Ordering symmetric sparse matrices for small profile and wavefront[1]

J. K. Reid and J. A. Scott[2]

Abstract

The ordering of large sparse symmetric matrices for small profile and wavefront or for small bandwidth is important for the efficiency of frontal and variable-band solvers. In this report, we look at the computation of pseudoperipheral nodes and compare the effectiveness of using an algorithm based on level-set structures with using the spectral method as the basis of the Reverse Cuthill-McKee algorithm for bandwidth reduction. We also consider a number of ways of improving the performance and efficiency of Sloan's algorithm for profile and wavefront reduction, including the use of different weights, the use of supervariables, and implementing the priority queue as a binary heap. We also examine the use of the spectral ordering in combination with Sloan's algorithm. The design of software to implement the reverse Cuthill-McKee algorithm and a modified Sloan's algorithm is discussed. Extensive numerical experiments that justify our choice of algorithm are reported on.

Keywords: sparse matrices, symmetric pattern, reordering algorithms, profile reduction, Sloan algorithm, Reverse Cuthill-McKee algorithm, spectral method.

AMS(MOS) subject classifications: 65F50, 68R10.

ACM classification system: G.1.3.

Computing and Information Systems Department,
Rutherford Appleton Laboratory,
Chilton, Didcot,
Oxon OX11 0QX.

February 1998.

---

[1] Available by anonymous ftp from `matisa.cc.rl.ac.uk` in directory `pub/reports` in the file `rsRAL98016.ps.gz`

[2] Email addresses: `jkr@rl.ac.uk` and `J.A.Scott@rl.ac.uk`

CONTENTS

1 Introduction

We consider the ordering of symmetric sparse matrices for small profile and wavefront or for small bandwidth. We are primarily concerned with matrices that are positive definite, so we work only with the pattern of the matrix and do not take into account any permutations needed for numerical stability. The work is useful also for a matrix that is non-definite or is symmetric only in the pattern of its entries, but in these cases it must be appreciated that the actual factorization may be more expensive and require more storage. For finite-element applications, we assume that the matrix has been assembled. We treat unassembled finite-element matrices differently and this will be the subject of a separate report.

In recent years, much attention has been paid to the problem of ordering symmetric sparse systems (see, for example, Paulino, Menezes, Gattass, and Mukherjee 1994 for a discussion and list of references). One method which has been widely used for profile reduction is that of Sloan (1986, 1989). Sloan exploits the close relationship between a symmetric matrix $\mathbf{A} = \{a_{ij}\}$ of order n and its undirected graph with n nodes. Two nodes i and j are neighbours (or are adjacent) in the graph if and only if a_{ij} is nonzero. Sloan's algorithm has two distinct phases. In the first, a start node and an end node are chosen. In the second phase, the chosen start node is numbered first and a list of nodes that are eligible to be numbered next is formed. At each stage of the numbering, the list of eligible nodes comprises the neighbours of nodes which have already been numbered and their neighbours. The next node to be numbered is selected from the list of eligible nodes by means of a priority function. A node has a high priority if it causes either no increase or only a small increase to the current front size and is at a large distance from the end node.

The Harwell Subroutine Library code `MC40` (Duff, Reid, and Scott 1989) implements the ordering algorithm of Sloan (1986) and has been in satisfactory use for a decade. We decided that a revision was needed mainly because Kumfert and Pothen (1997) have found that, for the larger problems that are handled nowadays, there is a considerable efficiency gain from the use of a binary heap to manage the list of eligible nodes in the second phase of Sloan's algorithm. Another reason is for the economy of working with supervariables (sets of variables for which the corresponding matrix columns have identical patterns) when the number of supervariables is significantly less than the number of variables. We have added an option that permits users to provide a global priority vector because Kumfert and Pothen have found that the final ordering can be significantly better if we use a hybrid algorithm that combines a spectral ordering (see, for example, Barnard, Pothen, and Simon 1995) with the Sloan algorithm. We have also taken the opportunity to revise the code in a number of other ways, including adding an option for performing the Reverse Cuthill-McKee algorithm and allowing the user to specify the weights in Sloan's algorithm. The new code is called `MC60`,

and we also provide a simple driver called MC61.

This report is organized as follows. In Section 2, we briefly review frontal and variable-band methods. In Section 3, we look at computing start and end nodes for Sloan's algorithm. We examine modifications to improve the performance of the algorithm of Gibbs, Poole, and Stockmeyer (1976) for finding a pseudodiameter. We also look at using the spectral method to find a pseudodiameter and consider the effect of interchanging the ends of the pseudodiameter on the quality of the Reverse Cuthill-McKee algorithm. The numbering phase of Sloan's algorithm is considered in Section 4. We discuss the priority function and compare the performance of a simple sequential search for finding the node of highest priority with that of a binary heap implementation. We look at the effect of adjusting the weights in the priority function and of using the spectral pseudodiameter for the start and end nodes. In Section 5, we describe a modified version of the hybrid method of Kumfert and Pothen (1997). The design of our codes MC60 and MC61 is discussed in Section 6. Finally, a concluding discussion is given in Section 7.

To illustrate our ideas and findings, throughout this report we use the test examples of Everstine (1979) and of Kumfert and Pothen (1997). It should be noted that, although the Everstine problems have been widely used for testing algorithms of this kind, they are small by current standards. Their order varies from 59 to 2680 and Sloan reports root-mean-square wavefronts, following his ordering, varying from 3 to 40. The orders for the Kumfert and Pothen set vary from 6019 to 100196 and the root-mean-square wavefronts, following Sloan's ordering, vary from 59 to 1399.

All the results presented in this report are for Fortran 77 code compiled with the EPC (Edinburgh Portable Compilers, Ltd) Fortran 90 compiler with optimization −O running on a 143 MHz Sun Ultra 1. All timings are in CPU seconds.

2 Background

Two methods for solving large sparse symmetric systems of equations **Ax** = **b** that are widely used, especially in finite-element analysis, are the variable-band (profile) and frontal methods. The efficiency of these methods is affected substantially by the ordering of the variables. In this section, we briefly discuss the parameters which measure the efficiency of variable-band and frontal methods.

2.1 Frontal and variable-band methods

Sloan's method aims to find an elimination order that is suitable for the frontal method applied to a symmetric and positive-definite matrix. A variable is in the front if it has not yet been eliminated but is adjacent to a variable that has been eliminated or is about to be eliminated. There is an underlying assumption that full matrix storage is used for the frontal matrix (the submatrix corresponding to the rows and columns of the front) and that no advantage is taken of zeros within it. The order of the frontal matrix is known as the **wavefront**. Of interest is

- the maximum wavefront, since this affects the in-core storage needed,
- the sum of the wavefronts, known as the **profile**, since this is the total storage needed for either of the factors, and
- the root-mean-square wavefront, since the work performed when eliminating a variable is proportional to the square of the current wavefront.

The elimination order is also relevant for the variable-band method. Here, instead of the maximum wavefront, of relevance is

- the maximum semibandwidth, which affects the number of matrix rows that need to be held in-core at once (unless we are willing to reread parts of the factors from disk).

If no advantage is taken of zeros within the band, the profile is of relevance because again it is the total storage needed for either of the factors. The root-mean-square wavefront is also important since it bears the same relationship to the work performed as for the frontal method.

2.2 The row-by-row frontal method for unsymmetric matrices

In the frontal method, the matrix **A** need not be assembled explicitly. Instead, the assembly and elimination operations are interleaved with each variable being eliminated as soon as its row and column are fully summed. If the matrix is already assembled, a frontal code may accept the matrix row-by-row and treat each row as an element matrix (see, for example, Duff, Erisman, and Reid 1986, Section 10.6). We will call this the **row-by-row** frontal method. The Harwell Subroutine Library frontal code `MA42` (Duff and Scott 1996) includes such an option. In the row-by-row frontal method, a variable is eliminated when its index appears in an index list for the entries of a row for the last time. We use the terms **row front size** and **column front size** for the numbers of rows and columns in the rectangular frontal matrix involved. For efficiency, the rows need to be numbered for small row and column front sizes. Of interest here are

- the maximum row and column front sizes, and
- the root-mean-square row and column front sizes

for the permuted matrix.

If the matrix **A** is unsymmetric but has a symmetric sparsity pattern, Sloan's method may be used to give an efficient elimination order. We can then obtain a row ordering for the row-by-row frontal method by first ordering all the rows that have an entry in the column of the first variable in the elimination order, then any remaining rows that have an entry in the column of the second variable, and so on.

Given this row order, a row-by-row frontal code will use a very similar elimination order to that specified, but it is unlikely to be identical since several columns may become fully summed at the same time. Usually the sequence of numbers of rows in the front will be identical to the sequence of wavefront sizes, but even this need not be so. It may happen that a column becomes fully summed before its variable is reached in the specified elimination order; in this case, it can be eliminated at once and the numbers of rows in the front will be less than the corresponding wavefront sizes until the variable's position in the specified elimination order is reached. For example, consider the matrix with entries

$$\begin{array}{cccc} \times & \times & \times & \\ \times & \times & & \times \\ \times & & \times & \\ & \times & & \times \end{array}$$

and the natural elimination order 1, 2, 3, 4. Rows 1 to 3 are loaded into the front, then variable 1 is eliminated. At this point, variable 3 is fully summed and can be eliminated although it has not yet been reached in the elimination order.

3 Finding start and end nodes for Sloan's method

In this section, we consider finding pairs of nodes that are at maximum or nearly maximum distance apart, since experience has shown that such nodes are good candidates for starting nodes for profile and wavefront reduction algorithms and for bandwidth reduction algorithms (see, for example, Gibbs 1976, Gibbs, Poole, and Stockmeyer 1976, Sloan and Randolph 1983, Sloan 1986). In our discussion we assume that the matrix **A** is irreducible so that its associated graph G is connected (if not, we work with each component of the graph separately).

We first introduce notation that we will use throughout this report and recall some standard terminology and concepts from elementary graph theory (see Gibbs *et al.* 1976). The **degree** of a node $P \in G$ is the number of nodes that are adjacent to P. The **distance** $d(P, Q)$ between two nodes P and Q in G is defined to be the length of the shortest path connecting them (one less that the number of nodes on the path). The **diameter length** of G is the greatest distance between any two nodes in G. A **diameter** is a shortest path between two nodes P and Q whose distance apart is equal to the diameter length. A **pseudodiameter** is either a diameter or a shortest path between two nodes in G whose distance apart is slightly less than the diameter length. An end of a pseudodiameter is called a **pseudoperipheral node**. It is convenient here to refer to pseudodiameters, but actually we will only ever be interested in the pairs of pseudoperipheral nodes that define them.

3.1 Finding a pseudodiameter using level sets

Gibbs, Poole, and Stockmeyer (1976) find a pseudodiameter by constructing level-set structures. The algorithm we propose is a modified version of their procedure. The **level-set structure** rooted at a node P is defined as the partitioning of the nodes in G into level sets $L_1(P), L_2(P), ..., L_h(P)$ such that

(i) $L_1(P) = \{P\}$ and

(ii) for $i > 1$, $L_i(P)$ is the set of all nodes that are adjacent to nodes in $L_{i-1}(P)$ but are not in $L_1(P), L_2(P), ..., L_{i-1}(P)$.

Note that all the nodes in $L_{i+1}(P)$ are at the distance i from P. The level-set structure rooted at P is denoted by $\mathbf{L}(P)$. We refer to the number h of level sets in a level-set structure as its **depth** and the greatest number of nodes in a level set as its **width**. Gibbs, Poole, and Stockmeyer choose a starting node P of minimum degree and generate $\mathbf{L}(P)$. They then generate the level structures rooted at each of the nodes in the final level set $L_h(P)$. If the level-set structure $\mathbf{L}(Q)$ rooted at such a node Q has a greater depth than $\mathbf{L}(P)$, the whole process is recommenced with Q replacing P.

Constructing level-set structures for all the nodes in the final level set is obviously expensive. George and Liu (1979) therefore recommended terminating the construction of any level-set structure whose width reaches or exceeds that of the narrowest level-set structure so far found. The significance of the width is that it is closely related to the wavefront of the matrix if the level-set structure is used for ordering. Lewis (1982) recommended that the nodes should also be sorted by degree, since pseudoperipheral nodes usually have low degree. Sloan (1986) incorporated both these modifications into his algorithm for finding pseudoperipheral nodes. He also used the empirical observation that nodes with high degrees are not often selected as potential start or end nodes to introduce a shrinking strategy that reduces the number of nodes in the final level set L_h for which level-set structures are generated. Sloan chose to shrink L_h by taking the first `int`(m/2)+1 nodes (sorted in ascending sequence of degree), where `int` is the Fortran `int` function (truncation towards zero) and m is the number of nodes in L_h.

In earlier work (Duff, Reid, and Scott 1989), we tried the strategy of rejecting any node in L_h that had a neighbour that had already been tested, but rejected this on the grounds of its being more expensive than Sloan's shrinking strategy. Instead, we decided to limit the search to one representative of each degree, which we found to be significantly more economical while having little effect on the quality of the final ordering. This strategy was used in the code `MC40`. When publishing a Fortran implementation of his algorithm, Sloan (1989, p. 2655) followed us, saying 'this often minimizes the number of level structures that need to be generated without affecting the quality of the pseudoperipheral nodes'.

Since two nodes may have the same degree while being well separated with quite different level-set structures, the strategy that we now recommend is to consider up to five nodes in the final level set in order of increasing degree, omitting any that is a neighbour of a node already considered. We follow George and Liu in terminating the construction of any level-set structure whose width reaches or exceeds that of the narrowest level-set structure so far found. We will refer to our procedure as the **MGPS** (modified Gibbs Poole Stockmeyer) algorithm. On the Kumfert and Pothen test matrices, we found no case where the depth was increased by using this strategy in place of that proposed of Duff *et al.* (1989), but for a few (notably *nasasarb* and *onera_dual)* the width was significantly reduced. The computation times were generally very similar. We present in Table 1 the cases that showed different widths. Column 3 shows the number of nodes of the final level set that were considered.

Problem	Code	No. tries	Time	Level-set Depth	Level-set Width	RCM semi-bandwidth
barth5	MC40	1	0.06	103	380	394
	MGPS	5	0.13	103	359	373
copter2	MC40	2	0.45	54	2226	2322
	MGPS	2	0.45	54	2204	2280
nasasarb	MC40	3	0.92	176	864	882
	MGPS	5	1.16	176	540	577
onera_dual	MC40	1	0.58	84	3270	3479
	MGPS	2	0.58	84	2712	2768
shuttle_eddy	MC40	4	0.07	176	236	239
	MGPS	5	0.07	176	225	227
tandem_vtx	MC40	7	0.28	30	1471	1602
	MGPS	5	0.25	30	1472	1565

Table 1. The cases where the new pseudodiameter algorithm gave different widths.

Cuthill and McKee (1969) proposed that the ordering associated with the level-set structure be used as a basis for an ordering for the variable-band method and George (1971) found that there are advantages in reversing the resulting order. For an explanation of why this is so, see Duff, Erisman, and Reid (1986, page 155). We provide this ordering as an option in our new codes `MC60` and `MC61` (see Section 6). In Table 1, we also present the resulting semibandwidths when the level-set structures computed by our new strategy and by the `MC40` strategy are used for the Reverse Cuthill-McKee ordering. We refer to this as the **RCM** ordering.

We remark that our limit of five nodes in the final level set is somewhat arbitrary, but without such a limit, we found that we could do significantly more work without improving the quality of the final result.

3.2 Interchanging the ends of the pseudodiameter

In almost all of our test problems, we found that the widths seen from the two ends of the pseudodiameter were different, sometimes by a significant amount. The columns labelled Widths MGPS in Table 2 show the widths from the opposite ends. If the width is important, there seems to be no alternative to computing it from both ends and choosing as the start node the one whose level-set structure has the lesser width. We do this in our codes `MC60` and `MC61`. This usually involves no overhead, but can require one more level-set structure to be constructed to find the distances to the end node (we need to do this anyway whenever the most recently computed level-set structure is not of least width, see Section 6.3).

Problem	Depths			Widths				RCM semibandwidths			
	MGPS	Spectral		MGPS		Spectral		MGPS		Spectral	
barth	**70**	64	69	192	**180**	190	192	200	**194**	201	199
barth4	**71**	65	66	212	196	**184**	202	220	198	**188**	208
barth5	**103**	102	102	**359**	392	399	377	**373**	403	412	389
bcsstk30	**34**	32	29	2504	2639	2639	**2504**	2827	**2814**	**2814**	2827
commanche_dual	**157**	140	156	152	127	**123**	127	158	134	**129**	134
copter1	**42**	**42**	**42**	917	**915**	**915**	917	**935**	963	963	**935**
copter2	**54**	53	53	2204	2609	2869	**2197**	2280	2754	2950	**2270**
finance256	**56**	53	52	**2010**	**2010**	**2010**	**2010**	2016	2016	**2014**	**2014**
finance512	**88**	87	87	**1208**	1211	1211	1211	**1307**	**1307**	1319	1319
ford1	**143**	130	134	**252**	303	319	295	**257**	312	332	307
ford2	**244**	234	230	956	961	1009	**939**	962	986	1021	**955**
nasasrb	**176**	161	173	**540**	864	1068	900	**577**	881	1080	944
onera_dual	**84**	80	72	**2712**	5074	3780	3454	**2768**	5164	3809	3535
pds10	**16**	**16**	**16**	3419	3290	3419	**2917**	4117	3803	4117	**3392**
shuttle_eddy	**176**	170	**176**	225	**167**	198	**167**	227	**177**	201	**177**
skirt	**57**	**57**	56	1879	**1776**	1913	1972	2071	**1994**	2071	2174
tandem_dual	103	**104**	98	**2139**	2331	2285	2167	2206	2387	2325	**2173**
tandem_vtx	30	**31**	**31**	**1472**	1774	1584	1508	**1565**	1848	1681	1571

Table 2. Depths, widths, and RCM semibandwidths, using the MGPS and spectral orderings from both ends of the pseudodiameter.

3.3 The spectral pseudodiameter

In this section, we briefly discuss a recent method that has been proposed for finding a pseudodiameter, without constructing level-set structures. Barnard, Pothen, and Simon (1995) described a spectral algorithm that associates a Laplacian matrix **L** with the given matrix **A** with a symmetric sparsity pattern,

$$\mathbf{L} = \{l_{ij}\} = \begin{cases} -1 & i \neq j,\ a_{ij} \neq 0 \\ 0 & i \neq j,\ a_{ij} = 0 \\ \sum_{i \neq k} |l_{ij}| & i = j. \end{cases}$$

An eigenvector corresponding to the smallest positive eigenvalue of the Laplacian matrix is termed a **Fiedler vector**. The spectral permutation of the variables is computed by sorting the components of a Fiedler vector into monotonically nonincreasing or nondecreasing order and Barnard *et al.* found that this gave good profile sizes.

Paulino, Menezes, Gattass, and Mukherjee (1994) suggested using the first and last nodes of the spectral permutation to define a pseudodiameter. They take the better of the RCM orderings based on these two nodes. In Table 2, we show the depths, widths, and RCM semibandwidths that result from using the two ends of the MGPS and spectral pseudodiameters. In our experiments, the Fiedler vector was obtained using Chaco 2.0

(Hendrickson and Leland 1995). We used the SymmLQ/RQI option and the input parameters were chosen to be the same as those used by Kumfert and Pothen (1997).

We remark that for the spectral pseudodiameter, if the level-set structure rooted at one end is constructed, the other end does not necessarily lie in the final level set. Therefore the depths as well as the widths of the level-set structures rooted at each end of the pseudodiameter can differ. We see from our results that only for *tandem_vtx* and one end of *tandem_dual* does the spectral pseudodiameter yield a level-set structure with a greater depth than the MGPS pseudodiameter, but in about half the cases, it produces a narrower width and smaller RCM semibandwidth. We highlight in bold the greatest depths, the narrowest widths, and the smallest RCM semibandwidths. For both methods, the importance of using the end of the pseudodiameter with the narrower level-set structure is apparent. Paulino *et al.* report the results of taking the better of the two ends for the spectral method, but do not try reversing the ends for the Gibbs Poole Stockmeyer algorithm. It is our belief that it is because of this that in their paper the spectral method yielded slightly improved results and not because the spectral pseudodiameter is inherently superior for bandwidth reduction algorithms. In our code, we always use the end of the pseudodiameter that yields the level-set structure with the lesser width.

3.4 Using the supervariable graph

For efficiency, our new codes offer the option of working with supervariables. A **supervariable** is defined to be the set of variables that correspond to a set of columns of **A** with identical patterns. A permutation is constructed that places the variables of each supervariable together. The pattern of **A** is replaced by that of the permuted matrix represented as supervariables (that is, by its condensed or compressed equivalent). We call this the **supervariable graph** and is used in place of the graph associated with the original matrix. The potential savings in the computation times by using the supervariable graph are illustrated by Duff *et al.* (see also Table 8 in Section 7).

When a node is introduced into a level set in the original graph, all the nodes of its supervariable will be introduced too, unless one of them is the start node. The other nodes of the start supervariable will be in level 2 if the corresponding matrix rows have diagonal entries and in level 3 otherwise. Thus there is no significant difference between the properties of the variable and supervariable graphs. The depths will be the same, except for the trivial cases where the depth is 2 or 3.

In processing the supervariable graph, we take the numbers of variables in the supervariables into account when calculating the width of a level-set structure, but not for the

degrees of the supervariables in the list of potential start nodes. Our reasoning for these choices is that

- the width has a direct bearing on the wavefront or semibandwidth when the ordering is used without alteration and calculating it is a small overhead in the loop that adds supervariables to the level set; and
- the supervariable degree is likely to have greater topological relevance and, if there are substantially fewer supervariables than variables, is significantly cheaper to calculate.

4 Sloan's algorithm

In this section, we discuss the second phase of Sloan's algorithm, that is, the numbering phase. We will again assume that the matrix **A** is irreducible. It is straightforward to apply the algorithm to each component of a reducible (block diagonal) matrix and Sloan's code (and ours) allows for this.

4.1 The Sloan priority function

In the first phase of his algorithm, Sloan finds a pseudodiameter and, in the second phase, uses this to guide his ordering. One end *s* of the pseudodiameter is used as the start node and the other *e* is used as a target end node. In fact, Sloan ensures that the position of a variable in his ordering is not very far away from one for which the distance from the target end node is monotonic decreasing. He is able to improve the profile and wavefront by localized reordering. He begins at the start node *s* and uses the priority function

$$P_i = -W_1 c_i + W_2 d(i,e) \qquad (1)$$

for node *i*, where W_1 and W_2 are integer weights, c_i (which he calls the **current degree**) is the amount that the wavefront will increase if node *i* is numbered next, and $d(i,e)$ is the distance to the target end node. At each stage, the next node in the ordering is chosen from a list of eligible nodes to maximize P_i. Thus, a balance is kept between the aim of keeping the number of nodes in the front small and including nodes that have been left behind (further away from the target end node than other candidates). Based on his numerical experiments, Sloan recommends the pair (2,1) for the weights. Following further experiments, we also used these values in our Harwell Subroutine Library code `MC40`.

If the current degree c_i of a node drops to zero, it should be chosen as the next node to be

renumbered since eliminating the corresponding variable next is bound to reduce the size of the front. Since testing for zero c_i is not a big overhead, we do this in our code. This can only improve the quality of the result, but such a node is likely to be chosen anyway if the ratio W_1/W_2 is large because c_i cannot be negative.

4.2 Sloan's search

For his list of eligible nodes, Sloan takes all nodes that are in the front (neighbours of one or more renumbered nodes) or are neighbours of one or more nodes in the front. He performs a simple sequential search of the list to find the node with highest priority that is to be numbered next. Sloan noted that the simple search was faster than using a binary heap search for most of Everstine's (1979) test problems, but suggested that the binary heap search will inevitably become the method of choice for large problems where the root-mean-square wavefront exceeds several hundred nodes The recent work of Kumfert and Pothen (1997) confirms this expectation. To make our code efficient both on the small problems used by Sloan and the much larger problems which are common today, we commence with code that performs Sloan's simple search, but switch to code that uses a binary heap if the number of eligible nodes exceeds a threshold. Our experience is that the performance is not very sensitive to this threshold. Based on our numerical experiments, we use a threshold of 100 in our code. We show timings for six test problems in Table 3, chosen to illustrate the performance in cases that each vary from the next by a factor of about 10 in the time taken when a simple search is used (threshold *n*). For small problems, our method with threshold 100 can be slightly less efficient than the simple search (probably because once we have switched to the heap, we do not return to the simple search), but there are substantial gains on the largest cases.

	DWT 59	DWT 221	DWT 869	*shuttle_eddy*	*tandem_vtx*	*onera_dual*
Order (*n*)	59	221	869	10,429	18,454	85,567
Threshold						
0	0.00042	0.0028	0.015	0.16	0.53	1.7
10	0.00041	0.0028	0.015	0.16	0.53	1.7
20	0.00035	0.0028	0.016	0.16	0.53	1.7
50	0.00035	0.0026	0.015	0.16	0.53	1.7
100		0.0025	0.014	0.16	0.53	1.7
500			0.014	0.19	0.54	1.8
1000				0.19	0.64	1.8
n	0.00035	0.0024	0.014	0.19	1.51	25.4

Table 3. Effect on reordering times of the threshold for using the heap.

4.3 Managing the binary heap

We hold the list of indices of eligible nodes in an array `QUEUE`, with the root in `QUEUE(1)`, its children in `QUEUE(2)` and `QUEUE(3)`, the children of `QUEUE(2)` in `QUEUE(4)` and `QUEUE(5)`, etc. Thus the parent of `QUEUE(J)` is always in `QUEUE(J/2)`. We ensure that the priority value for a node is never more than that of its parent. As a result, the root is always the node with the highest priority, so no search is needed to choose the next node for renumbering.

To restore the binary heap after removing the root, we move its child with greater priority into its place, then do the same for the child, continuing until the bottom of the heap is reached. About $\log_2 l$ steps are needed for a list of length l.

When a node is added to the list of eligible nodes, it is added to the bottom of the heap. To ensure that the heap still has the required properties, we need to compare the value of the priority function with that of its parent. If necessary, an interchange with the parent is made and the same comparison is made at the parent node. This continues until the root is reached or a correctly ordered parent and child is reached. At most $\log_2 l$ steps are needed for a list of length l.

The part of the algorithm which is potentially expensive is maintaining the priorities of the eligible nodes as nodes are renumbered. When the value of the priority function of an eligible node changes, it is always an increase caused by a neighbour being included in the front. We need to compare the new value of the priority function with that of its parent as in the previous paragraph. At most $\log_2 l$ steps are again needed, but our experience is that in most cases, no interchanges at all are needed.

4.4 Interchanging the start and end nodes

We observed in Table 1 that which end of the pseudodiameter is chosen as the start node can have a marked effect on the width of the level-set structure and on the RCM semibandwidth. To a lesser extent, which end is chosen as the start node affects the Sloan algorithm. In Table 4, we show the level-set widths and the Sloan profiles for the Kumfert and Pothen test set. For each problem, we report the results for the better of the pairs of weights (2,1) and (16,1) (see Section 4.5 for a discussion of the choice of weights). In column 2, we give the narrowest width and in column 4, the corresponding profile. We highlight the best profile (if the profiles for both ends of the pseudodiameter differ by less than 2%, both are highlighted). It can be seen that for most problems the final profile is not very sensitive to which end is used as the start node but there appears to be a slight advantage in choosing the pseudoperipheral node that gives the narrowest width as the start node. We therefore take this node as the start node

s in our code, since we feel that the added expense of running the second phase of Sloan's algorithm using both nodes would not be justified.

Problem	MGPS Widths		Sloan Profiles	
barth	180	192	**0.47**	**0.47**
barth4	196	212	**0.33**	**0.34**
barth5	359	392	**1.44**	1.49
bcsstk30	1208	1211	16.15	**11.16**
commanche_dual	127	152	**0.33**	**0.33**
copter1	915	917	**6.05**	**6.05**
copter2	2204	2609	**37.96**	38.87
finance256	2012	2012	**6.35**	**6.35**
finance512	2504	2639	**11.91**	**11.94**
ford1	252	303	**2.35**	2.61
ford2	956	961	**41.05**	**41.66**
nasasrb	540	864	**19.01**	**18.63**
onera_dual	2712	5074	**87.75**	103.41
pds10	3290	3419	**9.36**	12.53
shuttle_eddy	167	225	0.62	**0.59**
skirt	1776	1879	36.60	**34.16**
tandem_dual	2139	2331	**66.21**	72.98
tandem_vtx	1472	1774	**5.72**	**5.75**

Table 4. MGPS widths and Sloan profiles (in millions) for the two pseudoperipheral nodes.

4.5 Adjusting Sloan's weights

As already mentioned, Sloan recommends the pair (2,1) for the weights. However, the results of Kumfert and Pothen (1997) indicate that, for some problems, there are considerable advantages in using other values. We have examined the profile sizes for the 13 pairs of weights (1,64), (1,32), (1,16), ..., (1,1), (2,1), ..., (64,1) on all the Everstine and Kumfert and Pothen test matrices. Some examples illustrating our findings are shown in Table 5, where percentage increases from the best value are shown. In both test sets, there are cases for which the profile rises rapidly for large values of W_1/W_2. Kumfert and Pothen call these problems **class two** and the rest **class one**.

The first three examples in Table 5 are class-one problems and the rest are class-two problems. The examples *barth5* and *finance512* were used by Kumfert and Pothen to exemplify classes one and two, respectively. We see from the table that, for class-one problems, it may be important to choose a large value of W_1/W_2. For class-two problems, (1,1) or Sloan's choice of (2,1) both seem reasonable. Our results for the whole test set show that using the weights (2,1) rarely gives profiles for class-two problems that are more than 5% bigger than the best of those we computed. From the Kumfert and Pothen test set, the

class-two problem for which the Sloan choice gave the worst result was *skirt*. Results for this problem are given in the final column of Table 5. For class-one problems, it seems to be rarely advantageous to go beyond a ratio of 16, and can be slightly disadvantageous. Kumfert and Pothen have no suggestion for predicting to which class a problem belongs. It seems to us that, if the class of problem is not known, it is necessary to try more than one pair of weights. To allow for this, in our code `MC60` the weights are input parameters which must be set by the user. The default option in the driver `MC61` is to compute orderings for the pairs (2,1) and (16,1) and to choose the one with the smallest profile. `MC61` also allows the user to specify other choices for the weights.

	barth5	*copter2*	*onera_dual*	*finance512*	*ford1*	*skirt*
Weights						
(1,64)	100.3	53.5	55.0	32.1	10.9	23.0
(1,32)	100.3	53.5	55.0	32.0	10.9	23.0
(1,16)	100.3	53.7	55.0	32.1	10.9	22.6
(1,8)	100.3	53.4	55.0	32.0	10.7	13.4
(1,4)	100.4	51.3	55.0	30.6	9.8	2.7
(1,2)	99.1	41.7	52.2	16.1	7.7	0.0
(1,1)	88.5	26.4	41.7	7.4	4.2	3.3
(2,1)	73.5	13.7	27.8	1.2	0.0	16.1
(4,1)	47.8	7.5	12.2	0.0	0.9	46.4
(8,1)	14.8	5.8	1.6	44.9	8.5	80.6
(16,1)	1.5	0.0	0.0	318.0	10.3	130.7
(32,1)	0.0	0.9	0.3	796.9	24.6	131.6
(64,1)	0.0	0.9	0.3	954.4	25.8	131.7

Table 5. Percentage increases in profiles for different weights.

Both Sloan and Kumfert and Pothen use an integer priority function, but this seems to us to be an unnecessary restriction. We use real values, which have the same storage requirement in the usual case of 4-byte integers and 4-byte reals. We found that there was an increase in execution time, but it was very slight. Using reals means that no tests are needed to ensure that integer overflow does not occur.

Sloan's algorithm will generally avoid very large semibandwidths simply because of not departing far from the underlying rooted level-set structure ordering, but this may be give an ordering that is not satisfactory for an out-of-core variable-band solver. One possibility is to increase the weight W_1, but the direct use of the Reverse Cuthill-Mckee order is likely to be better from this point of view.

4.6 Other adjustments of the priority function

Kumfert and Pothen (1997) point out that the current degree c_i varies between 0 and $\Delta + 1$, where Δ is the maximum degree of a node, while $d(i, e)$ varies between 0 and h, the level-set depth. They therefore suggest replacing (1) by the priority function

$$P_i = -W_1 \, \mathtt{int}(h/\Delta) c_i + W_2 d(i, e). \qquad (2)$$

Our feeling is that it is inappropriate to take the depth into account. What matters is the local nature of the graph and, for the second term in P_i, it is to which level sets the candidate nodes belong. This varies by one from each level set to the next so is already properly normalized.

Using (2), we have examined the profile sizes for the 13 pairs of weights used in the previous section on all the Everstine and Kumfert and Pothen test matrices without seeing any evidence that normalization is needed. Therefore, we do not use (2) in our implementation of Sloan's algorithm.

Strictly speaking, the equations (1) and (2) do not define the priority function fully since we give maximum priority to a node with $c_i = 0$. Thus the priority function is really nonlinear in c_i. Nick Gould suggests [Private Communication] that further nonlinearity might be helpful, but we have not investigated this.

4.7 Other start and end nodes

Recall from Section 3.3 that Paulino *et al.* suggested using the spectral method to find a pseudodiameter and then using the better of the RCM orderings based on the two ends of this pseudodiameter. We can also use the spectral pseudodiameter to give start and target end nodes (s, e) for the numbering phase of Sloan's method. We again choose the start node to be the end of the pseudodiameter which gives the narrowest level-set structure. This is not a big overhead as it just requires one more level-set structure to be constructed. We found the overall quality of the results to be very similar to those obtained using the MGPS pseudodiameter – spectral start and end nodes were better on some problems and worse on others. This is illustrated by the results presented in column 'Sloan Spectral' of Table 6 in Section 5.

5 The hybrid method

Kumfert and Pothen (1997) observed that spectral orderings do well in a global sense but often do poorly locally. They therefore proposed using the spectral method to provide a global ordering to guide Sloan's method. Their results showed that this can yield a much better final ordering than using either the spectral method alone or Sloan's method with the rooted level-set structure ordering. Kumfert and Pothen propose the priority function

$$P_i = -W_1 \,\mathtt{int}(n/\Delta) c_i + W_2 d(i,e) - W_3 p_i, \tag{3}$$

where p_i is the position of node i in the spectral ordering and call this the **hybrid** method. The normalization has been changed to balance the maximum values of the factors for W_1 and the new W_3. They use the spectral pseudodiameter to find the end node e and leave the distance $d(i,e)$ unnormalized, which gives the second term in (3) only a small influence. Although in their paper they report that choosing W_2 to be equal to one generally does significantly better than setting W_2 to zero, they later say [Private Communication] that they regret including this term.

To make the normalization similar to that of the priority function (1), in place of (3) we have chosen to use the priority function

$$P_i = -W_1 \, c_i - W_2 \, (h/n) p_i, \tag{4}$$

where h is the level-set depth. This makes the factor for W_2 vary up to h, as in (1). Our results are summarized in the column 'Hybrid Perm.' of Table 6. For some problems, including *tandem_dual* and *onera_dual*, we found a significant improvement by using the whole spectral order as a guide for Sloan's method. It appears that the spectral ordering of the interior nodes is important. We see no possible justification for using both the level-set order and the spectral order to guide the Sloan algorithm and our results are comparable with those reported by Kumfert and Pothen. We again tried a range of values for W_1 and W_2. For class-two problems, we found that a value of W_1/W_2 that was smaller than that used by Sloan was advantageous. Based on our numerical experiments, for the hybrid method for this class, we recommend using the weights (1,2) rather than (2,1). In Table 6, for the hybrid method we take the better of the results for the weights (1,2) and (16,1). In general, the best weight for p_i must depend on the quality of the permutation and the higher weight that we have found useful with the spectral order indicates that it is of good quality.

For the hybrid method, we again experimented with interchanging the ends of the pseudodiameter. We constructed the level-set structures rooted at the two ends and selected as the starting node the one with the narrowest level structure. We found that for some problems this gave a reduction in the profile but for other problems it gave an increase. We

do not think the differences in the profile are large enough to justify the expense of running the Sloan algorithm from both nodes so in our code we use only one.

We have also tried using the Fiedler vector from the spectral method directly, again adjusting the normalization so that the factor for W_2 in (4) varies up to the depth h. We found (see column 'Hybrid Vector' of Table 6) that overall this did not significantly improve the quality of the results. We also show in Table 6 (column 'Spectral') the profiles for the spectral ordering. A comparison of columns 5 and 7 demonstrate that it is worthwhile to use Sloan's method to refine the spectral ordering.

Problem	Sloan `MC40`	Sloan MGPS	Sloan Spectral	Hybrid Perm.	Hybrid Vector	Spectral
barth	0.49	0.47	0.48	**0.40**	**0.40**	0.46
barth4	0.45	0.33	0.37	**0.29**	**0.29**	0.33
barth5	2.43	1.44	1.48	**1.29**	**1.29**	1.42
bcsstk30	15.72	16.15	14.86	**7.88**	8.20	9.14
commanche_dual	0.44	**0.33**	**0.34**	0.35	0.35	0.35
copter1	7.09	**6.05**	**6.05**	6.11	6.11	7.61
copter2	43.24	37.96	35.28	**32.78**	**32.78**	42.00
finance256	6.57	**6.35**	6.51	**6.44**	6.70	9.17
finance512	12.22	11.91	14.25	11.72	**11.41**	19.13
ford1	2.34	2.35	2.74	1.95	**1.88**	2.17
ford2	40.63	41.05	41.78	**35.97**	**35.64**	40.30
nasasrb	**18.35**	19.01	19.38	19.30	19.21	25.10
onera_dual	113.67	87.75	81.88	**46.67**	**46.66**	53.39
pds10	13.68	9.36	9.87	**8.81**	**8.89**	16.06
shuttle_eddy	**0.59**	0.62	0.62	**0.59**	**0.59**	0.76
skirt	34.12	36.60	33.38	**27.87**	29.26	30.51
tandem_dual	87.79	66.21	79.85	**42.22**	**42.22**	48.38
tandem_vtx	6.29	5.72	5.46	**5.22**	**5.22**	6.19

Table 6. Profiles (in millions) with different algorithms.

In Table 6, we highlight in bold the smallest profile for each problem and any within 2% of the smallest. We also show `MC40` profiles for these problems. Note that tie-breaking can affect all these results so that too much notice should not be taken of small differences. For example, `MC40` gives slightly better profiles than Sloan MGPS on six problems but generally its results are less good because it uses only the weights (2,1).

The hybrid method was intended for very large problems, but we felt that it would be of interest to see how it performs on some of the Everstine problems, since these have been widely used as a test set and very good profiles have been obtained for them by Armstrong (1985) using simulated annealing (which would not be suitable for everyday software). In Table 7, we compare Armstrong's profiles with those obtained with the hybrid method and

with the Sloan MGPS method. On this size of problem, there seems to be little advantage in using the hybrid algorithm; the profiles are within 2% of each other in five cases, and each is significantly better than the other in two cases. In one case, however, the hybrid profile is less than Armstrong's.

n	Sloan MGPS	Hybrid Perm.	Armstrong
758	7.3	7.5	7.1
869	13.9	15.7	13.2
878	19.4	19.2	17.8
918	17.0	17.3	15.9
992	33.5	33.4	32.5
1005	34.7	30.8	32.5
1007	22.7	20.4	19.9
1242	36.5	39.8	33.1
2680	89.7	91.4	84.9

Table 7. Profiles (in thousands) for the three algorithms on the nine largest Everstine problems.

We remark that although we have only used the spectral ordering in the hybrid algorithm, any input ordering can be used. Our codes are written to allow this.

6 Software design

In this section, we discuss the design of new codes for reordering sparse symmetric matrices. Our new subroutines are named according to the naming convention of the Harwell Subroutine Library (HSL 1995). The codes themselves are available; please contact one of the authors for details of price and conditions of use.

Our previous code `MC40` provided the user with a single subroutine. It accepted the strictly lower triangular part of the matrix and returned the permutation and the values of the profiles for the original and permuted orderings. While the design of our new software includes a simple driver, `MC61`, we have decided that it is very worthwhile to give the user the greatest possible flexibility so in the `MC60` package we provide user entries to the component parts of the reordering algorithm. These are described in the following subsections. For further details, the user should refer to the specification sheets (see appendices).

6.1 MC60A

MC60A accepts the pattern of the lower-triangular part of the matrix **A** and constructs the pattern of the whole matrix. Each is held by columns, with pointers to the column starts. For economy of storage, the work is done in place. A first pass looks for any out-of-range or repeated indices and removes them, or terminates if this has been requested. A second pass counts the number of entries that need to be added to each row to include the upper triangle. A third pass works through the rows in reverse order, moving them back to allow space for the additional entries. A final pass inserts the additional entries. There are extensive checks on the data. If the user already has the pattern of the whole matrix and does not wish to checks to made on the data, MC60A is not needed.

6.2 MC60B

MC60B constructs supervariables, given the pattern of the whole matrix. This is done in $O(n+\tau)$ time, where n is the order of the matrix and τ is the number of entries, by working progressively so that after j steps we have the supervariable structure for the submatrix of the first j columns. We start with all variables in one supervariable (for the submatrix with no columns), then split it into two according to which rows do or do not have an entry in column 1, then split these according to the entries in column 2, etc. The splitting is done by moving the variables one at a time to the new supervariable. Further details are given by Duff and Reid (1996).

Note that this strategy requires the user to provide the indices of the entries on the diagonal since these affect whether the structures of columns are identical. This contrasts with MC40, which assumes that the diagonal entries are all nonzero.

Unless all the supervariables consist of a single variable, we now compress the pattern. This is held in the form of the index list for column 1, followed by the index list for column 2, etc., with pointers to column starts. In order that this compression can be performed in place, we identify for each supervariable the member variable with least index as its key variable. This permits us to visit the supervariables in the order of their key variables, picking up the index list of that variable, using it to construct the supervariable index list, and placing the constructed list in its final position without fear of overwriting any information needed later. Note that the supervariable pattern and the map of variable to supervariable indices provides a complete representation of the original pattern. For efficiency, we also hold an array of numbers of variables in supervariables.

The use of MC60B is optional. If it is known that there are few supervariables, MC60B will not be needed. On the other hand, MC60B may be used in combination with another algorithm

for choosing an ordering.

6.3 MC60C

MC60C controls the main part of the algorithm. It works with the supervariable graph (Section 3.4) and returns a supervariable permutation. It allows for the matrix being reducible (a permutation of a block diagonal matrix). In this case, each diagonal block of the permuted matrix will correspond to a component of the graph (set of nodes with no connections to other nodes). It orders any trivial components first by choosing any nodes that have degree zero. It then orders each nontrivial component in turn by calling other subroutines, which allows these other subroutines to work with a single component.

The user has to choose whether Sloan's algorithm or RCM is required for each component. For Sloan's algorithm, the user may specify a global priority vector whose components p_i are used in the priority function (4). This will normally come from a spectral ordering, but is not restricted to this. Apart from this case, a pseudodiameter must be found.

By default, MC60C calls MC60H to compute a pseudodiameter, as explained in Section 3.1. MC60H also distinguishes between the ends, as explained in Section 3.2. Alternatively, the user may specify the two end nodes. In either case, the level-set structure rooted on the end node is needed to provide distances to the end node in Sloan's method and the ordering itself for the RCM method. This is always returned by MC60H; if the end nodes are specified, the level-set structure is computed by calling MC60L directly from MC60C (MC60L is also called from MC60H).

It is this need for the level-set structure that leads to the choice of end node possibly adding an overhead (see Section 3.2). If we do not mind which end is used and if the most recent level-set structure was constructed in full (the construction is terminated early if its width is found to be greater than the narrowest encountered so far, see Section 3.1), a final call of MC60L is not needed.

If Sloan's method is required, this is performed by MC60J, using weights W_1 and W_2 supplied by the user and switching from a simple search to using a binary heap if the front gets large (see Section 4.2). If RCM is required, we have only to reverse the order that we already have.

6.4 MC60D, MC60E, MC60F, and MC60G

MC60D, MC60E, MC60F, and MC60G are simple utilities that convert the supervariable ordering to an ordering for variables or rows, or provide statistics.

MC60D constructs the permutation for the variables that corresponds to a given permutation for the supervariables.

MC60E uses a given permutation for the supervariables to construct the corresponding ordering for the rows, as required by a row-by-row frontal solver such as MA42 (equation entry).

MC60F uses a given permutation for the supervariables to compute the profile, the maximum wavefront, the semibandwidth, and the root-mean-square wavefront for the permuted matrix.

MC60G uses a given row order to compute the maximum row and column front sizes and the root-mean-square row and column front sizes for a row-by-row frontal method.

6.5 The driver MC61

For our driver MC61, there are just two entries. The subroutine MC61I must be called to provide default values for the parameters that control the execution of the package. If the user wishes to use values other than the defaults, the corresponding parameters should be reset after the call to MC61I. MC61A accepts the pattern of the lower-triangular part of **A**, performs full checks on the data, and either

- chooses a permutation of the variables that aims to reduce the profile and wavefront of the matrix,
- chooses a permutation of the variables that aims to reduce the bandwidth of the matrix, or
- constructs an ordering for the rows that is efficient when used with a row-by-row frontal solver.

Although MC61 has a user interface which is similar to that of MC40, it provides a much wider range of options. The user may choose whether or not to use supervariables. The user may also specify the weights for the Sloan priority function (1) and can optionally supply the vector $\{p_i\}$ and weights for the hybrid priority function (3).

7 Concluding discussion

In this report, we have discussed the design and development of a software package, MC60, for computing a symmetric permutation to reduce the profile and wavefront of a large sparse matrix with a symmetric sparsity pattern. The driver MC61 provides the user with a straightforward interface to MC60. In the next release (Release 13) of the Harwell Subroutine Library, MC61 will supersede MC40.

As we discussed in Section 4.5, if Sloan's algorithm (combined with the MGPS algorithm for finding a pseudodiameter) is selected, we recommend computing profiles for the pairs of weights (2,1) and (16,1) and taking the best. This is the default option in MC61. Although the pseudodiameter does not need to be recomputed, the use of two pairs of weights does represent an overhead. To illustrate this and to compare the efficiency of the old and new ordering codes, in Table 8 we report timings for MC40 and MC61 for the Kumfert and Pothen test examples. For MC61 we show times for a single pair of weights and for two pairs of weights, using variables and using supervariables. The results in column 8 are those for the default MC61 parameters.

				MC61			
				One pair weights		Two pairs weights	
Problem	Order	Nsup	MC40	Var.	SVar.	Var.	Svar.
barth	6,691	6,691	0.17	0.18	0.21	0.28	0.31
barth4	6,019	6,019	0.12	0.15	0.14	0.21	0.24
barth5	15,606	15,606	0.56	0.45	0.52	0.71	0.78
bcsstk30	28,924	**9,289**	6.29	3.82	1.91	6.40	2.28
commanche_dual	7,920	7,920	0.13	0.13	0.16	0.22	0.24
copter1	17,222	17,222	1.44	0.57	0.70	0.99	1.09
copter2	55,476	55,476	11.55	2.39	2.80	4.12	4.52
finance256	37,376	37,376	1.93	1.10	1.29	1.85	2.04
finance512	74,752	74,752	3.41	2.12	2.50	3.57	3.95
ford1	18,728	18,210	0.60	0.38	0.44	0.65	0.69
ford2	100,196	97,906	8.68	2.63	2.92	4.39	4.60
nasasrb	54,870	**24,954**	4.59	5.05	2.99	8.51	3.85
onera_dual	85,567	85,567	21.83	2.50	2.84	4.16	4.51
pds10	16,558	16,558	5.40	0.56	0.65	0.92	1.02
shuttle_eddy	10,429	10,363	0.22	0.30	0.35	0.47	0.53
skirt	45,361	**14,956**	8.97	4.83	2.58	8.20	3.17
tandem_dual	94,069	94,069	16.96	2.57	2.90	4.12	4.51
tandem_vtx	18,454	18,454	1.33	0.81	0.94	1.27	1.40

Table 8. Timings for MC40 and MC61. Nsup denotes the number of supervariables. Var. indicates that variables are used. Svar. indicates that supervariables are used.

Kumfert and Pothen report that the hybrid method is more than six times more expensive than the Sloan method. In this study we do not include timings for the hybrid method because the Chaco package that we use to find the Fiedler vector is written in C and we do not

currently have a Fortran code within the Harwell Subroutine Library for computing the Fiedler vector.

As anticipated, for the larger problems, using the binary heap gives significant savings so that, for these problems, `MC61` with two pairs of weights is still generally faster than `MC40`. For the smaller problems, `MC61` can be slower than `MC40`, but the quality of the `MC61` ordering is usually superior. Only three of the test problems, *skirt, nasarb,* and *bcsstk30,* have significantly fewer supervariables than variables (highlighted in bold). For these problems, we see there is a substantial saving in the execution times when supervariables are used. For the other problems, searching for supervariables increases the reordering time but the increase is generally limited to about 15 per cent for a single pair of weights and about 10 per cent for two pairs. We therefore recommend that, unless the user knows his or her problem does not compress well, supervariables with two pairs of weights should be used, and this is the default in `MC61`.

8 Acknowledgements

We are grateful to Gary Kumfert and Alex Pothen of Old Dominion University and to Scott Sloan of the University of Newcastle, New South Wales for helpful discussions. We would like to thank Gary Kumfert for providing us with the test examples used this report. Finally, we would like to thank Scott Sloan, Alex Pothen, and our colleagues Iain Duff and Nick Gould for their helpful comments on drafts of this report.

9 References

Armstrong, B. A. (1985). Near-minimal matrix profiles and wavefronts for testing nodal resequencing algorithms. *Int. J. Numer. Meth. Engng.* **21**, 1785-1790.

Barnard, S. T., Pothen, A., and Simon, H. (1995). A spectral algorithm for envelope reduction of sparse matrices. *Numerical Linear Algebra with Applications* **2**, 317-334.

Cuthill, E. and McKee, J. (1969). Reducing the bandwidth of sparse symmetric matrices. Proceedings 24th National Conference of the Association for Computing Machinery, Brandon Press, New Jersey, 157-172.

Duff, I. S. and Reid, J. K. (1996). Exploiting zeros on the diagonal in the direct solution of indefinite sparse symmetric linear systems. *ACM Trans. Math. Softw.* **22**, 227-257.

Duff, I. S. and Scott, J. A. (1996). The design of a new frontal code for solving sparse, unsymmetric systems. *ACM Trans. Math. Softw.* **22**, 30-45.

Duff, I. S., Erisman, A. M., and Reid, J. K. (1986). Direct methods for sparse matrices. Oxford University Press, London.

Duff, I. S., Reid, J. K., and Scott, J. A. (1989). The use of profile reduction algorithms with a frontal code. *Int. J. Numer. Meth. Engng.* **28**, 2555-2568.

Everstine, G. C. (1979). A comparison of three resequencing algorithms for the reduction of matrix profile and wavefront. *Int. J. Numer. Meth. Engng.* **14**, 837-853.

George, A. (1971). Computer implementation of the finite-element method. Report STAN CS-71-208, Ph.D Thesis, Department of Computer Science, Stanford University, Stanford, California.

George, A. and Liu, J. W. H. (1979). An implementation of a pseudoperipheral node finder. *ACM Trans. Math. Softw.* **5**, 284-295.

Gibbs, N. E., Poole, W. G., Jr., and Stockmeyer, P. K. (1976). An algorithm for reducing the bandwidth and profile of a sparse matrix. *SIAM J. Numer. Anal.* **13**, 236-250.

Gibbs, N. E. (1976). A hybrid profile reduction algorithm. *ACM Trans. Math. Softw.* **2**, 378-387.

Hendrickson, B. and Leland, R. (1995). The Chaco user's guide: Version 2.0, Tech. Rep. SAND94-2692, Sandia National Laboratories, Albuquerque, NM.

HSL (1995). Harwell Subroutine Library Catalogue (Release 12). AEA Technology, Harwell.

Kumfert, G. and Pothen, A. (1997). Two improved algorithms for envelope and wavefront

reduction. *BIT* **18**, 559-590.

Lewis, J. G. (1982). Implementation of the Gibbs-Poole-Stockmeyer and Gibbs-King algorithms. *ACM Trans. Math. Softw.* **8**, 180-189 and 190-194.

Paulino, G. H., Menezes, I. V. M., Gattass, M., and Mukherjee, S. (1994). A new algorithm for finding a pseudoperipheral vertex or the endpoints of a pseudodiameter in a graph. *Communications in Numer. Meth. Engng.* **10**, 913-926.

Sloan, S. W. (1986). An algorithm for profile and wavefront reduction of sparse matrices. *Int. J. Numer. Meth. Engng.* **23**, 239-251.

Sloan, S. W. (1989). A Fortran program for profile and wavefront reduction. *Int. J. Numer. Meth. Engng.* **28**, 2651-2679.

Sloan, S. W. and M. F. Randolph (1983). Automatic element reordering for finite element analysis with frontal solution schemes. *Int. J. Numer. Meth. Engng.* **19**, 1153-1181.

Appendix 1. MC60 specification document

1 SUMMARY

This subroutine uses a variant of Sloan's method to calculate a symmetric permutation that aims to **reduce the profile and wavefront of a sparse matrix A with a symmetric sparsity pattern.** Alternatively, the Reverse Cuthill-McKee Method may be requested to reduce the bandwidth. There are optional facilities for looking for sets of columns with identical patterns and taking advantage of them. There is also an option for computing a row order that would be appropriate for use with a row-by-row frontal solver (for example, the equation entry to MA42). These optional facilities may also be used independently.

ATTRIBUTES — **Versions:** MC60A, MC60AD. **Calls:** None. **Date:** January 1998. **Origin:** J. K. Reid and J. A. Scott (Rutherford Appleton Laboratory). **Conditions on external use:** (i), (ii), (iii) and (iv).

2 HOW TO USE THE PACKAGE

2.1 Argument lists

There are seven entries to this package:

MC60A accepts the pattern of the lower-triangular part of the matrix and constructs the pattern of the whole matrix. There are extensive checks on the data. This is the usual initial entry.

MC60B looks for sets of columns with identical patterns. We refer to the set of variables that correspond to a set of identical columns as a supervariable. A permutation is constructed that places the variables of each supervariable together. The pattern is replaced by that of the permuted matrix represented as supervariables (that is, by its condensed equivalent).

MC60C chooses a supervariable permutation that aims to reduce the profile and wavefront or the bandwidth. It also provides pseudoperipheral pairs of nodes of the components of the supervariable graph of the matrix.

MC60D constructs the permutation for the variables that corresponds to a given permutation for the supervariables.

MC60E uses a given permutation for the supervariables to construct the corresponding row order, as required by a row-by row frontal solver, such as MA42.

MC60F uses a given permutation for the supervariables to compute the profile, the maximum wavefront, the semibandwidth, and the root-mean-square wavefront for the permuted matrix.

MC60G uses a given row order to compute the maximum row and column front sizes and the root-mean-square row and column front sizes for a row-by-row frontal method.

Only the initial entry provides extensive checks on the data. If it is known that there are few supervariables, MC60B will not be needed. On the other hand, MC60B may be used in combination with another algorithm for choosing a permutation.

2.1.1 Initial entry

The single precision version

```
CALL MC60A(N,LIRN,IRN,ICPTR,ICNTL,IW,INFO)
```

The double precision version

```
CALL MC60AD(N,LIRN,IRN,ICPTR,ICNTL,IW,INFO)
```

N is an INTEGER variable that must be set by the user to the order of the matrix. N is not altered. **Restriction:** $N \geq 1$.

LIRN is an INTEGER variable that must be set by the user to the length of the array IRN, which must be large enough to hold the pattern of the whole matrix (excluding duplicated and out-of-range indices). 2*(ICPTR(N+1)-1) is always sufficient. LIRN is not altered.

IRN is an INTEGER array of length LIRN whose leading part must be set by the user to hold the row indices of

the entries in the lower triangle (including those on the diagonal) of the matrix **A**. The entries of each column must be contiguous. The entries of column J must precede those of column J+1, J=1,2, ..., N-1, and there must be no wasted space between the columns. Row indices within a column may be in any order. On successful return, the array will be changed to hold the row indices of the whole matrix **A** in the same format.

ICPTR is an INTEGER array of length N+1 that must be set by the user so that ICPTR(J) points to the position in the array IRN of the first entry in column J, J=1,2, ..., N, and ICPTR(N+1)–1 must be the position of the last entry. On successful return, ICPTR holds corresponding data for the revised IRN.

ICNTL is an INTEGER array of length 2 that controls the action:

ICNTL(1) controls whether the computation terminates if duplicated or out-of-range indices are detected:

0 – Terminates if any duplicated or out-of-range indices found.

1 – Any duplicated or out-of-range indices are ignored.

ICNTL(2) controls printing of diagnostic messages:

0 – No diagnostic messages are required.

> 0 – The unit number for diagnostic messages.

ICNTL is not altered.

IW is an INTEGER array of length N that is used by the subroutine as workspace.

INFO is an INTEGER array of length 4 that need not be set by the user.

INFO(1) is used as an error flag. On a successful exit, it is set to:

0 – No out-of-range or duplicated indices.

1 – Some out-of-range or duplicated indices when ICNTL(1)=1.

If a fatal error has been detected, it is set to a negative value:

–1 – N < 1 or LIRN less than ICPTR(N+1)–1. Immediate return with IRN and ICPTR unchanged.

–2 – LIRN is too small. INFO(4) is set to the minimum value that will suffice. If ICNTL(1)=1, any out-of-range or duplicated variable indices will have been excluded from IRN and ICPTR. Otherwise, IRN and ICPTR are unchanged.

–3 – ICNTL(1)=0 and one or more variable indices either lies outside the lower triangle of the matrix or is duplicated (see INFO(2) and INFO(3)). IRN and ICPTR are unchanged.

INFO(2) holds the number of variable indices in IRN found to be out-of-range.

INFO(3) holds the number of indices in IRN that represent duplicates of previous entries.

INFO(4) holds the minimum value that will suffice for LIRN, unless INFO(1)=–1.

2.1.2 To find supervariables and compress the pattern

The single precision version

```
CALL MC60B(N,LIRN,IRN,ICPTR,NSUP,SVAR,VARS,IW)
```

The double precision version

```
CALL MC60BD(N,LIRN,IRN,ICPTR,NSUP,SVAR,VARS,IW)
```

N is an INTEGER variable that must be set by the user to the order of the matrix. N is not altered.

LIRN is an INTEGER variable that must be set by the user to the length of the array IRN. LIRN is not altered.

IRN is an INTEGER array of length LIRN. On entry, it may be as returned by MC60A/AD. Alternatively, it may be set by the user to hold the row indices of the whole matrix. The entries of each column must be contiguous. The entries of column J must precede those of column J+1, J=1,2, ..., N-1, and there must

be no wasted space between the columns. The row indices within a column may be in any order. Columns with no entries are permitted. No checks on the format are performed. On successful return, IRN holds the row indices of the entries in the condensed matrix, using the same format.

ICPTR is an INTEGER array of length N+1. On entry, it may be as returned by MC60A/AD. Alternatively, it may be set by the user so that ICPTR(J) points to the position in the array IRN of the first entry in column J, J=1,2, ..., N, and ICPTR(N+1)-1 points to the last entry. On successful return, ICPTR holds corresponding data for the condensed matrix.

NSUP is an INTEGER variable that need not be set on entry. On return, it holds the number of supervariables.

SVAR is an INTEGER array of length N that need not be set on entry. On successful return, SVAR(I) holds the supervariable to which variable I belongs, I=1,2, ..., N.

VARS is an INTEGER array of length N that need not be set on entry. On successful return, VARS(IS) holds the number of variables in supervariable IS, IS=1,2, ..., NSUP.

IW is an INTEGER array of length 2*N+2 that is used by the subroutine as workspace.

2.1.3 To find supervariable permutation

A normal call to MC60C/CD will follow a successful call to MC60B/BD, in which case the arguments N, NSUP, LIRN, IRN, ICPTR, and VARS should be unchanged since return from MC60B/BD. However, it may be called independently, so we describe these arguments as if they were provided afresh.

By making each supervariable consist of a single variable, MC60C/CD may also be used to find a permutation for the variables. In this case, NSUP must equal N and all elements of VARS must equal 1; N, LIRN, IRN, ICPTR will normally be unchanged since return from MC60A/AD, but they may be provided afresh. We do not recommend this option unless it is known that the problem has few supervariables.

The single precision version

```
CALL MC60C(N,NSUP,LIRN,IRN,ICPTR,VARS,JCNTL,PERMSV,WEIGHT,PAIR,INFO,IW,W)
```

The double precision version

```
CALL MC60CD(N,NSUP,LIRN,IRN,ICPTR,VARS,JCNTL,PERMSV,WEIGHT,PAIR,INFO,IW,W)
```

N is an INTEGER variable that must be set by the user to the order of the matrix. This argument is not altered.

NSUP is an INTEGER variable that must be set to hold the number of supervariables. NSUP is not altered.

LIRN is an INTEGER variable that must be set by the user to the length of the array IRN. LIRN is not altered.

IRN is an INTEGER array of length LIRN. It must be set by the user to hold the row indices of the condensed matrix. The entries of each column must be contiguous. The entries of column J must precede those of column J+1, J=1,2, ..., NSUP-1, and there must be no wasted space between the columns. The row indices within a column may be in any order. No checks on the format are performed. IRN is not altered.

ICPTR is an INTEGER array of length NSUP+1. It must be set by the user so that ICPTR(J) points to the position in the array IRN of the first entry in column J, J=1,2, ..., NSUP, and ICPTR(NSUP+1)-1 points to the last entry. ICPTR is not altered.

VARS is an INTEGER array of length NSUP. VARS(IS) must hold the number of variables in supervariable IS, IS=1,2, ..., NSUP. VARS is not altered.

JCNTL is an INTEGER array of length 2 that controls the action:

JCNTL(1) controls the choice of algorithm:

- 0 – Sloan's algorithm for reducing profile and wavefront.
- 1 – Reverse Cuthill-McKee algorithm (RCM) for reducing bandwidth.

JCNTL(2) controls algorithmic details:

- 0 – Automatic choice of pseudoperipheral pairs.
- 1 – Pseudoperipheral pairs specified in PAIR.

2 – Global priority vector given in PERMSV (Sloan's algorithm only).

JCNTL is not altered.

PERMSV is an INTEGER array of length NSUP. It need be set on entry only if JCNTL(2)=2, and in this case must hold positive global priority values for the supervariables; this may be a permutation, in which case the supervariable for which PERMSV(IS) = 1 is likely to be chosen first and the supervariable for which PERMSV(IS) = NSUP is likely to be chosen last. On exit, the position of supervariable IS in the new ordering is given in all cases by PERMSV(IS), IS = 1,2, ..., NSUP.

WEIGHT is a REAL (DOUBLE PRECISION in the D version) array of length 2. For Sloan's algorithm (JCNTL(1)=0), it must be set to the weights W_1 and W_2 in the priority function that is minimized when choosing the next supervariable in the order. The value of the function is

$$W_1\, deg(s) + W_2\, \nu\, glob(s)$$

where $deg(s)$ is the number of variables that will enter the front if supervariable s is chosen next, ν is a normalizing factor (see Section 4.3), and $glob(s)$ is the (positive) global priority value of supervariable s (generated automatically or provided in PERMSV). The choice of weights is discussed in Section 4.3.

PAIR is an INTEGER array of shape (2,NSUP/2). If JCNTL(2)=0, it need not set on entry and on return PAIR(1,IC), PAIR(2,IC) hold the pseudoperipheral pair for nontrivial component IC, IC = 1,2, ..., INFO(1). The first component is the largest. If JCNTL(2)=1, it must be set on entry to the pseudoperipheral pairs of the components and is not altered; the first component need not be the largest. If JCNTL(2)=2, it is not used.

INFO is a INTEGER array of length 4 that need not be set by the user. On exit,

INFO(1) holds the number of nontrivial components (two or more nodes) in the graph of the condensed matrix,

INFO(2) holds the number of variables in the largest component of the graph,

INFO(3) holds the number of level sets in the level-set structure of the largest component, and

INFO(4) holds the width of the level-set structure of the largest component.

IW is an INTEGER array of length 3*NSUP+1 that is used by the subroutine as workspace.

W is a REAL (DOUBLE PRECISION in the D version) array of length NSUP that is used by the subroutine as workspace.

2.1.4 To find permutation for variables from supervariable permutation

The single precision version

```
CALL MC60D(N,NSUP,SVAR,VARS,PERMSV,PERM,POSSV)
```

The double precision version

```
CALL MC60DD(N,NSUP,SVAR,VARS,PERMSV,PERM,POSSV)
```

N is an INTEGER variable that must hold the order of the matrix. N is not altered.

NSUP is an INTEGER variable that must hold the number of supervariables. NSUP is not altered.

SVAR is an INTEGER array of length N. SVAR(I) must hold the supervariable to which variable I belongs, I=1,2, ..., N. SVAR is not altered.

VARS is an INTEGER array of length NSUP. VARS(IS) must hold the number of variables in supervariable IS, IS=1,2, ..., NSUP. VARS is not altered.

PERMSV is an INTEGER array of length NSUP. It may be as returned by MC60C/CD. Alternatively, it may be set by the user so that PERMSV(IS) holds the position to which supervariable IS is permuted, IS=1,2, ..., NSUP. PERMSV is not altered..

PERM is an INTEGER array of length N that need not be set by the user. On return, PERM(I) holds the position of variable I, I=1,2, ..., N, in the permuted list of variables.

POSSV is an INTEGER array of length NSUP that need not be set by the user. On return, POSSV(IS) holds the position of the first variable of supervariable IS, IS=1,2, ..., NSUP, in the permuted list of variables.

2.1.5 To find a row-by-row frontal order from a supervariable permutation

The single precision version

```
CALL MC60E(N,NSUP,LIRN,IRN,ICPTR,SVAR,VARS,PERMSV,PERM,IW)
```

The double precision version

```
CALL MC60ED(N,NSUP,LIRN,IRN,ICPTR,SVAR,VARS,PERMSV,PERM,IW)
```

N is an INTEGER variable that must hold the order of the matrix. N is not altered.

NSUP is an INTEGER variable that must hold the number of supervariables. NSUP is not altered.

LIRN is an INTEGER variable that must be set by the user to the length of the array IRN. LIRN is not altered.

IRN is an INTEGER array of length LIRN. It must be as for MC60C/CD. No checks on the format are performed. IRN is not altered.

ICPTR is an INTEGER array of length NSUP+1. It must be as for MC60C/CD. No checks on the format are performed. ICPTR is not altered.

SVAR is an INTEGER array of length N. SVAR(I) must hold the supervariable to which variable I belongs, I=1,2, ..., N. SVAR is not altered.

VARS is an INTEGER array of length NSUP. VARS(IS) must hold the number of variables in supervariable IS, IS=1,2, ..., NSUP. VARS is not altered.

PERMSV is an INTEGER array of length NSUP. It may be as returned by MC60C/CD. Alternatively, it may be set by the user so that PERMSV(IS) holds the new index for supervariable IS, IS=1,2, ..., NSUP. On return, the row-by-row order for the rows of the condensed matrix is PERMSV(1), PERMSV(2), ..., PERMSV(NSUP).

PERM is an INTEGER array of length N that need not be set by the user. On return, the row-by-row order for the rows of the matrix **A** is PERM(1), PERM(2), ..., PERM(N).

IW is an INTEGER array of length NSUP that is used by the subroutine as workspace.

2.1.6 To compute the profile and wavefront for a supervariable permutation

The single precision version

```
CALL MC60F(N,NSUP,LIRN,IRN,ICPTR,VARS,PERMSV,IW,RINFO)
```

The double precision version

```
CALL MC60FD(N,NSUP,LIRN,IRN,ICPTR,VARS,PERMSV,IW,RINFO)
```

N is an INTEGER variable that must hold the order of the matrix. N is not altered.

NSUP is an INTEGER variable that must hold the number of supervariables. NSUP is not altered.

LIRN is an INTEGER variable that must be set by the user to the length of the array IRN. LIRN is not altered.

IRN is an INTEGER array of length LIRN. It must be as for MC60C/CD. No checks on the format are performed. IRN is not altered.

ICPTR is an INTEGER array of length NSUP+1. It must be as for MC60C/CD. No checks on the format are performed. ICPTR is not altered.

VARS is an INTEGER array of length NSUP. VARS(IS) must hold the number of variables in supervariable IS, IS=1,2, ..., NSUP. VARS is not altered.

PERMSV is an INTEGER array of length NSUP. It may be as returned by MC60C/CD. Alternatively, it may be set by the user so that PERMSV(IS) holds the new index for supervariable IS, IS=1,2, ..., NSUP. If data for the original order are required, PERMSV(IS) should be set to IS, IS=1,2, ..., NSUP. PERMSV is not altered.

IW is an INTEGER array of length 2*NSUP+1 that is used by the subroutine as workspace.

RINFO is a REAL (DOUBLE PRECISION in the D version) array of length 4 that need not be set by the user. On exit RINFO(1), RINFO(2), RINFO(3), and RINFO(4) hold, respectively, the profile, the maximum wavefront, the semibandwidth, and the root-mean-square wavefront for the permuted matrix.

2.1.7 To compute the front sizes for a row-by-row frontal method

The single precision version

```
CALL MC60G(N,NSUP,LIRN,IRN,ICPTR,VARS,PERMSV,IW,RINFO)
```

The double precision version

```
CALL MC60GD(N,NSUP,LIRN,IRN,ICPTR,VARS,PERMSV,IW,RINFO)
```

N is an INTEGER variable that must hold the order of the matrix. N is not altered.

NSUP is an INTEGER variable that must hold the number of supervariables. NSUP is not altered.

LIRN is an INTEGER variable that must be set by the user to the length of the array IRN. LIRN is not altered.

IRN is an INTEGER array of length LIRN. It must be as for MC60C/CD. No checks on the format are performed. IRN is not altered.

ICPTR is an INTEGER array of length NSUP+1. It must be as for MC60C/CD. No checks on the format are performed. ICPTR is not altered.

VARS is an INTEGER array of length NSUP. VARS(IS) must hold the number of variables in supervariable IS, IS=1,2, ..., NSUP. VARS is not altered.

PERMSV is an INTEGER array of length NSUP. It may be as returned by MC60E/ED. Alternatively, it may be set by the user so that the row-by-row order for the condensed matrix is PERMSV(1), PERMSV(2), ..., PERMSV(NSUP). PERMSV is not altered.

IW is an INTEGER array of length NSUP that is used by the subroutine as workspace.

RINFO is a REAL (DOUBLE PRECISION in the D version) array of length 4 that need not be set by the user. On exit RINFO(1), RINFO(2), RINFO(3), and RINFO(4) hold, respectively, the maximum row and column front sizes and the root-mean-square row and column front sizes.

3 GENERAL INFORMATION

Use of common: None.

Other routines called directly: The subroutines documented here call the following subroutines of the MC60 package: MC60H/HD, MC60J/JD, MC60L/LD, MC60O/OD, and MC60P/PD.

Input/output: If ICNTL(2) > 0 , diagnostic messages on unit ICNTL(2) (MC60A/AD only).

Restrictions: N ≥ 1.

4 METHOD

4.1 MC60A

For economy of storage, MC60A performs its work in place. A first pass looks for any out-of-range or repeated indices and removes them, or terminates if this has been requested. A second pass counts the number of entries that need to be added to each row to include the upper triangle. A third pass works through the rows in reverse order, moving them back to allow space for the additional entries. A final pass inserts the additional entries. There are extensive checks on the data. If the user already has the pattern of the whole matrix and does not wish to checks to made on the data, MC60A is not needed.

4.2 MC60B

MC60B constructs supervariables in $O(n+\tau)$ time, where n is the order of the matrix and τ is the number of entries, by working progressively so that after j steps we have the supervariable structure for the submatrix of the first j columns. We start with all variables in one supervariable (for the submatrix with no columns), then split it into two according to which rows do or do not have an entry in column 1, then split these according to the entries in column 2, etc. The splitting is done by moving the variables one at a time to the new supervariable. Further details are given by Reid and Scott (1998).

Note that this strategy requires the user to provide the indices of the entries on the diagonal since these affect whether the structures of columns are identical. This contrasts with MC40, which assumes that the diagonal entries are all nonzero.

The use of MC60B is optional. If it is known that there are few supervariables, MC60B will not be needed. On the other hand, MC60B may be used in combination with another algorithm for choosing an ordering.

4.3 MC60C

MC60C controls the main part of the algorithm. It works with the supervariable graph, which has *nsup* nodes and an edge between nodes i and j if entry i,j is present in the condensed matrix. It allows for the matrix being reducible (a permutation of a block diagonal matrix). In this case, each diagonal block of the permuted matrix will correspond to a component of the graph (set of nodes with no connections to other nodes). It orders any trivial components first by choosing any nodes that have no connections to other nodes. It then orders each nontrivial component in turn by calling other subroutines, which allows these other subroutines to work with a single component.

If JCNTL(2)=0, a pair of well-separated nodes (a pseudoperipheral pair) is selected for each nontrivial component by using a procedure which is a modification of that given by Gibbs, Poole, and Stockmeyer (1976). Further details are given by Reid and Scott (1998). Alternatively, the pairs may be specified by the user (JCNTL(2)=1).

Given a pseudoperipheral node, a *level-set structure rooted on the node* consists of the node itself at level 1 and at each level i the nodes that are neighbours of nodes at level i–1 but are not members of levels 1, 2, ..., i–1. The depth is the number of level sets and the width is the greatest number of variables associated with all the nodes of a single level. Once we have a pseudoperipheral pair, we chose the node whose rooted level set has the greater depth or, if they have the same depth, the lesser width. The ordering associated with this node by taking its last level set first, then its penultimate level set, etc. is used directly for the RCM method. The corresponding level-set indices are used as a global priority vector for Sloan's algorithm. The global priority vector may also be supplied by the user, for example, from a spectral ordering (see Barnard, Pothen, and Simon 1995), in which case we normalize it with the factor v, chosen to make the range the same as if the level-set indices were in use.

In the method of Sloan (1986), each successive node is chosen to minimize a weighted average of the global priority function and the number of variables that will enter the front if this node is chosen next. For the weights, Sloan recommends the pair (2,1) if the global priority vector is based on a rooted level-set structure. For global priority vectors based on the spectral method, we found that (1,2) was often to be preferred. However the global priority vector was generated, we have found that on some problems the weights (16,1) are significantly better. By default, our driver MC61 therefore calls MC60C twice, either with weights (2,1) and (16,1) or with weights (1,2) and (16,1), and takes the better result. It also checks that there is an improvement over the natural order.

Further details about the MC60C algorithms and their performance are given by Reid and Scott (1998).

4.4 MC60D, MC60E, MC60F, and MC60G

The remaining subroutines perform straightforward tasks of converting a supervariable ordering to an ordering for variables or rows, or providing statistics.

References

Barnard, S. T., Pothen, A., and Simon, H. (1995). A spectral algorithm for envelope reduction of sparse matrices. Numerical Linear Algebra with Applications **2**, 317-334.

Gibbs, N.E., Poole, W.G., and Stockmeyer, P.K. (1976). An algorithm for reducing the profile and bandwidth of

a sparse matrix. SIAM J. Numer. Anal. **13**, 236-250.

Reid, J.K. and J.A. Scott. (1998). Ordering symmetric sparse matrices for small profile and wavefront. Technical Report RAL-TR-98-016, Rutherford Appleton Laboratory.

Sloan, S.W. (1986). An algorithm for profile and wavefront reduction of sparse matrices. Inter. J. Numer. Meth. Engng **23**, 239-251.

5 EXAMPLE OF USE

The following program provides a simple example of the use of `MC60`. It works with or without looking for supervariables.

```
      INTEGER LIRN,MAXN
      PARAMETER (LIRN=20, MAXN=5)
      INTEGER CASE,N,NNZ,IRN(LIRN),ICPTR(MAXN+1),ICNTL(2),
     *      IW(3*MAXN+1),INFO(4),NSUP,I,SVAR(MAXN),VARS(MAXN),
     *      PERM(MAXN),PERMSV(MAXN),JCNTL(2),PAIR(2,MAXN/2)
      REAL WEIGHT(2),W(MAXN),RINFO(4)
      CHARACTER ALG

C Set parameter values
      ICNTL(1) = 0
      ICNTL(2) = 6
      JCNTL(1) = 0
      JCNTL(2) = 0
      WEIGHT(1) = 2.0
      WEIGHT(2) = 1.0

      DO 20 CASE = 1,2
C Read in data for the lower-triangular part
        READ (5,*) N,NNZ
        READ (5,*) (IRN(I),I = 1,NNZ)
        READ (5,*) (ICPTR(I),I = 1,N+1)

C Construct pattern of whole matrix
        CALL MC60A(N,LIRN,IRN,ICPTR,ICNTL,IW,INFO)

C Check for an error return
        IF (INFO(1).NE.0) THEN
          WRITE(6,'(A,2I3)')' MC60A failed with INFO=',INFO
          STOP
        END IF

        READ(5,*) ALG
        IF (ALG.EQ.'S') THEN
C Work with supervariables
          CALL MC60B(N,LIRN,IRN,ICPTR,NSUP,SVAR,VARS,IW)
          WRITE(6,'(A,5I4)') 'The number of supervariables is', NSUP
          CALL MC60C(N,NSUP,LIRN,IRN,ICPTR,VARS,JCNTL,PERMSV,WEIGHT,
     *            PAIR,INFO,IW,W)
          CALL MC60F(N,NSUP,LIRN,IRN,ICPTR,VARS,PERMSV,IW,RINFO)
          CALL MC60D(N,NSUP,SVAR,VARS,PERMSV,PERM,IW)
        ELSE
C Work with variables
          NSUP = N
          DO 10 I = 1,N
            VARS(I) = 1
   10     CONTINUE
          CALL MC60C(N,NSUP,LIRN,IRN,ICPTR,VARS,JCNTL,PERM,WEIGHT,
     *            PAIR,INFO,IW,W)
          CALL MC60F(N,NSUP,LIRN,IRN,ICPTR,VARS,PERM,IW,RINFO)
        END IF
```

```
          WRITE(6,'(A,5I4)') 'The chosen permutation is', PERM
          WRITE(6,'(A,F4.0,/)') 'The profile is', RINFO(1)

   20 CONTINUE

          END
```

Suppose we wish to reduce the profile of a matrix with the following sparsity pattern:

$$\mathbf{A} = \begin{pmatrix} \times & \times & \times & \times & \times \\ \times & \times & \times & & \\ \times & \times & \times & & \\ \times & & & \times & \\ \times & & & & \times \end{pmatrix}.$$

The following input data works with and then without supervariables. In each case, the lower-triangular part is provided column by column. format.

```
5 10
1  2  3  4  5  2  3  3  4  5
1  6  8  9 10 11
Variables

5 10
1  2  3  4  5  2  3  3  4  5
1  6  8  9 10 11
Supervariables
```

This produces the output:

```
The chosen permutation is   3   5   4   1   2
The profile is 10.

The number of supervariables is   4
The chosen permutation is   3   5   4   1   2
The profile is 10.
```

The pattern of the reordered matrix is:

$$\begin{pmatrix} \times & & \times & & \\ & \times & \times & & \\ \times & \times & \times & \times & \times \\ & & \times & \times & \times \\ & & \times & \times & \times \end{pmatrix}.$$

Appendix 2. MC61 specification document

1 SUMMARY

Let **A** be an $n \times n$ sparse matrix with a symmetric sparsity pattern. Given the sparsity pattern of **A**, this subroutine uses a variant of Sloan's method to calculate a symmetric permutation that aims to **reduce the profile and wavefront** of **A**. Alternatively, the Reverse Cuthill-McKee (RCM) method may be requested to reduce the bandwidth, or the user may request an ordering for the rows of **A** that is efficient when used with a row-by-row frontal solver (for example, equation entry to `MA42`).

`MC61` provides the user with a straightforward interface to the `MC60` package when detailed control of the steps in constructing a symmetric permutation or row ordering is not required.

ATTRIBUTES — **Remark:** `MC61` supersedes `MC40`. **Versions:** `MC61A`, `MC61AD`. **Calls:** `MC60`. **Date:** January 1998. **Origin:** J. K. Reid and J. A. Scott (Rutherford Appleton Laboratory). **Conditions on external use:** (i), (ii), (iii) and (iv).

2 HOW TO USE THE PACKAGE

2.1 Argument lists

There are two entries:

(a) The subroutine `MC61I/ID` must be called to initialize the parameters that control the execution of the package.

(b) `MC61A/AD` either chooses a variable permutation that aims to reduce the profile and wavefront or bandwidth of the matrix or constructs an ordering for the rows that is efficient when used with a row-by-row frontal solver, such as `MA42` (equation entry) or `MA43`.

2.1.1 The initialization subroutine

To initialize control parameters, the user should make a call of the following form:

The single precision version

```
CALL MC61I(ICNTL,CNTL)
```

The double precision version

```
CALL MC61ID(ICNTL,CNTL)
```

`ICNTL` is an `INTEGER` array of length `10` that need not be set by the user. This array is used to hold control parameters. On exit, `ICNTL` contains default values. If the user wishes to use values other than the defaults, the corresponding entries in `ICNTL` should be reset after the call to `MC61I/ID`. The default values are as follows:

`ICNTL(1)` is the stream number for error messages. It has the default value `6`. Printing of error messages is suppressed if `ICNTL(1)` ≤ 0.

`ICNTL(2)` is the stream number for warning messages. It has the default value `6`. Printing of warning messages is suppressed if `ICNTL(2)` ≤ 0.

`ICNTL(3)` controls the action taken if duplicate or out-of-range indices are detected. If `ICNTL(3)=0` and such indices are detected, the computation terminates with `IRN` and `ICPTR` unchanged. If `ICNTL(3)=1`, a warning is issued and the computation continues with such indices ignored. The default value is `0`.

`ICNTL(4)` controls whether supervariables are used (a supervariable is a set of variables that correspond to a set of identical columns and the condensed matrix is the representation of **A** by supervariables). If `ICNTL(4)` $= 0$, supervariables are used and are not used if `ICNTL(4)` $= 1$. If the problem has significantly fewer supervariables than variables, using supervariables will substantially reduce the execution time and amount of integer workspace used. The default value is `0`.

ICNTL(5) indicates whether the user wishes to supply a global priority function. If ICNTL(5) = 0, no priority function is supplied; if ICNTL(5) = 1, a priority function is supplied in PERM. The default value is 0.

ICNTL(6) controls whether the user wishes to supply the weights for the priority function (JOB = 1 or 3) (see Section 4). If ICNTL(6) = 0, the code will use the pairs of weights (2,1) and (16,1) and will return results for whichever pair yields the best permutation; if ICNTL(6) = 1, the pairs of weights (1,2) and (16,1) will be used and results for whichever pair yields the best permutation will be returned; if ICNTL(6) = 2, the weights in CNTL(1) and CNTL(2) will be used. The weights which are used to give the final permutation are returned in RINFO(9) and RINFO(10). The default value is 0.

ICNTL(7) to ICNTL(10) are given the default value 0. They are currently not used but may be used in a later release of the code.

CNTL is a REAL (DOUBLE PRECISION in the D version) array of length 5 that need not be set by the user. This array is used to hold control parameters. On exit, CNTL contains default values. If the user wishes to use values other than the defaults, the corresponding entries in CNTL should be reset after the call to MC61I/ID. The default values are as follows:

CNTL(1) and CNTL(2) hold the weights W_1 and W_2 that are used in the priority function in the case ICNTL(6) = 2. The default values are 2.0 and 1.0, respectively.

CNTL(3) to CNTL(5) are given the default value zero. They are currently not used but may be used in a later release of the code.

2.1.2 To find a variable permutation or a row ordering

The single precision version

```
CALL MC61A(JOB,N,LIRN,IRN,ICPTR,PERM,LIW,IW,W,ICNTL,CNTL,INFO,RINFO)
```

The double precision version

```
CALL MC61AD(JOB,N,LIRN,IRN,ICPTR,PERM,LIW,IW,W,ICNTL,CNTL,INFO,RINFO)
```

JOB is an INTEGER variable that must be set by the user to 1 if a variable permutation to reduce the profile and wavefront of the matrix is required, to 2 if a variable permutation to reduce the bandwidth is required, and to 3 if a row ordering for a row-by-row frontal solver is required. This argument is not altered. **Restriction:** JOB = 1, 2, or 3.

N is an INTEGER variable that must be set by the user to the order of the matrix **A**. This argument is not altered. **Restriction:** N ≥ 1.

LIRN is an INTEGER variable that must be set by the user to the length of the array IRN, which must be large enough to hold the sparsity pattern of the whole matrix. 2*(ICPTR(N+1)-1) is always sufficient. This argument is not altered.

IRN is an INTEGER array of length LIRN whose leading part must be set by the user to hold the row indices of the entries in the lower triangle of the matrix, including those on the diagonal. The entries of each column must be contiguous. The entries of column J must precede those of column J+1 (J=1,2, ..., N-1), and there must be no wasted space between the columns. Row indices within a column may be in any order. On successful exit, IRN holds the row entries of the condensed matrix (upper and lower triangular parts), using the same format.

ICPTR is an INTEGER array of length N+1 that must be set by the user so that ICPTR(J) points to the position in the array IRN of the first entry in column J (J=1,2, ..., N), and ICPTR(N+1)-1 must be the position of the last entry. On successful exit, ICPTR holds corresponding data for the condensed matrix.

PERM is an INTEGER array of length N. This array need be set on entry only if ICNTL(5) = 1 (the default is ICNTL(5) = 0). If ICNTL(5) = 1, PERM must be set by the user to hold positive global priority values for the variables (see Section 4); this may be a permutation, in which case the variable for which PERM(I) = 1 is likely have a low index in the new ordering and the variable for which PERM(I) = N is likely to appear towards then end of the new ordering. In all cases, on exit, the new ordering is contained in PERM. If a

variable permutation is requested (JOB = 1 or 2), the new index for variable I is given by PERM(I) (I = 1,2, ..., N). If a row order is requested (JOB = 3), the order in which the rows should be presented to the row-by-row frontal solver is PERM(1), PERM(2), ..., PERM(N).

LIW is an INTEGER variable that must be set by the user to the length of the array IW. LIW must be at least 2+4*N and 2+8*N is always sufficient. If ICNTL(5) = 1 (the default is 0), a sufficient value is 2+7*N and if ICNTL(5) = 1 and ICNTL(6) = 2 (the default is 0), a sufficient value is 2+6*N. Note that if supervariables are used (ICNTL(4) = 0, which is the default), 2+2*N+max(2*N, 6*nsup) is always sufficient, where nsup is the number of supervariables (see INFO(2)). This argument is not altered. **Restriction:** LIW ≥ 2+4*N.

IW is an INTEGER array of length LIW that is used by the routine as workspace.

W is a REAL (DOUBLE PRECISION in the D version) array of length N that is used by the routine as workspace.

ICNTL is an INTEGER array of length 10 that must be set by the user to hold control parameters. This argument is not altered.

CNTL is a REAL (DOUBLE PRECISION in the D version) array of length 5 that must be set by the user to hold control parameters. This argument is not altered.

INFO is an INTEGER array of length 10 that need not be set by the user. On exit, INFO returns the following information:

INFO(1) is used as an error flag. If a call to MC61A/AD is successful, on exit INFO(1) has value 0. A negative value for INFO(1) is associated with a fatal error. If ICNTL(1) > 0, a self-explanatory message is, in each case, output on unit ICNTL(1). The negative values for INFO(1) are:

–1 – JOB is not equal to 1, 2, or 3, or N < 1, or LIRN < ICPTR(N+1)-1. Immediate return with input parameters unchanged.

–2 – LIRN is too small. INFO(6) is set to the minimum value which will suffice for LIRN. If ICNTL(3)=1, any out-of-range or duplicated variable indices will have been excluded from IRN and ICPTR. Otherwise, the input parameters are unchanged.

–3 – LIW is too small. INFO(3) is set to a value which will suffice for LIW.

–4 – ICNTL(3)=0 and one or more variable indices either lies outside the lower triangle of the matrix or is duplicated. Further information is contained in INFO(4) and INFO(5).

A positive value of INFO(1) is associated with a warning message. If ICNTL(2) > 0, a self-explanatory message is, in each case, output on unit ICNTL(2) and further information is contained in INFO(3) to INFO(6).

Note that if a fatal error is detected during a call to MC61A/AD, the information contained in INFO and RINFO will be incomplete.

INFO(2) holds, on successful exit, the total number of supervariables in the problem. If supervariables are not used (ICNTL(4) = 1), INFO(2) is set to N.

INFO(3) holds the amount of workspace used by the routine. If the user has provided insufficient workspace (INFO(1) = –3), INFO(3) is set to a value which will suffice for LIW.

INFO(4) holds the number of out-of-range indices in IRN.

INFO(5) holds the number of duplicate indices in IRN.

INFO(6) holds the minimum value which will suffice for LIRN.

INFO(7) to INFO(10) are currently not used but may be used in a later release of the code.

RINFO is a REAL (DOUBLE PRECISION in the D version) array of length 15 that need not be set by the user. If

JOB = 1 or 2, on successful exit RINFO returns the following information:

RINFO(1) holds the profile of the matrix **A**.

RINFO(2) holds the maximum wavefront of the matrix **A**.

RINFO(3) holds the semibandwidth of the matrix **A**.

RINFO(4) holds the root mean squared wavefront of the matrix **A**.

RINFO(5) holds the profile of the permuted matrix.

RINFO(6) holds the maximum wavefront of the permuted matrix.

RINFO(7) holds the semibandwidth of the permuted matrix.

RINFO(8) holds the root mean squared wavefront of the permuted matrix.

If JOB = 3, on successful exit RINFO returns the following information:

RINFO(1) and RINFO(2) hold the maximum row and column front sizes for the original row order 1, 2, ..., N.

RINFO(3) and RINFO(4) hold the root mean squared row and column front sizes for the original row order 1, 2, ..., N.

RINFO(5) and RINFO(6) hold the maximum row and column front sizes for the new row order PERM(1), PERM(2), ..., PERM(N).

RINFO(7) and RINFO(8) hold the root mean squared row and column front sizes for the new row order PERM(1), PERM(2), ..., PERM(N).

In addition, if JOB = 1 or 3, on successful exit, RINFO(9) and RINFO(10) hold the pair of weights that are used to give the final permutation (see ICNTL(6)).

RINFO(11) to RINFO(15) are currently not used but may be used in a later release of the code.

3 GENERAL INFORMATION

Use of common: None.

Other routines called directly: MC60A/AD, MC60B/BD, MC60C/CD, MC60D/DD, MC60E/ED, MC60F/FD, MC60G/GD.

Input/output: In the event of errors, diagnostic messages are printed. Stream ICNTL(1) is used for error messages and stream ICNTL(2) for warnings.

Restrictions: JOB = 1, 2, or 3, N ≥ 1, LIW ≥ 2+4*N.

4 METHOD

MC61 first calls MC60A to construct the pattern of the whole matrix **A** and to perform extensive checks on the data. If JOB = 1 or 2, MC60F computes the profile, the maximum wavefront, the semibandwidth, and the root mean squared wavefront of **A**, and if JOB = 3, MC60G computes the maximum row and column front sizes and the root mean squared row and column front sizes for the original row order.

If supervariables are being used (ICNTL(4) = 0), MC61 calls MC60B to look for sets of columns with identical patterns. A permutation is constructed that places the variables of each supervariable together. The pattern of **A** is replaced by that of the permuted matrix represented as supervariables (that is, by its condensed equivalent).

MC61 then calls MC60C to construct a supervariable (ICNTL(4) = 0) or variable (ICNTL(4) = 1) permutation that aims to reduce either the profile and wavefront of the matrix or the bandwidth. If JOB = 1 or 3, the priority function for supervariable s is

$$W_1 deg(s) + W_2 vglob(s),$$

where W_1 and W_2 are weights, $deg(s)$ is the number of supervariables that will enter the front if supervariable s

is chosen next, v is a normalizing factor, and $glob(s)$ is the positive global priority function (generated automatically or provided in PERM). If JOB = 2, the Reverse Cuthill McKee algorithm is used to reduce the bandwidth.

If JOB = 1 or 2, MC60F computes the profile, the maximum wavefront, the semibandwidth, and the root mean squared wavefront for the permuted matrix and, if supervariables are being used, MC60D constructs the permutation for the variables that corresponds to the permutation for the supervariables. If JOB = 3, MC60E uses the supervariable permutation to construct an ordering for the rows, as required by a row-by-row frontal solver, and MC60G computes the maximum row and column front sizes and the root mean squared row and column front sizes for the new row order.

Further details of the method are given in the MC60 specification document and by Reid and Scott (1998).

References

Reid, J.K. and J.A. Scott. (1998). Ordering symmetric sparse matrices for small profile and wavefront. Technical Report RAL-TR-98-016, Rutherford Appleton Laboratory.

5 EXAMPLE OF USE

The following program provides an example of the use of MC61. We wish to reduce the profile of a matrix with the following sparsity pattern.

$$\mathbf{A} = \begin{pmatrix} \times & \times & \times & \times & \times \\ \times & \times & & & \\ \times & & \times & & \\ \times & & & \times & \\ \times & & & & \times \end{pmatrix}.$$

The input data will be the lower triangle of **A** in column-wise format. Using the program:

```
C Code to illustrate use of MC61

      INTEGER LIRN,MAXN
      PARAMETER (LIRN=20, MAXN=5)
      INTEGER I,JOB,LIW,N,NZ
      INTEGER IRN(LIRN),ICPTR(MAXN+1),PERM(MAXN),IW(8*MAXN+2),
     +        ICNTL(10),INFO(10)
      REAL CNTL(5),RINFO(15),W(MAXN)

C Read in data
      READ (5,*) N,NZ
      READ (5,*) (IRN(I),I = 1,NZ)
      READ (5,*) (ICPTR(I),I = 1,N+1)

C Set control parameters
      CALL MC61I(ICNTL,CNTL)
C Prepare to call MC61A
      JOB = 1
      LIW = 8*N + 2
      CALL MC61A(JOB,N,LIRN,IRN,ICPTR,PERM,LIW,IW,W,ICNTL,CNTL,
     +           INFO,RINFO)

C Check for an error return
      IF (INFO(1).LT.0) THEN
         WRITE (6,*) ' Unexpected error return!'
         STOP
      END IF

C Write out the profile
      WRITE (6,'(/A,F6.0)') ' The profile initially  is ', RINFO(1)
      WRITE (6,'(/A,F6.0)') ' The profile after MC61 is ', RINFO(5)
      END
```

on the data

```
5  9
1  2  3  4  5  2  3  4  5
1  6  7  8  9  10
```

produces the output:

```
The profile initially  is   15.

The profile after MC61 is    9.
```