GALAHAD SCALE

USER DOCUMENTATION GALAHAD Optimization Library version 2.5

1 SUMMARY

This package calculates and applies **shift and scale factors** for the variables and constraints to try to equiilibrate the **quadratic programming problem**

$$\text{minimize } \tfrac{1}{2}\mathbf{x}^T\mathbf{H}\mathbf{x} + \mathbf{g}^T\mathbf{x} + f \tag{1.1}$$

subject to the general linear constraints

$$c_i^l \leq \mathbf{a}_i^T\mathbf{x} \leq c_i^u, \quad i = 1, \ldots, m, \tag{1.2}$$

and the simple bound constraints

$$x_j^l \leq x_j \leq x_j^u, \quad j = 1, \ldots, n, \tag{1.3}$$

where the n by n symmetric matrix $\mathbf{H}$, the vectors $\mathbf{g}$, $\mathbf{a}_i$, $\mathbf{c}^l$, $\mathbf{c}^u$, $\mathbf{x}^l$, $\mathbf{x}^u$ and the scalar f are given. Full advantage is taken of any zero coefficients in the matrix $\mathbf{H}$, as well as the matrix $\mathbf{A}$, whose rows are the vectors $\mathbf{a}_i^T$, $i = 1, \ldots, m$. Any of the constraint bounds c_i^l, c_i^u, x_j^l and x_j^u may be infinite.

The derived type is also capable of supporting **parametric** quadratic programming problems, in which an additional objective term $\theta\delta\mathbf{g}^T\mathbf{x} + \theta\delta f$ is included, and the trajectory of solution are required for all $0 \leq \theta \leq \theta_{\max}$ for which

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

and

$$x_j^l + \theta x_j^l \leq x_j \leq x_j^u + \delta x_j^u, \quad j = 1, \ldots, n.$$

New variables $\mathbf{X}_s^{-1}(\mathbf{x} - \mathbf{x}_s)$ are calculated, involving the matrix of diagonal variable scaling factors $\mathbf{X}_s$ and a corrsponding vector of shifts $\mathbf{x}_s$. Likelwise the constraint values are transformed to be $\mathbf{C}_s^{-1}(\mathbf{A}\mathbf{x} - \mathbf{c}_s)$, involving the matrix of diagonal constraint scaling factors $\mathbf{C}_s$ and vector of corrsponding shifts $\mathbf{c}_s$. The value of the objective function is transformed to be $F_s^{-1}(q(x) - f_s)$ using an objective scaling factor F_s and shift f_s.

ATTRIBUTES — Versions: `GALAHAD_SCALE_single`, `GALAHAD_SCALE_double`. **Uses:** `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_SPECFILE`, `GALAHAD_TOOLS`, `GALAHAD_SMT`, `GALAHAD_QPT`, `GALAHAD_TRANS`. **Date:** January 2011. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003.

2 HOW TO USE THE PACKAGE

Access to the package requires a `USE` statement such as

Single precision version

```
USE GALAHAD_SCALE_single
```

Double precision version

```
USE GALAHAD_SCALE_double
```

If it is required to use both modules at the same time, the derived types `SMT_type`, `QPT_problem_type`, `SCALE_trans_type`, `SCALE_control_type`, `SCALE_inform_type` and `SCALE_data_type` (Section 2.2) and the subroutines `SCALE_initialize`, `SCALE_get`, `SCALE_apply`, `SCALE_recover`, `SCALE_terminate`, (Section 2.3) and `SCALE_read_specfile` (Section 2.5) must be renamed on one of the `USE` statements.

All use is subject to licence. See `http://galahad.rl.ac.uk/galahad-www/cou.html`.
For any commercial application, a separate license must be signed.

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), ...,`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**.

2.2 The derived data types

Six derived data types are accessible from the package.

2.2.1 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 default `INTEGER`, that holds the number of rows in the matrix.

`n` is a scalar component of type default `INTEGER`, that holds the number of columns in the matrix.

`ne` is a scalar variable of type default `INTEGER`, that holds the number of matrix entries.

`type` is a rank-one allocatable array of type default `CHARACTER`, that is used to indicate the matrix storage scheme used. Its precise length and content depends on the type of matrix to be stored (see §2.2.2).

All use is subject to licence. See `http://galahad.rl.ac.uk/galahad-www/cou.html`.
For any commercial application, a separate license must be signed.

`val` is a rank-one allocatable array of type default `REAL` (double precision in `GALAHAD_SCALE_double`) 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 default `INTEGER`, 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 default `INTEGER`, 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 default `INTEGER`, and dimension at least `m + 1`, that may hold the pointers to the first entry in each row (see §2.1.3).

2.2.2 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:

`n` is a scalar variable of type default `INTEGER`, that holds the number of optimization variables, n.

`m` is a scalar variable of type default `INTEGER`, 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`, and for the diagonal storage scheme (see Section 2.1.4), the first eight components of `H%type` must contain the string `DIAGONAL`.

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 `SCALE_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 default `INTEGER`, 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 three schemes.

`H%val` is a rank-one allocatable array of type default `REAL` (double precision in `GALAHAD_SCALE_double`), 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 default `INTEGER`, 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 three schemes.

`H%col` is a rank-one allocatable array variable of type default `INTEGER`, 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.

All use is subject to licence. See `http://galahad.rl.ac.uk/galahad-www/cou.html`.
For any commercial application, a separate license must be signed.

H%ptr is a rank-one allocatable array of dimension n+1 and type default INTEGER, 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.

G is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_SCALE_double), that holds the gradient $\mathbf{g}$ of the linear term of the quadratic objective function. The j-th component of G, $j = 1,\ldots,n$, contains $\mathbf{g}_j$.

DG is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_SCALE_double), that may hold the gradient $\delta\mathbf{g}$ of the parametric linear term of the quadratic objective function. The j-th component of DG, $j = 1,\ldots,n$, contains δg_j.

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

df is a scalar variable of type default REAL (double precision in GALAHAD_SCALE_double), that holds the parametric constant term, δf, 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 SCALE_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 default INTEGER, 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 default REAL (double precision in GALAHAD_SCALE_double), 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 default INTEGER, 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 default INTEGER, 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 default INTEGER, 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 default REAL (double precision in GALAHAD_SCALE_double), 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 $\mathbf{c}_i^l$. 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 2.2.4).

All use is subject to licence. See http://galahad.rl.ac.uk/galahad-www/cou.html.
For any commercial application, a separate license must be signed.

C_u is a rank-one allocatable array of dimension m and type default REAL (double precision in GALAHAD_SCALE_double), 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 2.2.4).

DC_l is a rank-one allocatable array of dimension m and type default REAL (double precision in GALAHAD_SCALE_double), that may hold the vector of parametric lower bounds $\delta\mathbf{c}^l$ on the general constraints. The i-th component of DC_l, $i = 1,\ldots,m$, contains δc^l_i. Only components corresponding to finite lower bounds c^l_i need be set.

DC_u is a rank-one allocatable array of dimension m and type default REAL (double precision in GALAHAD_SCALE_double), that may hold the vector of parametric upper bounds $\delta\mathbf{c}^u$ on the general constraints. The i-th component of DC_u, $i = 1,\ldots,m$, contains δc^u_i. Only components corresponding to finite upper bounds c^u_i need be set.

X_l is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_SCALE_double), 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 2.2.4).

X_u is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_SCALE_double), 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 2.2.4).

DX_l is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_SCALE_double), that may hold the vector of parametric lower bounds $\delta\mathbf{x}^l$ on the variables. The j-th component of DX_l, $j = 1,\ldots,n$, contains δx^l_j. Only components corresponding to finite lower bounds x^l_j need be set.

DX_u is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_SCALE_double), that may hold the vector of parametric upper bounds $\delta\mathbf{x}^u$ on the variables. The j-th component of DX_u, $j = 1,\ldots,n$, contains δx^u_j. Only components corresponding to finite upper bounds x^u_j need be set.

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

C is a rank-one allocatable array of dimension m and type default REAL (double precision in GALAHAD_SCALE_double), 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 default REAL (double precision in GALAHAD_SCALE_double), 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.

Z is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_SCALE_double), 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.

2.2.3 The derived data type for holding the scaling factors and shifts

The derived data type SCALE_trans_type is used to hold the computed scaling factors and shifts. The components of SCALE_trans_type are:

X_scale is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_SCALE_double), that holds the variable scale factors. The j-th component of X_scale, $j = 1,\ldots,n$, contains the scale factor to be applied to x_j.

**All use is subject to licence. See http://galahad.rl.ac.uk/galahad-www/cou.html.
For any commercial application, a separate license must be signed.**

X_shift is a rank-one allocatable array of dimension n and type default REAL (double precision in GALAHAD_SCALE_-double), that holds the variable shifts if appropriate. The j-th component of X_shift, $j = 1,\ldots,n$, contains the shift to be applied to x_j.

C_scale is a rank-one allocatable array of dimension m and type default REAL (double precision in GALAHAD_SCALE_-double), that holds the constraint scale factors. The i-th component of C_scale, $i = 1,\ldots,m$, contains the scale factor to be applied to the i-th constraint.

C_shift is a rank-one allocatable array of dimension m and type default REAL (double precision in GALAHAD_SCALE-_double), that holds the constraint shifts if appropriate. The i-th component of C_shift, $i = 1,\ldots,m$, contains the shift to be applied to the i-th constraint.

f_scale is a scalar variable of type default REAL (double precision in GALAHAD_SCALE_double), that holds the scale factor for the objective function.

f_shift is a scalar variable of type default REAL (double precision in GALAHAD_SCALE_double), that holds the shift for the objective function.

2.2.4 The derived data type for holding control parameters

The derived data type SCALE_control_type is used to hold controlling data. Default values may be obtained by calling SCALE_initialize (see Section 2.3.1), while components may also be changed by calling SCALE_read_specfile (see Section 2.5.1). The components of SCALE_control_type are:

error is a scalar variable of type default INTEGER, that holds the stream number for error messages. Printing of error messages in SCALE_get, SCALE_apply, SCALE_recover and SCALE_terminate is suppressed if error ≤ 0. The default is error = 6.

out is a scalar variable of type default INTEGER, that holds the stream number for informational messages. Printing of informational messages in SCALE_get, SCALE_apply, SCALE_recover and SCALE_terminate is suppressed if out < 0. The default is out = 6.

print_level is a scalar variable of type default INTEGER, that is used to control the amount of informational output which is required. No informational output will occur if print_level ≤ 0. If print_level $= 1$, a single line of output will be produced for each iteration of the process. If print_level ≥ 2, this output will be increased to provide significant detail of each iteration. The default is print_level = 0.

maxit is a scalar variable of type default INTEGER, that holds the maximum number of scaling iterations which will be allowed in SCALE_get. The default is maxit = 100.

shift_x is a scalar variable of type default INTEGER, that should be set be larger than 0 if shifts should be applied to the variables **x**. No shifts will be applied if shift_x ≤ 0. The default is shift_x = 0.

scale_x is a scalar variable of type default INTEGER, that should be set be larger than 0 if scaling should be applied to the variables **x**. No scaling will be applied if scale_x ≤ 0. The default is scale_x = 0.

shift_c is a scalar variable of type default INTEGER, that should be set be larger than 0 if shifts should be applied to the general constraints. No shifts will be applied if shift_c ≤ 0. The default is shift_c = 0.

scale_c is a scalar variable of type default INTEGER, that should be set be larger than 0 if scaling should be applied to the general constraints No scaling will be applied if scale_c ≤ 0. The default is scale_c = 0.

shift_f is a scalar variable of type default INTEGER, that should be set be larger than 0 if shifts should be applied to the objective function. No shifts will be applied if shift_f ≤ 0. The default is shift_f = 0.

All use is subject to licence. See http://galahad.rl.ac.uk/galahad-www/cou.html.
For any commercial application, a separate license must be signed.

`scale_f` is a scalar variable of type default `INTEGER`, that should be set be larger than 0 if scaling should be applied to the onjective function. No scaling will be applied if `scale_f` ≤ 0. The default is `scale_f = 0`.

`infinity` is a scalar variable of type default `REAL` (double precision in `GALAHAD_SCALE_double`), that is used to specify which constraint bounds are infinite. Any bound larger than `infinity` in modulus will be regarded as infinite. The default is `infinity` = 10^{19}.

`stop_tol` is a scalar variable of type default `REAL` (double precision in `GALAHAD_SCALE_double`), that is specifes the stopping tolerance using for the scaling iteration if required.

`scale_x_min` is a scalar variable of type default `REAL` (double precision in `GALAHAD_SCALE_double`), that is used to specify the minimum permitted variable scale factor. The default is `scale_x_min` = 1, and any specified non-positive value of `scale_x_min` will be interpreted as the default.

`scale_c_min` is a scalar variable of type default `REAL` (double precision in `GALAHAD_SCALE_double`), that is used to specify the minimum permitted constraint scale factor. The default is `scale_c_min` = 1, and any specified non-positive value of `scale_c_min` will be interpreted as the default.

`space_critical` is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if space is critical when allocating arrays and `.FALSE.` otherwise. The package may run faster if `space_critical` is `.FALSE.` but at the possible expense of a larger storage requirement. The default is `space_critical = .FALSE.`.

`deallocate_error_fatal` is a scalar variable of type default `LOGICAL`, that must be set `.TRUE.` if the user wishes to terminate execution if a deallocation fails, and `.FALSE.` if an attempt to continue will be made. The default is `deallocate_error_fatal = .FALSE.`.

`prefix` is a scalar variable of type default `CHARACTER` and length 30, that may be used to provide a user-selected character string to preface every line of printed output. Specifically, each line of output will be prefaced by the string `prefix(2:LEN(TRIM(prefix))-1)`, thus ignoring the first and last non-null components of the supplied string. If the user does not want to preface lines by such a string, they may use the default `prefix = ""`.

2.2.5 The derived data type for holding informational parameters

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

`status` is a scalar variable of type default `INTEGER`, that gives the exit status of the algorithm. See Section 2.4 for details.

`alloc_status` is a scalar variable of type default `INTEGER`, that gives the status of the last attempted array allocation or deallocation. This will be 0 if `status = 0`.

`bad_alloc` is a scalar variable of type default `CHARACTER` and length 80, that gives the name of the last internal array for which there were allocation or deallocation errors. This will be the null string if `status = 0`.

`deviation` is a scalar variable of type default `REAL` (double precision in `GALAHAD_SCALE_double`), that holds the value of the deviation from double-stocasticity when appropriate.

2.2.6 The derived data type for holding problem data

The derived data type `SCALE_data_type` is used to hold all the data for a particular problem, or sequences of problems with the same structure, between calls of `SCALE` procedures. This data should be preserved, untouched, from the initial call to `SCALE_initialize` to the final call to `SCALE_terminate`.

All use is subject to licence. See `http://galahad.rl.ac.uk/galahad-www/cou.html`.
For any commercial application, a separate license must be signed.

2.3 Argument lists and calling sequences

There are five procedures for user calls (see Section 2.5 for further features):

1. The subroutine `SCALE_initialize` is used to set default values, and initialize private data, before solving one or more problems with the same sparsity and bound structure.
2. The subroutine `SCALE_get` is called to compute the scaling factors.
3. The subroutine `SCALE_apply` is called to apply the scaling factors to the data of a QP problem.
4. The subroutine `SCALE_recover` is called to undo the effects of the scaling factors previously applied to a QP problem.
5. The subroutine `SCALE_terminate` is provided to allow the user to automatically deallocate array components of the private data, allocated by `SCALE_get`, at the end of the solution process.

2.3.1 The initialization subroutine

Default values are provided as follows:

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

`data` is a scalar `INTENT(INOUT)` argument of type `SCALE_data_type` (see Section 2.2.6). It is used to hold data about the problem being scaled.

`control` is a scalar `INTENT(OUT)` argument of type `SCALE_control_type` (see Section 2.2.4). On exit, `control` contains default values for the components as described in Section 2.2.4. These values should only be changed after calling `SCALE_initialize`.

`inform` is a scalar `INTENT(OUT)` argument of type `SCALE_inform_type` (see Section 2.2.5). A successful call to `SCALE_initialize` is indicated when the component `status` has the value 0. For other return values of `status`, see Section 2.4.

2.3.2 The subroutine that computes the scaling factors

The scaling factors and shifts are calculated as follows:

```
CALL SCALE_get( prob, scale, trans, data, control, inform )
```

`prob` is a scalar `INTENT(IN)` argument of type `QPT_problem_type` (see Section 2.2.2). It is used to hold data about the problem being scaled. The user must allocate all the array components for the non-parametric problem (1.1)–(1.3), and set values for these components.

The components `prob%X`, `prob%C`, `prob%Y` and `prob%Z` should be set to "typical" estimates of the primal variables, $\mathbf{x}$, general constraint values $\mathbf{Ax}$, Lagrange multipliers for the general constraints, $\mathbf{y}$ and dual variables for the bound constraints, $\mathbf{z}$, respectively. **Restrictions:** `prob%n` > 0, `prob%m` ≥ 0, `prob%A_type` $\in$ {`'DENSE'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'` }, and `prob%H_type` $\in$ {`'DENSE'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'`, `'DIAGONAL'` }.

`scale` is a scalar `INTENT(IN)` argument of type default `INTEGER`, that is used to control problem scaling. Possible values and their consequences are:

$\leq$ `0` or > 7. No scaling will be performed

`1.` Scaling and shifts will be calculated to try to map all variables and constraints to have values between 0 and 1.

**All use is subject to licence. See `http://galahad.rl.ac.uk/galahad-www/cou.html`.
For any commercial application, a separate license must be signed.**

2. The symmetric Curtis-Reid method will be applied to compute scalings to normalize the rows of the matrix

$$\mathbf{K} = \begin{pmatrix} \mathbf{H} & \mathbf{A}^T \\ \mathbf{A} & 0 \end{pmatrix}.$$

3. The unsymmetric Curtis-Reid method will be applied to normalize the rows and columns of **A**.
4. Scaling will be applied to equilibrate the norms of the rows of **A**.
5. Strategy 2 will be followed by strategy 4.
6. Strategy 3 will be followed by strategy 4.
7. Scaling will be applied to equilibrate the rows and columns of **K** using the Sinkhorn-Knopp strategy.

`trans` is a scalar `INTENT(INOUT)` argument of type `SCALE_data_type` (see Section 2.2.3) whose components will be filled as appropriate on output with the scaling factors and shifts for the requested scaling strategy.

`data` is a scalar `INTENT(INOUT)` argument of type `SCALE_data_type` (see Section 2.2.6). 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 `SCALE_initialize`.

`control` is a scalar `INTENT(IN)` argument of type `SCALE_control_type` (see Section 2.2.4). Default values may be assigned by calling `SCALE_initialize` prior to the first call to `SCALE_get`.

`inform` is a scalar `INTENT(INOUT)` argument of type `SCALE_inform_type` (see Section 2.2.5). A successful call to `SCALE_get` is indicated when the component `status` has the value 0. For other return values of `status`, see Section 2.4.

2.3.3 The subroutine that applies the scaling factors and shifts

The scaling factors and shifts are applied to the quadratic programming problem data as follows:

```
CALL SCALE_apply( prob, trans, data, control, inform )
```

The arguments `prob`, `trans`, `data`, `control` and `inform` are as described for `SCALE_get` except that `prob` is now `INTENT(INOUT)` while `trans` is `INTENT(IN)`. On exit, the scalings and shifts recorded in `trans` will be applied to the quadratic programming data input in `prob`; the transformed problem data will be output in `prob`. The transformation will only be applied to the parametric components $\delta\mathbf{g}$, δf, $\delta\mathbf{c}^l$, $\delta\mathbf{c}^u$, $\delta\mathbf{x}^l$ and $\delta\mathbf{x}^u$ of the problem when `prob%DG` is allocated.

A successful call to `SCALE_apply` is indicated when the component `inform%status` has the value 0. For other return values of `inform%status`, see Section 2.4.

2.3.4 The subroutine that "undoes" the scaling factors and shifts

The effects of the scaling factors and shifts on the quadratic programming problem data are "undone" as follows:

```
CALL SCALE_recover( prob, trans, data, control, inform )
```

The arguments `prob`, `trans`, `data`, `control` and `inform` are exactly as described for `SCALE_apply` except that now on exit, the inverses of the scalings and shifts recorded in `trans` will be applied to the quadratic programming data input in `prob`; the unscaled problem data will be output in `prob`. The reverse transformation will only be applied to the parametric components $\delta\mathbf{g}$, δf, $\delta\mathbf{c}^l$, $\delta\mathbf{c}^u$, $\delta\mathbf{x}^l$ and $\delta\mathbf{x}^u$ of the problem when `prob%DG` is allocated.

A successful call to `SCALE_recover` is indicated when the component `inform%status` has the value 0. For other return values of `inform%status`, see Section 2.4.

All use is subject to licence. See `http://galahad.rl.ac.uk/galahad-www/cou.html`.
For any commercial application, a separate license must be signed.

2.3.5 The termination subroutine

All previously allocated arrays are deallocated as follows:

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

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

`control` is a scalar `INTENT(IN)` argument of type `SCALE_control_type` exactly as for `SCALE_get`.

`inform` is a scalar `INTENT(OUT)` argument of type `SCALE_inform_type` exactly as for `SCALE_get`. Only the component `status` will be set on exit, and a successful call to `SCALE_terminate` is indicated when this component `status` has the value 0. For other return values of `status`, see Section 2.4.

2.4 Warning and error messages

A negative value of `inform%status` on exit from `SCALE_get`, `SCALE_apply`, `SCALE_recover` or `SCALE_terminate` indicates that an error has occurred. No further calls should be made until the error has been corrected. Possible values are:

-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. One of the restrictions `prob%n` > 0, `prob%m` ≥ 0, or the requirements that `prob%A_type` contains its relevant string `'DENSE'`, `'COORDINATE'` or `'SPARSE_BY_ROWS'` and `prob%H_type` contain its relevant string `'DENSE'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'` or `'DIAGONAL'` has been violated.

A positive value of `inform%status` is a warning. Possible values are:

18. Too many scaling iterations have been performed. This may happen if `control%maxit` is too small.

2.5 Further features

In this section, we describe an alternative means of setting control parameters, that is components of the variable `control` of type `SCALE_control_type` (see Section 2.2.4), by reading an appropriate data specification file using the subroutine `SCALE_read_specfile`. This facility is useful as it allows a user to change `SCALE` control parameters without editing and recompiling programs that call `SCALE`.

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 `SCALE_read_specfile` must start with a "`BEGIN SCALE`" command and end with an "`END`" command. The syntax of the specfile is thus defined as follows:

All use is subject to licence. See `http://galahad.rl.ac.uk/galahad-www/cou.html`.
For any commercial application, a separate license must be signed.

```
( .. lines ignored by QP_read_specfile .. )
  BEGIN CQP
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by QP_read_specfile .. )
```

where keyword and value are two strings separated by (at least) one blank. The "BEGIN SCALE" 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 QP SPECIFICATION
```

and

```
END QP SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN SCALE" 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 three different types, namely integer, logical or real. 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".

The specification file must be open for input when SCALE_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 SCALE_read_specfile.

2.5.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL SCALE_read_specfile( control, device )
```

control is a scalar INTENT(INOUT)argument of type SCALE_control_type (see Section 2.2.4). Default values should have already been set, perhaps by calling SCALE_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 2.2.4) of control that each affects are given in Table 2.1.

device is a scalar INTENT(IN)argument of type default INTEGER, 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.

2.6 Information printed

If control%print_level is positive, information about the progress of the algorithm will be printed on unit control-%out. If control%print_level > 0, a few lines of output indicating the progress of the computation of the scaling factotrs and shifts may be given.

**All use is subject to licence. See http://galahad.rl.ac.uk/galahad-www/cou.html.
For any commercial application, a separate license must be signed.**

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
maximum-number-of-iterations	%maxit	integer
shift-x	%shift_x	integer
scale-x	%scale_x	integer
shift-c	%shift_c	integer
scale-c	%scale_c	integer
shift-f	%shift_f	integer
scale-f	%scale_f	integer
infinity-value	%infinity	real
stop-tolerance	%stop_tol	real
smallest-x-scaling	%scale_x_min	real
smallest-c-scaling	%scale_x_min	real
space-critical	%space_critical	logical
deallocate-error-fatal	%deallocate_error_fatal	logical
output-line-prefix	%prefix	character

Table 2.1: 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: SCALE_get , SCALE_apply and SCALE_recover call the GALAHAD packages GALAHAD_SYMBOLS, GALAHAD_SPACE, GALAHAD_SPECFILE, GALAHAD_TOOLS, GALAHAD_SMT, GALAHAD_QPT and GALAHAD_TRANS.

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', 'DIAGONAL' }.

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

4 METHOD

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

$$\mathbf{A}\mathbf{x} = \mathbf{c} \tag{4.1}$$

and

$$\mathbf{c}^l \leq \mathbf{c} \leq \mathbf{c}^u, \;\; \mathbf{x}^l \leq \mathbf{x} \leq \mathbf{x}^u, \tag{4.2}$$

the dual optimality conditions

$$\mathbf{H}\mathbf{x} + \mathbf{g} = \mathbf{A}^T\mathbf{y} + \mathbf{z}, \;\; \mathbf{y} = \mathbf{y}^l + \mathbf{y}^u \;\text{ and }\; \mathbf{z} = \mathbf{z}^l + \mathbf{z}^u, \tag{4.3}$$

All use is subject to licence. See http://galahad.rl.ac.uk/galahad-www/cou.html .
For any commercial application, a separate license must be signed.

and

$$\mathbf{y}^l \geq 0, \;\; \mathbf{y}^u \leq 0, \;\; \mathbf{z}^l \geq 0 \;\text{ and }\; \mathbf{z}^u \leq 0, \tag{4.4}$$

and the complementary slackness conditions

$$(\mathbf{A}\mathbf{x} - \mathbf{c}^l)^T \mathbf{y}^l = 0, \;\; (\mathbf{A}\mathbf{x} - \mathbf{c}^u)^T \mathbf{y}^u = 0, \;\; (\mathbf{x} - \mathbf{x}^l)^T \mathbf{z}^l = 0 \;\text{ and }\; (\mathbf{x} - \mathbf{x}^u)^T \mathbf{z}^u = 0, \tag{4.5}$$

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 Curtis-Reid symmetric and unsymmetric matrix scaling procedures are described in

A. R. Curtis and J. K. Reid (1972). On the automatic scaling of matrices for Gaussian elimination. IMA J. Appl. Math. **10(1)** 118-124.

The Sinkhorn-Knopp scaling strategy that aims to scale a symmetric matrix so that it is doubly stochastic (i.e., its rows and columns have unit norm) was proposed by

R. Sinkhorn and P. Knopp (1967). Concerning nonnegative matrices and doubly stochastic matrices. Pacific J. Math. **21(2)** 343-348.

The other strategies are "home grown".

5 EXAMPLE OF USE

Suppose we are considering the quadratic program $\frac{1}{2}x_1^2 + x_2^2 + x_2 x_3 + \frac{3}{2}x_3^2 + 2x_2 + 1$ subject to the the general linear constraints $1 \leq 2x_1 + x_2 \leq 2$ and $x_2 + x_3 = 2$, and simple bounds $-1 \leq x_1 \leq 1$ and $x_3 \leq 2$. Then, on writing the data for this problem as

$$\mathbf{H} = \begin{pmatrix} 1 & & \\ & 2 & 1 \\ & 1 & 3 \end{pmatrix}, \;\; \mathbf{g} = \begin{pmatrix} 0 \\ 2 \\ 0 \end{pmatrix}, \;\; \mathbf{x}^l = \begin{pmatrix} -1 \\ -\infty \\ -\infty \end{pmatrix}, \;\; \mathbf{x}^u = \begin{pmatrix} 1 \\ \infty \\ 2 \end{pmatrix}$$

and

$$\mathbf{A} = \begin{pmatrix} 2 & 1 & \\ & 1 & 1 \end{pmatrix}, \;\; \mathbf{c}^l = \begin{pmatrix} 1 \\ 2 \end{pmatrix}, \;\text{ and }\; \mathbf{c}^u = \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

in sparse co-ordinate format, we may transform the problem using Sinkhorn-Knopp scaling using the following code:

```
! THIS VERSION: GALAHAD 2.4 - 17/01/2011 AT 15:30 GMT.
   PROGRAM GALAHAD_SCALE_EXAMPLE
   USE GALAHAD_SCALE_double                       ! double precision version
   USE GALAHAD_SMT_double
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 )      ! set precision
   REAL ( KIND = wp ), PARAMETER :: infinity = 10.0_wp ** 20
   TYPE ( QPT_problem_type ) :: p
   TYPE ( SCALE_trans_type ) :: trans
   TYPE ( SCALE_data_type ) :: data
   TYPE ( SCALE_control_type ) :: control
   TYPE ( SCALE_inform_type ) :: inform
   INTEGER :: s, scale
   INTEGER, PARAMETER :: n = 3, m = 2, h_ne = 4, a_ne = 4
! start problem data
   ALLOCATE( p%G( n ), p%X_l( n ), p%X_u( n ) )
   ALLOCATE( p%C( m ), p%C_l( m ), p%C_u( m ) )
   ALLOCATE( p%X( n ), p%Y( m ), p%Z( n ) )
   p%n = n ; p%m = m ; p%f = 1.0_wp               ! dimensions & obj constant
```

All use is subject to licence. See `http://galahad.rl.ac.uk/galahad-www/cou.html`.
For any commercial application, a separate license must be signed.

```
   p%G = (/ 0.0_wp, 2.0_wp, 0.0_wp /)              ! objective gradient
   p%C_l = (/ 1.0_wp, 2.0_wp /)                    ! constraint lower bound
   p%C_u = (/ 2.0_wp, 2.0_wp /)                    ! constraint upper bound
   p%X_l = (/ - 1.0_wp, - infinity, - infinity /) ! variable lower bound
   p%X_u = (/ 1.0_wp, infinity, 2.0_wp /)          ! variable upper bound
   p%X = 0.0_wp ; p%Y = 0.0_wp ; p%Z = 0.0_wp      ! typical values for x, y & z
   p%C = 0.0_wp                                    ! c = A * x
!  sparse co-ordinate storage format
   CALL SMT_put( p%H%type, 'COORDINATE', s )       ! specify co-ordinate
   CALL SMT_put( p%A%type, 'COORDINATE', s )       ! storage for H and A
   ALLOCATE( p%H%val( h_ne ), p%H%row( h_ne ), p%H%col( h_ne ) )
   ALLOCATE( p%A%val( a_ne ), p%A%row( a_ne ), p%A%col( a_ne ) )
   p%H%val = (/ 1.0_wp, 2.0_wp, 1.0_wp, 3.0_wp /) ! Hessian H
   p%H%row = (/ 1, 2, 2, 3 /)                      ! NB lower triangle
   p%H%col = (/ 1, 2, 1, 3 /) ; p%H%ne = h_ne
   p%A%val = (/ 2.0_wp, 1.0_wp, 1.0_wp, 1.0_wp /) ! Jacobian A
   p%A%row = (/ 1, 1, 2, 2 /)
   p%A%col = (/ 1, 2, 2, 3 /) ; p%A%ne = a_ne
! problem data complete - compute and apply scale factors
   CALL SCALE_initialize( data, control, inform ) ! Initialize controls
   control%infinity = infinity                     ! Set infinity
   scale = 7                                       ! Sinkhorn-Knopp scaling
   CALL SCALE_get( p, scale, trans, data, control, inform ) ! Get scalings
   IF ( inform%status == 0 ) THEN                  !  Successful return
     WRITE( 6, "( ' variable scalings : ', /, ( 5ES12.4 ) )" ) trans%X_scale
     WRITE( 6, "( ' constraint scalings : ', /, ( 5ES12.4 ) )" ) trans%C_scale
   ELSE                                            !  Error returns
     WRITE( 6, "( ' SCALE_get exit status = ', I6 ) " ) inform%status
   END IF
   CALL SCALE_apply( p, trans, data, control, inform )
   IF ( inform%status == 0 ) THEN                  !  Successful return
     WRITE( 6, "( ' scaled A : ', /, ( 5ES12.4 ) )" ) p%A%val
   ELSE                                            !  Error returns
     WRITE( 6, "( ' SCALE_get exit status = ', I6 ) " ) inform%status
   END IF
   CALL SCALE_terminate( data, control, inform, trans )  !  delete workspace
   END PROGRAM GALAHAD_SCALE_EXAMPLE
```

This produces the following output:

```
 variable scalings :
  7.0711E-01  7.0711E-01  5.7735E-01
 constraint scalings :
  7.0711E-01  1.2968E+00
 scaled A :
  1.0000E+00  5.0000E-01  9.1700E-01  7.4873E-01
```

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

```
!  sparse co-ordinate storage format
...
! problem data complete
```

by

```
! sparse row-wise storage format
   CALL SMT_put( p%H%type, 'SPARSE_BY_ROWS' )  ! Specify sparse-by-row
```

All use is subject to licence. See `http://galahad.rl.ac.uk/galahad-www/cou.html`**.**
For any commercial application, a separate license must be signed.

```
   ALLOCATE( p%H%val( h_ne ), p%H%col( h_ne ), p%H%ptr( n + 1 ) )
   ALLOCATE( p%A%val( a_ne ), p%A%col( a_ne ), p%A%ptr( m + 1 ) )
   p%H%val = (/ 1.0_wp, 2.0_wp, 1.0_wp, 3.0_wp /) ! Hessian H
   p%H%col = (/ 1, 2, 3, 3 /)                     ! NB lower triangular
   p%H%ptr = (/ 1, 2, 3, 5 /)                     ! Set row pointers
! problem data complete
```

or using a dense storage format with the replacement lines

```
! dense storage format
   CALL SMT_put( p%H%type, 'DENSE' )  ! Specify dense
   ALLOCATE( p%H%val( n * ( n + 1 ) / 2 ) )
   p%H%val = (/ 1.0_wp, 0.0_wp, 2.0_wp, 0.0_wp, 1.0_wp, 3.0_wp /) ! Hessian
! problem data complete
```

respectively.

If instead **H** had been the diagonal matrix

$$\mathbf{H} = \begin{pmatrix} 1 & & \\ & 2 & \\ & & 3 \end{pmatrix}$$

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 = (/ 1.0_wp, 2.0_wp, 3.0_wp /) ! Hessian values
```

All use is subject to licence. See `http://galahad.rl.ac.uk/galahad-www/cou.html`**.**
For any commercial application, a separate license must be signed.