
    case 3: // dense
        st = 'D';
        int H_dense_ne = n*(n+1)/2; // number of elements of H
        int A_dense_ne = m*n; // number of elements of A
        double H_dense[] = {1.0,
                            0.0, 0.0,
                            0.0, 0.0, 0.0,
                            0.0, 0.0, 0.0, 0.0,
                            0.0, 0.0, 0.0, 0.0, 0.0,
                            0.0, 0.0, 0.0, 0.0, 0.0, 0.0};
        double A_dense[] = {0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                            0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
                            0.0, 0.0, 1.0, 1.0, 1.0, 0.0,
                            0.0, 0.0, 1.0, 0.0, 0.0, 1.0,
                            0.0, 0.0, 0.0, 1.0, 1.0, 1.0};
        presolve_import_problem( &control, &data, &status, n, m,
                    "dense", H_dense_ne, NULL, NULL, NULL, H_dense, g,
                    f, "dense", A_dense_ne, NULL, NULL, NULL, A_dense,
                    c_l, c_u, x_l, x_u,
                    &n_trans, &m_trans, &H_ne_trans, &A_ne_trans );
        break;
    case 4: // diagonal
        st = 'L';
        presolve_import_problem( &control, &data, &status, n, m,
                    "diagonal", n, NULL, NULL, NULL, H_val, g, f,
                    "sparse_by_rows", A_ne, NULL, A_col, A_ptr, A_val,
                    c_l, c_u, x_l, x_u,
                    &n_trans, &m_trans, &H_ne_trans, &A_ne_trans );
        break;
    case 5: // scaled identity
        st = 'S';
        presolve_import_problem( &control, &data, &status, n, m,
                "scaled_identity", 1, NULL, NULL, NULL, H_val, g, f,
                "sparse_by_rows", A_ne, NULL, A_col, A_ptr, A_val,
                c_l, c_u, x_l, x_u,
                &n_trans, &m_trans, &H_ne_trans, &A_ne_trans );
        break;
    case 6: // identity
        st = 'I';
        presolve_import_problem( &control, &data, &status, n, m,
                    "identity", 0, NULL, NULL, NULL, NULL, g, f,
                    "sparse_by_rows", A_ne, NULL, A_col, A_ptr, A_val,
                    c_l, c_u, x_l, x_u,
                    &n_trans, &m_trans, &H_ne_trans, &A_ne_trans );
        break;
    case 7: // zero
        st = 'Z';
        presolve_import_problem( &control, &data, &status, n, m,
                    "zero", 0, NULL, NULL, NULL, NULL, g, f,
                    "sparse_by_rows", A_ne, NULL, A_col, A_ptr, A_val,
                    c_l, c_u, x_l, x_u,
                    &n_trans, &m_trans, &H_ne_trans, &A_ne_trans );
        break;
    }
//printf("%c: n, m, h_ne, a_ne = %2i, %2i, %2i, %2i\n",
//           st, n_trans, m_trans, H_ne_trans, A_ne_trans);
double f_trans;  // transformed constant term in the objective
int H_ptr_trans[n_trans+1]; // transformed Hessian row pointers
int H_col_trans[H_ne_trans]; // transformed Hessian column indices
double H_val_trans[H_ne_trans]; // transformed Hessian values
double g_trans[n_trans]; // transformed gradient
int A_ptr_trans[m_trans+1]; // transformed Jacobian row pointers
int A_col_trans[A_ne_trans]; // transformed Jacobian column indices
double A_val_trans[A_ne_trans]; // transformed Jacobian values
double x_l_trans[n_trans]; // transformed lower variable bounds
double x_u_trans[n_trans]; // transformed upper variable bounds
double c_l_trans[m_trans]; // transformed lower constraint bounds
double c_u_trans[m_trans]; // transformed upper constraint bounds
double y_l_trans[m_trans]; // transformed lower multiplier bounds
double y_u_trans[m_trans]; // transformed upper multiplier bounds
```

```
        double z_l_trans[n_trans]; // transformed lower dual variable bounds
        double z_u_trans[n_trans]; // transformed upper dual variable bounds
        presolve_transform_problem( &data, &status, n_trans, m_trans,
                                    H_ne_trans, H_col_trans, H_ptr_trans,
                                    H_val_trans, g_trans, &f_trans, A_ne_trans,
                                    A_col_trans, A_ptr_trans, A_val_trans,
                                    c_l_trans, c_u_trans, x_l_trans, x_u_trans,
                                    y_l_trans, y_u_trans, z_l_trans, z_u_trans );
        double x_trans[n_trans]; // transformed variables
        for( int i = 0; i < n_trans; i++) x_trans[i] = 0.0;
        double c_trans[m_trans]; // transformed constraints
        for( int i = 0; i < m_trans; i++) c_trans[i] = 0.0;
        double y_trans[m_trans]; // transformed Lagrange multipliers
        for( int i = 0; i < m_trans; i++) y_trans[i] = 0.0;
        double z_trans[n_trans]; // transformed dual variables
        for( int i = 0; i < n_trans; i++) z_trans[i] = 0.0;
        double x[n]; // primal variables
        double c[m]; // constraint values
        double y[m]; // Lagrange multipliers
        double z[n]; // dual variables
        //printf("%c: n_trans, m_trans, n, m = %2i, %2i, %2i, %2i\n",
        //           st, n_trans, m_trans, n, m );
        presolve_restore_solution( &data, &status, n_trans, m_trans,
                x_trans, c_trans, y_trans, z_trans, n, m, x, c, y, z );
        presolve_information( &data, &inform, &status );
        if(inform.status == 0){
            printf("%c:%6i transformations, n, m = %2i, %2i, status = %1i\n",
                   st, inform.nbr_transforms, n_trans, m_trans, inform.status);
        }else{
            printf("%c: PRESOLVE_solve exit status = %1i\n", st, inform.status);
        }
        //printf("x: ");
        //for( int i = 0; i < n; i++) printf("%f ", x[i]);
        //printf("\n");
        //printf("gradient: ");
        //for( int i = 0; i < n; i++) printf("%f ", g[i]);
        //printf("\n");
        // Delete internal workspace
        presolve_terminate( &data, &control, &inform );
    }
}
```

Index