Science and Technology Facilities Council

HSL_MC64

1 SUMMARY

Given a sparse matrix $A = \{a_{ij}\}_{m\times n}$, $m \geq n$, `HSL_MC64` attempts to **find row and column permutation vectors** that make the permuted matrix have n entries on its diagonal. If the matrix is structurally nonsingular, the subroutine optionally returns permutations that maximize the smallest element on the diagonal, maximize the sum of the diagonal entries, or maximize the product of the diagonal entries of the permuted matrix. For the latter option, the subroutine also **finds scaling factors** that may be used to scale the original matrix so that the nonzero diagonal entries of the permuted and scaled matrix are one in absolute value and all the off-diagonal entries are less than or equal to one in absolute value. The natural logarithms of the scaling factors u_i, $i = 1, ..., m$, for the rows and v_j, $j = 1, ..., n$, for the columns are returned so that the scaled matrix $B = \{b_{ij}\}_{n\times n}$ has entries

$$b_{ij} = a_{ij}\, exp(u_i + v_j).$$

In this Fortran 95 version, there are added facilities from the original `mc64` code for working on rectangular and symmetric matrices. We use the term *matching* to mean a set of entries in the matrix no two of which are in the same row or column. For the rectangular case, the user can permute the matching to the diagonal and identify the rows in the structurally nonsingular block. The matching will lie on the diagonal of the permuted matrix. For the symmetric case, the user must only supply the lower triangle and, if a scaling is computed, it will be a symmetric scaling with the same property as in the unsymmetric case. In this case, a symmetric permutation is returned but the matching will not normally be on the diagonal of the permuted matrix. If the matrix is structurally singular and symmetric the user may call the maximum product matching only. In this case the matching returned will not be complete, but is optimal on a submatrix. The scaling will be such that the normal scaling property holds in that case.

ATTRIBUTES — Version: 2.4.3 (1 November 2023) **Interfaces:** C, Fortran, MATLAB. **Types:** Real (single,double), Complex (single,double). **Calls:** `HSL_MC34`, `MC64`, `HSL_ZD11`. **Original date:** July 2007. Version 2.0.0 May 2009. **Origin:** I. S. Duff, Rutherford Appleton Laboratory, and J. Koster, Trondheim. Also J. D. Hogg, Rutherford Appleton Laboratory (Version 2.1.0 and later). **Language:** Fortran 2003 subset (F95 + TR155581 + C interoperability). **Remark:** `HSL_MC64` is a Fortran 95 encapsulation of `MC64` and offers additional facilities to the Fortran 77 version. The work of the first author was supported by EPSRC Grant EP/E053351/1.

2 HOW TO USE THE PACKAGE

2.1 C interface to Fortran code

This package is written in Fortran and a wrapper is provided for C programmers This wrapper may only implement a subset of the full functionality described i the Fortran user documentation.

The wrapper will automatically convert between 0-based (C) and 1-based (Fortran) array indexing, so may be used transparently from C. This conversion involves both time and memory overheads that may be avoided by supplying data that is already stored using 1-based indexing. The conversion may be disabled by setting the control parameter `control.f_arrays=1` and supplying all data using 1-based indexing. With 0-based indexing, the matrix is treated as having rows and columns $0, 1, ...n-1$. In this document, we assume 0-based indexing.

The wrapper uses the Fortran 2003 interoperability features. **Matching C and Fortran compilers must be used**, for example, gcc and gfortran, or icc and ifort. If the Fortran compiler is not used to link the user's program, additional Fortran compiler libraries may need to be linked explicitly.

2.2 Calling sequences

Access to the package requires inclusion of the header file

Single precision version
`#include "hsl_mc64s.h"`

Double precision version
`#include "hsl_mc64d.h"`

Single precision complex version
`#include "hsl_mc64c.h"`

Double precision complex version
`#include "hsl_mc64z.h"`

It is not possible to use more than one version at the same time.
The following subroutines are available to the user:

(a) `mc64_default_control` sets default values for members of the `mc64_control` data type needed by other subroutines.

(b) `mc64_matching` accepts the matrix *A* and finds the required permutations (and possibly a scaling).

2.3 Input of the matrix

The user should supply the matrix *A* in standard HSL format. If *A* is symmetric, only the **lower** triangular part should be supplied. HSL standard format is compressed sparse column format with the entries within each column ordered by increasing row index. There is no requirement that zero entries on the diagonal are explicitly included. Before using `HSL_MC64`, the HSL package `HSL_MC69` may be used to check for errors and to handle duplicates (`HSL_MC69` sums them) and out-of-range entries (`HSL_MC69` removes them).

If the user's data is held using another standard sparse matrix format (such as coordinate format or sparse compressed row format), we recommend using a conversion routine from `HSL_MC69` to put the data into standard HSL format. The input of *A* is illustrated in Section 5.

2.4 Argument lists and calling sequences

2.4.1 Package types

The complex versions require C99 support for the `double complex` and `float complex` types. The real versions do not require C99 support.
We use the following type definitions in the different versions of the package:
Single precision version

```
typedef float pkgtype
```

Double precision version

```
typedef double pkgtype
```

Single complex version

```
typedef float complex pkgtype
```

Double complex version

```
typedef double complex pkgtype
```

Elsewhere, for *single* and *single complex* versions replace `double` with `float`.

2.4.2 The default setting subroutine

Default values for members of the `mc64_control` structure may be set by a call to `mc64_default_control`.

```
void mc64_default_control(struct mc64_control *control);
```

`control` has its members set to their default values, as described in Section 2.8.

2.5 To find the column permutation (and possibly a scaling)

```
void mc64_matching(int job, int matrix_type, int m, int n, const int *ptr,
   const int *row, const pkgtype *val, const struct mc64_control *control,
   struct mc64_info *info, int *perm, double *scale);
```

`job` must be set by the user to control the action. It is not altered by the subroutine. Possible values for `job` are:

- `1` Compute a row and column permutation of the matrix so that the permuted matrix has as many entries on its diagonal as possible. The values on the diagonal are of arbitrary size.
- `2` Compute a row and column permutation of the matrix so that the smallest value on the diagonal of the permuted matrix is maximized.
- `3` Compute a row and column permutation of the matrix so that the smallest value on the diagonal of the permuted matrix is maximized. The algorithm differs from the one used for `job=2` and may have quite a different performance (see Section 4).
- `4` Compute a row and column permutation of the matrix so that the sum of the diagonal entries of the permuted matrix is maximized.
- `5` Compute a row and column permutation of the matrix so that the product of the diagonal entries of the permuted matrix is maximized. Vectors are also computed to scale the matrix so that the nonzero diagonal entries of the permuted matrix are one in absolute value and all the off-diagonal entries are less than or equal to one in absolute value. In the symmetric case the scaling will be symmeterised as described in Section 4.

`matrix_type` indicates if the matrix is symmetric or not. Values of 3 or greater are treated as symmetric, all other values are treated as unsymmetric/rectangular depending on the values of `m` and `n` (this is consistent with the matrix types defined by the `HSL_MC69` package).

`m` holds the number of rows m in the matrix A. **Restriction:** `m` $\geq$ `n`.

`n` holds the number of columns n in the matrix A. **Restriction:** `n` $\geq$ `1`.

`ptr` is a rank-1 array of size at least `n+1`. `ptr[j]` must be set by the user so that `ptr[j]` is the position in `row` of the first entry in column `j` and `ptr[n]` must be set to the number of matrix entries being input by the user.

`row` is a rank-1 array of size at least `ptr[n]`. It must hold the row indices of the entries of A with those for column 0 preceding those for column 1, and so on (within each column, the row indices may be in arbitrary order). If A is symmetric, only those entries in the lower triangle should be stored.

`val` is a rank-1 array of size at least `ptr[n]`, the leading part of which holds the values of the entries stored by columns such that `val[k]` is the value of the entry in `row[k]`.

`control` is used to control the actions of the package, see Section 2.8.

`info` is use to return information about the execution of the package, as explained in Section 2.9.

`perm` is an rank-1 array of size (`m+n`). On exit, `abs(perm[i])`, `i` $=$ `0,1,...,m` $+$ `n` $-$ `1`, will be set to permutation vectors. The entries $i =$ `0,1,...m` $-$ `1` hold a permutation vector such that row i of the original matrix is row `abs(perm[i])` in the permuted matrix. Column `j` of the original matrix is column `abs(perm[m+j])` in the permuted matrix. For rows and columns that are unmatched, the value of `perm` is negative. (Observe that if control.f_arrays==0 then it will be impossible to distinguish if row 0 is matched or not. If this is required, either refer to info.strucrank or use Fortran array indexing).

`scale` may be `NULL`. If it is non-null and `job` $=$ `5`, it must be a rank-1 array of size (`m+n`) and, on exit, `scale[i]`, `i` $=$ `0,1,...,m-1`, and `scale[m+j]`, `j` $=$ `0,2,...,n-1` will be set to the row and column scaling vectors u_i and v_j as discussed in Section 1. Note that these scale factors are applied to the original (unpermuted) matrix. If `JOB` $\neq 5$, scale is not accessed.

2.6 Warning and error messages

A negative value of `info.flag` on exit from `mc64_matching` indicates that an error has occurred. No further calls should be made until the error has been corrected. Possible values are:

- 1 Value of `job` out of range. `job` $>$ `5` or `job` $\leq$ `0`. `info.more` is set to the value of `job`. Note that further options for `job` may be available in later releases.

- 2 Value of `n` out of range. `n` $<$ `1`. `info.more` is set to the value of `n`.

- 3 Value of `ptr[n]` out of range (`ptr[n]` $<$ `1`). `info.more` is set to the value of `ptr[n]`.

- 4 Value of `m` $<$ `n`. `info.more` is set to the value of `m`.

- 5 Failure of an allocate or deallocate statement. `info.stat` is set to the `STAT` value. `info.more` is set to the value of `n`.

- 6 Index out of range. `info.more` is set to the position in array `row` where out of range index found.

- 7 Duplicate entry. `info.more` is set to the position in the array `row` where duplicate found.

- 8 A symmetric matrix was indicated but there were entries in the upper triangle in column `info.more`.

A positive value of `info.flag` on exit from `mc64_matching` is used to warn the user that the data may be faulty or that the subroutine cannot guarantee the solution obtained. Possible values are:

+ 1 Matrix is structurally singular. The cardinality of the matching is less than `n`.

+ 2 The returned scaling factors are large and may cause overflow when used to scale the matrix (`job=5`, only).

2.7 The derived data types

Two derived data types are employed from the package. For each problem, the user must employ derived types defined by the module to declare structures for controlling the operation of the routine and returning information.

2.8 The control derived data type

The derived data type `mc64_control` is used to control the action. The components, which are automatically given default by a call to `mc64_default_control`, are:

`int lp` is used as the Fortran output stream for error messages. If it is negative, these messages will be suppressed. The default value is `6` (stdout).

`int wp` is used as the Fortran output stream for warning messages. If it is negative, these messages will be suppressed. The default value is `6` (stdout).

`int sp` is used as the output stream for diagnostic messages. If it is negative, these messages will be suppressed. The default value is `-1`.

`int ldiag` is used to control the amount of informational output which is required. No informational output will occur if `ldiag` $\leq$ `0`. The levels are:

- `<1` No messages are output.
- `1` Only error messages are output.
- `2` Error and warning messages output.
- `3` As for `2` plus scalar parameters, the first 10 entries of array parameters, and the control parameters on the first entry to `mc64_matching`.
- `>3` All data will be printed on entry and exit.

The default is `ldiag = 2`.

`int checking` is used by `mc64_matching` to specify whether the input data is checked for out-of-range indices and duplicates. This will be done if `checking` is set to zero. These checks will not be performed if `checking` is set to any other value. `mc64_matching` will run faster without the checking but, if either out-of-range indices or duplicates are present, it may fail so the user must be sure of the data before changing the default. If the user has already performed checking or conversion using `HSL_MC69` this option is redundant and should be disabled. The default value is `0`.

2.9 The derived data type for holding information

The derived data type `mc64_info` is used to hold information from the execution of `mc64_matching`. The components are:

`int flag` is used as an error/warning flag. Negative values indicate a fatal error and positive values are associated with a warning. Details in Section 2.6.

`int more` provides further information in the case of an error, see Section 2.6.

`int strucrank` gives the number of entries that can be placed on the diagonal (the structural rank).

`int stat` is set, in the case of the failure of an allocate or deallocate statement, to the value of the Fortran `stat` parameter.

3 GENERAL INFORMATION

Input/output: Output is provided under the control of `control.ldiag`. If a non-negative value is set in `control.lp`, `control.wp`, or `control.sp`, then error, warning or diagnostic information will be printed to the corresponding unit number, respectively. However if a value is negative, printing is suppressed.

Restrictions: m≥n, n≥1.

4 METHOD

The algorithms used in the code and a discussion of its performance are given in detail in the paper [4], and we give only a very brief summary here. Earlier work on these codes and a study of their use in solving systems by both iterative and direct methods is given in [3].

The algorithm that is used for `job = 1` is `MC21`, a depth-first search algorithm with look-ahead technique [2].

The algorithm that is used for `job = 2` solves a bottleneck bipartite matching problem. The algorithm starts with a partial matching and extends this by repeatedly searching the bipartite graph corresponding to the matrix for a path from an unmatched column node to any unmatched row node and for which the smallest weight of any edge in this path is maximal.

The algorithm for `job = 3` is based on repeated applications of an `MC21` type algorithm to matrices A_e obtained from the original $m\times n$ matrix A by deleting those entries from A that are below a certain threshold value e. If a column permutation of cardinality n exists for A_e, the threshold value e is increased; otherwise, e is decreased. This is repeated until e is such that a maximum matching of size n exists for matrix A_e, but not for $A_{e'}$, where $e' > e$. This algorithm is described in detail in [3].

The algorithm that is used for `job = 4` and `5` solves a weighted bipartite matching problem. The algorithm starts with a partial matching and extends this by repeatedly searching the corresponding bipartite graph for a path from an unmatched column node to any unmatched row node whose length is shortest. These paths are found using an algorithm similar to that of Dijkstra [1].

If the input matrix is symmetric, then the matrix is first expanded using `MC34` before calling `mc64`. A symmetric permutation will be returned which means that any matching entries that are off-diagonal in the original matrix will not be permuted to the diagonal. The scaling, if requested, is set to the square root of the product of the row and column scaling computed by `mc64` and will maintain the property that there is at least one entry in every row and column of the scaled matrix with absolute value one and all other entries have modulus less than or equal to one.

If the matrix is symmetric, structurally singular and `job = 5`, a partial matching representing a non-singular submatrix of maximum rank will be returned. The scaling, if requested, is set as for the non-singular case on this submatrix. For unmatched rows and columns, it is set such that

$$s_i = \log\frac{1}{\max_{k\in index(\bar{A})}|a_{ik}s_k|} \qquad \text{with the convention } \frac{1}{0}=1.$$

where $\bar{A}$ is the non-singular submatrix of maximum rank.

If the matrix is rectangular, the permuted matrix will have matching entries in the n positions

$$a(permi(i), permc(i)), i = 1, ...n$$

and the scaled matrix will have at least one entry of modulus one in each column and row with no entries of modulus greater than one.

References

[1] Dijkstra, E. W. A note on two problems in connection with graphs. Numerische Mathematik **1**, 269-271, 1959.

[2] Duff, I. S. Algorithm 575. Permutations for a zero-free diagonal. ACM Trans Math Softw. **7**, 387-390, 1981.

[3] Duff, I. S. and Koster, J. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. SIAM J Matrix Analysis and Applications **20** (4), 889-901, 1999.

[4] Duff, I. S. and Koster, J. On algorithms for permuting large entries to the diagonal of a sparse matrix. SIAM J Matrix Analysis and Applications **22** (4), 973-996, 2001.

5 EXAMPLE OF USE

The following program reads sparse matrices by columns and calls `HSL_MC64` to compute a permutation and scaling. It prints the permutation, the scaling and the scaled matrix.

```
/* hsl_mc64ds.c */

#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "hsl_mc64d.h"

int main(void) {
   struct mc64_control control;
   struct mc64_info info;

   int matrix_type;
   int *ptr, *row;
   double *val;

   int *perm;
   int eye, i, j, jay, job, k, kase, m, n, ne;

   double *scale, *a;

   for(kase=1; kase<=3; kase++) {
      /* KASE = 1 ... square matrix
       * KASE = 2 ... rectangular matrix
       * KASE = 3 ... symmetric matrix */
      switch(kase) {
      case 1:
         printf("Square matrix\n");
         matrix_type = 2;
         break;
      case 2:
         printf("\n\nRectangular matrix\n");
         matrix_type = 1;
         break;
```

```
case 3:
   printf("\n\nSymmetric matrix\n");
   matrix_type = 4;
   break;
}

/* Read matrix order and number of entries */
scanf("%d %d %d", &m, &n, &ne);

/* Allocate arrays of appropriate sizes */
ptr = (int *) malloc((n+1)*sizeof(int));
row = (int *) malloc(ne*sizeof(int));
val = (double *) malloc(ne*sizeof(double));
perm = (int *) malloc((m+n)*sizeof(int));
scale = (double *) malloc((m+n)*sizeof(double));
a = (double *) malloc(m*n*sizeof(double));

/* Read matrix and right-hand side */
for(i=0; i<n+1; i++) scanf("%d", &ptr[i]);
for(i=0; i<ne; i++) scanf("%d", &row[i]);
for(i=0; i<ne; i++) scanf("%lf", &val[i]);

/* Initialise control */
mc64_default_control(&control);

/* Get matching */
job = 5;
mc64_matching(job,matrix_type,m,n,ptr,row,val,&control,&info,perm,scale);
if(info.flag<0) {
   printf("Failure of mc64_matching with info.flag=%d", info.flag);
   free(ptr); free(row); free(val); free(perm); free(scale); free(a);
   return 1;
}

/* Print permutations */
printf("Row permutation\n");
for(i=0; i<m; i++) printf("%8d", perm[i]);
printf("\nColumn permutation\n");
for(i=m;i<m+n;i++) printf("%8d", perm[i]);
if(matrix_type>=3) {
   /* Print symmetric scaling */
   printf("\nSymmetric scaling\n");
   for(i=0;i<n;i++) printf(" %lf", exp(scale[i]));
} else {
   /* Print row scaling */
   printf("\nRow scaling\n");
   for(i=0;i<m;i++) printf(" %lf", exp(scale[i]));
   /* Print column scaling */
   printf("\nColumn scaling\n");
```

```
         for(i=m;i<m+n;i++) printf(" %lf", exp(scale[i]));
      }
      printf("\n");

      /* Scale matrix and put scaled and permuted entries in full array */
      for(i=0; i<m*n; i++) a[i] = 0.0;
      for(j=0; j<n; j++) {
         /* EYE and JAY are indices in permuted matrix */
         jay = abs(perm[m+j]);
         for(k=ptr[j]; k<ptr[j+1]; k++) {
            i = row[k];
            eye = abs(perm[i]);
            /* Scale and permute the matrix */
            a[jay*m+eye] = exp(scale[i])*val[k]*exp(scale[m+j]);
            /* Set symmetric counterpart if appropriate */
            if (matrix_type>=3) a[eye*m+jay] = a[jay*m+eye];
         }
      }

      /* Write out scaled matrix */
      printf("Scaled matrix\n");
      for(i=0; i<m; i++) {
         for(j=0; j<n; j++) {
            printf(" %lf", a[j*m+i]);
         }
         printf("\n");
      }

      free(ptr); free(row); free(val); free(perm); free(scale); free(a);
   }

   return 0;
}
```

with the following data

```
3 3 5
0 2 4 5
1 2 1 2 0
8.0 3.0 2.0 1.0 4.0
3 2 6
0 3 6
0 1 2 0 1 2
5.0 2.0 1.0 1.0 3.0 4.0
3 3 4
0 1 3 4
1 1 2 2
2.0 1.0 1.0 4.0
```

This produces the following output:

```
Square matrix
Row permutation
       0       1       2
Column permutation
       1       2       0
Row scaling
 1.000000 1.000000 2.000000
Column scaling
 0.125000 0.500000 0.250000
Scaled matrix
 1.000000 0.000000 0.000000
 0.000000 1.000000 1.000000
 0.000000 0.750000 1.000000


Rectangular matrix
Row permutation
       0       1      -2
Column permutation
       0       1
Row scaling
 1.000000 1.333333 1.000000
Column scaling
 0.200000 0.250000
Scaled matrix
 1.000000 0.250000
 0.533333 1.000000
 0.200000 1.000000


Symmetric matrix
Row permutation
       1       0       2
Column permutation
       1       0       2
Symmetric scaling
 0.707107 0.707107 0.500000
Scaled matrix
 0.500000 1.000000 0.353553
 1.000000 0.000000 0.000000
 0.353553 0.000000 1.000000
```