GALAHAD PRESOLVE

USER DOCUMENTATION GALAHAD Optimization Library version 5.0

1 SUMMARY

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.

The package GALAHAD_PRESOLVE applies presolving techniques to a **linear**

$$\text{minimize } \ell(\mathbf{x}) = f + \mathbf{g}^T\mathbf{x} \tag{1.1}$$

or **quadratic program**

$$\text{minimize } q(\mathbf{x}) = f + \mathbf{g}^T\mathbf{x} + \tfrac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} \tag{1.2}$$

subject to the general linear constraints

$$c_i^l \leq \mathbf{a}_i^T\mathbf{x} \leq c_i^u, \quad i = 1,\ldots,m, \tag{1.3}$$

and simple bounds

$$x_j^l \leq x_j \leq x_j^u, \quad j = 1,\ldots,n, \tag{1.4}$$

where the scalar f, the n-dimensional vectors $\mathbf{g}$, $\mathbf{x}^l$ and $\mathbf{x}^u$, the m-dimensional vectors $\mathbf{c}^l$ and $\mathbf{c}^u$, the $n \times n$ symmetric matrix $\mathbf{H}$ and the $m \times n$ matrix $\mathbf{A}$ (whose rows are the vectors $\mathbf{a}_i^T$) are given. Furthermore, bounds on the Lagrange multipliers $\mathbf{y}$ associated with the general linear constraints and on the dual variables $\mathbf{z}$ associated with the simple bound constraints

$$y_i^l \leq \mathbf{y}_i \leq y_i^u, \quad i = 1,\ldots,m,$$

and

$$z_i^l \leq \mathbf{z}_i \leq z_i^u, \quad i = 1,\ldots,n,$$

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

ATTRIBUTES — Versions: GALAHAD_PRESOLVE_single, GALAHAD_PRESOLVE_double. **Uses:** GALAHAD_SMT, GALAHAD_QPT, GALAHAD_SPECFILE, GALAHAD_SORT, GALAHAD_SYMBOLS. **Date:** March 2002. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory, and Ph. L. Toint, The University of Namur, Belgium. **Language:** Fortran 95 + TR 15581 or Fortran 2003.

2 HOW TO USE THE PACKAGE

The package is available using both single and double precision reals, and either 32-bit or 64-bit integers. Access to the 32-bit integer, single precision version requires the USE statement

```
USE GALAHAD_PRESOLVE_single
```

with the obvious substitution GALAHAD_PRESOLVE_double, GALAHAD_PRESOLVE_single_64 and GALAHAD_PRESOLVE_double_64 for the other variants.

If it is required to use more than one of the modules at the same time, the derived types SMT_problem_type, QPT_problem_type, PRESOLVE_control_type, PRESOLVE_inform_type and PRESOLVE_data_type (Section 2.3) and the five subroutines PRESOLVE_initialize, PRESOLVE_read_specfile, PRESOLVE_apply, PRESOLVE_restore, PRESOLVE_terminate, (Section 2.4) must be renamed on one of the USE statements.

2.1 Matrix storage formats

Both the Hessian matrix **H** and the constraint Jacobian **A** may be stored in a variety of input formats.

2.1.1 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. Component $n*(i-1)+j$ of the storage array `A%val` will hold the value a_{ij} for $i = 1,\ldots,m$, $j = 1,\ldots,n$. Since **H** is symmetric, only the lower triangular part (that is the part h_{ij} for $1 \leq j \leq i \leq n$) need be held. In this case the lower triangle will be stored by rows, that is component $i*(i-1)/2+j$ of the storage array `H%val` will hold the value h_{ij} (and, by symmetry, h_{ji}) for $1 \leq j \leq i \leq n$.

2.1.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l-th entry of **A**, its row index i, column index j and value a_{ij} are stored in the l-th components of the integer arrays `A%row`, `A%col` and real array `A%val`, respectively. The order is unimportant, but the total number of entries `A%ne` is also required. The same scheme is applicable to **H** (thus requiring integer arrays `H%row`, `H%col`, a real array `H%val` and an integer value `H%ne`), except that only the entries in the lower triangle need be stored.

2.1.3 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 a integer array `A%ptr` holds the position of the first entry in this row, while `A%ptr` $(m+1)$ holds the total number of entries plus one. The column indices j and values a_{ij} of the entries in the i-th row are stored in components $l =$ `A%ptr`(i), $\ldots$,`A%ptr` $(i+1)-1$ of the integer array `A%col`, and real array `A%val`, respectively. The same scheme is applicable to **H** (thus requiring integer arrays `H%ptr`, `H%col`, and a real array `H%val`), except that only the entries in the lower triangle need be stored.

For sparse matrices, this scheme almost always requires less storage than its predecessor.

2.1.4 Diagonal storage format

If **H** is diagonal (i.e., $h_{ij} = 0$ for all $1 \leq i \neq j \leq n$) only the diagonals entries h_{ii}, $1 \leq i \leq n$, need be stored, and the first n components of the array `H%val` may be used for the purpose. There is no sensible equivalent for the non-square **A**.

2.1.5 Zero storage format

If **H** is the zero matrix (i.e., $h_{ij} = 0$ for all $1 \leq i \leq j \leq n$), no entries need be stored.

2.2 Real and integer kinds

We use the terms integer and real to refer to the fortran keywords `REAL(rp_)` and `INTEGER(ip_)`, where `rp_` and `ip_` are the relevant kind values for the real and integer types employed by the particular module in use. The former are equivalent to default `REAL` for the single precision versions and `DOUBLE PRECISION` for the double precision cases, and correspond to `rp_ = real32` and `rp_ = real64`, respectively, as supplied by the fortran `iso_fortran_env` module. The latter are default (32-bit) and long (64-bit) integers, and correspond to `ip_ = int32` and `ip_ = int64`, respectively, again from the `iso_fortran_env` module.

2.3 The derived data types

Six derived data types are accessible from the package.

2.3.1 The derived data type for holding matrices

The derived data type SMT_TYPE is used to hold the matrices **A** and **H**. The components of SMT_TYPE used here are:

m is a scalar component of type INTEGER(ip_), that holds the number of rows in the matrix.

n is a scalar component of type INTEGER(ip_), that holds the number of columns in the matrix.

ne is a scalar variable of type INTEGER(ip_), that either holds the number of matrix entries or is used to flag the storage scheme used.

val is a rank-one allocatable array of type REAL(rp_) and dimension at least ne, that holds the values of the entries. Each pair of off-diagonal entries $h_{ij} = h_{ji}$ of a *symmetric* matrix **H** is represented as a single entry (see §2.1.1–2.1.3). Any duplicated entries that appear in the sparse co-ordinate or row-wise schemes will be summed.

row is a rank-one allocatable array of type INTEGER(ip_), and dimension at least ne, that may hold the row indices of the entries. (see §2.1.2).

col is a rank-one allocatable array of type INTEGER(ip_), and dimension at least ne, that may the column indices of the entries (see §2.1.2–2.1.3).

ptr is a rank-one allocatable array of type INTEGER(ip_), and dimension at least m + 1, that may hold the pointers to the first entry in each row (see §2.1.3).

2.3.2 The derived data type for holding the problem

The derived data type QPT_problem_type is used to hold the problem. The components of QPT_problem_type are:

new_problem_structure is a scalar variable of type default LOGICAL, that is .TRUE. if this is the first (or only) problem in a sequence of problems with identical "structure" to be attempted, and .FALSE. if a previous problem with the same "structure" (but different numerical data) has been solved. We say that two problems have the same structure if they differ only in their components of type REAL, which means that they share the same dimensions and matrix sparsity patterns. See Section 4 for a description of how the package may be applied to more than one problem with the same structure.

When a .TRUE. value is specified for this component on entry in PRESOLVE_apply (see below), this routine performs extensive checks on the consistency of the problem structure and also allocates the necessary problem dependent workspace. It is thus mandatory that the .TRUE. value is used on the first call to PRESOLVE_apply, but the .FALSE. value should be used for any subsequent call to this routine for problems with the same structure.

n is a scalar variable of type INTEGER(ip_), that holds the number of optimization variables, n.

m is a scalar variable of type INTEGER(ip_), that holds the number of general linear constraints, m.

H is scalar variable of type SMT_TYPE that holds the Hessian matrix **H**. The following components are used:

H%type is an allocatable array of rank one and type default CHARACTER, that is used to indicate the storage scheme used. If the dense storage scheme (see Section 2.1.1) is used, the first five components of H%type must contain the string DENSE. For the sparse co-ordinate scheme (see Section 2.1.2), the first ten components of H%type must contain the string COORDINATE, for the sparse row-wise storage scheme (see Section 2.1.3), the first fourteen components of H%type must contain the string SPARSE_BY_ROWS, for the

diagonal storage scheme (see Section 2.1.4), the first eight components of H%type must contain the string DIAGONAL, and for the zero storage scheme (see Section 2.1.4), the first four components of H%type must contain the string ZERO.

For convenience, the procedure SMT_put may be used to allocate sufficient space and insert the required keyword into H%type. For example, if prob is of derived type PRESOLVE_problem_type and involves a Hessian we wish to store using the co-ordinate scheme, we may simply

```
CALL SMT_put( prob%H%type, 'COORDINATE' )
```

See the documentation for the GALAHAD package SMT for further details on the use of SMT_put.

H%ne is a scalar variable of type INTEGER(ip_), that holds the number of entries in the **lower triangular** part of **H** in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be set for any of the other four schemes.

H%val is a rank-one allocatable array of type REAL(rp_), that holds the values of the entries of the **lower triangular** part of the Hessian matrix **H** in any of the storage schemes discussed in Section 2.1.

H%row is a rank-one allocatable array of type INTEGER(ip_), that holds the row indices of the **lower triangular** part of **H** in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be allocated for any of the other four schemes.

H%col is a rank-one allocatable array variable of type INTEGER(ip_), that holds the column indices of the **lower triangular** part of **H** in either the sparse co-ordinate (see Section 2.1.2), or the sparse row-wise (see Section 2.1.3) storage scheme. It need not be allocated when the dense or diagonal storage schemes are used.

H%ptr is a rank-one allocatable array of dimension n+1 and type INTEGER(ip_), 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 (see Section 2.1.3). It need not be allocated when the other schemes are used.

gradient_kind is a scalar variable of type INTEGER(ip_), that is used to indicate whether the components of the gradient **g** have special or general values. Possible values for gradient_kind are:

0 In this case, $\mathbf{g} = 0$.

1 In this case, $g_i = 1$ for $i = 1, \ldots, n$.

$\neq$ 0,1 In this case, general values of **g** will be used, and will be provided by the user in the component G.

G is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the gradient **g** of the linear term of the quadratic objective function. The j-th component of G, $j = 1, \ldots, n$, contains $\mathbf{g}_j$. If gradient_kind = 0, 1, G need not be allocated.

f is a scalar variable of type REAL(rp_), that holds the constant term, f, in the objective function.

A is scalar variable of type SMT_TYPE that holds the Jacobian matrix **A**. The following components are used:

A%type is an allocatable array of rank one and type default CHARACTER, that is used to indicate the storage scheme used. If the dense storage scheme (see Section 2.1.1) is used, the first five components of A%type must contain the string DENSE. For the sparse co-ordinate scheme (see Section 2.1.2), the first ten components of A%type must contain the string COORDINATE, while for the sparse row-wise storage scheme (see Section 2.1.3), the first fourteen components of A%type must contain the string SPARSE_BY_ROWS.

Just as for H%type above, the procedure SMT_put may be used to allocate sufficient space and insert the required keyword into A%type. Once again, if prob is of derived type PRESOLVE_problem_type and involves a Jacobian we wish to store using the sparse row-wise storage scheme, we may simply

```
CALL SMT_put( prob%A%type, 'SPARSE_BY_ROWS' )
```

A%ne is a scalar variable of type INTEGER(ip_), that holds the number of entries in **A** in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be set for either of the other two schemes.

A%val is a rank-one allocatable array of type REAL(rp_), that holds the values of the entries of the Jacobian matrix **A** in any of the storage schemes discussed in Section 2.1.

A%row is a rank-one allocatable array of type INTEGER(ip_), that holds the row indices of **A** in the sparse co-ordinate storage scheme (see Section 2.1.2). It need not be allocated for either of the other two schemes.

A%col is a rank-one allocatable array variable of type INTEGER(ip_), that holds the column indices of **A** in either the sparse co-ordinate (see Section 2.1.2), or the sparse row-wise (see Section 2.1.3) storage scheme. It need not be allocated when the dense storage scheme is used.

A%ptr is a rank-one allocatable array of dimension m+1 and type INTEGER(ip_), 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 (see Section 2.1.3). It need not be allocated when the other schemes are used.

C_l is a rank-one allocatable array of dimension m and type REAL(rp_), that holds the vector of lower bounds $\mathbf{c}^l$ on the general constraints. The i-th component of C_l, $i = 1,\ldots,m$, contains $\mathbf{c}^l_i$. Infinite bounds are allowed by setting the corresponding components of C_l to any value smaller than -infinity, where infinity is a component of the control array control (see Section 2.3.3).

C_u is a rank-one allocatable array of dimension m and type REAL(rp_), that holds the vector of upper bounds $\mathbf{c}^u$ on the general constraints. The i-th component of C_u, $i = 1,\ldots,m$, contains $\mathbf{c}^u_i$. Infinite bounds are allowed by setting the corresponding components of C_u to any value larger than infinity, where infinity is a component of the control array control (see Section 2.3.3).

C_status is a rank-one allocatable array of dimension m and type INTEGER(ip_), that holds the status of the problem constraints (active or inactive). A constraint is said to be inactive if it is not included in the formulation of the considered quadratic program.

X_l is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the vector of lower bounds $\mathbf{x}^l$ on the the variables. The j-th component of X_l, $j = 1,\ldots,n$, contains $\mathbf{x}^l_j$. Infinite bounds are allowed by setting the corresponding components of X_l to any value smaller than -infinity, where infinity is a component of the control array control (see Section 2.3.3).

X_u is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the vector of upper bounds $\mathbf{x}^u$ on the variables. The j-th component of X_u, $j = 1,\ldots,n$, contains $\mathbf{x}^u_j$. Infinite bounds are allowed by setting the corresponding components of X_u to any value larger than that infinity, where infinity is a component of the control array control (see Section 2.3.3).

X_status is a rank-one allocatable array of dimension n and type INTEGER(ip_), that holds the status of the problem variables (active or inactive). Variable j is said to be inactive if its value is fixed to the current value of problem%X(j), in which case it can be seen as a parameter of the quadratic program.

Y_l is a rank-one allocatable array of dimension m and type REAL(rp_), that holds the vector of lower bounds $\mathbf{y}^l$ on the the multipliers. The j-th component of Y_l, $j = 1,\ldots,m$, contains $\mathbf{y}^l_j$. Infinite bounds are allowed by setting the corresponding components of Y_l to any value smaller than -infinity, where infinity is a component of the control array control (see Section 2.3.3).

Y_u is a rank-one allocatable array of dimension m and type REAL(rp_), that holds the vector of upper bounds $\mathbf{y}^u$ on the multipliers. The j-th component of Y_u, $j = 1,\ldots,m$, contains $\mathbf{y}^u_j$. Infinite bounds are allowed by setting the corresponding components of Y_u to any value larger than that infinity, where infinity is a component of the control array control (see Section 2.3.3).

Z_l is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the vector of lower bounds $\mathbf{z}^l$ on the the dual variables. The j-th component of Z_l, $j = 1,\ldots,n$, contains $\mathbf{z}^l_j$. Infinite bounds are allowed by setting the corresponding components of Z_l to any value smaller than -infinity, where infinity is a component of the control array control (see Section 2.3.3).

Z_u is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the vector of upper bounds $\mathbf{z}^u$ on the dual variables. The j-th component of Z_u, $j = 1,\ldots,n$, contains $\mathbf{z}^u_j$. Infinite bounds are allowed by setting the corresponding components of Z_u to any value larger than that infinity, where infinity is a component of the control array control (see Section 2.3.3).

X is a rank-one allocatable array of dimension n and type REAL(rp_), that holds the values $\mathbf{x}$ of the optimization variables. The j-th component of X, $j = 1,\ldots,n$, contains x_j.

Z is a rank-one allocatable array of dimension n and type default REAL(rp_), that holds the values $\mathbf{z}$ of estimates of the dual variables corresponding to the simple bound constraints (see Section 4). The j-th component of Z, $j = 1,\ldots,n$, contains z_j.

C is a rank-one allocatable array of dimension m and type default REAL(rp_), that holds the values $\mathbf{Ax}$ of the constraints. The i-th component of C, $i = 1,\ldots,m$, contains $\mathbf{a}_i^T\mathbf{x} \equiv (\mathbf{Ax})_i$.

Y is a rank-one allocatable array of dimension m and type REAL(rp_), that holds the values $\mathbf{y}$ of estimates of the Lagrange multipliers corresponding to the general linear constraints (see Section 4). The i-th component of Y, $i = 1,\ldots,m$, contains y_i.

2.3.3 The derived data type for holding control parameters

The derived data type PRESOLVE_control_type is used to hold controlling data. Default values may be obtained by calling PRESOLVE_initialize (see Section 2.4.1), while individual components may also be changed by calling PRESOLVE_read_specfile (see Section 2.6.1). The components of PRESOLVE_control_type are:

termination is a scalar variable of type INTEGER(ip_), that determines the strategy for terminating the presolve analysis. Possible values are:

1. presolving continues so long as one of the sizes of the problem (n, m, sizes of $\mathbf{A}$ and $\mathbf{H}$) is being reduced.
2. presolving continues so long as further problem transformations are possible.

Note that the maximum number of analysis passes (max_nbr_passes) and the maximum number of problem transformations (max_nbr_transforms) set an upper limit on the presolving effort irrespective of the choice of termination. The only effect of this latter parameter is to allow for early termination. The default is termination = 1.

max_nbr_transforms is a scalar variable of type INTEGER(ip_), that determines the maximum number of problem transformations. The default is max_nbr_transforms = 1000000.

max_nbr_passes is a scalar variable of type INTEGER(ip_), that determines the maximum number of analysis passes for problem analysis during a single call to PRESOLVE_apply. The default is max_nbr_passes = 25.

c_accuracy is a scalar variable of type REAL(rp_), that holds 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. The default is c_accuracy = 10^{-4} in single precision, and c_accuracy = 10^{-6} in double precision.

z_accuracy is a scalar variable of type REAL(rp_), that holds 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. The default is z_accuracy = 10^{-4} in single precision, and z_accuracy = 10^{-6} in double precision.

infinity is a scalar variable of type REAL(rp_), that holds the value beyond which a number is deemed equal to plus infinity (minus infinity being defined as its opposite) The default is infinity = 10^{19}.

out is a scalar variable of type INTEGER(ip_), that holds the unit number associated with the device used for printout. The default is out = 6.

errout is a scalar variable of type INTEGER(ip_), that holds the unit number associated with the device used for error ouput. The default is errout = 6.

print_level is a scalar variable of type INTEGER(ip_), that holds the level of printout requested by the user. See Section 2.7. The default is print_level = 0.

dual_transformations is a scalar variable of type default LOGICAL, that has the value .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. The value .FALSE. prevents dual transformations to be applied, thus allowing for inexact solution of the reduced problem. The setting of this control parameter overides the values of get_z, get_z_bounds, get_y, get_y_bounds, dual_constraints_freq, singleton_columns_freq, doubleton_columns_freq, z_accuracy and check_dual_feasibility. The default is dual_transformations = .TRUE..

redundant_xc is a scalar variable of type default LOGICAL, that has the value .TRUE. if redundant variables and constraints (that is variables that don't occur in the objective function and are either unbounded above with all their coefficients in **A** being positive for constraints that are unbounded above and negative for constraints that are unbounded below, or unbounded below with all their coefficients in **A** being positive for constraints that are unbounded below or negative for all constraints that are unbounded above) are to be removed from the problem with their associated constraints before any other problem transformation is attempted. The default is redundant_xc = .TRUE..

primal_constraints_freq is a scalar variable of type INTEGER(ip_), that holds the frequency of primal constraints analysis in terms of presolving passes. A value of 2 indicates that primal constraints are analyzed every 2 presolving passes. A zero value indicates that they are never analyzed. The default is primal_constraints_freq = 1.

dual_constraints_freq is a scalar variable of type INTEGER(ip_), that holds the frequency of dual constraints analysis in terms of presolving passes. A value of 2 indicates that dual constraints are analyzed every 2 presolving passes. A zero value indicates that they are never analyzed. The default is dual_constraints_freq = 1.

singleton_columns_freq is a scalar variable of type INTEGER(ip_), that holds the frequency of singleton column analysis in terms of presolving passes. A value of 2 indicates that singleton columns are analyzed every 2 presolving passes. A zero value indicates that they are never analyzed. The default is singleton_columns_freq = 1.

doubleton_columns_freq is a scalar variable of type INTEGER(ip_), that holds 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. The default is doubleton_columns_freq = 1.

unc_variables_freq is a scalar variable of type INTEGER(ip_), that holds the frequency of the attempts to fix linearly unconstrained variables, expressed in terms of presolving passes. A value of 2 indicates that attempts are made every 2 presolving passes. A zero value indicates that no attempt is ever made. The default is unc_variables_freq = 1.

dependent_variables_freq is a scalar variable of type default INTEGER(ip_), that holds the frequency of search for dependent variables in terms of presolving passes. A value of 2 indicates that dependent variables are searched for every 2 presolving passes. A zero value indicates that no attempt is ever made to detect such variables. The default is dependent_variables_freq = 1.

sparsify_rows_freq is a scalar variable of type INTEGER(ip_), that holds the frequency of the attempts to make A sparser in terms of presolving passes. A value of 2 indicates that attempts are made every 2 presolving passes. A zero value indicates that no attempt is ever made. The default is sparsify_rows_freq = 1.

max_fill is a scalar variable of type INTEGER(ip_), that holds 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 max_fill is chosen. If max_fill is negative, no limit is put on row fill. The default is max_fill = -1 (no limit).

transf_file_nbr is a scalar variable of type INTEGER(ip_), that holds the unit number to be associated with the file(s) used for saving problem transformations on a disk file. The default is transf_file_nbr = 52.

transf_buffer_size is a scalar variable of type INTEGER(ip_), that holds the number of transformations that can be kept in memory at once (that is without being saved on a disk file). The default is transf_buffer_size = 50000.

transf_file_status is a scalar variable of type INTEGER(ip_), that holds 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.

The default is transf_file_status = 0.

transf_file_name is a scalar variable of type INTEGER(ip_), that holds the name of the file (to be) used for storing problem transformation on disk. Note that this parameter must be identical for all calls to PRESOLVE that follows PRESOLVE_read_specfile. It can then only be changed after calling PRESOLVE_terminate. The default is transf_file_name = transf.sav.

y_sign is a scalar variable of type INTEGER(ip_), that determines the convention of sign used for the multipliers associated with the general linear constraints. Possible values are:

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.

The default is y_sign = 1.

inactive_y is a scalar variable of type INTEGER(ip_), that determines whether or not the multipliers corresponding to general linear constraints that are inactive at the unreduced point corresponding to the reduced point on input of PRESOLVE_restore must be set to zero. Possible values are:

0. all multipliers corresponding to inactive general linear constraints are forced to zero, possibly at the expense of deteriorating the dual feasibility condition. Note that this option is inactive unless get_y = get_c get_c_bounds = .TRUE..

1. multipliers corresponding to inactive general linear constraints are left unaltered.

The default is inactive_y = 1.

z_sign is a scalar variable of type INTEGER(ip_), that determines the convention of sign used for the dual variables associated with the bound constraints. Possible values are:

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.

The default is z_sign = 1.

inactive_z is a scalar variable of type INTEGER(ip_), that determines whether or not the dual variables corresponding to bound constraints that are inactive at the unreduced point corresponding to the reduced point on input of PRESOLVE_restore must be set to zero. Possible values are:

0. all dual variables corresponding to inactive bounds are forced to zero, possibly at the expense of deteriorating the dual feasibility condition. Note that this option is inactive unless get_z = get_x get_x_bounds = .TRUE..

1. dual variables corresponding to inactive bounds are left unaltered.

The default is inactive_z = 1.

final_x_bounds is a scalar variable of type INTEGER(ip_), that holds 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 * problem%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 * problem%n must be allocated;

Note that his parameter must be identical for all calls to PRESOLVE following PRESOLVE_read_specfile. The default is final_x_bounds = 0.

final_z_bounds is a scalar variable of type INTEGER(ip_), that holds 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 * problem%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 * problem%n must be allocated;

Note that his parameter must be identical for all calls to PRESOLVE following PRESOLVE_read_specfile. The default is final_z_bounds = 0.

final_c_bounds is a scalar variable of type INTEGER(ip_), that holds 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 * problem%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 * problem%n must be allocated;

Note that his parameter must be identical for all calls to PRESOLVE following PRESOLVE_read_specfile. If different from 0, its value must be equal to that of final_x_bounds. The default is final_c_bounds = 0.

final_y_bounds is a scalar variable of type INTEGER(ip_), that holds 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 * problem%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 * problem%n must be allocated;

Note that his parameter must be identical for all calls to PRESOLVE following PRESOLVE_read_specfile. The default is final_y_bounds = 0.

check_primal_feasibility is a scalar variable of type default INTEGER(ip_), that holds the level of feasibility check (on the values of $\mathbf{x}$) 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 $\mathbf{x}$ and a message issued if the computed value does not match the input value, or if it is out of bounds (if print_level $\geq$ 2);

2. the same as for 1, but PRESOLVE is terminated if an incompatibilty is detected.

The default is check_primal_feasibility = 0.

check_dual_feasibility is a scalar variable of type default INTEGER(ip_), that holds the level of dual feasibility check (on the values of $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$) 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 $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ and a message issued if the computed value does not match the input value, or if it is out of bounds (if print_level $\geq$ 2);

2. the same as for 1, but PRESOLVE is terminated if an incompatibilty is detected.

The default is check_dual_feasibility = 0.

get_q is a scalar variable of type default LOGICAL, that must be set to .TRUE. if the value of the objective function must be reconstructed by PRESOLVE_restore from the (solved) reduced problem. The default is get_q = .TRUE..

get_f is a scalar variable of type default LOGICAL, that must be set to .TRUE. if the value of the objective function's independent term is to be be reconstructed byPRESOLVE_ restore from the (solved) reduced problem. The default is get_f = .TRUE..

get_g is a scalar variable of type default LOGICAL, that must be set to .TRUE. if the values of the objective function's gradient is to be be reconstructed by PRESOLVE_restore from the (solved) reduced problem. The default is get_g = .TRUE..

get_H is a scalar variable of type default LOGICAL, that must be set to .TRUE. if the values of the objective function's Hessian is to be be reconstructed by PRESOLVE_restore from the (solved) reduced problem. The default is get_H = .TRUE..

get_A is a scalar variable of type default LOGICAL, that must be set to .TRUE. if the values of the constraints' Jacobian is to be be reconstructed by PRESOLVE_restore from the (solved) reduced problem. The default is get_A = .TRUE..

get_x is a scalar variable of type default LOGICAL, that must be set to .TRUE. if the value of the variables must be reconstructed by PRESOLVE_restore from the (solved) reduced problem. The default is get_x = .TRUE..

get_x_bounds is a scalar variable of type default LOGICAL, that must be set to .TRUE. if the values of the bounds on the problem variables must be reconstructed by PRESOLVE_restore from the (solved) reduced problem. The default is get_x_bounds = .TRUE..

get_z is a scalar variable of type default LOGICAL, that must be set to .TRUE. if the value of the dual variables must be reconstructed by PRESOLVE_restore from the (solved) reduced problem. The default is get_z = .TRUE..

get_z_bounds is a scalar variable of type default LOGICAL, that must be set to .TRUE. if the values of the bounds on the problem dual variables must be reconstructed by PRESOLVE_restore from the (solved) reduced problem. If set, this may require to store specific additional information on the problem transformations, therefore increasing the storage needed for these transformations. Note that this parameter must be identical for all calls to PRESOLVE following PRESOLVE_read_specfile. The default is get_z_bounds = .TRUE..

get_c is a scalar variable of type default LOGICAL, that must be set to .TRUE. if the values of the constraints must be reconstructed by PRESOLVE_restore from the (solved) reduced problem. The default is get_c = .TRUE..

get_c_bounds is a scalar variable of type default LOGICAL, that must be set to .TRUE. if the values of the bounds on the problem constraints must be reconstructed by PRESOLVE_restore from the (solved) reduced problem. The default is get_c_bounds = .TRUE..

get_y is a scalar variable of type default LOGICAL, that must be set to .TRUE. if the values of the multipliers must be reconstructed by PRESOLVE_restore from the (solved) reduced problem. The default is get_y = .TRUE..

get_y_bounds is a scalar variable of type default LOGICAL, that must be set to .TRUE. if the values of the bounds on the problem multipliers must be reconstructed by PRESOLVE_restore from the (solved) reduced problem. If set, this may require to store specific additional information on the problem transformations, therefore increasing the storage needed for these transformations. Note that this parameter must be identical for all calls to PRESOLVE following PRESOLVE_read_specfile. The default is get_y_bounds = .TRUE..

pivot_tol is a scalar variable of type REAL(rp_), that holds the relative pivot tolerance above which pivoting is considered as numerically stable in transforming the coefficient matrix $\mathbf{A}$. A zero value corresponds to a totally unsafeguarded pivoting strategy (potentially unstable). The default is pivot_tol = 10^{-6} in single precision, and pivot_tol = 10^{-10} in double precision.

min_rel_improve is a scalar variable of type default REAL(rp_), that holds the minimum relative improvement in the bounds on $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ 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 $\mathbf{x}$, $\mathbf{y}$ or $\mathbf{z}$, and if newlower is a tentative tighter lower bound on the same quantity, it is only accepted if

$$\texttt{newlower} \geq \texttt{lower} + \texttt{min\_rel\_improve} * \max(1, |\texttt{lower}|).$$

Similarly, a tentative tighter upper bound newupper only replaces the current upper bound upper if

$$\texttt{newupper} \leq \texttt{upper} - \texttt{min\_rel\_improve} * \max(1, |\texttt{upper}|).$$

Note that this parameter must exceed the machine precision significantly. The default is min_rel_improve = 10^{-6} in single precision, and min_rel_improve = 10^{-10} in double precision.

max_growth_factor is a scalar variable of type default REAL(rp_), that holds the maximum ratio that is allowed for the absolute value of any data item of the reduced problem compared to the maximum absolute value of any data item of the original problem. In the course of the presolving process, any transformation that would result in violating this bound is skipped. The default is min_growth_factor = 10^4 in single precision, and min_growth_factor = 10^8 in double precision.

2.3.4 The derived data type for holding informational parameters

The derived data type PRESOLVE_inform_type is used to hold parameters that give information about the progress and needs of the algorithm. The components of PRESOLVE_inform_type are:

status is a scalar variable of type INTEGER(ip_), that gives the exit status of the algorithm. See Sections 2.5 and 2.7 for details.

message is a character array of 3 lines of 80 characters each, containing a description of the exit condition on exit, typically including more information than indicated in the description of status above. It is printed out on device errout at the end of execution unless print_level is 0.

nbr_transforms is a scalar variable of type INTEGER(ip_), that gives the the final number of problem transformations, as reported to the user at exit.

2.3.5 The derived data type for holding problem data

The derived data type PRESOLVE_data_type is used to hold all the data for a particular problem, or sequences of problems with the same structure, between calls of PRESOLVE procedures. This data should be preserved, untouched, from the initial call to PRESOLVE_initialize to the final call to PRESOLVE_terminate.

2.4 Argument lists and calling sequences

There are five procedures for user calls (see Section 2.6 for further features):

1. The subroutine PRESOLVE_initialize is used to set default values, and initialize private data, before presolving one or more problems with the same sparsity and bound structure.
2. The subroutine packagename_read_specfile is used to read the packagename specfile in order to possibly modify the algoritmic default parameters (see Section 2.6.1).
3. The subroutine PRESOLVE_apply is called to presolve the problem, that is to reduce it by applying suitable problem transformations and permute it to standard form.
4. The subroutine PRESOLVE_restore restores the (solved) reduced problem to the original definition of variables and constraints;
5. The subroutine PRESOLVE_terminate is provided to allow the user to automatically deallocate array components of the private data, allocated by PRESOLVE, at the end of the solution process. It is important to do this if the data object is re-used for another problem **with a different structure** since PRESOLVE_initialize cannot test for this situation, and any existing associated targets will subsequently become unreachable.

2.4.1 The initialization subroutine

Default values for the control parameters are provided as follows:

```
CALL PRESOLVE_initialize( control, inform, data )
```

control is a scalar INTENT(OUT) argument of type PRESOLVE_control_type (see Section 2.3.3). On exit, control contains default values for the components as described in Section 2.3.3. These values should only be changed after calling PRESOLVE_initialize.

inform is a scalar INTENT(OUT) argument of type PRESOLVE_inform_type (see Section 2.3.4). A successful call to the routine PRESOLVE_initialize is indicated when the component status has the value 0. For other return values of status, see Section 2.5.

data is a scalar INTENT(INOUT) argument of type PRESOLVE_data_type (see Section 2.3.5). It is used to hold data about the problem being solved. It should never be altered by the user.

2.4.2 The subroutine that applies presolving transformations to the problem

The presolving algorithm is called as follows:

```
CALL PRESOLVE_apply( problem, control, inform, data )
```

Such a call must always be preceded by a call to PRESOLVE_initialize.

problem is a scalar INTENT(INOUT) argument of type QPT_problem_type (see Section 2.3.2) that contains the problem statement. It is used to hold data about the problem being solved. Users are free to choose whichever of the three matrix formats described in Section 2.1 is appropriate for **A** and **H** for their application. Some components of the problem structure need not be allocated or set on input, in which case they will be assigned suitable default values. The components in question, their size and the associated default values are given in Table 2.1.

component	size	default	component	size	default
X_l	n	$-\infty$	C_l	m	$-\infty$
X_u	n	$+\infty$	C_u	m	$+\infty$
X	n	(problem dependent)	C	m	(problem dependent)
X_status	n	1	C_status	m	1
Z_l	n	$-\infty$	Y_l	m	$-\infty$
Z_u	n	$+\infty$	Y_u	m	$+\infty$
Z	n	(problem dependent)	Y	m	(problem dependent)

Table 2.1: Defaults for unallocated array components of problem.

If the array problem%X_status is allocated on entry, then possible value of its j-th component are as follows:

-2. the j-th variable is inactive in the sense that the quadratic program under consideration ignores it (this is equivalent of fixing x_j to problem%X(j); which obviously requires problem%X to be allocated);

1. the j-th variable is active (i.e. not inactive).

The meaning of the i-th component of problem%C_status is identical, except that it relates to the i-th constraint:

-2. the i-th constraint is inactive in the sense that the quadratic program under consideration ignores it;

1. the j-th variable is active (i.e. not inactive).

On exit, the problem structure will contain the reduced problem, **with its Hessian and Jacobian matrices stored in sparse row-wise format**; exceptionally, if problem%H%ne = 0, no values or indices of the Hessian will be returned. Values for **x**, **z**, **c**, **y** and f will be provided, that are feasible for the reduced problem. Note that frequently not all the space allocated for the original problem is used by the reduced one. However, crucial information that is necessary to restore the problem to its original variables/constraints remains stored in the problem structure, beyond that specified by the dimensions of the reduced problem. Thus modification (for instance by a QP algorithm) of the reduced problem data is possible (except for problem%X_status and problem%C_status, which should always remain unchanged), but no other data within the problem structure should be altered before calling PRESOLVE_restore.

control is a scalar INTENT(INOUT) argument of type PRESOLVE_control_type (see Section 2.3.3). Default values may be assigned by calling PRESOLVE_initialize prior to the first call to PRESOLVE_apply.

inform is a scalar INTENT(INOUT) argument of type PRESOLVE_inform_type (see Section 2.3.4). A successful call to the routine PRESOLVE_apply is indicated when the component status has the value 0. For other return values of status, see Section 2.5.

data is a scalar INTENT(INOUT) argument of type PRESOLVE_data_type (see Section 2.3.5). It is used to hold data about the problem being solved. It must never be altered by the user since the last call to any of the PRESOLVE routines.

2.4.3 The restoration subroutine

The (solved) reduced problem is restored in the original variables/constraints and matrix format by calling

```
CALL PRESOLVE_restore( problem, control, inform, data )
```

The choice of which components of the problem to restore is governed by the different %get_* components of the control structure (see Section 2.3.3).

problem is a scalar INTENT(INOUT) argument of type QPT_problem_type (see Section 2.3.2). On entry, it is used to hold data about the solved reduced problem. The values of problem%X_status and problem%C_status should not have been altered since the exit from PRESOLVE_apply.

On exit, the problem structure will contain selected components of the solved problem restored to the original variables/constraints and/or matrix format. The selection of these elements is specified by setting the get_* components of the control dummy argument (see Section 2.3.3).

control is a scalar INTENT(INOUT) argument of type PRESOLVE_control_type (see Section 2.3.3). In particular, its get_* components specify which elements of the (solved) reduced problem must be restored to the original formulation.

inform is a scalar INTENT(INOUT) argument of type PRESOLVE_inform__type (see Section 2.3.4). A successful call to the routine PRESOLVE_restoire is indicated when the component status has the value 0. For other return values of status, see Section 2.5.

data is a scalar INTENT(INOUT) argument of type PRESOLVE_data_type (see Section 2.3.5). It is used to hold data about the problem being solved. It must not have been altered by the user since the last call to any of the PRESOLVE routines.

2.4.4 The termination subroutine

All previously allocated workspace arrays are deallocated as follows:

```
CALL PRESOLVE_terminate( control, inform, data )
```

control is a scalar INTENT(IN) argument of type PRESOLVE_control_type exactly as for PRESOLVE_initialize.

inform is a scalar INTENT(OUT) argument of type PRESOLVE_inform_type exactly as for PRESOLVE_initialize. A successful call to PRESOLVE_terminate is indicated when the component status has the value 0. For other return values of status, see Section 2.5.

data is a scalar INTENT(INOUT) argument of type PRESOLVE_data_type exactly as for PRESOLVE_solve, which must not have been altered by the user since the last call to PRESOLVE_initialize. On exit, array components will have been deallocated.

Note that a call to this routine is mandatory before PRESOLVE_apply is called for a new quadratic program whose structure differs from the current one.

2.5 Warning and error messages

A negative value of info%status on exit from PRESOLVE_initialize, PRESOLVE_read_specfile, PRESOLVE_apply, PRESOLVE_restore, or PRESOLVE_terminate indicates that an error has occurred. No further calls should be made to the four three of these routines until the error has been corrected. Possible values are:

1. The maximum number of problem transformation has been reached. Note that 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 max_nbr_transforms.

-1. A workspace allocation failed.

-2. A file intended for saving problem transformations could not be opened.

-3. An input-output error occurred while saving transformations on the relevant disk file.

-4. The machine/compiler only supports less than 8 bits in a single integer (this error is thus very unlikely).

-21. The problem appears to be primal infeasible.

-22. The problem appears to be dual infeasible.

-23. The dimension of the gradient problem%G is not equal to the number of variables in the problem problem%n.

-24. The dimension of the vector problem%H_val containing the entries of the Hessian is erroneously specified.

-25. The dimension of the vector problem%H_ptr containing the addresses of the first entry of each Hessian row is erroneously specified.

-26. The dimension of the vector problem%H_col containing the column indices of the nonzero Hessian entries is erroneously specified.

-27. The dimension of the vector problem%H_row containing the row indices of the nonzero Hessian entries is erroneously specified.

-28. The dimension of the vector problem%A_val containing the entries of the Jacobian is erroneously specified.

-29. The dimension of the vector problem%A_ptr containing the addresses of the first entry of each Jacobian row is erroneously specified.

-30. The dimension of the vector problem%A_col containing the column indices of the nonzero Jacobian entries is erroneously specified.

-31. The dimension of the vector problem%A_row containing the row indices of the nonzero Jacobian entries is erroneously specified;

-32. The dimension of the vector problem%X of variables is incompatible with the problem dimension problem%n.

-33. The dimension of the vector problem%X_l of lower bounds on the variables is incompatible with the problem dimension problem%n.

-34. The dimension of the vector problem%X_u of upper bounds on the variables is incompatible with the problem dimension problem%n.

-35. The dimension of the vector problem%Z of dual variables is incompatible with the problem dimension problem%n.

-36. The dimension of the vector problem%Z_l of lower bounds on the dual variables is incompatible with the problem dimension problem%n.

-37. The dimension of the vector problem%Z_u of upper bounds on the dual variables is incompatible with the problem dimension problem%n.

-38. The dimension of the vector problem%C of constraints values is incompatible with the problem dimension problem%m.

-39. The dimension of the vector problem%C_l of lower bounds on the constraints is incompatible with the problem dimension problem%m.

-40. The dimension of the vector problem%C_u of upper bounds on the constraints is incompatible with the problem dimension problem%m.

-41. The dimension of the vector problem%Y of multipliers values is incompatible with the problem dimension problem%m.

-42. The dimension of the vector problem%Y_l of lower bounds on the multipliers is incompatible with the problem dimension problem%m.

-43. The dimension of the vector problem%Y_u of upper bounds on the multipliers is incompatible with the problem dimension problem%m.

-44. The problem structure has not been set or has been cleaned up before an attempt to apply PRESOLVE_apply.

-45. The problem has not been analyzed before an attempt to permute it.

-46. The problem has not been permuted or fully reduced before an attempt to restore it.

-47. 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. One of the files containing saved problem transformations has been corrupted between writing and reading.

-49. The dimension of the vector problem%X_status of variables' status is incompatible with the problem dimension problem%n.

-50. The dimension of the vector problem%C_status of constraints' status is incompatible with the problem dimension problem%m.

-52. The problem does not contain any (active) variable (problem%n ≤ 0).

-53. The problem contains a negative number of constraints (problem%m < 0).

-54. The vectors are too long for the quicksort sorting routine (see the GALAHAD SORT module).

-55. The value of a variable that is obtained in PRESOLVE_restore 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. The value of a constraint that is obtained by recomputing its value on input of PRESOLVE_restore from the current $\mathbf{x}$ is incompatible with its declared value or its bounds. This may caused the restored problem to be infeasible.

-57. The value of a dual variable that is obtained by recomputing its value on input of PRESOLVE_restore (assuming dual feasibility) from the current values of $(\mathbf{x},\mathbf{y},\mathbf{z})$ is incompatible with its declared value. This may caused the restored problem to be infeasible or suboptimal.

-58. A dual variable whose value is nonzero because the corresponding primal is at an artificial bound cannot be zeroed while maintaining dual feasibility (in PRESOLVE_restore). This can happen when $(\mathbf{x},\mathbf{y},\mathbf{z})$ on input of this routine are not (sufficiently) optimal.

-60. A keyword was not recognized in the analysis of the specification file.

-61. A value was not recognized in the analysis of the specification file.

-63. The vector problem%G has not been allocated although it has general values.

-65. The vector problem%A_val has not been allocated although problem%m > 0.

-66. The vector problem%A_ptr has not been allocated although problem%m > 0 and **A** is stored in row-wise sparse format.

-67. The vector problem%A_col has not been allocated although problem%m > 0 and **A** is stored in row-wise sparse format or sparse coordinate format.

-68. The vector problem%A_row has not been allocated although problem%m > 0 and **A** is stored in sparse coordinate format.

-69. The vector problem%H_val has not been allocated although problem%H_ne $= -2$ or problem%H_ne $= -1$ or problem%H_ne > 0.

-70. The vector problem%H_ptr has not been allocated although **H** is stored in row-wise sparse format.

-71. The vector problem%H_col has not been allocated although **H** is stored in row-wise sparse format or problem%H_ne > 0 and **H** is stored sparse coordinate format.

-72. The vector problem%H_row has not been allocated although problem%H_ne > 0 and **H** is stored in sparse coordinate format.

-73. The value of problem%A_ne is erroneously specified.

-74. The value of problem%H_ne is erroneously specified.

2.6 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable control of type PRESOLVE_control_type (see Section 2.3.3), by reading an appropriate data specification file using the subroutine PRESOLVE_read_specfile. This facility is useful as it allows a user to change PRESOLVE control parameters without editing and recompiling programs that call PRESOLVE.

A specification file, or specfile, is a data file containing a number of "specification commands". Each command occurs on a separate line, and comprises a "keyword", which is a string (in a close-to-natural language) used to identify a control parameter, and an (optional) "value", which defines the value to be assigned to the given control parameter. All keywords and values are case insensitive, keywords may be preceded by one or more blanks but values must not contain blanks, and each value must be separated from its keyword by at least one blank. Values must not contain more than 30 characters, and each line of the specfile is limited to 80 characters, including the blanks separating keyword and value.

The portion of the specification file used by PRESOLVE_read_specfile must start with a "BEGIN PRESOLVE" command and end with an "END" command. The syntax of the specfile is thus defined as follows:

```
( .. lines ignored by PRESOLVE_read_specfile .. )
  BEGIN PRESOLVE
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by PRESOLVE_read_specfile .. )
```

where keyword and tt value are two strings separated by (at least) one blank. The "BEGIN PRESOLVE" and "END" delimiter command lines may contain additional (trailing) strings so long as such strings are separated by one or more blanks, so that lines such as

```
BEGIN PRESOLVE SPECIFICATION
```

and

```
END PRESOLVE SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN PRESOLVE" and "END" delimiters, specification commands may occur in any order. Blank lines and lines whose first non-blank character is ! or * are ignored. The content of a line after a ! or * character is also ignored (as is the ! or * character itself). This provides an easy manner to "comment out" some specification commands, or to comment specific values of certain control parameters.

The value of a control parameters may be of five different types, namely integer, logical, real, string or symbol. Integer and real values may be expressed in any relevant Fortran integer and floating-point formats (respectively). Permitted values for logical parameters are "ON", "TRUE", ".TRUE.", "T", "YES", "Y", or "OFF", "NO", "N", "FALSE", ".FALSE." and "F". Empty values are also allowed for logical control parameters, and are interpreted as "TRUE". String are specified as a sequence of characters.

The specification file must be open for input when PRESOLVE_read_specfile is called, and the associated device number passed to the routine in device (see below). Note that the corresponding file is REWINDed, which makes it possible to combine the specifications for more than one program/routine. For the same reason, the file is not closed by PRESOLVE_read_specfile.

2.6.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL PRESOLVE_read_specfile( device, control, inform )
```

device is a scalar INTENT(IN) argument of type INTEGER(ip_), that must be set to the unit number on which the specfile has been opened. If device is not open, control will not be altered and execution will continue, but an error message will be printed on unit control%error.

control is a scalar INTENT(INOUT) argument of type PRESOLVE_control_type (see Section 2.3.3). Default values should have already been set, perhaps by calling PRESOLVE_initialize. On exit, individual components of control may have been changed according to the commands found in the specfile. Specfile commands and the component (see Section 2.3.3) of control that each affects are given in Tables 2.2 and 2.3.

inform is a scalar INTENT(OUT) argument of type PRESOLVE_inform_type (see Section 2.3.4).

2.7 Information printed

The level of output produced by values of control%print_level is as follows:

0. no printout is produced,

1. only the major steps in the analysis is reported, that is headers of the main preprocessing phases and, for each pass, the number of transformations of each type applied,

2. in addition, reports the nature of each problem transformation,

3. in addition, reports more details on each of the main presolve loops constituents,

4. reports considerable detail, including information on unsuccessful attempts to apply presolving transformations,

5. reports a completely silly amount of information.

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
presolve-termination-strategy	%termination	integer
maximum-number-of-transformations	%max_nbr_transforms	integer
maximum-number-of-passes	%max_nbr_passes	integer
constraints-accuracy	%c_accuracy	real
dual-variables-accuracy	%z_accuracy	real
allow-dual-transformations	%dual_transformations	logical
remove-redundant-variables-constraints	%redundant_xc	logical
primal-constraints-analysis-frequency	%primal_constraints_freq	integer
dual-constraints-analysis-frequency	%dual_constraints_freq	integer
singleton-columns-analysis-frequency	%singleton_columns_freq	integer
doubleton-columns-analysis-frequency	%doubleton_columns_freq	integer
unconstrained-variables-analysis-frequency	%unc_variables_freq	integer
dependent-variables-analysis-frequency	%dependent_variables_freq	integer
row-sparsification-frequency	%sparsify_rows_freq	integer
maximum-percentage-row-fill	%max_fill	integer
transformations-buffer-size	%transf_buffer_size	integer
transformations-file-device	%transf_file_nbr	integer
transformations-file-status	%transf_file_status	integer
transformations-file-name	%transf_file_name	string
primal-feasibility-check	%check_primal_feasibility	integer
dual-feasibility-check	%check_dual_feasibility	integer
active-multipliers-sign	%y_sign	integer
inactive-multipliers-value	%inactive_y	integer
active-dual-variables-sign	%z_sign	integer
inactive-dual-variables-value	%inactive_z	integer
primal-variables-bound-status	%final_x_bounds	integer

Table 2.2: Specfile commands and associated components of control .

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: PRESOLVE calls the GALAHAD modules GALAHAD_SMT, GALAHAD_QPT, GALAHAD_SPECFILE, GALAHAD_SORT, and GALAHAD_SYMBOLS.

Input/output: Output is under control of the arguments control%error, control%out and control%print_level.

Restrictions: prob%n > 0, prob%m ≥ 0, prob%A_type and prob%H_type $\in \{$'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS'$\}$.

Portability: ISO Fortran 95 + TR 15581 or Fortran 2003. The package is thread-safe.

command	component of control	value type
dual-variables-bound-status	%final_z_bounds	integer
constraints-bound-status	%final_c_bounds	integer
multipliers-bound-status	%final_y_bounds	integer
infinity-value	%infinity	real
pivoting-threshold	%pivot_tol	real
minimum-relative-bound-improvement	%min_rel_improve	real
maximum-growth-factor	%max_growth_factor	real
compute-quadratic-value	%get_q	logical
compute-objective-constant	%get_f	logical
compute-gradient	%get_g	logical
compute-Hessian	%get_H	logical
compute-constraints-matrix	%get_A	logical
compute-primal-variables-values	%get_x	logical
compute-primal-variables-bounds	%get_x_bounds	logical
compute-dual-variables-values	%get_z	logical
compute-dual-variables-bounds	%get_z_bounds	logical
compute-contraints-values	%get_c	logical
compute-constraints-bounds	%get_c_bounds	logical
compute-multipliers-values	%get_y	logical
compute-multipliers-bounds	%get_y_bounds	logical

Table 2.3: Specfile commands and associated components of control (continued).

4 METHOD

The required solution $\mathbf{x}$ of the problem necessarily satisfies the primal optimality conditions

$$\mathbf{A}\mathbf{x} = \mathbf{c}$$

and

$$\mathbf{c}^l \leq \mathbf{c} \leq \mathbf{c}^u, \ \mathbf{x}^l \leq \mathbf{x} \leq \mathbf{x}^u,$$

the dual optimality conditions

$$\mathbf{H}\mathbf{x} + \mathbf{g} = \mathbf{A}^T\mathbf{y} + \mathbf{z}, \ \mathbf{y} = \mathbf{y}^l + \mathbf{y}^u \ \text{and} \ \mathbf{z} = \mathbf{z}^l + \mathbf{z}^u,$$

and

$$\mathbf{y}^l \geq 0, \ \mathbf{y}^u \leq 0, \ \mathbf{z}^l \geq 0 \ \text{and} \ \mathbf{z}^u \leq 0,$$

and the complementary slackness conditions

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

where the vectors $\mathbf{y}$ and $\mathbf{z}$ are known as the Lagrange multipliers for the general linear constraints, and the dual variables for the bounds, respectively, and where the vector inequalities hold componentwise. 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			$\mathbf{x}$		
non-negativity	0	$\leq$	$\mathbf{x}$		
lower	$\mathbf{x}^l$	$\leq$	$\mathbf{x}$		
range	$\mathbf{x}^l$	$\leq$	$\mathbf{x}$	$\leq$	$\mathbf{x}^u$
upper			$\mathbf{x}$	$\leq$	$\mathbf{x}^u$
non-positivity			$\mathbf{x}$	$\leq$	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	$\leq$	$\mathbf{Ax}$		
equality	$\mathbf{c}^l$	$=$	$\mathbf{Ax}$		
lower	$\mathbf{c}^l$	$\leq$	$\mathbf{Ax}$		
range	$\mathbf{c}^l$	$\leq$	$\mathbf{Ax}$	$\leq$	$\mathbf{c}^u$
upper			$\mathbf{Ax}$	$\leq$	$\mathbf{c}^u$
non-positivity			$\mathbf{Ax}$	$\leq$	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:

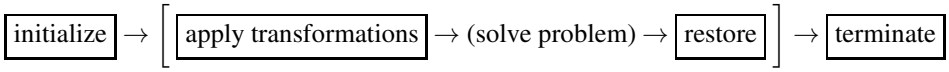
1. remove empty and singletons rows,
2. try to eliminate variables that are linearly unconstrained,
3. attempt to exploit the presence of linear singleton columns,
4. attempt to exploit the presence of linear doubleton columns,
5. complete the analysis of the dual constraints,
6. remove empty and singletons 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

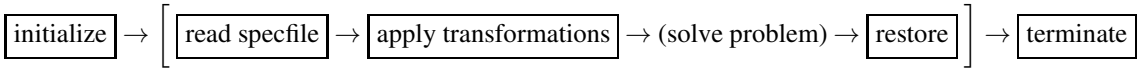
solver, thus ensuring sufficiently small primal-dual feasibility and complementarity. Finally, the solution of the simplified problem is re-translated in the variables/constraints/format of the original problem formulation by a "restoration" phase.

If the number of problem transformations exceeds control%transf_buffer_size, the transformation buffer size (see Section 2.3.3), then they are saved in a "history" file, whose name may be chosen by specifying the control%transf_file_name control parameter (see Section 2.3.3). When this is the case, this file is subsequently reread by PRESOLVE_restore. It must not be altered by the user.

At the overall level, the presolving process follows one of the two sequences:



or



where the procedure's control parameter may be modified by reading the specfile (see Section 2.6), 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 (see Section 2.4). In the above diagrams, brackated subsequence of steps means that they can be repeated with problem having the same structure. The value of the new_problem_structure component of problem must be .TRUE. on entry of PRESOLVE_apply 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

$$L(\mathbf{x},\mathbf{y},\mathbf{z}) = q(\mathbf{x}) - \texttt{y\_sign} * \mathbf{y}^T(\mathbf{A}\mathbf{x} - \mathbf{c}) - \texttt{z\_sign} * \mathbf{z},$$

(considering only active constraints $\mathbf{A}\mathbf{x} = \mathbf{c}$), 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 (see Section 2.3.3).

References:

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.

5 EXAMPLE OF USE

Suppose that we wish to solve the quadratic program (1.2)–(1.4) with the data $n = 6$, $m = 5$, $f = 1$, $\mathbf{g} = (1\ 1\ 1\ 1\ 1\ 1)^T$,

$$\mathbf{H} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}, \qquad \mathbf{A} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \end{pmatrix},$$

$\mathbf{x}^l = (0\ 0\ 0\ 0\ 0\ 0)^T$, $\mathbf{x}^u = (1\ 1\ 1\ 1\ 1\ 1)^T$, $\mathbf{c}^l = (0\ 0\ 2\ 1\ 3)^T$ and $\mathbf{c}^u = (1\ 1\ 3\ 3\ 3)^T$, using the quadratic programming solver `QPSOLVER` after applying the `PRESOLVE` package and then restoring the final solution to the original variable formulation. We may use the following code—note that we require some output from `PRESOLVE` by setting `control%print_level` to `1`, and that calling `QPSOLVER` is actually unnecessary since the problem of our example is completely reduced to a single feasible point (which must then be the solution) after presolving.

```
   PROGRAM GALAHAD_PRESOLVE_EXAMPLE
   USE GALAHAD_QPT_double                          ! Double precision
   USE GALAHAD_PRESOLVE_double                     ! Double precision
   USE GALAHAD_SYMBOLS                             ! The GALAHAD symbols
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D0 )        ! Set precision
   REAL ( KIND = wp ), PARAMETER  :: infinity = 10._wp ** 20
   REAL ( KIND = wp ), PARAMETER  :: r0 = 0.0_wp, r1 = 1.0_wp
   REAL ( KIND = wp ), PARAMETER  :: r2 = 2.0_wp, r3 = 3.0_wp
   TYPE ( QPT_problem_type )      :: problem
   TYPE ( PRESOLVE_control_type ) :: control
   TYPE ( PRESOLVE_inform_type )  :: inform
   TYPE ( PRESOLVE_data_type )    :: data
   INTEGER :: j, n, m, a_ne, h_ne
! start problem data
   n = 6; m = 5; h_ne = 1; a_ne = 8
   problem%new_problem_structure = .TRUE.
   problem%n = n; problem%m = m; problem%f = r1
   ALLOCATE( problem%G( n )  , problem%X_l( n ), problem%X_u( n ) )
   ALLOCATE( problem%C_l( m ), problem%C_u( m ) )
   problem%gradient_kind = 1
   problem%C_l = (/  r0, r0, r2, r1, r3  /)
   problem%C_u = (/  r1, r1, r3, r3, r3  /)
   problem%X_l = (/ -r3, r0, r0, r0, r0, r0 /)
   problem%X_u = (/  r3, r1, r1, r1, r1, r1 /)
! sparse coordinate format
   CALL SMT_put( problem%H%type, 'COORDINATE' )
   CALL SMT_put( problem%A%type, 'COORDINATE' )
   ALLOCATE( problem%H%val( h_ne ) )
   ALLOCATE( problem%H%col( h_ne ), problem%H%row( h_ne ) )
   ALLOCATE( problem%A%val( a_ne ) )
   ALLOCATE( problem%A%col( a_ne ), problem%A%row( a_ne ) )
   problem%H%val = (/ r1  /)
   problem%H%row = (/  1  /)
   problem%H%col = (/  1  /)
   problem%A%val = (/ r1, r1, r1, r1, r1, r1, r1, r1 /)
   problem%A%row = (/  3,  3,  3,  4,  4,  5,  5,  5 /)
   problem%A%col = (/  3,  4,  5,  3,  6,  4,  5,  6 /)
```

```
   problem%a_ne  = a_ne; problem%h_ne = h_ne
! problem data complete
! write the original formulation
   CALL QPT_write_problem( 6, problem )
! set the default PRESOLVE control parameters
   CALL PRESOLVE_initialize( control, inform, data )
   IF ( inform%status /= 0 ) STOP
   control%print_level = 1  ! Ask for some output
! apply presolving to reduce the problem
   CALL PRESOLVE_apply( problem, control, inform, data )
   IF ( inform%status /= 0 ) STOP
! write the reduced problem
   CALL QPT_write_problem( 6, problem )
! solve the reduced problem
   ! CALL QPSOLVER (unnecessary here, because the reduced problem has a
   ! single feasible point in this example)
! restore the solved reduced problem to the original formulation
   CALL PRESOLVE_restore( problem, control, inform, data )
   IF ( inform%status /= 0 ) STOP
! write the final solution in the original variables
   WRITE( 6, "( /, '  The problem solution X is', /, )" )
   DO j = 1, n
      WRITE( 6, '(3x,''x('',I1,'') = '', ES12.4)' ) j, problem%X( j )
   END DO
! deallocate internal workspace
   CALL PRESOLVE_terminate( control, inform, data )
   END PROGRAM GALAHAD_PRESOLVE_EXAMPLE
```

This produces the following output:

```
    ============== PROBLEM ====================

    n =  6

    variables

                   lower       upper

   x(   1) =    -3.0000E+00  3.0000E+00
   x(   2) =     0.0000E+00  1.0000E+00
   x(   3) =     0.0000E+00  1.0000E+00
   x(   4) =     0.0000E+00  1.0000E+00
   x(   5) =     0.0000E+00  1.0000E+00
   x(   6) =     0.0000E+00  1.0000E+00

    m =  5

    constraints

                   lower       upper

   c(   1) =     0.0000E+00  1.0000E+00
   c(   2) =     0.0000E+00  1.0000E+00
   c(   3) =     2.0000E+00  3.0000E+00
   c(   4) =     1.0000E+00  3.0000E+00
```

```
 c(   5) =     3.0000E+00  3.0000E+00

  Jacobian

 A(   3,   3) =   1.0000E+00
 A(   3,   4) =   1.0000E+00
 A(   3,   5) =   1.0000E+00
 A(   4,   3) =   1.0000E+00
 A(   4,   6) =   1.0000E+00
 A(   5,   4) =   1.0000E+00
 A(   5,   5) =   1.0000E+00
 A(   5,   6) =   1.0000E+00


  objective function constant term =   1.0000E+00


  gradient

 g(   1) =     1.0000E+00
 g(   2) =     1.0000E+00
 g(   3) =     1.0000E+00
 g(   4) =     1.0000E+00
 g(   5) =     1.0000E+00
 g(   6) =     1.0000E+00

  Hessian

 H(   1,   1) =   1.0000E+00

  ============ END OF PROBLEM =================


*******************************************
*                                         *
*         GALAHAD presolve for QPs        *
*                                         *
*            problem analysis             *
*                                         *
*******************************************


============ starting problem analysis ============

checking bounds on x, y, z, and c: 0 transformations
redundant variables and constraints: 0 transformations

============= main processing loop 1  =============
  ( n = 6 , m = 5 , a_ne = 8 , h_ne = 1 )

removing empty and singleton rows: 2 transformations
analyzing special linear columns: 3 transformations
analyzing dual constraints: 0 transformations
removing empty and singleton rows: 0 transformations
checking dependent variables: 2  transformations
```

```
analyzing primal constraints: 5 transformations
checking bounds on x, y, z, and c: 0 transformations

============= main processing loop 2  =============
  ( n = 1 , m = 2 , a_ne = 2 , h_ne = 0 )

removing empty and singleton rows: 2 transformations
analyzing special linear columns: 2 transformations

======== end of the main processing loop ( loop = 2 ) ========

all variables and constraints have been eliminated!

No permutation necessary.

******************** Bye ******************


  =============== PROBLEM =====================

  n =  0

  m =  0

  current objective function value =   3.5000E+00


  objective function constant term =   3.5000E+00


  ============ END OF PROBLEM =================


*******************************************
*                                         *
*         GALAHAD PRESOLVE for QPs        *
*                                         *
*            problem restoration          *
*                                         *
*******************************************

verifying user-defined presolve control parameters
===  starting historical loop
===  end of the historical loop

Problem successfully restored.

******************** Bye ******************


 The problem solution X is

 x(1) =  -1.0000E+00
 x(2) =   0.0000E+00
 x(3) =   0.0000E+00
```

```
   x(4) =   1.0000E+00
   x(5) =   1.0000E+00
   x(6) =   1.0000E+00

  ******************************************
  *                                        *
  *         GALAHAD PRESOLVE for QPs       *
  *                                        *
  *            workspace cleanup           *
  *                                        *
  ******************************************



  ******************** Bye ******************
```

The same problem may be solved holding the data in a sparse row-wise storage format by replacing the lines

```
! sparse coordinate format
  ......
! problem data complete
```

by

```
! sparse row-wise storage format
   CALL SMT_put( problem%H%type, 'SPARSE_BY_ROWS' )
   CALL SMT_put( problem%A%type, 'SPARSE_BY_ROWS' )
   ALLOCATE( problem%H%val( h_ne ) )
   ALLOCATE( problem%H%ptr( n+1 ), problem%H%col( h_ne ) )
   ALLOCATE( problem%A%val( a_ne ) )
   ALLOCATE( problem%A%ptr( m+1 ), problem%A%col( a_ne ) )
   problem%H%val = (/ r1  /)
   problem%H%ptr = (/  1,  2,  2,  2,  2,  2,  2  /)
   problem%H%col = (/  1  /)
   problem%A%val = (/ r1, r1, r1, r1, r1, r1, r1, r1 /)
   problem%A%ptr = (/  1,  1,  1,  4,  6,  9  /)
   problem%A%col = (/  3,  4,  5,  3,  6,  4,  5,  6 /)
! problem data complete
```

or using a dense storage format with the replacement lines

```
! dense storage format
   CALL SMT_put( problem%H%type, 'DENSE' )
   CALL SMT_put( problem%A%type, 'DENSE' )
   ALLOCATE( problem%H%val( n*(n+1)/2 ) )
   ALLOCATE( problem%A%val( n*m ) )
   problem%H%val = (/ r1,                          &
                      r0, r0,                      &
                      r0, r0, r0,                  &
                      r0, r0, r0, r0,              &
                      r0, r0, r0, r0, r0,          &
                      r0, r0, r0, r0, r0, r0  /)
   problem%A%val = (/ r0, r0, r0, r0, r0, r0,      &
                      r0, r0, r0, r0, r0, r0,      &
                      r0, r0, r1, r1, r1, r0,      &
```

```
                      r0, r0, r1, r0, r0, r1,       &
                      r0, r0, r0, r1, r1, r1  /)
! problem data complete
```

respectively. (If instead **H** had been the diagonal matrix

$$\mathbf{H} = \begin{pmatrix} 1 & & \\ & 0 & \\ & & 3 \end{pmatrix}$$

but the other data is as before, the diagonal storage scheme might be used for **H**, and in this case we would instead

```
   CALL SMT_put( prob%H%type, 'DIAGONAL' )  ! Specify dense storage for H
   ALLOCATE( p%H%val( n ) )
   p%H%val = (/ r1, r0, r0, r0, r0, r0, r0 /) ! Hessian values
```

Notice here that zero diagonal entries are stored.) We could also make use of the `PRESOLVE_read_specfile` routine to set the printing level, in which case the statement

```
   control%print_level = GALAHAD_TRACE             ! Ask for some output
```

is replaced by

```
! open specfile
   OPEN( 57, FILE = 'PRESOLVE.SPC', STATUS = 'OLD' )
! read its content (asking for some output)
   CALL PRESOLVE_read_specfile( 57, control, inform )
! close it
   CLOSE( 57 )
```

where we assume that the file `PRESOLVE.SPC` exists in the current directory and contains the lines

```
BEGIN PRESOLVE SPECIFICATION
   print-level 1
END PRESOLVE SPECIFICATION
```