



Science and
Technology
Facilities Council



GALAHAD

MIQR

USER DOCUMENTATION

GALAHAD Optimization Library version 5.5

1 SUMMARY

Given a real matrix m by n matrix $\mathbf{A}$, form a multilevel incomplete QR factorization $\mathbf{QR}$ so that $\mathbf{A} \approx \mathbf{QR}$ or optionally $\mathbf{A}^T \approx \mathbf{QR}$. Only the n by n triangular factor $\mathbf{R}$ is retained, and facilities are provided to allow the solutions of the systems $\mathbf{Ru} = \mathbf{v}$ and $\mathbf{R}^T \mathbf{x} = \mathbf{z}$ for given vectors $\mathbf{v}$ and $\mathbf{z}$. The matrix $\mathbf{R}$ is particularly helpful as a preconditioner when applying an iterative method to solve the least-squares problems $\min \|\mathbf{Ax} - \mathbf{b}\|_2$ or $\min \|\mathbf{A}^T \mathbf{y} - \mathbf{c}\|_2$. Full advantage is taken of any zero coefficients in the matrices $\mathbf{A}$.

ATTRIBUTES — Versions: GALAHAD_MIQR_single, GALAHAD_MIQR_double. **Uses:** GALAHAD_CLOCK, GALAHAD_SYMBOLS, GALAHAD_SPACE, GALAHAD_SMT, GALAHAD_NORMS, GALAHAD_CONVERT, GALAHAD_SPECFILE, **Date:** June 2014. **Origin:** N. I. M. Gould, Rutherford Appleton Laboratory. **Language:** Fortran 95 + TR 15581 or Fortran 2003.

2 HOW TO USE THE PACKAGE

The package is available with single, double and (if available) quadruple 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_MIQR_single
```

with the obvious substitution `GALAHAD_MIQR_double`, `GALAHAD_MIQR_quadruple`, `GALAHAD_MIQR_single_64`, `GALAHAD_MIQR_double_64` and `GALAHAD_MIQR_quadruple_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`, `MIQR_time_type`, `MIQR_control_type`, `MIQR_inform_type` and `MIQR_data_type` (§2.3) and the subroutines `MIQR_initialize`, `MIQR_inform`, `MIQR_apply`, `MIQR_terminate`, (§2.4) and `MIQR_read_specfile` (§2.6) must be renamed on one of the `USE` statements.

2.1 Matrix storage formats

The input matrix $\mathbf{A}$ may be stored in a variety of input formats.

2.1.1 Dense storage format

The matrix $\mathbf{A}$ 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 $i = 1, \dots, m$, $j = 1, \dots, n$.

2.1.2 Sparse co-ordinate storage format

Only the nonzero entries of the matrices are stored. For the l -th entry of $\mathbf{A}$, 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.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

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 $\mathbf{A}$, 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 $l = \text{A\%ptr}(i), \dots, \text{A\%ptr}(i + 1) - 1$ of the integer array `A%col`, and real array `A%val`, respectively.

2.1.4 Sparse column-wise storage format

Again only the nonzero entries are stored, but this time they are ordered so that those in column j appear directly before those in column $j + 1$. For the j -th column of $\mathbf{A}$, the j -th component of a integer array `A%ptr` holds the position of the first entry in this column, while `A%ptr (n + 1)` holds the total number of entries plus one. The row indices i and values a_{ij} of the entries in the j -th row are stored in components $l = \text{A\%ptr}(j), \dots, \text{A\%ptr}(j + 1) - 1$ of the integer array `A%row`, and real array `A%val`, respectively.

For sparse matrices, the row and column schemes almost always requires less storage than the coordinate and dense schemes.

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, `DOUBLE PRECISION` for the double precision cases and quadruple-precision if 128-bit reals are available, and correspond to `rp_ = real32`, `rp_ = real64` and `rp_ = real128` respectively as defined 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

Five 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 matrix $\mathbf{A}$. 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.

`type` is a rank-one allocatable array of type default `CHARACTER`, that is used to indicate the storage scheme used. If the dense storage scheme (see §2.1.1), is used, the first five components of `type` must contain the string `DENSE`. For the sparse co-ordinate scheme (see §2.1.2), the first ten components of `type` must contain the string `COORDINATE`, for the sparse row-wise storage scheme (see §2.1.3), the first fourteen components of `type` must contain the string `SPARSE_BY_ROWS`, and for the sparse column-wise storage scheme (see §2.1.4), the first seventeen components of `type` must contain the string `SPARSE_BY_COLUMNS`.

For convenience, the procedure `SMT_put` may be used to allocate sufficient space and insert the required keyword into `type`. For example, if $\mathbf{A}$ is of derived type `SMT_type` and we wish to use the co-ordinate storage scheme, we may simply

```
CALL SMT_put( A%type, 'COORDINATE', istat )
```

All use is subject to the conditions of a BSD-3-Clause License.
 See <https://www.galahad.rl.ac.uk/download/> for full details.

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

- `ne` is a scalar variable of type `INTEGER(ip_)`, that holds the number of matrix entries.
- `val` is a rank-one allocatable array of type `REAL(rp_)` and dimension at least `ne`, that holds the values of the entries. 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 and §2.1.4).
- `col` is a rank-one allocatable array of type `INTEGER(ip_)`, and dimension at least `ne`, that may hold 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) or of dimension at least `n + 1`, that may hold the pointers to the first entry in each column (see §2.1.4).

2.3.2 The derived data type for holding control parameters

The derived data type `MIQR_control_type` is used to hold controlling data. Default values may be obtained by calling `MIQR_initialize` (see §2.4.1), while components may also be changed by calling `GALAHAD_MIQR_read_spec` (see §2.6.1). The components of `MIQR_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 `MIQR_form`, `MIQR_apply` and `MIQR_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 `MIQR_form`, `MIQR_apply` 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 level of the process. If `print_level ≥ 2`, this output will be increased to provide significant detail of the factorization. The default is `print_level = 0`.
- `max_level` is a scalar variable of type `INTEGER(ip_)`, that is used to specify the maximum level allowed when a multi-level factorization (MIQR) is attempted (see `multi_level` below). The default is `max_level = 4`.
- `max_order` is a scalar variable of type `INTEGER(ip_)`, that is used to specify the maximum number of columns that will be processed per level when a multi-level factorization (MIQR) is attempted (see `multi_level` below). Any non-positive value will be interpreted as `n`. The default is `max_order = -1`.
- `max_fill` is a scalar variable of type `INTEGER(ip_)`, that is used is used to control the incomplete factorization. In particular the maximum number of elements allowed in each column of $\mathbf{R}$ will not exceed `max_fill`. Any negative value will be interpreted as `n`. The default is `max_fill = 100`.
- `max_fill_q` is a scalar variable of type `INTEGER(ip_)`, that is used is used to control the incomplete factorization. In particular the maximum number of elements allowed in each column of $\mathbf{Q}$ will not exceed `max_fill_q`. Any negative value will be interpreted as `m`. The default is `max_fill_q = 100`.
- `increase_size` is a scalar variable of type `INTEGER(ip_)`, that is used increase array sizes in chunks of this when needed. The default is `increase_size = 100`.
- `buffer` is a scalar variable of type `INTEGER(ip_)`, that is used to specify the unit for any out-of-core writing when expanding arrays needed to store $\mathbf{R}$ and other intermediary data. The default is `buffer = 70`.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

`smallest_diag` is a scalar variable of type `REAL(rp_)`, that is used to identify those diagonal entries of $\mathbf{R}$ that are considered to be zero (and thus indicate rank deficiency). Any diagonal entry in the $\mathbf{R}$ factor that is smaller than `smallest_dia` will be judged to be zero, and modified accordingly. The default is `smallest_diag = 10-10`.

`tol_level` is a scalar variable of type `REAL(rp_)`, that is used as a tolerance for stopping multi-level phase. In particular, the multi-level phase ends if the dimension of the reduced problem is no smaller than `tol_level` times that of the previous reduced problem. The default is `tol_level = 0.3`.

`tol_orthogonal` is a scalar variable of type `REAL(rp_)`, that is used to judge if two vectors are roughly orthogonal; specifically vectors u and v are orthogonal if $|u^T v| \leq \text{tol_orthogonal} * \|u\| \|v\|$. The default is `tol_orthogonal = 0.0`.

`tol_orthogonal_increase` is a scalar variable of type `REAL(rp_)`, that is used to indicate the increase in the orthogonality tolerance that will be applied at each successive level. The default is `tol_orthogonal_increase = 0.01`.

`average_max_fill` is a scalar variable of type `REAL(rp_)`, that is used to control the incomplete factorization. In particular the maximum number of elements allowed in each column of $\mathbf{R}$ will not exceed `average_max_fill * ne / n`. The default is `average_max_fill = 6.0`.

`average_max_fill_q` is a scalar variable of type `REAL(rp_)`, that is used to control the incomplete factorization. In particular the maximum number of elements allowed in each column of $\mathbf{Q}$ will not exceed `average_max_fill_q * ne / m`. The default is `average_max_fill_q = 24.0`.

`tol_drop` is a scalar variable of type `REAL(rp_)`, that is used as a dropping tolerance for small generated entries. Any entry smaller than `tol_drop` will be excluded from the factorization. The default is `tol_drop = 0.01`.

`transpose` is a scalar variable of type default `LOGICAL`, that is used to indicate whether the factorization of $\mathbf{A}^T$ should be found rather than that of $\mathbf{A}$. The default is `transpose = .FALSE..`

`multi_level` is a scalar variable of type default `LOGICAL`, that is used to specify whether a multi-level incomplete factorization (MIQR) will be attempted or whether an incomplete QR factorization (IQR) suffices. The default is `multi_level = .TRUE..`

`sort` is a scalar variable of type default `LOGICAL`, that is used to specify whether the nodes of the graph of $\mathbf{A}^T \mathbf{A}$ should be sorted in order of increasing degree. This often improves the quality of the multilevel factorization. The default is `sort = .TRUE..`

`deallocate_after_factorization` is a scalar variable of type default `LOGICAL`, that is used to specify whether temporary workspace should be deallocated after every factorization. This may save space at the expense of multiple allocations if many factorizations are required. The default is `deallocate_after_factorization = .FALSE..`

`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 = ""`.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

`CONVERT_control` is a scalar variable of type `CONVERT_control_type` whose components are used to control the conversion of the input matrix type into the column-wise scheme used internally by `GALAHAD_MIQR`, as performed by the package `GALAHAD_CONVERT`. See the specification sheet for the package `GALAHAD_CONVERT` for details, and appropriate default values.

2.3.3 The derived data type for holding timing information

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

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

`form` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent computing the multi-level incomplete factorization.

`levels` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent in the multi-level phase of the factorization.

`iqr` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent in the incomplete QR phase of the factorization.

`apply` is a scalar variable of type `REAL(rp_)`, that gives the CPU time spent solving systems involving the computed factor **R**.

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

`clock_form` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent computing the multi-level incomplete factorization.

`clock_levels` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent in the multi-level phase of the factorization.

`clock_iqr` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent in the incomplete QR phase of the factorization.

`clock_apply` is a scalar variable of type `REAL(rp_)`, that gives the elapsed system clock time spent solving systems involving the computed factor **R**.

2.3.4 The derived data type for holding informational parameters

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

`status` is a scalar variable of type `INTEGER(ip_)`, that gives the exit status of the algorithm. See §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`.

`entries_in_factors` is a scalar variable of type `INTEGER(ip_)`, that gives the number of nonzeros in the incomplete factor **R**.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

`drop` is a scalar variable of type `INTEGER(ip_)`, that gives the number of entries that were dropped during the incomplete factorization.

`zero_diagonals` is a scalar variable of type `INTEGER(ip_)`, that gives the number of diagonal entries of $\mathbf{R}$ that were judged to be zero during the incomplete factorization.

`time` is a scalar variable of type `MIQR_time_type` whose components are used to hold elapsed CPU and system clock times for the various parts of the calculation (see Section 2.3.3).

`CONVERT_inform` is a scalar variable of type `CONVERT_inform_type` whose components are used to provide information concerning the conversion of the input matrix type into the column-wise scheme used internally by `GALAHAD_MIQR`, as performed by the package `GALAHAD_CONVERT`. See the specification sheet for the package `GALAHAD_CONVERT` for details, and appropriate default values.

2.3.5 The derived data type for holding problem data

The derived data type `MIQR_data_type` is used to hold all the data for the problem and the workspace arrays used to construct the multi-level incomplete factorization between calls of `MIQR` procedures. This data should be preserved, untouched, from the initial call to `MIQR_initialize` to the final call to `MIQR_terminate`.

2.4 Argument lists and calling sequences

There are four procedures for user calls (see §2.6 for further features):

1. The subroutine `MIQR_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 `MIQR_form` is called to form the multi-level incomplete factorization.
3. The subroutine `MIQR_apply` is called to apply the computed factor $\mathbf{R}$ to solve systems $\mathbf{R}\mathbf{x} = \mathbf{b}$ or $\mathbf{R}^T \mathbf{x} = \mathbf{b}$ for a given vector $\mathbf{b}$.
4. The subroutine `MIQR_terminate` is provided to allow the user to automatically deallocate array components of the private data, allocated by `MIQR_form` at the end of the solution process.

2.4.1 The initialization subroutine

Default values are provided as follows:

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

`data` is a scalar `INTENT(INOUT)` argument of type `MIQR_data_type` (see §2.3.5). It is used to hold data about the problem being solved.

`control` is a scalar `INTENT(OUT)` argument of type `MIQR_control_type` (see §2.3.2). On exit, `control` contains default values for the components as described in §2.3.2. These values should only be changed after calling `MIQR_initialize`.

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

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

2.4.2 The subroutine for forming the multi-level incomplete factorization

The multi-level incomplete QR factorization $\mathbf{A} \approx \mathbf{QR}$ or $\mathbf{A}^T \approx \mathbf{QR}$ or is formed as follows:

```
CALL MIQR_form( A, data, control, inform )
```

A is a scalar INTENT(IN) argument of type SMT_type whose components must be set to specify the data defining the matrix **A** (see §2.3.1).

data is a scalar INTENT(INOUT) argument of type MIQR_data_type (see §2.3.5). It is used to hold data about the factors obtained. It must not have been altered **by the user** since the last call to MIQR_initialize.

control is a scalar INTENT(IN) argument of type MIQR_control_type (see §2.3.2). Default values may be assigned by calling MIQR_initialize prior to the first call to MIQR_form.

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

2.4.3 The subroutine for solving systems involving the incomplete factors

Given the right-hand side **b**, one or other of the systems $\mathbf{Rx} = \mathbf{b}$ or $\mathbf{R}^T \mathbf{x} = \mathbf{b}$ may be solved as follows:

```
CALL MIQR_apply( SOL, transpose, data, inform )
```

SOL is a rank-one INTENT(INOUT) array of type default REAL that must be set on entry to hold the components of the vector **y**. On successful exit, the components of SOL will contain the solution **x**.

transpose is a scalar INTENT(IN) argument of type default LOGICAL, that should be set .TRUE. if the user wishes to solve $\mathbf{R}^T \mathbf{x} = \mathbf{b}$ and .FALSE. if the solution to $\mathbf{Rx} = \mathbf{b}$ is required.

data is a scalar INTENT(INOUT) argument of type MIQR_data_type (see §2.3.5). It is used to hold data about the factors obtained. It must not have been altered **by the user** since the last call to MIQR_initialize.

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

2.4.4 The termination subroutine

All previously allocated arrays are deallocated as follows:

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

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

control is a scalar INTENT(IN) argument of type MIQR_control_type exactly as for MIQR_form.

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

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

2.5 Warning and error messages

A negative value of `inform%status` on exit from `MIQR_form`, `MIQR_apply` or `MIQR_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 $A_n > 0$ or $a_m > 0$ or requirements that `prob%A_type` contain the string 'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS' or 'SPARSE_BY_COLUMNS' has been violated.

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 `MIQR_control_type` (see §2.3.2), by reading an appropriate data specification file using the subroutine `MIQR_read_specfile`. This facility is useful as it allows a user to change `MIQR` control parameters without editing and recompiling programs that call `MIQR`.

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

```
( .. lines ignored by MIQR_read_specfile .. )
BEGIN MIQR
  keyword      value
  .....      .....
  keyword      value
END
( .. lines ignored by MIQR_read_specfile .. )
```

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

and

```
END MIQR SPECIFICATION
```

are acceptable. Furthermore, between the "BEGIN MIQR" 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 `!`

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

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 MIQR_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 MIQR_read_specfile.

2.6.1 To read control parameters from a specification file

Control parameters may be read from a file as follows:

```
CALL MIQR_read_specfile( control, device )
```

`control` is a scalar `INTENT(INOUT)` argument of type `MIQR_control_type` (see §2.3.2). Default values should have already been set, perhaps by calling `MIQR_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 §2.3.2) of `control` that each affects are given in Table 2.1.

command	component of control	value type
error-printout-device	%error	integer
printout-device	%out	integer
print-level	%print_level	integer
max-level-allowed	%max_level	integer
max-order-allowed-per-level	%max_order	integer
out-of-core-buffer	%buffer	integer
increase-array-size-by	%increase_size	real
max-entries-per-column	%max_fill	real
max-entries-per-column-of-q	%max_fill_q	real
smallest-diagonal-factor-allowed	%smallest_diag	real
level-stop-tolerance	%tol_level	real
orthogonality-tolerance	%tol_orthogonal	real
orthogonality-tolerance-increase	%tol_orthogonal_increase	real
dropping-tolerance	%tol_drop	real
proportion-max-entries-per-column	%average_max_fill	real
proportion-max-entries-per-column-of-q	%average_max_fill_q	real
factorize-transpose	%transpose	logical
use-multi-level	%multi_level	logical
sort-vertices	%sort	logical
deallocate-workspace-after-factorization	%deallocate_after_factorization	logical
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`.

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

`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 , statistics concerning the formation of $\mathbf{R}$ as well as warning and error messages will be reported.

3 GENERAL INFORMATION

Use of common: None.

Workspace: Provided automatically by the module.

Other routines called directly: None.

Other modules used directly: `MIQR_form` calls the GALAHAD packages `GALAHAD_CLOCK`, `GALAHAD_SYMBOLS`, `GALAHAD_SPACE`, `GALAHAD_SMT`, `GALAHAD_NORMS`, `GALAHAD_CONVERT` and `GALAHAD_SPECFILE`,

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

Restrictions: $A_{nn} > 0$, $a_{mm} > 0$, $A_type \in \{'DENSE', 'COORDINATE', 'SPARSE_BY_ROWS', 'SPARSE_BY_COLUMNS'\}$.

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

4 METHOD

Given the matrix $\mathbf{A}$, a decomposition

$$\mathbf{A} \approx \mathbf{Q}_0 \begin{pmatrix} \mathbf{R}_0 & \mathbf{S}_0 \\ & \mathbf{A}_1 \end{pmatrix}$$

is found. Here $\mathbf{R}_0$ is upper triangular and $\mathbf{Q}_0$ is constructed as normalized, structurally-orthogonal columns of $\mathbf{A}$, or columns that are approximately so. The same approach is then applied recursively to $\mathbf{A}_1$ to obtain $\mathbf{R}_1$ and $\mathbf{A}_2$, etc. The recursion ends either before or at a prescribed level k , and thereafter an incomplete QR factorization of the remaining block $\mathbf{A}_k$ is computed.

The basic algorithm is a slight generalisation of that given by

Na Li and Yousef Saad (2006). MIQR: A Multilevel Incomplete QR preconditioner for large sparse least-squares problems. *SIAM. J. Matrix Anal. & Appl.*, **28**(2) 524–550,

and follows in many aspects the design of the C package

<http://www-users.cs.umn.edu/~saad/software/MIQR.tar.gz>

The principal use of $\mathbf{R}$ is as a preconditioner when solving linear least-squares problems via an iterative method. In particular, the minimizer of $\|\mathbf{Ax} - \mathbf{b}\|_2$ satisfies the normal equations $\mathbf{A}^T \mathbf{Ax} = \mathbf{A}^T \mathbf{b}$, and if $\mathbf{A} = \mathbf{QR}$ with orthogonal $\mathbf{Q}$, it follows that we may find $\mathbf{x}$ by forward and back substitution from $\mathbf{R}^T \mathbf{Rx} = \mathbf{A}^T \mathbf{b}$. Moreover $\mathbf{R}^{-T} \mathbf{A}^T \mathbf{AR}^{-1} = \mathbf{I}$, the n by n identity matrix. Since the matrix $\mathbf{R}$ computed by MIQR is incomplete, we expect instead that $\mathbf{R}^{-T} \mathbf{A}^T \mathbf{AR}^{-1} \approx \mathbf{I}$, and this may be used to advantage by iterative methods like CGNE and LSQR. See §2.5 of the specification sheets for the packages `GALAHAD_LSTR`, `GALAHAD_LSRT` and `GALAHAD_L2RT` for uses within GALAHAD.

All use is subject to the conditions of a BSD-3-Clause License.
 See <https://www.galahad.rl.ac.uk/download/> for full details.

5 EXAMPLE OF USE

Suppose

$$\mathbf{A} = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ & 5 \end{pmatrix},$$

that we wish to form a multi-level incomplete factorization of $\mathbf{A}$, and then to solve the resulting systems

$$\mathbf{R}^T \mathbf{z} = \mathbf{b} \text{ and } \mathbf{R} \mathbf{x} = \mathbf{z}, \text{ where } \mathbf{b} = \begin{pmatrix} 14 \\ 42 \\ 75 \end{pmatrix}.$$

Then storing the matrices in sparse row format, we may use the following code:

```
! THIS VERSION: GALAHAD 2.6 - 13/05/2014 AT 15:00 GMT.
PROGRAM GALAHAD_MIQR_EXAMPLE
USE GALAHAD_MIQR_double           ! double precision version
IMPLICIT NONE
INTEGER, PARAMETER :: wp = KIND( 1.0D+0 ) ! set precision
TYPE ( SMT_type ) :: A
TYPE ( MIQR_data_type ) :: data
TYPE ( MIQR_control_type ) :: control
TYPE ( MIQR_inform_type ) :: inform
REAL ( KIND = wp ), ALLOCATABLE, DIMENSION( : ) :: SOL
INTEGER :: s
! set problem data
A%m = 4 ; A%n = 3 ; A%ne = 5
! sparse row-wise storage format
CALL SMT_put( A%type, 'SPARSE_BY_ROWS', s ) ! storage for A
ALLOCATE( A%ptr( A%m + 1 ), A%col( A%ne ), A%val( A%ne ) )
A%val = (/ 1.0_wp, 2.0_wp, 3.0_wp, 4.0_wp, 5.0_wp /) ! matrix A
A%col = (/ 1, 2, 1, 2, 3 /)
A%ptr = (/ 1, 3, 4, 5, 6 /) ! set row pointers
! problem data complete
CALL MIQR_initialize( data, control, inform ) ! Initialize control parameters
CALL MIQR_form( A, data, control, inform ) ! form factors
ALLOCATE( SOL( A%n ) )
SOL = (/ 14.0, 42.0, 75.0 /) ! set b
CALL MIQR_apply( SOL, .TRUE., data, inform ) ! solve R^T z = b
WRITE( 6, "( ' z ', /, ( 5ES12.4 ) )" ) SOL
CALL MIQR_apply( SOL, .FALSE., data, inform ) ! solve R x = z
WRITE( 6, "( ' x ', /, ( 5ES12.4 ) )" ) SOL
CALL MIQR_terminate( data, control, inform ) ! delete internal workspace
END PROGRAM GALAHAD_MIQR_EXAMPLE
```

This produces the following output:

```
z
1.5000E+01  4.4272E+00  8.8544E+00
x
1.0000E+00  2.0000E+00  3.0000E+00
```

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

```
! sparse row-wise storage format
```

All use is subject to the conditions of a BSD-3-Clause License.
See <https://www.galahad.rl.ac.uk/download/> for full details.

```
...
! problem data complete
```

by

```
! sparse column-wise storage format
  CALL SMT_put( A%type, 'SPARSE_BY_COLUMNS', i ) ! storage for A
  ALLOCATE( A%val( A%ne ), A%row( A%ne ), A%ptr( A%n + 1 ) )
  A%val = (/ 1.0_wp, 3.0_wp, 2.0_wp, 4.0_wp, 5.0_wp /) ! matrix A
  A%row = (/ 1, 2, 1, 3, 4 /)
  A%ptr = (/ 1, 3, 5, 6 /) ! set column pointers
! problem data complete
```

or using a sparse co-ordinate storage format with the replacement lines

```
! sparse co-ordinate storage format
  CALL SMT_put( A%type, 'COORDINATE', i ) ! storage for A
  ALLOCATE( A%val( A%ne ), A%row( A%ne ), A%col( A%ne ) )
  A%val = (/ 1.0_wp, 2.0_wp, 3.0_wp, 4.0_wp, 5 /) ! matrix A
  A%row = (/ 1, 1, 2, 3, 4 /)
  A%col = (/ 1, 2, 1, 2, 3 /)
! problem data complete
```

or using a dense storage format with the replacement lines

```
! dense storage format
  CALL SMT_put( A%type, 'DENSE', i ) ! storage for A
  ALLOCATE( A%val( A%n * A%m ) )
  A%val = (/ 1.0_wp, 2.0_wp, 0.0_wp, 3.0_wp, 0.0_wp, 0.0_wp, 0.0_wp, 4.0_wp, &
    0.0_wp, 0.0_wp, 0.0_wp, 5.0_wp /) ! matrix A, dense by rows
! problem data complete
```

respectively.