qr_mumps

version 1.1

Users guide

Contents

1 Introduction

qr_mumps is a software package for the solution of sparse, linear systems on multicore computers. It implements a direct solution method based on the QR factorization of the input matrix. Therefore, it is suited to solving sparse **least-squares** problems $\min_x \|Ax - b\|_2$ and to computing the **minimum-norm** solution of sparse, underdetermined problems. It can obviously be used for solving square problems in which case the stability provided by the use of orthogonal transformations comes at the cost of a higher operation count with respect to solvers based on, e.g., the LU factorization. qr_mumps supports **real and complex**, **single or double** precision arithmetic.

As in all the sparse, direct solvers, the solution is achieved in three distinct phases:

Analysis : in this phase an analysis of the structural properties of the input matrix is performed in preparation for the numerical factorization phase. This includes computing a column permutation which reduces the amount of *fill-in* coefficients (i.e., nonzeroes introduced by the factorization). This step does not perform any floating-point operation and is, thus, commonly mush faster than the factorization and solve (depending on the number of right-hand sides) phases.

Factorization : at this step, the actual QR factorization is computed. This step is the most computationally intense and, therefore, the most time consuming.

Solution : once the factorization is done, the factors can be used to compute the solution of the problem through two operations:

Solve : this operation computed the solution of the triangular system $Rx = b$ or $R^T x = b$;

Apply : this operation applies the Q orthogonal matrix to a vector, i.e., $y = Qx$ or $y = Q^T x$.

These three steps have to be done in order but each of them can be performed multiple times. If, for example, the problem has to be solved against multiple right-hand sides (not all available at once), the analysis and factorization can be done only once while the solution is repeated for each right-hand side. By the same token, if the coefficients of a matrix are updated but not its structure, the analysis can be performed only once for multiple factorization and solution steps.

qr_mumps is built upon the large knowledge base and know-how developed by the members of the MUMPS[1] project. However, qr_mumps does not share any code with the MUMPS package and it is a completely independent software. qr_mumps is developed and maintained in a collaborative effort by the IRIT-CERFACS joint laboratory in Toulouse, the ROMA team at ENS-Lyon and the LaBRI laboratory in Bordeaux, France.

2 Algorithm

qr_mumps is based on the multifrontal factorization method. This method was first introduced by Duff and Reid [7] as a method for the factorization of sparse, symmetric linear systems and, since then, has been the object of numerous studies and the method of choice for several, high-performance, software packages such as MUMPS [1] and UMFPACK [4]. At the heart of this method is the concept of an *elimination tree*, extensively studied and formalized later by Liu [9]. This tree graph describes the dependencies among computational tasks in the multifrontal factorization. The multifrontal method can be adapted to the QR factorization of a sparse matrix thanks to the fact that the R factor of a matrix A and the Cholesky factor of the normal equation matrix $A^T A$ share the same structure under the hypothesis that the matrix A is *Strong Hall* (for a definition of this property see, for example, [2]). Based on this equivalence, the elimination tree for the QR factorization of A is the same as that for the Cholesky factorization of $A^T A$. In the case where the Strong Hall property does not hold, the elimination tree related to the Cholesky factorization of $A^T A$ can still be used although the resulting

[1] http://mumps.enseeiht.fr

QR factorization will perform more computations and consume more memory than what is really needed; alternatively, the matrix A can be permuted to a Block Triangular Form (BTF) where all the diagonal blocks are Strong Hall.

In a basic multifrontal method, the elimination tree has n nodes, where n is the number of columns in the input matrix A, each node representing one pivotal step of the QR factorization of A. Every node of the tree is associated with a dense matrix, known as *frontal matrix* that contains all the coefficients affected by the elimination of the corresponding pivot. The whole QR factorization consists in a bottom-up traversal of the tree where, at each node, two operations are performed:

- **assembly**: a set of rows from the original matrix is assembled together with data produced by the processing of child nodes to form the frontal matrix;
- **factorization**: one Householder reflector is computed and applied to the whole frontal matrix in order to annihilate all the subdiagonal elements in the first column. This step produces one row of the R factor of the original matrix and a complement which corresponds to the data that will be later assembled into the parent node (commonly referred to as a *contribution block*). The Q factor is defined implicitly by means of the Householder vectors computed on each front; the matrix that stores the coefficients of the computed Householder vectors, will be referred to as the H matrix from now on.

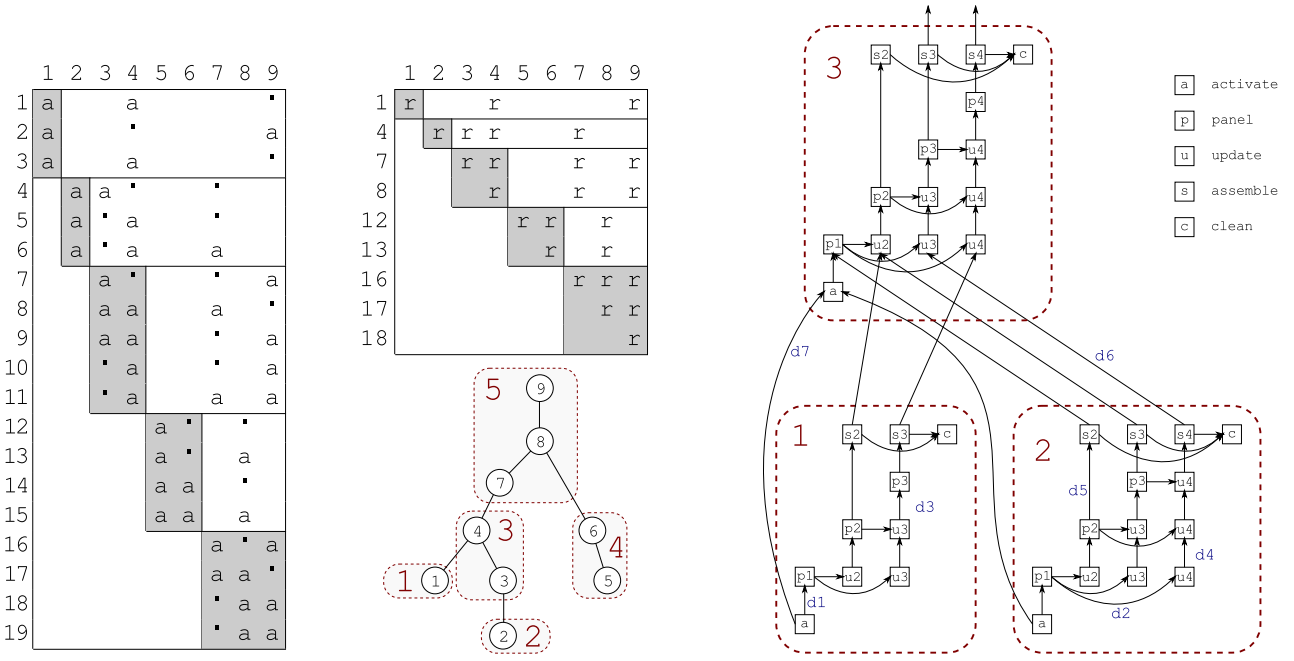

Figure 1: *(left)* Example of multifrontal QR factorization. The dots denote the fill-in coefficients. *(right)* The DAG associated with supernodes 1, 2 and 3 for a block-column size of one.

In practical implementations of the multifrontal QR factorization, and in `qr_mumps` , nodes of the elimination tree are amalgamated to form *super nodes*. The amalgamated pivots correspond to rows of R that have the same structure and can be eliminated at once within the same frontal matrix without producing any additional fill-in in the R factor. The elimination of amalgamated pivots and the consequent update of the trailing frontal submatrix can thus be performed by means of efficient Level-3 BLAS routines. Moreover, amalgamation reduces the number of assembly operations increasing the computations-to-communications ratio which results in better performance. The amalgamated elimination tree is also commonly referred to as *assembly tree*. Figure 1 *(left)* shows a sparse matrix along with the assembly tree and the resulting R factor.

Parallelism is exploited, in `qr_mumps` , thorough a fine-grained decomposition of the frontal matrices into block-columns. As illustrated in Figure 1 *(right)*, this allows representing the whole matrix factorization as DAG (Directed Acyclic Graph) where nodes represent sequential tasks, i.e. the execution of one elementary operation on a block-column, and edges the dependencies among them.

The tasks in the DAG are then scheduled dynamically according to a data-flow parallel execution model. This approach delivers high flexibility and concurrence which result in high performance on modern, multicore computers.

The method used in `qr_mumps` is described in full details in [3].

3 Features

3.1 Parallelism

`qr_mumps` is a parallel, multithreaded software based on the OpenMP standard. Therefore, it will run on multicore or, more generally, shared memory multiprocessor computers. `qr_mumps` does not run on distributed memory (e.g. clusters) parallel computers. As described in Section 2, parallelism is achieved through a decomposition of the workload into fine-grained computational tasks which basically correspond to the execution of a BLAS or LAPACK operation on a block-column. It is strongly recommended to use sequential BLAS and LAPACK libraries and let `qr_mumps` have full control of the parallelism. Equivalently, you can link to a multithreaded BLAS or LAPACK making sure the number of threads used by these libraries is equal to one when `qr_mumps` routines are called. For example, for OpenMP based BLAS libraries, such as Intel MKL, executing `call omp_set_num_threads(1)` right before a `qr_mumps` routine will do.

The number of threads used by `qr_mumps` can be controlled in two different ways:

1. by setting `QRM_NUM_THREADS` environment variable to the desired number of threads. In this case the number of threads will be the same throughout the execution of your program/application;

2. through the `qrm_set` (see Section 4.3.1 and 6). This allows a finer control over the parallelism; for example, a different number of threads can be used in two consecutive calls to the factorization phase.

The `qrm_set` routine has higher priority than the `QRM_NUM_THREADS` environment variable.

The granularity of the tasks is controlled by the `qrm_nb_` parameter (see Section 6) which is a blocking factor for partitioning internal data. Smaller values mean more parallelism; however, because this blocking factor is an upper-bound for the granularity of operations (or, more precisely for the granularity of calls to BLAS and LAPACK routines), it is recommended to choose reasonably large values in order to achieve high efficiency. Experimental results show that 120 gives a good compromise between concurrence and efficiency of elementary operations over a wide range of modern, multicore architectures (such as Intel, AMD or IBM processors).

3.2 Singletons detection

The analysis phase in a QR multifrontal solver heavily relies on the assumption that the structure of the R factor of A is the same as that of the Cholesky factor of A^TA. However, this assumption does not hold when the A matrix is not *strong Hall*, in which case the analysis may largely overestimate the fill-in. This problem can be addressed by permuting the input matrix to a Block Triangular Form (BTF): this will not only reduce the cost of the factorization phase because only the diagonal block must be factorized but, since these are strong Hall, will also make the analysis perfectly accurate (for further details please refer to Puglisi et al. [11]). `qr_mumps` does not directly provide the possibility to permute the matrix into BTF (it is obviously possible for the user to do it and then invoke `qr_mumps` separately on each diagonal block); instead it provides a feature called *singleton detection* introduced

by Tim Davis in the SPQR [5] package. This feature consists in permuting the input matrix as

$$PAQ = \begin{bmatrix} R_{11} & R_{12} \\ 0 & \tilde{A}_{22} \end{bmatrix}$$

where R_{11} is upper triangular. In this case on the $\tilde{A}_{22}$ submatrix will be factorized. It has to be noted that the singleton detection does not guarantee that the $\tilde{A}_{22}$ submatrix is strong Hall although it commonly makes the problem much more tractable.

The singleton detection feature can be turned on or off through the `qrm_set` routine (see Section 6).

3.3 Orderings

The cost of a sparse matrix factorization (both in terms of floating point operations and memory footprint) heavily depends on the amount of fill-in. In the case of a QR factorization, this can be reduced by conveniently permuting the columns of the input matrix. `qr_mumps` supports several permutation methods (commonly called pivot ordering methods) through third party software packages, namely COLAMD [6], SCOTCH [10] and Metis [8]. It is also possible for the user to provide his own column permutation of choice. Methods based on nested dissection (i.e., those in SCOTCH and Metis) commonly provide better results and are, therefore, highly recommended especially on larger matrices. The ordering method can be chosen using the `qrm_set` routine (see Section 6).

4 API

qr_mumps is developed in the Fortran 2003 language[2] but includes a C interface developed through the Fortran 2003 iso_c_binding feature. All the qr_mumps features are available from both interfaces although the Fortran one takes full advantage of the language features, such as the interface overloading, that are not available in C. The naming convention used in qr_mumps groups all the routine or data type names into two families depending on whether they depend on the arithmetic or not. Typed names always begin by _qrm_ where the first underscore _ becomes d, s, z, c for real double, real single, complex double or complex single arithmetic, respectively. Untyped names, instead, simply begin by qrm_. Note that thanks to interface overloading in Fortran all the typed interfaces of a routine can be conveniently grouped into a single untyped one; this is described in details in Section 4.4. All the interfaces described in the remainder of this section are for the real, single precision case. The interfaces for real double, complex single and complex double can be obtained by replacing sqrm with dqrm, cqrm and zqrm, respectively and real with real(kind(1.d0)), complex, complex(kind(1.d0)), respectively. All the routines that take vectors as input (e.g., _qrm_apply) can be called with either one vector (i.e. a rank-1 Fortran array x(:)) or multiple ones (i.e., a rank-2 Fortran array x(:,:)) through the same interface thanks to interface overloading. This is not possible for the C interface, in which case an extra argument is present in order to specify the number of vectors which are expected to be stored in column-major (i.e., Fortran style) format.

In this section only the Fortran API is presented. For each Fortran name (either of a routine or of a data type) the corresponding C name is obtained by adding the _c suffix. The number, type and order of arguments in the C routines is the same except for those routines that take dense vectors in which case, the C interface needs an extra argument specifying the number of vectors passed trough the same pointer. The user can refer to the code examples and to the _qrm_mumps.h file for the full details of the C interface.

4.1 Data types

4.1.1 _qrm_spmat_type

This data type is used to define a problem and all the information needed to process it. Specifically it contains the problem matrix, the parameters used to control the behavior of the qr_mumps operations done on it and the statistics collected by qr_mumps during the execution of these operations.

```
type sqrm_spmat_type
   ! Row and column indices
   integer, pointer :: irn(:), jcn(:)
   ! Numerical values
   real, pointer :: val(:)
   ! Number of rows, columns
   ! and nonzeroes
   integer :: m, n, nz
   ! A pointer to an array
   ! containing a column permutation
   ! provided by the user
   integer, pointer :: cperm_in(:)
   ! Integer control parameters
   integer :: icntl(20)
   ! Collected statistics
   integer(kind=8) :: gstats(10)
end type sqrm_spmat_type
```

[2] or, better, Fortran 95 plus a minor subset of the Fortran 2003 features.

- Matrix data: matrices can be stored in the `COO` (or coordinate) format through the `irn`, `jcn` and `val` fields containing the row indices, column indices and values, respectively and the `m`, `n` and `nz` containing the number of rows, columns and nonzeroes, respectively. `qr_mumps` uses a Fortran-style 1-based numbering and thus all row indices are expected to be between 1 and `m` and all the column indices between 1 and `n`. Duplicate entries are summed during the factorization, out-of-bound entries are ignored.
- `cperm_in`: this array can be used to provide a matrix column permutation and is only accessed by `qr_mumps` in this case.
- `icntl`: this array contains all the integer control parameters. Its content can be modified either directly or indirectly through the `qrm_set` routine (see Section 6).
- `cntl`: this array contains all the real control parameters. Its content can be modified either directly or indirectly through the `qrm_set` routine (see Section 6).
- `gstats`: this array contains all the statistics collected by `qr_mumps` . Its content can be accessed either directly or indirectly through the `qrm_get` routine (see Section 7).

4.2 Computational routines

4.2.1 `qrm_analyse`

This routine performs the analysis phase (see Section 1) on A or A^T.

```
interface qrm_analyse

   subroutine sqrm_analyse(qrm_mat, transp)
     type(sqrm_spmat_type):: qrm_mat
     character, optional  :: transp
   end subroutine sqrm_analyse

end interface qrm_analyse
```

Arguments:

- `qrm_mat`: the input problem.
- `transp`: whether the input matrix should be transposed or not. Can be either `'t'` or `'n'`. In the Fortran interface this parameter is optional and set by default to `'n'` if not passed.

4.2.2 `qrm_factorize`

This routine performs the factorization phase (see Section 1) on A or A^T. It can only be executed if the analysis is already done.

```
interface qrm_factorize

   subroutine sqrm_factorize(qrm_mat, transp)
     type(sqrm_spmat_type):: qrm_mat
     character, optional  :: transp
   end subroutine sqrm_factorize

end interface qrm_factorize
```

Arguments:

- `qrm_mat`: the input problem.
- `transp`: whether the input matrix should be transposed or not. Can be either `'t'` or `'n'`. In the Fortran interface this parameter is optional and set by default to `'n'` if not passed.

4.2.3 `qrm_apply`

This routine computes $b = Q \cdot b$ or $b = Q^T \cdot b$. It can only be executed once the factorization is done.

```
interface qrm_apply

   subroutine sqrm_apply1d(qrm_mat, transp, b)
     type(sqrm_spmat_type) :: qrm_mat
     character             :: transp
     real                  :: b(:)
   end subroutine sqrm_apply1d

   subroutine sqrm_apply2d(qrm_mat, transp, b)
     type(sqrm_spmat_type) :: qrm_mat
     character             :: transp
     real                  :: b(:,:)
   end subroutine sqrm_apply2d

end interface qrm_apply
```

Arguments:

- `qrm_mat`: the input problem.
- `transp`: whether to apply Q or Q^T. Can be either `'t'` or `'n'`.
- `b`: the b vector(s) to which Q or Q^T is applied.

4.2.4 `qrm_solve`

This routine solves the triangular system $R \cdot x = b$ or $R^T \cdot x = b$. It can only be executed once the factorization is done/

```
interface qrm_solve

   subroutine sqrm_solve1d(qrm_mat, transp, b, x)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: b(:)
     real                  :: x(:)
     character             :: transp
   end subroutine sqrm_solve1d

   subroutine sqrm_solve2d(qrm_mat, transp, b, x)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: b(:,:)
     real                  :: x(:,:)
     character             :: transp
   end subroutine sqrm_solve2d

end interface qrm_solve
```

Arguments:

- `qrm_mat`: the input problem.
- `transp`: whether to solve for R or R^T. Can be either `'t'` or `'n'`.
- `b`: the b right-hand side(s).
- `x`: the x solution vector(s).

4.2.5 `qrm_least_squares`

This subroutine can be used to solve a linear least squares problem $\min_x \|Ax - b\|_2$ in the case where the input matrix is square or overdetermined. It is a shortcut for the sequence

```
call qrm_analyse(qrm_mat, 'n')
call qrm_factorize(qrm_mat, 'n')
call qrm_apply(qrm_mat, 't', b)
call qrm_solve(qrm_mat, 'n', b, x)
```

```
interface qrm_least_squares

   subroutine sqrm_least_squares1d(qrm_mat, b, x)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: b(:)
     real                  :: x(:)
   end subroutine sqrm_least_squares1d

   subroutine sqrm_least_squares2d(qrm_mat, b, x)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: b(:,:)
     real                  :: x(:,:)
   end subroutine sqrm_least_squares2d

end interface qrm_least_squares
```

Arguments:

- `qrm_mat`: the input problem.
- `b`: the b right-hand side(s).
- `x`: the x solution vector(s).

4.2.6 `qrm_min_norm`

This subroutine can be used to solve a linear minimum norm problem in the case where the input matrix is square or underdetermined. It is a shortcut for the sequence

```
call qrm_analyse(qrm_mat, 't')
call qrm_factorize(qrm_mat, 't')
call qrm_solve(qrm_mat, 't', b, x)
call qrm_apply(qrm_mat, 'n', b)
```

```
interface qrm_min_norm
   subroutine sqrm_min_norm1d(qrm_mat, b, x)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: x(:)
     real                  :: b(:)
   end subroutine sqrm_min_norm1d

   subroutine sqrm_min_norm2d(qrm_mat, b, x)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: x(:,:)
     real                  :: b(:,:)
   end subroutine sqrm_min_norm2d

end interface qrm_min_norm
```

Arguments:

- `qrm_mat`: the input problem.
- `b`: the b right-hand side(s).
- `x`: the x solution vector(s).

4.2.7 `qrm_matmul`

This subroutine performs a matrix-vector product of the type $y = \alpha Ax + \beta y$ or $y = \alpha A^T x + \beta y$.

```
interface qrm_matmul

   subroutine sqrm_matmul1d(qrm_mat, transp, alpha, x, beta, y)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: y(:)
     real                  :: x(:)
     real                  :: alpha, beta
     character             :: transp
   end subroutine sqrm_matmul1d

   subroutine sqrm_matmul2d(qrm_mat, transp, alpha, x, beta, y)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: y(:,:)
     real                  :: x(:,:)
     real                  :: alpha, beta
     character             :: transp
   end subroutine sqrm_matmul2d

end interface qrm_matmul
```

Arguments:

- `qrm_mat`: the input problem.
- `transp`: whether to multiply by A or A^T. Can be either `'t'` or `'n'`.
- `alpha`, `beta` the α and β scalars

- `x`: the x vector(s).
- `y`: the y vector(s).

4.2.8 `qrm_matnrm`

This routine computes the one-norm $\|A\|_1$ or the infinity-norm $\|A\|_\infty$ of a matrix.

```
interface qrm_matnrm

   subroutine sqrm_matnrm(qrm_mat, ntype, nrm)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: nrm
     character             :: ntype
   end subroutine sqrm_matnrm

end interface qrm_matnrm
```

Arguments:

- `qrm_mat`: the input problem.
- `ntype`: the type of norm to be computed. It can be either `'i'` or `'1'` for the infinity and one norms, respectively.
- `nrm`: the computed norm.

4.2.9 `qrm_vecnrm`

This routine computes the one-norm $\|x\|_1$, the infinity-norm $\|x\|_\infty$ or the two-norm $\|x\|_2$ of a vector.

```
interface qrm_vecnrm

   subroutine sqrm_vecnrm1d(vec, n, ntype, nrm)
     real      :: vec(:)
     integer   :: n
     character :: ntype
     real      :: nrm
   end subroutine sqrm_vecnrm1d

   subroutine sqrm_vecnrm2d(vec, n, ntype, nrm)
     real      :: vec(:,:)
     integer   :: n
     character :: ntype
     real      :: nrm(:)
   end subroutine sqrm_vecnrm2d

end interface qrm_vecnrm
```

Arguments:

- `x`: the x vector(s).
- `n`: the size of the vector.

- `ntype`: the type of norm to be computed. It can be either `'i'`, `'1'` or `'2'` for the infinity, one and two norms, respectively.
- `nrm` the computed norm(s). If `x` is a rank-2 array (i.e., a multivector) this argument has to be a rank-1 array `nrm(:)` and each of its elements will contain the norm of the corresponding column of `x`.

4.2.10 `qrm_residual_norm`

This routine computes the scaled norm of the residual $\frac{\|b-Ax\|_\infty}{\|b\|_\infty+\|x\|_\infty\|A\|_\infty}$, i.e., the normwise backward error. It is a shortcut for the sequence

```
call qrm_vecnrm(b, qrm_mat%m, 'i', nrmb)
call qrm_vecnrm(x, qrm_mat%n, 'i', nrmx)
call qrm_matmul(qrm_mat, 'n', -1, x, 1, b)
call qrm_matnrm(qrm_mat, 'i', nrma)
call qrm_vecnrm(b, qrm_mat%m, 'i', nrmr)
nrm = nrmr/(nrmb+nrma*nrmx)
```

```
interface qrm_residual_norm

   subroutine sqrm_residual_norm1d(qrm_mat, b, x, nrm)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: b(:)
     real                  :: x(:)
     real                  :: nrm
   end subroutine sqrm_residual_norm1d

   subroutine sqrm_residual_norm2d(qrm_mat, b, x, nrm)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: b(:,:)
     real                  :: x(:,:)
     real                  :: nrm
   end subroutine sqrm_residual_norm2d

end interface qrm_residual_norm
```

Arguments:

- `qrm_mat`: the input problem.
- `b`: the b right-hand side(s). On output this argument contains the residual.
- `x`: the x solution vector(s).
- `nrm` the scaled residual norm. This argument is of type `real` for single precision arithmetic (both real and complex) and `real(kind(1.d0))` for double precision ones (both real and complex). If `x` and `b` are rank-2 arrays (i.e., multivectors) this argument has to be a rank-1 array `nrm(:)` and each coefficient will contain the scaled norm of the residual for the corresponding column of `x` and `b`.

4.2.11 `qrm_residual_orth`

Computes the quantity $\frac{\|A^Tr\|_2}{\|r\|_2}$ which can be used to evaluate the quality of the solution of a least squares problem (see [2], page 34). It is a shortcut for the sequence

```
  call qrm_matmul(qrm_mat, 't', 1, r, 0, atr)
  call qrm_vecnrm(r,   qrm_mat%m, '2', nrmr)
  call qrm_vecnrm(atr, qrm_mat%n, '2',  nrm)
  nrm = nrm/nrmr
```

```
interface qrm_residual_orth

   subroutine sqrm_residual_orth1d(qrm_mat, r, nrm)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: r(:)
     real                  :: nrm
   end subroutine sqrm_residual_orth1d

   subroutine sqrm_residual_orth2d(qrm_mat, r, nrm)
     type(sqrm_spmat_type) :: qrm_mat
     real                  :: r(:,:)
     real                  :: nrm
   end subroutine sqrm_residual_orth2d

end interface qrm_residual_orth
```

Arguments:

- `qrm_mat`: the input problem.
- `r`: the r residual(s).
- `nrm` the scaled $A^T r$ norm. This argument is of type `real` for single precision arithmetic (both real and complex) and `real(kind(1.d0))` for double precision ones (both real and complex). If `r` is a rank-2 array (i.e., a multivector) this argument has to be a rank-1 array `nrm(:)` and each coefficient will contain the scaled norm of $A^T r$ for the corresponding column of `r`.

4.3 Management routines

4.3.1 `qrm_set`

This family of routines is used to set control parameters that define the behavior of `qr_mumps` . In the Fortran API the `qrm_set` interfaces overloads all of them (see Section 4.4 for more details). These control parameters are explained in full details in Section 6.

```
interface qrm_set

   subroutine sqrm_pseti(qrm_mat, string, ival)
     type(sqrm_spmat_type) :: qrm_mat
     character(len=*)      :: string
     integer               :: ival
   end subroutine sqrm_pseti

   subroutine qrm_gseti(string, ival)
     character(len=*)      :: string
     integer               :: ival
   end subroutine qrm_gseti

end interface qrm_set
```

Arguments:

- qrm_mat: the input problem.
- string: a string describing the parameter to be set (see Section 6 for a full list).
- val: the parameter value.

4.3.2 qrm_get

This family of routines can be used to get the value of a control parameter or the get the value of information collected by qr_mumps during the execution (see Section 7 for a full list).

```
interface qrm_get

   subroutine sqrm_pgeti(qrm_mat, string, ival)
     type(sqrm_spmat_type) :: qrm_mat
     character(len=*)      :: string
     integer               :: ival
   end subroutine sqrm_pgeti

   subroutine sqrm_pgetii(qrm_mat, string, ival)
     type(sqrm_spmat_type) :: qrm_mat
     character(len=*)      :: string
     integer(kind=8)       :: ival
   end subroutine sqrm_pgetii

   subroutine qrm_ggeti(string, ival)
     character(len=*)      :: string
     integer               :: ival
   end subroutine qrm_ggeti

   subroutine qrm_ggetii(string, ival)
     character(len=*)      :: string
     integer(kind=8)       :: ival
   end subroutine qrm_ggetii

end interface qrm_get
```

Arguments:

- qrm_mat: the input problem.
- string: a string describing the parameter to be set (see Sections 6 and 7 for a full list).
- val: the returned parameter value.

4.3.3 qrm_err_check

This routine checks whether an error was detected during the execution of qr_mumps . See Section 5 for the details on error handling.

```
subroutine qrm_err_check( )
end subroutine qrm_err_check
```

4.3.4 qrm_spmat_init

This routine initializes a qrm_spmat_type data structure. this pretty much amounts to setting default values for all the control parameters related to a specific problem. No other routine can be executed on qrm_spmat if it has not been initialized.

```
interface qrm_spmat_init

   subroutine sqrm_spmat_init(qrm_spmat)
     type(sqrm_spmat_type) :: qrm_mat
   end subroutine sqrm_spmat_init

end interface qrm_spmat_init
```

Arguments:

- qrm_mat: the input problem.

4.3.5 qrm_spmat_destroy

This routine destroys an instance of the qrm_spmat_type data structure. By default, this means that it will cleanup all the additional data that has been produced and attached to it (and that is not visible to the user) during the execution of the various qr_mumps operations. In the Fortran interface an optional all parameter is present which allows to deallocate also the arrays containing the input matrix, i.e., irn, jcn and val. Note that in this case the memory counters will be decremented (see Section 7) and thus it doesn't make much sense to pass all=.true. unless these arrays have been allocated through the qrm_palloc routine.

```
interface qrm_spmat_destroy

   subroutine sqrm_spmat_destroy(qrm_mat, all)
     type(sqrm_spmat_type) :: qrm_mat
     logical, optional     :: all
   end subroutine sqrm_spmat_destroy

end interface qrm_spmat_destroy
```

Arguments:

- qrm_mat: the input problem.
- all: if set equal to .true. the original matrix arrays will be deallocated. This option is not available on the C interface.

4.3.6 qrm_palloc, qrm_aalloc, qrm_pdealloc and qrm_adealloc

These routines are used to allocate and deallocate Fortran pointers or allocatables. They're essentially wrappers around the Fortran allocate function and they're mostly used internally by qr_mumps too keep track of the amount of memory allocated. Input pointers and allocatables can be either 1D or 2D, integer, real or complex, single precision or double precision (all of these are available regardless of the arithmetic with which qr_mumps has been compiled). For the sake of brevity, only the interface of the 1D and 2D, single precision, real allocatable versions is given below. The qrm_palloc and qrm_pdealloc overload all the allocation and deallocation routines for pointers; the qrm_aalloc and qrm_adealloc do the same for the allocatables.

```
interface qrm_aalloc

   subroutine qrm_aalloc_s(a, m)
     real(kind(1.e0)), allocatable  :: a(:)
     integer                        :: m
   end subroutine qrm_aalloc_s

   subroutine qrm_aalloc_2s(a, m, n)
     real(kind(1.e0)), allocatable  :: a(:,:)
     integer                        :: m, n
   end subroutine qrm_aalloc_2s

   subroutine qrm_adealloc_s(a)
     real(kind(1.e0)), allocatable  :: a(:)
   end subroutine qrm_adealloc_s

   subroutine qrm_adealloc_2s(a)
     real(kind(1.e0)), allocatable  :: a(:,:)
   end subroutine qrm_adealloc_2s

end interface qrm_aalloc
```

Arguments:

- `a`: the input 1D or 2D pointer or allocatable array.
- `m`: the row size.
- `n`: the column size.

4.4 Interface overloading

The interface overloading feature of the Fortran language is heavily used inside `qr_mumps` . First of all, all the typed routines of the type `_qrm_xyz` are overloaded with a generic `qrm_xyz` interface. This means that, for example, a call to the `qrm_factorize(a)` routine will be interpreted as a call to `sqrm_factorize(a)` or as a call to `dqrm_factorize(a)` depending on whether `a` is of type `sqrm_spmat_type` or `dqrm_spmat_type`, respectively (i.e., single or double precision real, respectively). As said in Section 4.3 the `qrm_set` and `qrm_get` interfaces overload the routines in the corresponding families and the same holds for the allocation/deallocation routines (see Section 4.3.6. The advantages of the overloading are obvious. Take the following example:

```
  type(sqrm_spmat_type) :: qrm_mat
  real, allocatable     :: b(:), x(:)

  ! initialize the control data structure.
  call qrm_spmat_init(qrm_mat)
  ...
  ! allocate arrays for the input matrix
  call qrm_palloc(qrm_mat%irn, nz)
  call qrm_palloc(qrm_mat%jcn, nz)
  call qrm_palloc(qrm_mat%val, nz)
  call qrm_aalloc(b, m)
```

```
call qrm_aalloc(x, n)

! initialize the data
...

! solve the problem
call qrm_least_squares(qrm_mat, b, x)
...
```

In case the user wants to switch to double precision, only the declarations on the first two lines have to be modified and the rest of the code stays unchanged.

5 Error handling

The error management in `qr_mumps` is based on a stack of error messages. Every time an error is detected inside a routine, the corresponding error code and message is pushed onto the stack and the control is returned to the calling routine. This allows to identify the sequence of routine calls that lead to the error. The user can choose which action has to be performed by any called routine in the case where an error is detected; two option are possible:

1. `qrm_abort_`: the execution of the program is aborted and all the errors on the stack are dumped on the error unit (e.g., this can be screen, or a file. See Section 6 for how to set the unit).
2. `qrm_return_`: the called routine silently returns even if errors where detected. In this case it is up to the user to check whether errors have been detected and to handle them.

The action to be taken can be set through the `qrm_set` routine as explained in Section 6.

The `qrm_get` (see Section 7) or the `qrm_err_check` (see Section 4.3) can be used to check whether errors were detected by `qr_mumps` .

6 Control parameters

Control parameters define the behavior of `qr_mumps` and can be classified in two types:

- global: these parameters control the global behavior of `qr_mumps` and are not related to a specific problem, e.g., the unit for output messages.
- problem specific: these parameters control the behavior of `qr_mumps` on a specific problem, e.g., the ordering method to be used on the problem.

All the control parameters can be set through the `qrm_set` routine (see the interface in Section 4.3); problem specific control parameters can also be set by manually changing the coefficients of the `qrm_spmat_type%icntl` array.

6.1 Global parameters

The global parameters can be set through the `qrm_gseti` routine (or its generic interface `qrm_set`). A call

```
call qrm_get('qrm_param', val)
```

sets parameter `qrm_param` can be set to a value `val` where the latter can either be a preset value (a constant, predefined, value) or, simply, an integer.

Here is a list of the parameters, their meaning and the accepted values:

- qrm_ounit: val is an integer specifying the unit for output messages; if negative, output messages are suppressed. Default is 6.
- qrm_eunit: val is an integer specifying the unit for error messages; if negative, error messages are suppressed. Default is 0.
- qrm_error_action: specifies the action to be taken in case an error is detected inside qr_mumps . See Section 5 for more details. Default is qrm_abort_

6.2 Problem specific parameters

The problem specific parameters can be set through the qrm_pseti routine (or its generic interface qrm_set). A call

```
call qrm_get(qrm_mat, 'qrm_param', val)
```

sets, for the qrm_mat problem, parameter qrm_param can be set to a value val where the latter can either be a preset value (a constant, predefined, value) or, simply, an integer. Equivalently, a problem specific control parameter can be set like (nothe the underscore at the end of qrm_param_:

```
qrm_mat%icntl(qrm_param_) = val
```

Here is a list of the parameters, their meaning and the accepted values:

- qrm_ordering: this parameter specifies what permutation to apply to the columns of the input matrix in order to reduce the fill-in and, consequently, the operation count of the factorization and solve phases. This parameter is used by qr_mumps during the analysis phase and, therefore, has to be set before it starts. The following pre-defined values are accepted:
  - qrm_auto_: the choice is automatically made by qr_mumps . This is the default.
  - qrm_natural_: no permutation is applied.
  - qrm_given_: a column permutation is provided by the user through the qrm_spmat_type%cperm_in.
  - qrm_colamd_: the COLAMD software package (if installed) is used for computing the column permutation.
  - qrm_scotch_: the SCOTCH software package (if installed) is used for computing the column permutation.
  - qrm_metis_: the Metis software package (if installed) is used for computing the column permutation.
- qrm_sing: this parameter defines whether to perform the singleton detection or not (see Section 3.2 for the details). This parameter is used by qr_mumps during the analysis phase and, therefore, has to be set before it starts. Accepted values are:
  - qrm_yes_: perform the singletons detection.
  - qrm_no_: don't perform the singletons detection. This is the default.
- qrm_keeph: this parameter says whether the H matrix should be kept for later use or discarded. This parameter is used by qr_mumps during the factorization phase and, therefore, has to be set before it starts. Accepted value are:
  - qrm_yes_: the H matrix is kept. This is the default.
  - qrm_no_: the H matrix is discarded.

- qrm_nb: This parameter defines the granularity of parallel tasks. Smaller values mean higher concurrence. This parameter, however, implicitly defines an upper bound for the granularity of call to BLAS and LAPACK routines (defined by the qrm_ib parameter described below); therefore, excessively small values may result in poor performance. This parameter is used by qr_mumps during the factorization phase and, therefore, has to be set before it starts. The default value is 120 which was found to be best on a wide range of machines.

- qrm_ib: this parameter defines the granularity of BLAS/LAPACK operations. Larger values mean better efficiency but imply more fill-in and thus more flops and memory consumption (please refer to [3] for more details). The value of this parameter is upper-bounded by the qrm_nb parameter described above. This parameter is used by qr_mumps during the factorization phase and, therefore, has to be set before it starts. The default value is 120 which was found to be best on a wide range of machines.

- qrm_nthreads: this parameter defines the number of threads to be used during the factorization and solve phases. This parameter is used by qr_mumps during the factorization and/or solve phases and, therefore, has to be set before they start. Different numbers of threads may be used for the two phases. Setting the number of threads through the qrm_set routine has higher priority that the QRM_NUM_THREADS environment variable described in Section 3.1. The default value is one.

- qrm_rhsnb: in the case where multiple right-hand sides are passed to the qrm_apply or the qrm_solve routines, this parameter can be used to define a blocking of the right-hand sides. Multiple blocks will then be treated in parallel depending on the qrm_rhsnthreads parameter described below. This parameter is used by qr_mumps during the solve phase and, therefore, has to be set before it starts. By default, all the right-hand sides are treated in a single block.

- qrm_rhsnthreads: this parameter defines how many threads have to be used for solving against different blocks of right-hand sides in parallel (the blocks being defined by the qrm_rhsnb parameter above). The total number of threads used during the solve phase (i.e., one of the previously mentioned routines) is given by the product of qrm_nthreads and qrm_rhsnthreads. This parameter is used by qr_mumps during the solve phase and, therefore, has to be set before it starts.

7 Information parameters

Information parameters return information about the behavior of qr_mumps and can be classified in two types:

- global: these parameters describe the global behavior of qr_mumps and are not related to a specific problem, e.g., the peak amount of memory consumed by qr_mumps .

- problem specific: these parameters describe the behavior of qr_mumps on a specific problem, e.g., the total number of flops executed during the factorization of a matrix.

All the information parameters can be gotten through the qrm_get routine (see the interface in Section 4.3); problem specific control parameters can also be retrieved by manually reading the coefficients of the qrm_spmat_type%gstats array.

The qrm_get routine can also be used to retrieve the values of all the control parameters described in the previous section with the obvious usage.

7.1 Global parameters

```
call qrm_get('qrm_param', val)
```

- qrm_error: this parameter, of type integer may be used to check whether an error was detected during the execution of a qr_mumps routine. The value returned is of type integer and corresponds to the error code found on the bottom of the error stack described in Section 5 (i.e., the error that was detected first).

- qrm_max_mem: this parameter, of type integer (or, better, of type integer(kind=8)), returns the maximum amount of memory allocated by qr_mumps during its execution. Because there is no easy way to precisely track the memory consumption in a shared memory, parallel environment without slowing down the execution, qr_mumps only returns a loose upper bound on the memory peak during all parallel operations (qrm_factorize, qrm_solve and qrm_apply): this is computed as the sum of the peaks on all the involved threads. This behavior can be changed through the qrm_exact_mem variable described in Section 7.1.

- qrm_tot_mem: this parameter, of type integer (or, better, of type integer(kind=8)) , returns the total amount of memory allocated by qr_mumps at the moment when the qrm_get routine is called.

- qrm_exact_mem: this parameter decides whether the memory consumption within parallel sections of qr_mumps should be measured exactly or not. By default, this parameter is set to qrm_no_ which means that qr_mumps returns a loose upper bound of the memory peak consumption computed as the sum of the local peaks on each thread. Because these peaks may not be all attained at the same moment, the returned estimate may be pretty far above the actual peak. This is done to avoid frequent synchronizations within the parallel code that are necessary to regulate the access to shared variables used for storing the memory consumption. When this parameter is set to qrm_yes_, qr_mumps returns an exact value for the peak memory consumption achieved; although this may work okay on most problems, it may severely slow down the code execution.

7.2 Problem specific parameters

```
call qrm_get(qrm_mat, 'qrm_param', val)
```

- qrm_e_nnz_r: this parameter, of type integer (or, better, of type integer(kind=8)) returns an estimate, computed during the analysis phase, of the number of nonzero coefficients in the R factor. This value is only available after the qrm_analyse routine is executed.

- qrm_e_nnz_h: this parameter, of type integer (or, better, of type integer(kind=8)) returns an estimate, computed during the analysis phase, of the number of nonzero coefficients in the H matrix. This value is only available after the qrm_analyse routine is executed.

- qrm_e_facto_flops this parameter, of type integer (or, better, of type integer(kind=8)) returns an estimate, computed during the analysis phase, of the number of floating point operations performed during the factorization phase. This value is only available after the qrm_analyse routine is executed.

- qrm_nnz_r: this parameter, of type integer (or, better, of type integer(kind=8)) returns the actual number of the nonzero coefficients in the R factor after the factorization is done. This value is only available after the qrm_factorize routine is executed.

- qrm_nnz_h: this parameter, of type integer (or, better, of type integer(kind=8)) returns the actual number of the nonzero coefficients in the H matrix after the factorization is done. This value is only available after the qrm_factorize routine is executed.

- qrm_facto_flops: this parameter, of type integer (or, better, of type integer(kind=8)) returns the actual number of floating-point operation done during the factorization is done. This value is only available after the qrm_factorize routine is executed.

8 Example

The code below shows a basic example program that allocates and fills up a sparse matrix, runs the analysis, factorization and solve on it, computes the solution backward error and finally prints some information collected during the process.

```
program sqrm_test_small
  use qrm_mod
  implicit none

  type(sqrm_spmat_type)  :: qrm_mat
  integer                :: ierr, nargs, i, nrhs
  real, allocatable      :: b(:), x(:), r(:)
  real                   :: rnrm, onrm

  ! initialize the control data structure.
  call qrm_spmat_init(qrm_mat)

  ! allocate arrays for the input matrix
  call qrm_palloc(qrm_mat%irn, 13)
  call qrm_palloc(qrm_mat%jcn, 13)
  call qrm_palloc(qrm_mat%val, 13)
  ! initialize the input matrix
  qrm_mat%jcn = (/1,1,1,2,2,3,3,3,3,4,4,5,5/)
  qrm_mat%irn = (/2,3,6,1,6,2,4,5,7,2,3,2,4/)
  qrm_mat%val = (/0.7,0.6,0.4,0.1,0.1,0.3,0.6,0.7,0.2,0.5,0.2,0.1,0.6/)
  qrm_mat%m   = 7
  qrm_mat%n   = 5
  qrm_mat%nz  = 13

  write(*,'("Starting␣Analysis")')
  call qrm_analyse(qrm_mat)

  write(*,'("Starting␣Factorization")')
  call qrm_factorize(qrm_mat)

  call qrm_aalloc(b, qrm_mat%m)
  call qrm_aalloc(r, qrm_mat%m)
  call qrm_aalloc(x, qrm_mat%n)

  b = 1.e0
  ! as by is changed when applying Q', we save a copy in r for later use
  r = b
  call qrm_apply(qrm_mat, 't', b)
  call qrm_solve(qrm_mat, 'n', b, x)

  ! compute the residual
  call qrm_residual_norm(qrm_mat, r, x, rnrm)
  call qrm_residual_orth(qrm_mat, r, onrm)
  write(*,'("||r||/||A||␣␣␣␣=␣",e10.2)')rnrm
  write(*,'("||A^tr||/||r||␣=␣",e10.2)')onrm

  call qrm_adealloc(b)
  call qrm_adealloc(r)
  call qrm_adealloc(x)
  call qrm_spmat_destroy(qrm_mat, all=.true.)
  write(*,'("␣␣Nonzeroes␣in␣R␣␣␣␣␣␣␣␣␣␣:␣",i20)')qrm_mat%gstats(qrm_nnz_r_)
  write(*,'("␣␣Total␣flops␣at␣facto␣␣␣␣␣:␣",i20)')qrm_mat%gstats(qrm_facto_flops_)
  write(*,'("␣␣Memory␣peak␣␣␣␣␣␣␣␣␣␣␣␣␣:␣",f9.3,"␣MB")') &
       &real(qrm_max_mem,kind(1.d0))/1024.d0/1024.d0

  stop
end program sqrm_test_small
```

References

[1] Patrick R. Amestoy, Iain S. Duff, J. Koster, and Jean-Yves L'Excellent. `MUMPS`: a general purpose distributed memory sparse solver. In A. H. Gebremedhin, F. Manne, R. Moe, and T. Sørevik, editors, *Proceedings of PARA2000, the Fifth International Workshop on Applied Parallel Computing, Bergen, June 18-21*, pages 122–131. Springer-Verlag, 2000. Lecture Notes in Computer Science 1947.

[2] Å. Björck. *Numerical methods for Least Squares Problems*. SIAM, Philadelphia, 1996.

[3] A. Buttari. Fine-grained multithreading for the multifrontal QR factorization of sparse matrices, 2011. Submitted to SIAM SISC and APO technical report number RT-APO-11-6 [PDF].

[4] T. A. Davis. Algorithm 832: UMFPACK V4.3 — an unsymmetric-pattern multifrontal method. *ACM Transactions on Mathematical Software*, 30(2):196–199, 2004.

[5] Timothy A. Davis. Algorithm 915, suitesparseqr: Multifrontal multithreaded rank-revealing sparse qr factorization. *ACM Trans. Math. Softw.*, 38:8:1–8:22, December 2011.

[6] Timothy A. Davis, John R. Gilbert, Stefan I. Larimore, and Esmond G. Ng. A column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 30:353–376, September 2004.

[7] Iain S. Duff and John K. Reid. The multifrontal solution of indefinite sparse symmetric linear systems. *ACM Transactions on Mathematical Software*, 9:302–325, 1983.

[8] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, December 1998.

[9] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIMAX*, 11:134–172, 1990.

[10] F. Pellegrini. Scotch 5.0 User's guide. Technical Report, LaBRI, Université Bordeaux I, August 2007.

[11] Chiara Puglisi. *QR Factorization of large sparse overdetermined and square matrices using a multifrontal method in a multiprocessor environment*. Phd thesis, Institut National Polytechnique de Toulouse, 1993. Available as CERFACS report TH/PA/93/33.