GALAHAD CRO

USER DOCUMENTATION GALAHAD Optimization Library version 5.0

1 SUMMARY

This package provides a **crossover** from a solution to the **convex quadratic programming problem**

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

subject to the general linear constraints

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

and the simple bound constraints

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

found by an interior-point method to one in which the **matrix of defining active constraints/variables is of full rank.** Here, the n by n symmetric, positive-semi-definite matrix $\mathbf{H}$, the vectors $\mathbf{g}$, $\mathbf{a}_i$, $\mathbf{c}^l$, $\mathbf{c}^u$, $\mathbf{x}^l$, $\mathbf{x}^u$, the scalar f are given. In addition a solution $\mathbf{x}$ along with optimal Lagrange multipliers $\mathbf{y}$ for the general constraints and dual variables $\mathbf{z}$ for the simple bounds must be provided (see Section 4). These will be adjusted as necessary. Any of the constraint bounds c_i^l, c_i^u, x_j^l and x_j^u may be infinite. Full advantage is taken of any zero coefficients in the matrix $\mathbf{H}$ or the matrix $\mathbf{A}$ of vectors $\mathbf{a}_i$.

ATTRIBUTES — Versions: GALAHAD_CRO_single, GALAHAD_CRO_double. **Uses:** GALAHAD_CLOCK, GALAHAD_SYMBOLS, GALAHAD_SPACE, GALAHAD_SPECFILE, GALAHAD_TOOLS. GALAHAD_QPT, GALAHAD_SCU, GALAHAD_SLS, GALAHAD_ULS, **Date:** January 2012. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003. **Parallelism:** Some options may use OpenMP and its runtime library.

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_CRO_single
```

with the obvious substitution GALAHAD_CRO_double, GALAHAD_CRO_single_64 and GALAHAD_CRO_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_type, QPT_problem_type, NLPT_userdata_type, CRO_time_type, CRO_control_type, CRO_inform_type and CRO_data_type (Section 2.3) and the subroutines CRO_initialize, CRO_crossover, CRO_terminate, (Section 2.4) and CRO_read_specfile (Section 2.6) must be renamed on one of the USE statements.

2.1 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.

2.2 Parallel usage

OpenMP may be used by the GALAHAD_CRO package to provide parallelism for some solvers in shared memory environments. See the documentation for the GALAHAD package SLS for more details. To run in parallel, OpenMP must be enabled at compilation time by using the correct compiler flag (usually some variant of -openmp). The number of threads may be controlled at runtime by setting the environment variable OMP_NUM_THREADS.

MPI may also be used by the package to provide parallelism for some solvers in a distributed memory environment. To use this form of parallelism, MPI must be enabled at runtime by using the correct compiler flag (usually some variant of -lmpi). Although the MPI process will be started automatically when required, it should be stopped by the calling program once no further use of this form of parallelism is needed. Typically, this will be via statements of the form

```
CALL MPI_INITIALIZED( flag, ierr )
IF ( flag ) CALL MPI_FINALIZE( ierr )
```

The code may be compiled and run in serial mode.

2.3 The derived data types

Four derived data types are accessible from the package.

2.3.1 The derived data type for holding control parameters

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

error is a scalar variable of type INTEGER(ip_), that holds the stream number for error messages. Printing of error messages in CRO_crossover and CRO_terminate is suppressed if error ≤ 0. The default is error = 6.

out is a scalar variable of type INTEGER(ip_), that holds the stream number for informational messages. Printing of informational messages in CRO_crossover is suppressed if out < 0. The default is out = 6.

print_level is a scalar variable of type INTEGER(ip_), 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.

max_schur_complement is a scalar variable of type INTEGER(ip_), that specifies the maximum number of columns permitted in the Schur complement when updating the solution (see Section 4) before a re-factorization is triggered. The default is max_schur_complement = 75.

infinity is a scalar variable of type REAL(rp_), 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}.

feasibility_tolerance is a scalar variables of type REAL(rp_), that specifies the maximum violation of the KKT conditions that is permitted. The default iw feasibility_tolerance = u, where u is EPSILON(1.0) (EPSILON(1.0D0) in GALAHAD_CRO_double).

check_io is a scalar variable of type default LOGICAL, that must be set .TRUE. if the user wishes to check the input and output data and .FALSE. otherwise. The package may run faster if space_critical is .FALSE. but at the possible expense of errors The default is check_io = .TRUE..

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..

symmetric_linear_solver is a scalar variable of type default CHARACTER and length 30, that specifies the external package to be used to solve any symmetric linear system that might arise. Current possible choices are 'sils', 'ma27', 'ma57', 'ma77', 'ma86', 'ma97', ssids, 'pardiso' and 'wsmp', although only 'sils' and, for OMP 4.0-compliant compilers, 'ssids' are installed by default. See the documentation for the GALAHAD package SLS for further details. The default is symmetric_linear_solver = 'sils', but we recommend 'ma97' if it available.

unsymmetric_linear_solver is a scalar variable of type default CHARACTER and length 30, that specifies the external package to be used to solve any unsymmetric linear systems that might arise. Possible choices are 'gls', 'ma28' and 'ma48', although only 'gls' is installed by default. See the documentation for the GALAHAD package ULS for further details. The default is unsymmetric_linear_solver = 'gls', but we recommend 'ma48' if it available.

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 = "".

SLS_control is a scalar variable argument of type SLS_control_type that is used to pass control options to external packages used to solve any symmetric linear systems that might arise. See the documentation for the GALAHAD package SLS for further details. In particular, default values are as for SLS.

ULS_control is a scalar variable argument of type ULS_control_type that is used to pass control options to external packages used to solve any unsymmetric linear systems that might arise. See the documentation for the GALAHAD package ULS for further details. In particular, default values are as for ULS.

2.3.2 The derived data type for holding timing information

The derived data type CRO_time_type is used to hold elapsed CPU and system clock times for the various parts of the calculation. The components of CRO_time_type are:

total is a scalar variable of type REAL(rp_), that gives the total CPU time spent in the package.

analyse is a scalar variable of type REAL(rp_), that gives the CPU time spent analysing the required matrices prior to factorization.

factorize is a scalar variable of type REAL(rp_), that gives the CPU time spent factorizing the required matrices.

solve is a scalar variable of type REAL(rp_), that gives the CPU time spent computing corrections to the current solution.

clock_total is a scalar variable of type REAL(rp_), that gives the total elapsed system clock time spent in the package.

clock_analyse is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent analysing the required matrices prior to factorization.

clock_factorize is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent factorizing the required matrices.

clock_solve is a scalar variable of type REAL(rp_), that gives the elapsed system clock time spent computing corrections to the current solution.

2.3.3 The derived data type for holding informational parameters

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

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

alloc_status is a scalar variable of type INTEGER(ip_), 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.

dependent is a scalar variable of type INTEGER(ip_), that gives the number of dependent active constraints.

time is a scalar variable of type CRO_time_type whose components are used to hold elapsed CPU and system clock times for the various parts of the calculation (see Section 2.3.2).

SLS_control is a scalar variable argument of type SLS_control_type that is used to pass control options to external packages used to solve any symmetric linear systems that might arise. See the documentation for the GALAHAD package SLS for further details. In particular, default values are as for SLS.

ULS_control is a scalar variable argument of type ULS_control_type that is used to pass control options to external packages used to solve any unsymmetric linear systems that might arise. See the documentation for the GALAHAD package ULS for further details. In particular, default values are as for ULS.

scu_status is a scalar variable of type INTEGER(ip_), that gives the return status from the Schur-complement updating package GALAHAD_SCU. See the specification sheet for GALAHAD_SCU for details.

SCU_inform is a scalar variable of type SCU_info_type whose components are used to provide information about Schur-complement updating applied by the package GALAHAD_SCU. See the specification sheet for GALAHAD_SCU for details.

2.3.4 The derived data type for holding problem data

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

2.4 Argument lists and calling sequences

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

1. The subroutine CRO_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 CRO_crossover is called to solve the problem.

3. The subroutine CRO_terminate is provided to allow the user to automatically deallocate array components of the private data, allocated by CRO_crossover, at the end of the solution process.

We use square brackets [ ] to indicate OPTIONAL arguments.

2.4.1 The initialization subroutine

Default values are provided as follows:

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

data is a scalar INTENT(INOUT) argument of type CRO_data_type (see Section 2.3.4). It is used to hold private data used by the crossover algorithm.

control is a scalar INTENT(OUT) argument of type CRO_control_type (see Section 2.3.1). On exit, control contains default values for the components as described in Section 2.3.1. These values should only be changed after calling CRO_initialize.

inform is a scalar INTENT(OUT) argument of type CRO_inform_type (see Section 2.3.3). A successful call to CRO_initialize is indicated when the component status has the value 0. For other return values of status, see Section 2.5.

2.4.2 The crossover subroutine

The crossover algorithm is called as follows:

```
CALL CRO_crossover( n, m, m_equal, H_val, H_col, H_ptr, A_val, &
                    A_col, A_ptr, G, C_l, C_u, X_l, X_u, C, X, Y, &
                    Z, C_stat, X_stat, data, control, inform )
```

n is a scalar INTENT(IN) argument of type INTEGER(ip_), that must be set to the number of optimization variables, n. **Restriction:** n > 0.

m is a scalar INTENT(IN) argument of type INTEGER(ip_), that must be set to the number of general linear constraints, m. **Restriction:** m ≥ 0.

m_equal is a scalar INTENT(IN) argument of type INTEGER(ip_), that must be set to the number of general linear constraints that are equalities, i.e., whose lower and upper bounds coincide. **Restriction:** m $\geq$ m_equal > 0.

H_val is an INTENT(IN) rank-one array argument of dimension H_ptr(n+1)-1 and type REAL(rp_), that must be set to hold the values of the nonzero entries of the **lower triangular** part of the Hessian matrix **H**. The entries must be ordered so that those in row i appear directly before those in row $i+1$ with no gaps; the order within each row is unimportant.

H_col is an INTENT(IN) rank-one array argument of dimension H_ptr(n+1)-1 and type INTEGER(ip_), that must be set to hold the column indices of the **lower triangular** part of **H**. These must be ordered so that they correspond to the values stored in H_val.

H_ptr is an INTENT(IN) rank-one array argument of dimension n+1 and type INTEGER(ip_), whose j-th component, $j = 1, \ldots, n$, holds the starting position of row j of the **lower triangular** part of **H** as stored in H_val and H_col. The $n+1$-st component must be set to the total number of entries in the lower triangular part of **H** plus one.

A_val is an INTENT(IN) rank-one array argument of dimension A_ptr(m+1)-1 and type REAL(rp_), that must be set to hold the values of the nonzero entries of the Jacobian matrix **A**. The entries must be ordered so that those in row i appear directly before those in row $i+1$ with no gaps; the order within each row is unimportant. **Restriction:** the rows of **A must be ordered** so that the first m_equal are equality constraints.

A_col is an INTENT(IN) rank-one array argument of dimension A_ptr(m+1)-1 and type INTEGER(ip_), that must be set to hold the column indices of $\mathbf{A}$. These must be ordered so that they correspond to the values stored in A_val.

A_ptr is an INTENT(IN) rank-one array argument of dimension m+1 and type INTEGER(ip_), whose i-th component, $i = 1,\ldots,m$, holds the starting position of row i of $\mathbf{A}$ as stored in A_val and A_col. The $m+1$-st component must be set to the total number of entries in $\mathbf{A}$ plus one.

G is an INTENT(IN) rank-one array argument of dimension n and type REAL(rp_), that holds the gradient $\mathbf{g}$ of the linear term of the quadratic objective function. The j-th component of G, $j = 1,\ldots,n$, must be set to $\mathbf{g}_j$.

C_l is an INTENT(IN) rank-one array argument 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, $i = 1,\ldots,m$, must be set to $\mathbf{c}^l_i$. 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 2.3.1).

C_u is an INTENT(IN) rank-one array argument 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, $i = 1,\ldots,m$, must be set to $\mathbf{c}^u_i$. 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 2.3.1).

X_l is an INTENT(IN) rank-one array argument 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, $j = 1,\ldots,n$, must be set to $\mathbf{x}^l_j$. 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 2.3.1).

X_u is an INTENT(IN) rank-one array argument 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, $j = 1,\ldots,n$, must be set to $\mathbf{x}^u_j$. 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 2.3.1).

X is an INTENT(INOUT) rank-one array argument of dimension n and type REAL(rp_), that holds the values $\mathbf{x}$ of the optimization variables. The j-th component of X, $j = 1,\ldots,n$, must be set to x_j on input, and may have been adjusted to provide another solution on output.

C is an INTENT(INOUT) rank-one array argument of dimension m and type REAL(rp_), that holds the values $\mathbf{Ax}$ of the constraints. The i-th component of C, $i = 1,\ldots,m$, must be set to $\mathbf{a}_i^T\mathbf{x} \equiv (\mathbf{Ax})_i$ on input, and may have been adjusted to provide another solution on output.

Y is an INTENT(INOUT) rank-one array argument 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, $i = 1,\ldots,m$, must be set to y_i on input, and may have been adjusted to provide another solution on output.

Z is an INTENT(INOUT) rank-one array argument of dimension n and type 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, $j = 1,\ldots,n$, must be set to z_j on input, and may have been adjusted to provide another solution on output.

C_stat is an INTENT(IN) rank-one INTENT(INOUT) array argument of dimension m and type INTEGER(ip_), that indicates which of the general linear constraints are in the current active set. Possible input values for C_stat(i), i$= 1,\ldots,$ m, and their meanings are

<0 the i-th general constraint is in the active set on its lower bound, i.e., $\mathbf{a}_i^T\mathbf{x} = \mathbf{c}^l_i$,

>0 the i-th general constraint is in the active set on its upper bound, i.e., $\mathbf{a}_i^T\mathbf{x} = \mathbf{c}^u_i$, and

0 the i-th general constraint is not in the active set, i.e., $\mathbf{c}_i^l < \mathbf{a}_i^T \mathbf{x} < \mathbf{c}_i^u$.

On output, the value of C_stat(i) may have changed to mean

-1 the i-th general constraint is both independent and active on its lower bound,

-2 the i-th general constraint is on its lower bound but linearly dependent on others,

1 the i-th general constraint is both independent and active on its upper bound,

2 the i-th general constraint is on its upper bound but linearly dependent on others, and

0 the i-th general constraint is not in the active set.

X_stat is an INTENT(IN) rank-one INTENT(INOUT) array argument of dimension n and type INTEGER(ip_), that indicates which of the simple bound constraints are in the current active set. Possible input values for X_stat(j), j= $1,\ldots,$ n, and their meanings are

<0 the j-th simple bound constraint is in the active set on its lower bound, i.e., $\mathbf{x}_j = \mathbf{x}_j^l$,

>0 the j-th simple bound constraint is in the active set on its upper bound, i.e., $\mathbf{x}_j = \mathbf{x}_j^u$, and

0 the j-th simple bound constraint is not in the active set, i.e., $\mathbf{x}_j^l < \mathbf{x}_j < \mathbf{x}_j^u$.

On output, the value of X_stat(j) may have changed to mean

-1 the j-th simple bound constraint is both independent and active on its lower bound,

-2 the j-th simple bound constraint is on its lower bound but linearly dependent on others,

1 the j-th simple bound constraint is both independent and active on its upper bound,

2 the j-th simple bound constraint is on its upper bound but linearly dependent on others, and

0 the j-th simple bound constraint is not in the active set.

data is a scalar INTENT(INOUT) argument of type CRO_data_type (see Section 2.3.4). It is used to hold private data used by the crossover algorithm and must not have been altered **by the user** since the last call to CRO_initialize.

control is a scalar INTENT(IN) argument of type CRO_control_type (see Section 2.3.1). Default values may be assigned by calling CRO_initialize prior to the first call to CRO_crossover.

inform is a scalar INTENT(INOUT) argument of type CRO_inform_type (see Section 2.3.3). A successful call to CRO_crossover is indicated when the component status has the value 0. For other return values of status, see Section 2.5.

2.4.3 The termination subroutine

All previously allocated arrays are deallocated as follows:

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

data is a scalar INTENT(INOUT) argument of type CRO_data_type exactly as for CRO_crossover, that must not have been altered **by the user** since the last call to CRO_initialize. On exit, array components will have been deallocated.

control is a scalar INTENT(IN) argument of type CRO_control_type exactly as for CRO_crossover.

inform is a scalar INTENT(OUT) argument of type CRO_inform_type exactly as for CRO_crossover. Only the component status will be set on exit, and a successful call to CRO_terminate is indicated when this component status has the value 0. For other return values of status, see Section 2.5.

2.5 Warning and error messages

A negative value of inform%status on exit from CRO_crossover or CRO_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 or the requirement that prob%H_type contain its relevant string 'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS' or 'DIAGONAL' when **H** is available, has been violated.

-4. The bound constraints are inconsistent.

-7. The objective function appears to be unbounded from below on the feasible set.

-9. An error was reported by SLS_analyse. The return status from SLS_analyse is given in inform%SLS_inform%-status. See the documentation for the GALAHAD package SLS for further details.

-10. An error was reported by SLS_factorize or SCU_append. The return status from SLS_factorize is given in inform%SLS_inform%status and that from SCU_append in inform%scu_status. See the documentation for the GALAHAD packages SLS and SCU for further details.

-11. An error was reported by SLS_solve or SCU_solve. The return status from SLS_solve is given in inform%SLS_-inform%status and that from SCU_solve in inform%scu_status. See the documentation for the GALAHAD packages SLS and SCU for further details.

-13. An error was reported by ULS_factorize. The return status from ULS_factorize is given in inform%uls_-factorize_status. See the documentation for the GALAHAD package ULS for further details.

-14. An error was reported by ULS_solve. The return status from ULS_solve is given in inform%uls_solve_status. See the documentation for the GALAHAD package ULS for further details.

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 CRO_control_type (see Section 2.3.1), by reading an appropriate data specification file using the subroutine CRO_read_specfile. This facility is useful as it allows a user to change CRO control parameters without editing and recompiling programs that call CRO.

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

```
( .. lines ignored by CRO_read_specfile .. )
  BEGIN CRO
     keyword    value
     .......    .....
     keyword    value
  END
( .. lines ignored by CRO_read_specfile .. )
```

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

and

```
END CRO SPECIFICATION
```

are acceptable. Furthermore, between the "`BEGIN CRO`" 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 `CRO_read_specfile` is called, and the associated device number passed to the routine in device (see below). Note that the corresponding file is `REWIND`ed, which makes it possible to combine the specifications for more than one program/routine. For the same reason, the file is not closed by `CRO_read_specfile`.

2.6.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL CRO_read_specfile( control, device )
```

`control` is a scalar `INTENT(INOUT)` argument of type `CRO_control_type` (see Section 2.3.1). Default values should have already been set, perhaps by calling `CRO_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 2.3.1) of `control` that each affects are given in Table 2.1.

`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`.

2.7 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` $= 1$, a single line indicating how many dependent constraints will be removed. If `control%print_level` ≥ 2, this output will be increased to provide details of the dependent constraints, while if `control%print_level` ≥ 5, full debugging details (probably only of interest to the code developer) are provided.

command	component of `control`	value type
`error-printout-device`	`%error`	integer
`printout-device`	`%out`	integer
`print-level`	`%print_level`	integer
`maximum-dimension-of-schur-complement`	`%max_schur_complement`	integer
`infinity-value`	`%infinity`	real
`feasibility-tolerance`	`%feasibility_tol`	real
`check-input-output`	`%check_io`	logical
`space-critical`	`%space_critical`	logical
`deallocate-error-fatal`	`%deallocate_error_fatal`	logical
`output-line-prefix`	`%prefix`	character
`symmetric-linear-equation-solver`	`%symmetric_linear_solver`	character
`unsymmetric-linear-equation-solver`	`%unsymmetric_linear_solver`	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: `CRO_crossover` calls the GALAHAD packages `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_SPECFILE`, `GALAHAD_TOOLS`. `GALAHAD_QPT`, `GALAHAD_SCU`, `GALAHAD_SLS` and `GALAHAD_ULS`.

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

Restrictions: `n` > 0, `m` $\geq$ *m_equal*, `m_equal` ≥ 0, `prob%A_type` and `prob%H_type` $\in$ {`'DENSE'`, `'COORDINATE'`, `'SPARSE_BY_ROWS'`, `'DIAGONAL'` }. (if **H** and **A** are explicit).

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

4 METHOD

Any required solution **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

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

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.

Denote the active constraints by $\mathbf{A}_A\mathbf{x} = \mathbf{c}_A$ and the active bounds by $\mathbf{I}_A\mathbf{x} = \mathbf{x}_A$. Then any optimal solution satisfies the linear system

$$\begin{pmatrix} \mathbf{H} & -\mathbf{A}_A^T & -\mathbf{I}_A^T \\ \mathbf{A}_A & 0 & 0 \\ \mathbf{I}_A & 0 & 0 \end{pmatrix} \begin{pmatrix} \mathbf{x} \\ \mathbf{y}_A \\ \mathbf{z}_A \end{pmatrix} = \begin{pmatrix} -\mathbf{g} \\ \mathbf{c}_A \\ \mathbf{x}_A \end{pmatrix},$$

where $\mathbf{y}_A$ and $\mathbf{z}_A$ are the corresponding active Lagrange multipliers and dual variables respectively. Consequently the difference between any two solutions $(\Delta\mathbf{x}, \Delta\mathbf{y}, \Delta\mathbf{z})$ must satisfy

$$\begin{pmatrix} \mathbf{H} & -\mathbf{A}_A^T & -\mathbf{I}_A^T \\ \mathbf{A}_A & 0 & 0 \\ \mathbf{I}_A & 0 & 0 \end{pmatrix} \begin{pmatrix} \Delta\mathbf{x} \\ \Delta\mathbf{y}_A \\ \Delta\mathbf{z}_A \end{pmatrix} = 0. \tag{4.6}$$

Thus there can only be multiple solution if the coefficient matrix $\mathbf{K}$ of (4.6) is singular. The algorithm used in GALAHAD_CRO exploits this. The matrix $\mathbf{K}$ is checked for singularity using the GALAHAD package `GALAHAD_ULS`. If $\mathbf{K}$ is non singular, the solution is unique and the solution input by the user provides a linearly independent active set. Otherwise $\mathbf{K}$ is singular, and partitions $\mathbf{A}_A^T = (\mathbf{A}_{AB}^T \;\; \mathbf{A}_{AN}^T)$ and $\mathbf{I}_A^T = (\mathbf{I}_{AB}^T \;\; \mathbf{I}_{AN}^T)$ are found so that

$$\begin{pmatrix} \mathbf{H} & -\mathbf{A}_{AB}^T & -\mathbf{I}_{AB}^T \\ \mathbf{A}_{AB} & 0 & 0 \\ \mathbf{I}_{AB} & 0 & 0 \end{pmatrix}$$

is non-singular and the "non-basic" constraints $\mathbf{A}_{AN}^T$ and $\mathbf{I}_{AN}^T$ are linearly dependent on the "basic" ones $(\mathbf{A}_{AB}^T \;\; \mathbf{I}_{AB}^T)$. In this case (4.6) is equivalent to

$$\begin{pmatrix} \mathbf{H} & -\mathbf{A}_{AB}^T & -\mathbf{I}_{AB}^T \\ \mathbf{A}_{AB} & 0 & 0 \\ \mathbf{I}_{AB} & 0 & 0 \end{pmatrix} \begin{pmatrix} \Delta\mathbf{x} \\ \Delta\mathbf{y}_{AB} \\ \Delta\mathbf{z}_{AB} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{AN}^T \\ 0 \\ 0 \end{pmatrix} \Delta\mathbf{y}_{AN} + \begin{pmatrix} \mathbf{I}_{AN}^T \\ 0 \\ 0 \end{pmatrix} \Delta\mathbf{z}_{AN}. \tag{4.7}$$

Thus, starting from the user's $(\mathbf{x}, \mathbf{y}, \mathbf{z})$ and with a factorization of the coefficient matrix of (4.7) found by the GALAHAD package `GALAHAD_SLS`, the alternative solution $(\mathbf{x} + \alpha\mathbf{x}, \mathbf{y} + \alpha\mathbf{y}, \mathbf{z} + \alpha\mathbf{z})$, featuring $(\Delta\mathbf{x}, \Delta\mathbf{y}_{AB}, \Delta\mathbf{z}_{AB})$ from (4.7) in which successively one of the components of $\Delta\mathbf{y}_{AN}$ and $\Delta\mathbf{z}_{AN}$ in turn is non zero, is taken. The scalar α at each stage is chosen to be the largest possible that guarantees (4.4); this may happen when a non-basic multiplier/dual variable reaches zero, in which case the corresponding constraint is disregarded, or when this happens for a basic multiplier/dual variable, in which case this constraint is exchanged with the non-basic one under consideration and disregarded. The latter corresponds to changing the basic-non-basic partition in (4.7), and subsequent solutions may be found by updating the factorization of the coefficient matrix in (4.7) following the basic-non-basic swap using the GALAHAD package `GALAHAD_SCU`.

5 EXAMPLE OF USE

Suppose we have solved the quadratic program

$$\begin{array}{ll} \underset{\mathbf{x}}{\text{minimize}} & \frac{1}{2}\sum_{i=1}^{11} x_i^2 + \frac{1}{2}\sum_{i=1}^{10} x_i x_{i+1} + \frac{1}{2}x_1 - \frac{1}{2}x_2 - \sum_{i=1}^{10} x_i - \frac{1}{2}x_{11} \\ \text{subject to} & \sum_{i=1}^{11} x_i = 10, \;\; \sum_{i=3}^{11} x_i \geq 9, \;\; \sum_{i=2}^{11} x_i \leq 10 \\ \text{and} & x_1 \geq 0, \;\; x_i \geq 1 \;\text{ for }\; i = 2, \ldots, 11 \end{array}$$

(using, for example, GALAHAD's `CQP` package), and have found the primal-dual solution $\mathbf{x} = (0, 1, 1, \ldots, 1)$, $\mathbf{y} = (-1, \frac{3}{2}, -2)$ and $\mathbf{z} = (2, 4, \frac{5}{2}, \frac{5}{2}, \ldots \frac{5}{2})$ for which all variables and constraints are active; clearly such a solution has dependent active constraints. Then we may find a crossover solution in which the defining active set is linearly independent using the following code:

```
! THIS VERSION: GALAHAD 2.5 - 06/01/2012 AT 08:30 GMT.
   PROGRAM GALAHAD_CRO_EXAMPLE
   USE GALAHAD_CRO_double                    ! double precision version
   IMPLICIT NONE
   INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
   REAL ( KIND = wp ), PARAMETER :: infinity = 10.0_wp ** 20
   TYPE ( CRO_data_type ) :: data
   TYPE ( CRO_control_type ) :: control
   TYPE ( CRO_inform_type ) :: inform
   INTEGER :: i
   INTEGER, PARAMETER :: n = 11, m = 3, m_equal = 1, a_ne = 30, h_ne = 21
   INTEGER, DIMENSION( h_ne ) :: H_col
   INTEGER, DIMENSION( n + 1 ) :: H_ptr
   REAL ( KIND = wp ), DIMENSION( h_ne ) :: H_val
   INTEGER, DIMENSION( a_ne ) :: A_col
   INTEGER, DIMENSION( m + 1 ) :: A_ptr
   REAL ( KIND = wp ), DIMENSION( a_ne ) :: A_val
   REAL ( KIND = wp ), DIMENSION( n ) :: G, X_l, X_u, X, Z
   REAL ( KIND = wp ), DIMENSION( m ) :: C_l, C_u, C, Y
   INTEGER, DIMENSION( m ) :: C_stat
   INTEGER, DIMENSION( n ) :: X_stat
! start problem data
   H_val = (/ 1.0D+0, 5.0D-1, 1.0D+0, 5.0D-1, 1.0D+0, 5.0D-1, 1.0D+0, 5.0D-1,  &
              1.0D+0, 5.0D-1, 1.0D+0, 5.0D-1, 1.0D+0, 5.0D-1, 1.0D+0, 5.0D-1,  &
              1.0D+0, 5.0D-1, 1.0D+0, 5.0D-1, 1.0D+0 /) ! H values
   H_col = (/ 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9, 10, 10,    &
              11 /)                          ! H columns
   H_ptr = (/ 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22 /) ! pointers to H col
   A_val  = (/ 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, &
               1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, &
               1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, &
               1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0 /) ! A values
   A_col = (/ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 3, 4, 5, 6, 7, 8, 9, 10, 11,  &
              2, 3, 4, 5, 6, 7, 8, 9, 10, 11 /) ! A columns
   A_ptr = (/ 1, 12, 21, 31 /)              ! pointers to A columns
   G   = (/ 5.0D-1, -5.0D-1, -1.0D+0, -1.0D+0, -1.0D+0,  -1.0D+0, -1.0D+0,     &
          -1.0D+0, -1.0D+0, -1.0D+0, -5.0D-1 /) ! objective gradient
   C_l = (/  1.0D+1, 9.0D+0, - infinity /)       ! constraint lower bound
   C_u = (/  1.0D+1, infinity, 1.0D+1 /)         ! constraint upper bound
   X_l = (/ 0.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0,                    &
           1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0 /) ! variable lower bound
   X_u  = (/ infinity, infinity, infinity, infinity, infinity, infinity,       &
             infinity, infinity, infinity, infinity, infinity /) ! upper bound
   C = (/ 1.0D+1, 9.0D+0, 1.0D+1 /)              ! optimal constraint value
   X = (/ 0.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0, 1.0D+0,      &
          1.0D+0, 1.0D+0, 1.0D+0 /)              ! optimal variables
   Y = (/  -1.0D+0, 1.5D+0, -2.0D+0 /)           ! optimal Lagrange multipliers
   Z = (/ 2.0D+0, 4.0D+0, 2.5D+0, 2.5D+0, 2.5D+0, 2.5D+0,                      &
          2.5D+0, 2.5D+0, 2.5D+0, 2.5D+0, 2.5D+0 /) ! optimal dual variables
   C_stat = (/ -1, -1, 1 /)                         ! constraint status
   X_stat = (/ -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 /) ! variable status
! problem data complete
   CALL CRO_initialize( data, control, inform ) ! Initialize control parameters
```

```
CALL CRO_crossover( n, m, m_equal, H_val, H_col, H_ptr, A_val, A_col,      &
                    A_ptr, G, C_l, C_u, X_l, X_u, C, X, Y, Z, C_stat,      &
                    X_stat , data, control, inform )  ! crossover
IF ( inform%status == 0 ) THEN                     ! successful return
  WRITE( 6, "( '      x_l          x           x_u          z     stat', /,  &
&              ( 4ES12.4, I5 ) )" )                                         &
    ( X_l( i ), X( i ), X_u( i ), Z( i ), X_stat( i ), i = 1, n )
  WRITE( 6, "( '      c_l          c           c_u          y     stat', /,  &
&              ( 4ES12.4, I5 ) )" )                                         &
    ( C_l( i ), C( i ), C_u( i ), Y( i ), C_stat( i ), i = 1, m )
  WRITE( 6, "( ' CRO_solve exit status = ', I0 ) " ) inform%status
ELSE                                              ! error returns
  WRITE( 6, "( ' CRO_solve exit status = ', I0 ) " ) inform%status
END IF
CALL CRO_terminate( data, control, inform )     ! delete internal workspace
END PROGRAM GALAHAD_CRO_EXAMPLE
```

This produces the following output:

```
      x_l          x           x_u          z     stat
  0.0000E+00  0.0000E+00  1.0000E+20  1.0000E+00   -1
  1.0000E+00  1.0000E+00  1.0000E+20  1.0000E+00   -1
  1.0000E+00  1.0000E+00  1.0000E+20  1.0000E+00   -1
  1.0000E+00  1.0000E+00  1.0000E+20  1.0000E+00   -1
  1.0000E+00  1.0000E+00  1.0000E+20  1.0000E+00   -1
  1.0000E+00  1.0000E+00  1.0000E+20  1.0000E+00   -1
  1.0000E+00  1.0000E+00  1.0000E+20  1.0000E+00   -1
  1.0000E+00  1.0000E+00  1.0000E+20  1.0000E+00   -1
  1.0000E+00  1.0000E+00  1.0000E+20  1.0000E+00   -1
  1.0000E+00  1.0000E+00  1.0000E+20  1.0000E+00   -1
  1.0000E+00  1.0000E+00  1.0000E+20  1.0000E+00   -1
      c_l          c           c_u          y     stat
  1.0000E+01  1.0000E+01  1.0000E+01  0.0000E+00   -2
  9.0000E+00  9.0000E+00  1.0000E+20  0.0000E+00   -2
 -1.0000E+20  1.0000E+01  1.0000E+01  0.0000E+00    2
 CRO_solve exit status = 0
```

Notice that active variable 1 and constraints 2 and 3 are found to be active but linearly dependent.