MULTIGRAPH
Users' Guide 2.1

Randolph E. Bank

Department of Mathematics
University of California at San Diego
La Jolla, California 92093-0112

July, 2017

This work was supported by the National Science Foundation
under awards DMS-1318480 and DMS-1345013.

Contents

1 Data Structures **1**
- 1.1 Overview. . . . 1
- 1.2 Matrix Data Structures. . . . 3
- 1.3 The *ka* Data Structure. . . . 5

2 Multigraph Routines **7**
- 2.1 Overview. . . . 7
- 2.2 Subroutine *mginit*. . . . 7
- 2.3 Subroutine *mgilu*. . . . 9
- 2.4 Subroutine *mg*. . . . 10
- 2.5 Subroutines *cycle*, *mtxmlt* and *perm*. . . . 11

3 Graphics **15**
- 3.1 Overview. . . . 15
- 3.2 Subroutine *gphplt*. . . . 15
- 3.3 Subroutine *mtxplt*. . . . 16

4 Test Driver **21**
- 4.1 Overview. . . . 21
- 4.2 Terminal Mode. . . . 21
- 4.3 Web Browser Mode. . . . 24
- 4.4 Batch Mode. . . . 29
- 4.5 Array Dimensions and Initialization. . . . 29
- 4.6 Matrix Files. . . . 30
- 4.7 Matrix Generators. . . . 30
- 4.8 Journal Files. . . . 31
- 4.9 Machine Dependent Routines. . . . 32
  - 4.9.1 Arithmetic Specification. . . . 32
  - 4.9.2 Timing Routine. . . . 32
  - 4.9.3 Graphics Interface. . . . 32

Bibliography **31**

Chapter 1

Data Structures

1.1 Overview.

The multigraph package can be used to solve large sparse linear systems of equations of the form

$$Ax = b. \tag{1.1}$$

In this chapter, we discuss the main data structures used in the package, and give a brief overview of its overall structure. See [3, 4] for algorithmic details and some numerical results.

We assume that the sparsity pattern of A is symmetric, although the numerical values need not be. We will begin by describing the basic two-level method for solving (1.1) Let B be an $n \times n$ nonsingular matrix, called the *smoother*, which gives rise to the basic iterative method used in the multigraph preconditioner. In our case, B is an approximate factorization of A, i.e.,

$$B = (L + D)D^{-1}(D + U) \approx P^t AP, \tag{1.2}$$

where L is (strict) lower triangular, U is (strict) upper triangular with the same sparsity pattern as L^t, D is diagonal, and P is a permutation matrix.

Given an initial guess x_0, m steps of the smoothing procedure produce iterates x_k, $1 \le k \le m$, given by

$$\begin{aligned} r_{k-1} &= P^t(b - Ax_{k-1}) \\ B\delta_{k-1} &= r_{k-1} \\ x_k &= x_{k-1} + P^t\delta_{k-1} \end{aligned} \tag{1.3}$$

The second component of the two-level preconditioner is the *coarse grid correction*. Here we assume that the matrix A can be partitioned as

$$\hat{P}A\hat{P}^t = \begin{pmatrix} A_{ff} & A_{fc} \\ A_{cf} & A_{cc} \end{pmatrix} \tag{1.4}$$

where the subscripts f and c denote *fine* and *coarse*, respectively. Similar to the smoother, the partition of A in fine and coarse blocks involves a permutation matrix $\hat{P}$. The $\hat{n} \times \hat{n}$ coarse grid matrix $\hat{A}$ is given by

$$\begin{aligned}\hat{A} &= \begin{pmatrix} V_{cf} & I_{cc} \end{pmatrix} \begin{pmatrix} A_{ff} & A_{fc} \\ A_{cf} & A_{cc} \end{pmatrix} \begin{pmatrix} W_{fc} \\ I_{cc} \end{pmatrix} \\ &= V_{cf} A_{ff} W_{fc} + V_{cf} A_{fc} + A_{cf} W_{fc} + A_{cc}. \end{aligned} \tag{1.5}$$

The matrices V_{cf} and W_{fc}^t are $\hat{n} \times (n - \hat{n})$ matrices, with identical sparsity patterns; thus $\hat{A}$ has a symmetric sparsity pattern. If $A^t = A$, we require $V_{cf} = W_{fc}^t$, so $\hat{A}^t = \hat{A}$.

Let

$$\hat{V} = \begin{pmatrix} V_{cf} & I_{cc} \end{pmatrix} \hat{P}, \qquad \hat{W} = \hat{P}^t \begin{pmatrix} W_{fc} \\ I_{cc} \end{pmatrix}. \tag{1.6}$$

In standard multigrid terminology, the matrices $\hat{V}$ and $\hat{W}$ are called *restriction* and *prolongation*, respectively. Given an approximate solution x_m to (1.1), the coarse grid correction produces an iterate x_{m+1} as follows.

$$\begin{aligned} \hat{r} &= \hat{V}(b - Ax_m) \\ \hat{A}\hat{\delta} &= \hat{r} \\ x_{m+1} &= x_m + \hat{W}\hat{\delta} \end{aligned} \tag{1.7}$$

As is typical of multilevel methods, we define the *Two-Level Preconditioner* M implicitly in terms of the smoother and coarse grid correction. A single cycle takes an initial guess x_0 to a final guess x_{2m+1} as follows:

Two-Level Preconditioner

i. x_k for $1 \le k \le m$ are defined using (1.3).

ii. x_{m+1} is defined using (1.7).

iii. x_k for $m + 2 \le k \le 2m + 1$ are defined using (1.3).

The generalization from two-level to multilevel consists of applying recursion to the solution of the equation $\hat{A}\hat{\delta} = \hat{r}$ in (1.7). Let ℓ denote the number of levels in the recursion. Let $\hat{M} \equiv \hat{M}(\ell)$ denote the preconditioner for $\hat{A}$; if $\ell = 2$ then $\hat{M} = \hat{A}$. Then (1.7) is generalized to:

$$\begin{aligned} \hat{r} &= \hat{V}(b - Ax_m) \\ \hat{M}\hat{\delta} &= \hat{r} \\ x_{m+1} &= x_m + \hat{W}\hat{\delta} \end{aligned} \tag{1.8}$$

The general ℓ level preconditioner M is then defined as follows:

ℓ-Level Preconditioner

i. if $\ell = 1$, $M = A$; i.e., solve (1.1) directly.

ii. if $\ell > 1$, then, starting from initial guess x_0, compute x_{2m+1} using (iii)-(v):

iii. x_k for $1 \le k \le m$ are defined using (1.3).

iv. x_{m+1} is defined by (1.8), using $p = 1$ or $p = 2$ iterations of the $\ell - 1$ level scheme for $\hat{A}\hat{\delta} = \hat{r}$ to define $\hat{M}$, and with initial guess $\hat{\delta}_0 = 0$.

v. x_k for $m + 2 \le k \le 2m + 1$ are defined using (1.3).

The case $p = 1$ corresponds to the symmetric *V-cycle*, while the case $p = 2$ corresponds to the symmetric *W-cycle*. We note that there are other variants of both the V-cycle and the W-cycle, as well as other types of multilevel cycling strategies [7]. However, in our code we restrict attention to just the symmetric V-cycle, with $m = 1$ presmoothing and postsmoothing iterations.

For the coarse mesh solution ($\ell = 1$), our procedure is somewhat non-traditional. Instead of direct solution of (1.1), we compute an approximate solution using one smoothing iteration.

If A is symmetric then so is M, and the ℓ-Level Preconditioner is used as a preconditioner for the composite step conjugate gradient method (CSCG). In the nonsymmetric case, the ℓ-level Preconditioner is used in conjunction with the composite step biconjugate gradient method (CSBCG). See [1] for details of these Krylov space methods.

1.2 Matrix Data Structures.

Let A be an $n \times n$ matrix with elements A_{ij}, and a symmetric sparsity structure; that is, both A_{ij} and A_{ji} are treated as nonzero elements (i.e. stored and processed) if $|A_{ij}| + |A_{ji}| > 0$. All diagonal entries A_{ii} are treated as nonzero regardless of their numerical values.

Our data structure is a modified and generalized version of the data structure introduced in the (symmetric) Yale Sparse Matrix Package [6]. It is a row-wise version of the data structure described in [2]. In our scheme, the nonzero entries of A are stored in a linear array a, and accessed through an integer array ja. Let η_i be the number of nonzeros in the strict upper triangular part of row i, and set $\eta = \sum_{i=1}^{n} \eta_i$. The array ja is of length $n+1+\eta$ and the array a is of length $n+1+\eta$ if $A^t = A$. If $A^t \neq A$, then the array a is of length $n + 1 + 2\eta$. The entries of $ja(i)$ $1 \le i \le n + 1$ are pointers defined as follows:

$$ja(1) = n + 2$$
$$ja(i + 1) = ja(i) + \eta_i, \quad 1 \le i \le n$$

The locations $ja(i)$ to $ja(i + 1) - 1$ contain the η_i column indices corresponding to the row i in the strictly upper triangular matrix.

In a similar manner, the array a is defined as follows:

$$\begin{aligned} a(i) &= A_{ii}, \quad 1 \le i \le n \\ a(n+1) &\quad \text{is arbitrary} \\ a(k) &= A_{ij}, \quad 1 \le i \le n, \quad j = ja(k), \quad ja(i) \le k \le ja(i+1) - 1 \end{aligned}$$

If $A^t \neq A$, then

$$a(k+\eta) = A_{ji}, \quad 1 \le i \le n, \quad j = ja(k), \quad ja(i) \le k \le ja(i+1) - 1$$

In words, the diagonal is stored first, followed by the strict upper triangle stored row-wise. If $A^t \neq A$, then this is followed by the strict lower triangle stored column-wise. Since A is structurally symmetric, the column indexes for the upper triangle are identical to the row indexes for the lower triangle, and hence need not be duplicated in storage.

As an example, let

$$A = \begin{pmatrix} A_{11} & A_{12} & A_{13} & 0 & 0 \\ A_{21} & A_{22} & 0 & A_{24} & 0 \\ A_{31} & 0 & A_{33} & A_{34} & A_{35} \\ 0 & A_{42} & A_{43} & A_{44} & 0 \\ 0 & 0 & A_{53} & 0 & A_{55} \end{pmatrix}$$

Then

	1	2	3	4	5	6	7	8	9	10	11
ja	7	9	10	12	12	12	2	3	4	4	5
a	A_{11}	A_{22}	A_{33}	A_{44}	A_{55}		A_{12}	A_{13}	A_{24}	A_{34}	A_{35}
	Diagonal						Upper Triangle				

	12	13	14	15	16
ja					
a	A_{21}	A_{31}	A_{42}	A_{43}	A_{53}
	Lower Triangle				

If desired, the user can specify a block structure for the matrix A. This block structure is used only in the coarsening phase of the algorithm (i.e. in creating $\hat{V}$ and $\hat{W}$). If the matrix has *nblock* blocks, the user provides and integer array ib of length $nblock + 1$, defined as follows: Let ξ_i be the order of block i, for $1 \le i \le nblock$. Then

$$\begin{aligned} ib(1) &= 1 \\ ib(i+1) &= ib(i) + \xi_i, \quad 1 \le i \le nblock. \end{aligned}$$

For the case of just one block, one should set

$$\begin{aligned} ib(1) &= 1 \\ ib(2) &= n + 1 \end{aligned}$$

The data structure for storing $B = (L + D)D^{-1}(D + U)$ is quite analogous to that for A. It consists of two arrays, ju and u, corresponding to ja and a, respectively. The first $n+1$ entries of ju are pointers as in ja, while entries $ju(i)$ to $ju(i+1)-1$ contain column indices of the nonzeros of row i in of U. In the u array, the diagonal entries of D are stored in the first n entries. Entry $n+1$ is arbitrary. Next, the nonzero entries of U are stored, in correspondence to the column indices in ju. If $L^t \neq U$, the nonzero entries of L follow, stored column-wise.

The data structure we use for the $n \times \hat{n}$ matrix $\hat{W}$ and the $\hat{n} \times n$ matrix $\hat{V}$ are similar. It consists of an integer array jv and a real array v. The nonzero entries of $\hat{W}$ are stored row-wise, including the rows of the block I_{cc}. As usual, the first $n+1$ entries of jv are pointers; entries $jv(i)$ to $jv(i+1)-1$ contain column indices for row i of $\hat{W}$. In the v array, the nonzero entries of $\hat{W}$ are stored row-wise in correspondence with jv but shifted by $n+1$ since there is no diagonal part. If $\hat{V}^t \neq \hat{W}$, this is followed by the nonzeros of $\hat{V}$ stored column-wise.

1.3 The ka Data Structure.

To avoid excessive clutter in the calling sequences, all of the relevant matrices for all of the levels are stored in just two arrays, an integer array ja and a real array a. In order to keep track of the internal structure of these arrays, a matrix of pointers, ka, is created in subroutine *mginit* and used in subroutine *mg*. A casual user need not be concerned with this array (other than allocating storage for it), but it is available to the user should access to the various matrices generated by the multigraph method be desired. ka is a $10 \times (lvl + 1)$ integer array, where $lvl \leq maxlvl$ is the number levels employed by the method. Column i corresponds to variables associated mainly with level $lvl + 1 - i$; that is, the first column is associated with the finest level, the second column with the next finest level, and so on.

i	$ka(i, *)$
1	n, the order of the matrix
2	*nptr*, pointer for multilevel vector arrays
3	*japtr* pointer for the integer data structure ja
4	*iaptr* pointer for the real data structure a
5	*juptr* pointer for the integer data structure ju
6	*iuptr* pointer for the real data structure u
7	*jvptr* pointer for the integer data structure jv
8	*ivptr* pointer for the real data structure v
9	*iqptr* pointer for the inverse permutation for P
10	*ibptr* block labels, computed from the *ib* array

Table 1.1. *The ka array.*

Chapter 2

Multigraph Routines

2.1 Overview.

The multigraph implementation consists of four main routines, *mginit*, *mgilu*, *mg*, and *cycle*. Subroutine *mginit* is the initialization routine that creates the levels and their associated data structures. Subroutine *mgilu* performs a subset of the operations of *mginit*, and can be used when one solves a sequence of linear systems with a family of related matrices (e.g. in a Newton iteration). Subroutine *mgilu* computes new values for all of the real variables (a, u and v), while retaining the integer data structures produced by *mginit*; this significantly reduces the initialization time. Subroutine *mg* solves (1.1) using either the composite set conjugate gradient or composite step biconjugate gradient method. Subroutine *cycle* is the V-cycle preconditioner called by *mg*. It is documented separately, as it can be called directly as the preconditioner in other iterative solvers. For such a situation, we also provide subroutines *mtxmlt* and *perm* for matrix multiplication and reordering, respectively. Two other routines, *gphplt* and *mtxplt*, are visualization tools that are discussed in Chapter 3.

This version of the multigraph package is written in FORTRAN90. There is only one version of the source code. The precision of the arithmetic is governed through the module *mthdef* where the precision of integer and floating point numbers can be specified through the parameters *iknd* and *rknd*, respectively. Module *mthdef* is included in every subroutine and function in the package, and thus represents a global specification of precision.

2.2 Subroutine mginit.

mginit is called using the statement:

call mginit(n, ispd, nblock, ib, maxja, ja, maxa, a, ncfact, maxlvl,
maxfil, ka, lvl, dtol, method, iflag)

A discussion of these parameters follows.

- n is an integer specifying the order of the system of equations.
- $ispd$ is an integer specifying the symmetry of the matrix. $ispd = 1$ indicates that symmetric storage is used; $ispd = 0$ indicates that nonsymmetric storage is used.
- $nblock$ is an integer specifying the number of blocks in the matrix (see Section 1.2).
- ib is an integer array of size $nblock + 1$ containing the block structure, as defined in Section 1.2.
- $maxja$ is an integer specifying the size of the array ja.
- ja is an array of integers, containing all the integer data structures for all levels defined in Chapter 1. On input, the head of ja should contain the integer data structure corresponding the the linear system (1.1) to be solved.
- $maxa$ is an integer specifying the size of the array a. A good (but inexact) guide is to choose $maxa \sim maxja$ when $ispd = 1$ and $maxa \sim 2\,maxja$ when $ispd = 0$.
- a is an array of reals, containing all the real data structures for all levels defined in Chapter 1. On input, the head of a should contain the real data structure corresponding the the linear system (1.1) to be solved.
- $ncfact$ is an integer specifying the coarsening factor. If the matrix at a given level is of order n, then the matrix for the next coarser level will be of order $\hat{n} \approx n/ncfact$. We require $ncfact \geq 2$.
- $maxlvl$ is an integer specifying the maximum number of levels to be used.
- $maxfil$ is an integer specifying the maximum storage allowed for certain matrices. In particular, the ja and ju arrays for a system of order n_i will have maximum size $n_i + 1 + n_i\,maxfil$. Note that $maxfil$ controls the average number of nonzeros per row, but NOT necessarily the fill-in in any particular row.
- ka is a $10 \times (lvl+1)$ integer array, which on output contains pointers as defined in Section 1.3.
- lvl is an integer, which on output contains the number of levels actually generated by $mginit$. In particular $lvl \leq maxlvl$.
- $dtol$ is a nonnegative real, specifying the drop tolerance for the ILU factorizations.

- *method* is an integer specifying the smoother for the multigraph algorithm. $method = 0$ is the default ILU with drop tolerance; $method = 1$ is $ILU(0)$ ($ja \equiv ju$ at all levels); $method = 2$ is symmetric Gauss-Seidel ($ja \equiv ju$ and $a \equiv u$ at all levels). $method = 1$ and $method = 2$ are provided mainly as a baseline to compare with $method = 0$; however, for certain problems they can provide comparable performance using less time and space for the initialization, and therefore are independently useful.

- $iflag$ is an integer that on output contains the error flag. $iflag = 0$ signifies no error; $iflag = 20$ signifies insufficient storage. Although this could refer to $maxja$, $maxa$, or $lenz$, the typical failure is for $lenz$.

2.3 Subroutine mgilu.

Subroutine *mgilu* performs a subset of the computations of *mginit*. In particular, for a related family of matrices, one can save the level and fill-in structures (essentially the contents of the ja array) and simply compute new numerical values for matrix elements (the a array). One calls *mginit* for the first member of the family of matrices and then *mgilu* for the remainder. For example, in a Newton iteration, one might expect the changes in the Jacobian matrices to be sufficiently small that the level and fill-in structures could be used for all (or perhaps just several) Newton steps. Thus one would call *mginit* once to initialize the arrays and compute the first set of matrices, and then call *mgilu* for all other matrices, which would then reuse the level and fill-in structure from the call to *mginit*.

mgilu is called using the statement:

call mgilu(ja, a, lvl, ka)

A discussion of these parameters follows.

- ja is an array of integers, containing all the integer data structures for all levels defined in Chapter 1. This should be the output from the original call to *mginit*.

- a is an array of reals, containing all the real data structures for all levels defined in Chapter 1. On input, the head of a should contain the real data structure corresponding the the linear system (1.1) to be solved.

- lvl is an integer, which contains the number of levels. This should be the output from the original call to *mginit*.

- ka is a $10 \times (lvl+1)$ integer array, which contains pointers as defined in Section 1.3. This should be the output from the original call to *mginit*.

It is important to note that *mginit* reorders the original matrix stored in the ja and a data structures. *mgilu* assumes that the new matrix provided in a corresponds to this reordering. The inverse permutation array for the ordering can

be found using the pointer $iqptr = ka(9, 1)$. If $p(i)$ is the permutation, and $q(i)$ the inverse permutation, then $p(q(i)) = i$, $1 \leq i \leq n$.

As a convenience, we provide subroutine *jamap*0, which takes a pair (i, j) in the original ordering and provides pointers to the locations of A_{ij} and A_{ji} in the reordered data structures. *jamap*0 is called using the statement:

call jamap0(i, j, n, ispd, ij, ji, ja)

A discussion of these parameters follows.

- i and j are the indices for the desired matrix element, given in the original ordering.
- n is an integer specifying the order of the system of equations.
- *ispd* is an integer specifying the symmetry of the matrix. $ispd = 1$ indicates that symmetric storage is used; $ispd = 0$ indicates that nonsymmetric storage is used.
- On output, ij and ji are pointers to the a array where matrix entries A_{ij} and A_{ji}, respectively, are stored. $ij = ji$ if $i = j$ or $ispd = 1$, and $ij = ji = 0$ if entry (i, j) is not present in the data structure.
- *ja* is an array of integers, containing all the integer data structures for all levels defined in Chapter 1. This should be the output from the original call to *mginit*.

2.4 Subroutine mg.

Subroutine *mg* solves the linear system (1.1) using the output from *mginit* (or *mgilu*). In the nonsymmetric case, subroutine *mg* can also solve problems of the form

$$A^t x = b. \tag{2.1}$$

mg is called using the statement:

call mg(ispd, lvl, mxcg, eps1, ja, a, dr, br, ka, relerr, iflag, hist)

A discussion of these parameters follows.

- *ispd* is an integer specifying the symmetry of the matrix. $ispd = 1$ indicates that symmetric storage is used; $ispd = 0$ indicates that nonsymmetric storage is used. $ispd = -1$ indicates that nonsymmetric storage is used and one should solve (2.1).
- *lvl* is an integer specifying the number of levels. This should be the output from the call to *mginit*.
- *mxcg* is an integer specifying the maximum number of CSCG iterations ($ispd = 1$) or CSBCG iterations ($ispd \neq 1$).

- *eps*1 it the convergence tolerance. The iteration terminates when the residual norm is reduced by a factor of *eps*1 or when *mxcg* iterations is achieved, whichever occurs first.
- *ja* is an array of integers, containing all the integer data structures for all levels defined in Chapter 1. This should be the output from the call to *mginit*.
- *a* is an array of reals, containing all the real data structures for all levels defined in Chapter 1. This should be the output from a call to *mginit* or *mgilu*.
- *dr* is a real array of size n, which on output contains the solution of the linear system.
- *br* is a real array of size n, which on input contains the right hand side of the linear system.
- *ka* is a $10 \times (lvl+1)$ integer array, which contains pointers as defined in Section 1.3. This should be the output from the call to *mginit*.
- *relerr* is a real number which on output specifies the ratio of the norms of initial and final residuals.
- $iflag$ is an integer that on output contains the error flag. $iflag = 0$ signifies no error; $iflag = 12$ indicates that the error tolerance *eps*1 was not reached in *mxcg* iterations, but the iteration appeared to be converging. $iflag = -12$ indicates that the iteration appeared to diverge. *hist* is a real array of size 22, which collects data used by the graphics routine *gphplt*.

2.5 Subroutines cycle, mtxmlt and perm.

Subroutine *cycle* implements the V-cycle preconditioner, and is called as needed by *mg*. It is documented separately here, as it can be used as a preconditioner in other preconditioned iterative methods. *cycle* is called using the statement:

call cycle(ispd, lvl, ja, a, x, b, ka)

A discussion of these parameters follows.

- *ispd* is an integer specifying the symmetry of the matrix. $ispd = 1$ indicates that symmetric storage is used; $ispd = 0$ indicates that nonsymmetric storage is used. $ispd = -1$ indicates that nonsymmetric storage is used and one should solve (2.1).
- *lvl* is an integer specifying the number of levels. This should be the output from the call to *mginit*.
- *ja* is an array of integers, containing all the integer data structures for all levels defined in Chapter 1. This should be the output from the call to *mginit*.

- a is an array of reals, containing all the real data structures for all levels defined in Chapter 1. This should be the output from a call to *mginit* or *mgilu*.
- x is a real array of size n, which on output contains the approximate solution of the linear system.
- b is a real array of size n, which on input contains the right hand side of the linear system.
- ka is a $10 \times (lvl+1)$ integer array, which contains pointers as defined in Section 1.3. This should be the output from the call to *mginit*.

Subroutine *mtxmlt* computes $b = Ax$ or $b = A^t x$. It is a companion routine to *cycle* for use in a preconditioned iterative method. *mtxmlt* is called using the statement:

call mtxmlt(n, ja, a, x, b, ispd)

A discussion of these parameters follows.

- n is an integer specifying the order of the system of equations.
- ja is an array of integers, containing all the integer data structures for all levels defined in Chapter 1. This should be the output from the call to *mginit*.
- a is an array of reals, containing all the real data structures for all levels defined in Chapter 1. This should be the output from a call to *mginit* or *mgilu*.
- x is a real array of size n, which on contains the input vector.
- b is a real array of size n, which on output contains Ax or $A^t x$.
- $ispd$ is an integer specifying the symmetry of the matrix. $ispd = 1$ indicates that symmetric storage is used; $ispd = 0$ indicates that nonsymmetric storage is used. In both cases $b = Ax$ is computed. $ispd = -1$ indicates that nonsymmetric storage is used and $b = A^t x$ is computed.

Both *cycle* and *mtxmlt* assume that all vectors are ordered according the minimum degree ordering computed in *mginit*. If the input and output are provided in the original ordering, then subroutine *perm* should be called as necessary to reorder the data.

perm is called using the statement:

call perm(n, x, ja, isw)

A discussion of these parameters follows.

- n is an integer specifying the order of the system of equations.

- x is a real array of size n, which contains the vector to be reordered.
- ja is an array of integers, containing all the integer data structures for all levels defined in Chapter 1. This should be the output from the call to $mginit$.
- isw is an integer switch. If $isw = 1$, the input is assumed to be in the original order, and the output is reordered using the order generated in $mginit$. If $isw = -1$, the input is assumed to be ordered using the order provided by $mginit$ and the output is restored to the original order.

Chapter 3
Graphics

3.1 Overview.

The graphics tools associated with the multigraph package consist of subroutines *gphplt* and *mtxplt*. These routines are written in self-contained, portable FORTRAN, addressing the graphics output device through subroutines *pline*, $pfill$, $pframe$ and *pltutl*. The specifications for these routines are given in Section 4.9.3.

Subroutine *gphplt* displays various graphs and charts containing timings, convergence histories, and other items of interest. Subroutine *mtxplt* displays sparse matrices associated with the multigraph solver.

3.2 Subroutine gphplt.

gphplt is called using the statement

call gphplt(ip, rp, sp, hist, ka, time)

Subroutine *gphplt* uses three parameters specified in the *ip* array and one parameter specified in the *sp* array.

- $igrsw$ is an integer switch specifying the graphs to be drawn. The possibilities are given in Table 3.1,
- *idevce* is an integer switch specifying the graphics output device.
- *gpane* is an integer switch specifying *ipane* for the web broswer interface.
- *mxcolr* is an integer specifying the number of colors available; we assume $mxcolr \geq 2$.
- *gtitle* is an string specifying the title for the graph.
- $grfile$ is an string specifying the graphics output file for XPM and PostScript graphics.

igrsw	displayed graph
0	convergence history
1	storage profile
-1	timings
2	*ip* array
-2	*rp* array
3	*ka* array
-3	*sp* array

Table 3.1. *The values of igrsw.*

The case $igrsw = 0$ is probably the most useful. In the large frame, a convergence history of the multigraph iteration is displayed; iteration number appears on the x-axis, and $\log(relerr)$ appears on the y-axis. In one of the smaller frames, times for *mginit* and *mg* are displayed in a pie chart. In the other, storage statistics for various matrices are displayed; $\log(n)$ for each level appears on the x-axis, and the average number of nonzeros in *ja*, *ju* and *jv* for each level are displayed in different colors on the y-axis. The cases $igrsw = \pm 1$ are permutations of the three frames.

The case $igrsw = 2$ displays the *ip* array, an integer array of size 100 containing global parameters used by the test driver program. The case $igrsw = -2$ displays the *rp* array, a real array of size 100 containing global parameters used by the test driver program. The case $igrsw = -3$ displays the *sp* array, a *character*80* array of size 100 containing global parameters used by the test driver program. Finally, $igrsw = 3$ displays the sizes of all major arrays on all levels.

The remaining arguments are summarized by:

- *hist* is a real array of size 22, which contains the convergence history. It is the output from subroutine *mg*.
- *ka* is a $10 \times (lvl+1)$ integer array, which contains pointers as defined in Section 1.3. This should be the output from the call to *mginit*.
- *time* is a real array of size 2, containing the execution times of *mginit* and *mg*.

3.3 Subroutine mtxplt.

Subroutine *mtxplt* displays the sparsity structure of the stiffness matrix A, the LDU factors from the ILU, or the error matrix E associated with an approximate factorization. *mtxplt* is called using the statement

call mtxplt(ip, rp, sp, ja, a, ka)

Subroutine *mtxplt* uses several parameters specified in the *ip* and *rp* arrays and one parameter specified in the *sp* array.

- *imtxsw* specifies the matrix to be displayed, as summarized in Table 3.2. If $imtxsw > 0$, the magnitude of matrix elements is displayed; if $imtxsw < 0$, the (signed) value is displayed.
- *idevce* is an integer switch specifying the graphics output device.
- *mpane* is an integer switch specifying *ipane* for the web broswer interface.
- *mxcolr* is an integer specifying the number of colors available; we assume $mxcolr \geq 2$.
- *iscale* in an integer that specifies the scaling to be used for the cases $imtxsw = \pm 2, \pm 4, \pm 6$ as summarized in Table 3.2.
- *lines* is an integer that specifies the line drawing option, as summarized in Table 3.2.
- *numbrs* in an integer that specifies numbering options, as summarized in Table 3.2.
- (mx, my, mz) are three integers specifying the viewing perspective.
- *ncon* is an integer specifying the number of colors in the cases cases $imtxsw = \pm 2, \pm 4, \pm 6$.
- *level* is an integer, $1 \leq level \leq lvl$, specifying the level of the matrix to be displayed. If $level > lvl$ or $level < 1$, then *lvl* is used.
- $(smin, smax)$ are real numbers that optionally specify lower and upper bounds for the color range for the cases $imtxsw = \pm 2, \pm 4, \pm 6$. Matrix elements with values falling outside the given range are colored white.
- *rmag* is a real number specifying the magnification factor.
- $(cenx, ceny)$ are real numbers that specify the center of the picture when $rmag > 1$.
- *mtitle* is an string specifying the title for the graph.
- *grfile* is an string specifying the graphics output file for XPM and PostScript graphics.

The remaining arguments are summarized by:

- *ja* is an array of integers, containing all the integer data structures for all levels defined in Chapter 1. This should be the output from the call to *mginit*.
- *a* is an array of reals, containing all the real data structures for all levels defined in Chapter 1. This should be the output from a call to *mginit* or *mgilu*.

imtxsw	displayed matrix
±1	LDU colored by element type
±2	LDU colored by element size
±3	A colored by element type
±4	A colored by element size
±5	E colored by element type
±6	E colored by element size

iscale	scale
0	linear
1	logarithmic
2	$\sinh^{-1}$

lines	line drawing option
0	no lines
-2	matrix element boundaries

numbrs	labeling option
0	no labels
-1	matrix element values
-2	matrix element locations

Table 3.2. *The values of switches.*

- *ka* is a $10 \times (lvl+1)$ integer array, which contains pointers as defined in Section 1.3. This should be the output from the call to *mginit*.

Chapter 4

Test Driver

4.1 Overview.

Program *atest* is the test driver used in the development and testing of the multigraph solver. *atest* is a flexible program in that it accepts simple command strings directing it to call subroutines or perform other tasks. It is not limited to a fixed sequence of tasks on a particular run; any routine can be called as often as desired, with certain parameters reset for each call at the discretion of the user.

The program *atest* can operate in three modes, governed by the switch *mode*. If $mode = -1$, *atest* runs as an interactive program, accepting commands from the user via a terminal window. If $mode = 0$, *atest* runs interactively, accepting commands from the user via a web browser interface. The communication link between *atest* and the web browser is a socket, so that the client web browser can be running on a different device from the server running *atest*. Finally, if $mode = 1$, *atest* runs as a batch program, reading commands from a journal file and sending all output to appropriate output files.

A common command syntax is used for all three modes. This is described first for the case $mode = -1$ in Section 4.2. The extensions used in the web browser interface are described in Section 4.3.

Several files are written by *atest*. The file $bfile$ contains a complete record of all commands and printed output produced during the session. The file $jwfile$ contains a record of all commands read and processed during the session, formatted as a journal file. See Section 4.8 for a discussion of journal files. *atest* sets the default values $bfile = output.out$ and $jwfile = journl.jnl$. *atest* also creates a temporary file $jtfile = jnltmp.jnl$ which it uses in connection with journal files.

4.2 Terminal Mode.

In terminal mode, commands are entered from a terminal window in character strings of 80 characters, counting blanks. The syntax of a command can take several forms, but the root command is always a single letter. The commands that

are currently recognized by *atest* are summarized in Table 4.1.

Command	Action
f	call *mginit*
s	call *mg*
g	call *gphplt*
m	call *mtxplt*
l	create a linear system
r	read data set from a file
j	read journal file
q	quit

Table 4.1. *Available commands for atest.*

The terminal window prompt is the string *command:*. At this prompt, one can enter a command string (e.g., *s*), reset parameters as described below, or enter a blank line to see a list of the available commands. In this latter case the terminal window will appear as follows.

```
command:
factor f    solve  s    gphplt g    mtxplt m
linsys l    read   r    journl j    quit   q

command:
```

A syntax error in a given command string causes the entire string to be ignored. *atest* will display the string *command error* and present the command prompt for a new input string.

The most simple commands are just single lower case letters as shown in Table 4.1. However, associated with most commands are various parameters which can be reset before calling the given routine. To see a listing of the parameters associated with a given command and their current values, without executing the command itself, enter the command in upper case at the command prompt. For example, the command *M* will display the parameters which can be interactively reset in connection with *mtxplt*.

```
command:M
imtxsw i  2         iscale s  0         lines  l  0         numbrs n  0
mdevce d  3         mx     mx 1         my     my 1         mz     mz 1
ncon   c  11        level  l  0         mxcolr mc 100       smin   sn 0.0
smax   sx 0.0       rmag   m  1.0       cenx   cx 0.5       ceny   cy 0.5
mtitle t  "mtxplt"
grfile f  figxxx.ext

command:
```

These are eleven integer, five real, and two string parameters affecting subroutine *mtxplt* which can be interactively reset by the user. To the right of each

parameter is a one- or two-letter alias (to avoid typing long names), followed by the current value.

To reset some parameters associated with a command c (c = *s, f, g,* etc.), without invoking the command itself, one can type a string of the form

```
command:C name1=value1, name2=value2, ... , namek=valuek
```

Note that the root command appears in upper case. The *namek* refer to variable names or their aliases, and *valuek* refer to integer, real, or string values. Several parameters can be reset, with different entries separated by commas. Values for integer parameters should be integers, while values for real parameters can be specified using integer, fixed point, or exponential notation. There are three types of string parameters: *short*, *file*, and *long*. Short and file strings contain no blank characters, or special characters used by *atest* ("=,) and hence can be entered directly. Long strings, such a titles for graphics output, could have blanks and other reserved characters and must appear within double quotes. Long string parameters can contain any printable ASCII characters (other than double quotes). Blank spaces are ignored everywhere but within the value field of a long string parameter. A syntax error in the input line (e.g., a misspelled variable name) causes the entire command to be ignored and no variables to be reset. *atest* will respond *command error* and then ask for the next command. For example, here we reset $iscale = 1$, $ncon = 20$, $cenx = .3$, $rmag = 10$, and $mtitle =$ *A new title for mtxplt.* Subroutine *mtxplt* is not called, but the parameters are updated and redisplayed as

```
command:M s=1, ncon=20, cenx=.3, rmag=1.e1, t="A new title for mtxplt"
imtxsw i  2         iscale s  1         lines  l  0         numbrs n  0
mdevce d  3         mx     mx 1         my     my 1         mz     mz 1
ncon   c  20        level  l  0         mxcolr mc 100       smin   sn 0.0
smax   sx 0.0       rmag   m  10.0      cenx   cx 0.3       ceny   cy 0.5
mtitle t  "A new title for mtxplt"
grfile f  figxxx.ext

command:
```

One can reset some parameters for a given command c, and then invoke the command itself, using a string of the form

```
command:c name1=value1, name2=value2, ... , namek=valuek
```

Note that the only difference is that the root command now appears in lower case rather than upper case. Thus

```
command:m s=1, ncon=20, cenx=.3, rmag=1.e1, t="A new title for mtxplt"
```

resets the indicated parameters as in the previous example. However, instead of displaying the updated values, subroutine *mtxplt* is called.

Finally, the graphics commands (g and m) have a short form allowing one crucial parameter (*igrsw* and *imtxsw*, respectively) to be reset without typing even the alias. For example,

```
command:g0
```

is the short form for

```
command:g igrsw=0
```

The short and long forms of these commands cannot be mixed. Thus

```
command:g0, gdevce=1
```

is not valid.

4.3 Web Browser Mode.

When *mode* = 0, *atest* connects to a web browser interface via socket *webprt*. The *webgui* browser interface was written by Chris Deotte. It is designed as a graphical user interface for any C or Fortran package. Complete documentation is provided in [5]; here we will briefly discuss its use in here. When an *atest* application is launched with *mode* = 0, it displays a message of the form

```
webgui: Listening on port 15000 for web browser...
```

In this example *webprt* = 15000. To establish a connection with a web browser running on the same computer as the *atest* application, type

```
http://localhost:15000
```

in the web browser's address bar. If the web browser is running on a different client device, replace *localhost* with the IP address or the hostname of the computer running the *atest* application. Once the connection is established, the *webgui* interface should appear in the the web browser.

The functional capabilities of the web browser interface are the same as those for terminal window mode, but the possibilities for data entry are more varied. An example of the web browser interface interface appears in Figure 4.1. The web browser display has four main elements. There are three graphics panes, and in the upper left is the main control panel. The graphics interface will be described in more detail in Section 4.9.3. Here we discuss the main control panel.

The control panel contains two elements. The upper portion of the display contains *command buttons*. The bottom portion of the display is the *history window*. The command buttons stand in one to one correspondence with the basic *atest* command set shown in Table 4.1. To the right of each main command button is a smaller button labeled with a plus sign. Clicking on a command button is equivalent to the typed lower case version of the given command. For example, clicking on the button for the *mtxplt* command causes subroutine *mtxplt* to be called as in the command *m*. Clicking on the plus button to the right of any command is equivalent to the typed upper-case version of that command. In this case a drop down menu appears containing a display of all the parameters that can

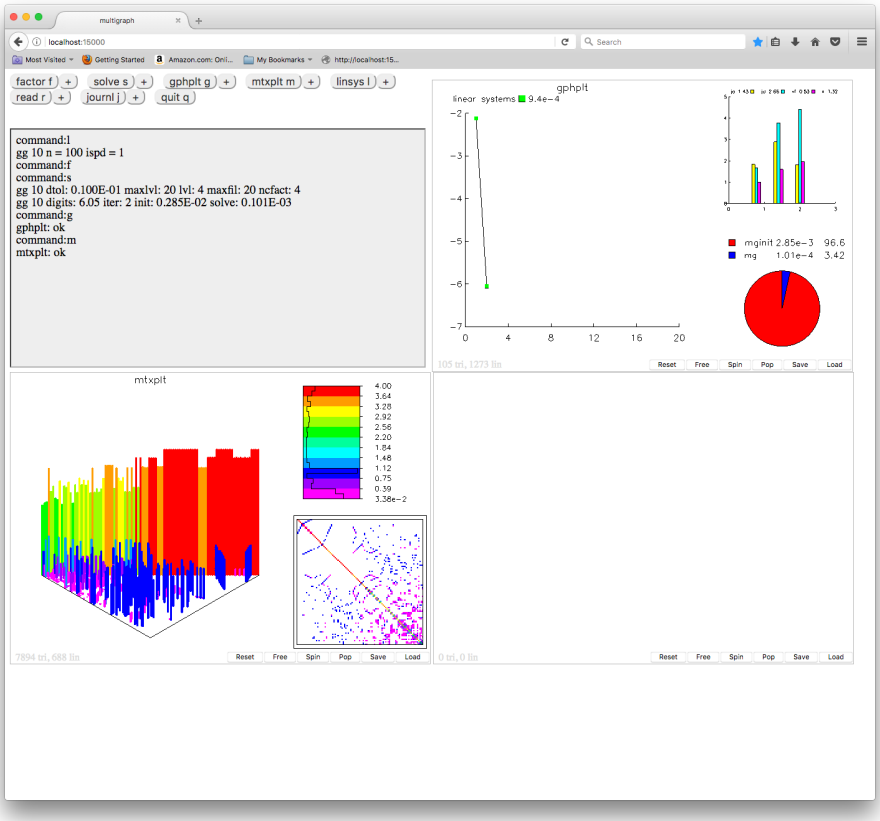

Figure 4.1. *The web browser interface.*

be modified by that command. For example, clicking on the plus button for the *mtxplt* command causes the parameters for the *mtxplt* command to be displayed as in the command M. This is shown is figure 4.2.

The parameters associated with a given command are displayed in the reset window in a format similar to terminal mode. However, each parameter value is displayed in one line text-editing window, and can be reset by typing in the new value. Some parameter names (e.g., *imtxsw* in Figure 4.2) appear in a button. If the parameter is a switch with several options, clicking on the name button causes the drop down menu to expand and display a suite of radio buttons indicating the various options associated with the parameter. Clicking on the desired option causes the parameter to be reset to the corresponding value. The radio buttons associated with the parameter *imtxsw* appear in Figure 4.3.

On the other hand, if the parameter is a file name, clicking on the name button causes the drop down menu to display a file selection widget. The file-selection widget for the $rwfile$ associated with the *read* command is shown in Figure 4.4.

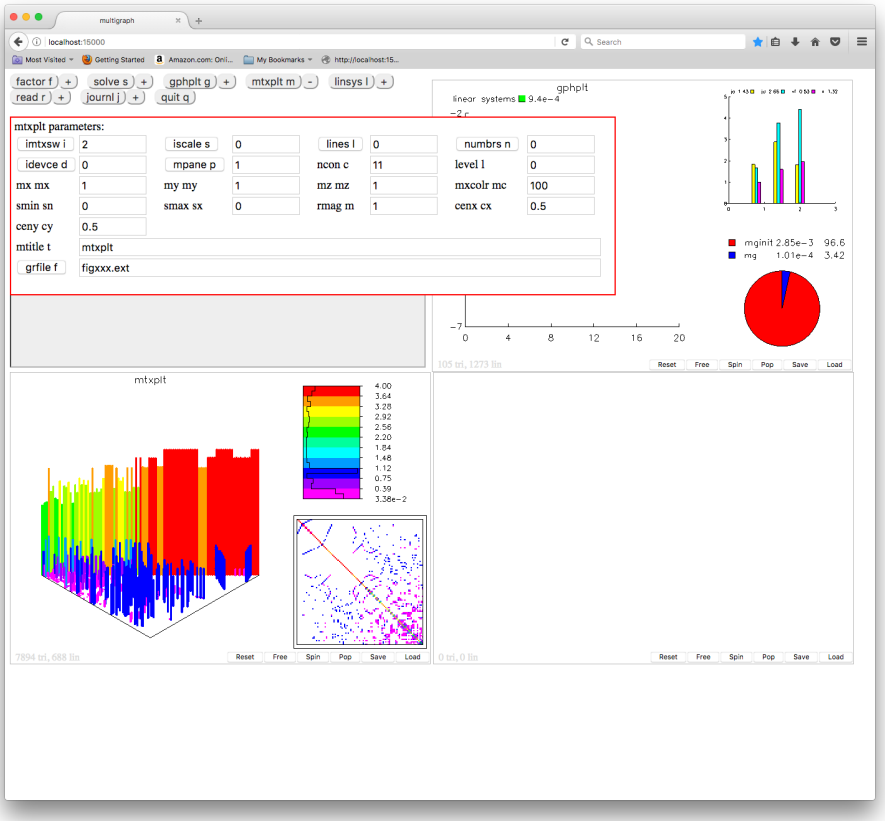

Figure 4.2. *Drop down menu for resetting parameters for mtxplt.*

The history window displays the contents of the output file, *bfile*, as it is created. When executing a journal file in web browser mode, if a graphics command is executed, depending on the graphics device selected, *atest* can pause after the picture is drawn, and create a small popup *click to continue* button. In this case, *atest* waits until the user dismisses the *click to continue* popup before continuing to execute the journal file. This allows time for the user to view the picture before processing the next command in the journal file.

The *webgui* interface has several additional features that are useful in certain situations. These appear in Table 4.2. To use these features one presses the OPTION key (ALT key on Windows machines) while also pressing the indicated key. When OPTION + C are pressed, the command buttons disappear and are replaced with a command line where commands may be entered as in terminal mode. Facsimiles of the displays created in terminal mode appear in the history window. Pressing OPTION + C again displays the command buttons.

OPTION + W changes the display from its default 2×2 layout to a single

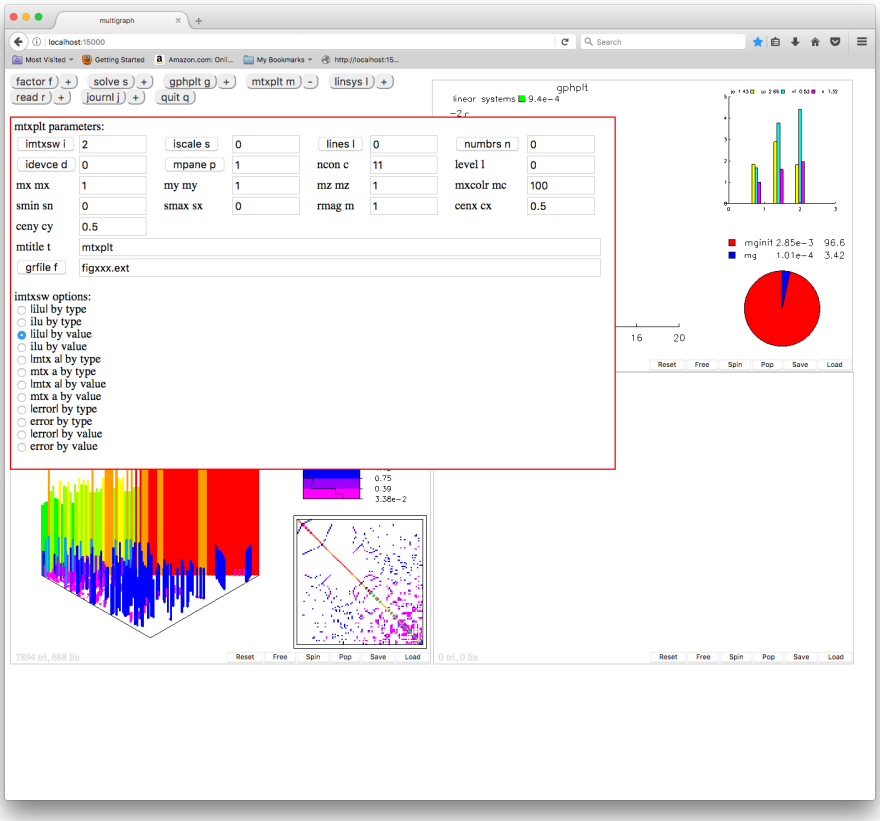

Figure 4.3. *Radio buttons for the parameter imtxsw.*

column 4×1 layout, with the control panel at the top, followed by the three graphics panes. Pressing OPTION + W a second time returns to the 2×2 display.

OPTION + F toggles the firewall on and off. When the firewall is on, *atest* displays the message

```
webgui: Only accepting ip address = x.x.x.x
```

where x.x.x.x is the IP address of the client device running the web browser. The *webgui* interface displays the message

```
FIREWALL  ON: Only your ip address (x.x.x.x) can access webgui.
```

at the bottom left of the main display. When the firewall is turned off, *atest* displays the message

```
webgui: Accepting all ip addresses.
```

while the message in the *webgui* interface disappears.

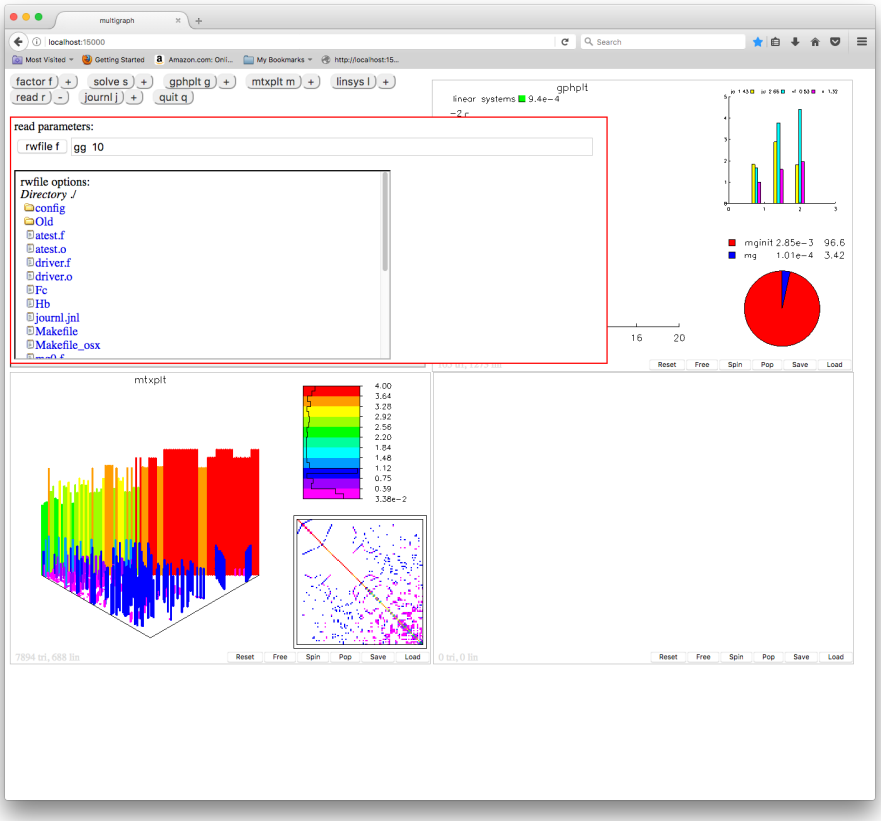

Figure 4.4. *File sector for the parameter rwfile.*

OPTION + E toggles endian flip on and off. Only one feature in the web browser interface (displaying 3D graphics objects) depends on the server and client machines having the same endianness. By default, the client assumes that client and server have the same endianness; if this is not the case, then OPTION + E flips the endianness as needed on the data it receives, and the *webgui* interface displays the message

```
ENDIAN FLIP: Client receiving flipped endianness of server.
```

The OPTION + I and OPTION + SAVE cases will be discussed in Section 4.9.3.

Finally, we remark that *webgui* uses the same database of Fortran character strings as the terminal window interface to define its displays, and returns command strings of the same type described in the terminal windows interface. Both the web browser interface and the terminal window interface are quite generic, in that neither contains direct links to any of the main routines in the package. Thus changes in the behavior of routines comprising the package have no impact on the interface

Key	Feature
OPTION + C	toggles between command buttons and command text field
OPTION + W	toggles between 2×2 and 4×1 layout
OPTION + F	toggles firewall on and off
OPTION + E	toggles endian flip on and off
OPTION + I	toggles displaying rotation, pan and zoom information
OPTION + SAVE	saves 3D objects as text instead of binary

Table 4.2. *Special Features of webgui.*

routines and at most modest impact on the database of character strings that define the displays.

4.4 Batch Mode.

When $mode = 1$, the *atest* driver runs as a batch program. All commands are read from the journal file specified in $jrfile$. Graphics output should be directed to files (PostScript and XPM) rather than to interactive displays.

4.5 Array Dimensions and Initialization.

atest has six labeled *common* blocks:

```
common /atest1/ip(100),rp(100),sp(100)
common /atest2/iu(100),ru(100),su(100)
common /atest3/mode,jnlsw,jnlr,jnlw,ibatch
common /atest4/jcmd,cmdtyp,list
common /atest5/idevce,ipane
common /atest6/nproc,myid,mpisw,mpirgn,mpiint,mpiflt
```

The functionality provided by blocks *atest*2, *atest*4 and *atest*6 is not used in the current implementation of the multigraph solver, but is embedded in the generic driver nonetheless.

The *ip*, *rp*, and *sp* and integer, real, and *character*80* arrays of size 100 that contain various global parameters associated with the driver, subroutines *mg*, *mginit*, *gphplt mtxplt*, etc. Their structure and current values can be displayed by appropriate calls to *gphplt*.

The arrays *iu*, *ru* and *su* are analogous to *ip*, *rp* and *sp* and are provided for user-defined variables used in *usrcmd* commands (no commands of this class are used in the multigraph package). *atest*3 contains internal control parameters used by *atest*; several have corresponding locations in the *ip* array, allowing the user to specify defaults as necessary. *atest*4 contains a *character*80* variable *list*, a *character*6* variable *cmdtyp* and an integer *jcmd*, used for communication between the main user interface routines and subroutine *reset*, part of the *usrcmd*.

The block *atest*5 contains integers specifying the current graphics output device. *atest*6 contains some MPI parameters read internally by *atest*; however MPI options are not available in this application.

The main program has a *parameter* statement where values of *maxn*, *maxja*, *maxa* and *lenw* are defined. In turn, these parameters are used to allocate storage for all the major arrays used by the package. *maxn* is the maximum order of linear systems to be solved; *maxja*, *maxa* and *lenw* are the sizes of the matrix arrays *ja*, *a*, and the work array *w*, respectively. Their sizes, relative to *maxn* are problem dependent, and may need to be adjusted by the user in any particular case.

4.6 Matrix Files.

The *read* command (*r*) will read a file containing data defining a matrix and right hand side. Although it increases the file size, matrix files are ASCII (as opposed to binary formats such as XDR) to make them readable by humans. The required format follows:

The first line of the file contains three integers: n, *ispd* and *nblock*, (in that order). $n \geq 1$ is the order of the system; $ispd = 0, 1$ specifies the symmetry structure, and $nblock \geq 1$ specifies the number of blocks. The next $nblock + 1$ lines each contain two integers and are of the form:

$$k \quad ib(k)$$

defining the *ib* array. The next n lines each contain one integer and one real, and are of the form:

$$k \quad b_k$$

defining the right hand side. The remaining lines all define matrix elements; each consists of two integers and one real and are of the form:

$$i \quad j \quad A_{ij}$$

The number of nonzeros is not directly specified; EOF (end-of-file) is treated as the end of matrix elements. Diagonal matrix entries should be defined, even if they are zero. If $ispd = 1$, then either a_{ij} or a_{ji} can be used to specify off-diagonal entries (specifying both causes no problems, but increases the file size). Within each major grouping (*ib*, right hand side, matrix) the entries can be specified in any order. All lines are free format (blank characters are used to separate entries).

4.7 Matrix Generators.

The driver provides a few routines to generate families of matrices of varying orders, for example to study the convergence of various multigraph strategies as a function of n. At the moment, six different classes of matrices are available, each arising from standard discretizations of simple PDE's on uniform meshes. The mesh has *ngrid* mesh points in each space dimension. The parameter *mtxtyp* specifies the matrix to be generated. A brief summary of each class follows:

- $mtxtyp = 0$ ($star5$): This is the usual 5-point star finite difference discretization for $-\Delta u$ on a uniform $ngrid \times ngrid$ square mesh. $n = ngrid^2$; $A_{ii} = 4$ for all diagonal entries, and $A_{ij} = -1$ for all nonzero off-diagonal entries.

- $mtxtyp = 1$ ($|star5|$): This is the same as $star5$ except $A_{ij} = 1$ for all nonzero off-diagonal entries. This is not really a PDE discretization, but provides a simple class of symmetric positive definite matrices which are NOT M-matrices.

- $mtxtyp = 2$ ($star7$): This is the usual 7-point star finite difference discretization for $-\Delta u$ on a uniform $ngrid \times ngrid \times ngrid$ cubic mesh in three space dimensions. $n = ngrid^3$; $A_{ii} = 6$ for all diagonal entries, and $A_{ij} = -1$ for all nonzero off-diagonal entries.

- $mtxtyp = 3$ ($stokes$): This is the mini-element discretization, with static condensation of cubic bubble functions, for the Stokes equations on a uniform $ngrid \times ngrid$ square mesh in two space dimensions. $n = 3\,ngrid^2$. These matrices are highly indefinite and correspond to stabilized saddle-point problems. For this class, we choose $nblock = 3$, with the three blocks corresponding to x-velocity, y-velocity, and pressure.

- $mtxtyp = 4$ ($star9$): This is the usual 9-point star finite element discretization for $-\Delta u$ on a uniform $ngrid \times ngrid$ square mesh. $n = ngrid^2$; $A_{ii} = 8$ for all diagonal entries, and $A_{ij} = -1$ for all nonzero off-diagonal entries.

- $mtxtyp = 5$ ($|star9|$): This is the same as $star9$ except $A_{ij} = 1$ for all nonzero off-diagonal entries. As with $|star5|$, this is not really a PDE discretization, but provides a second simple class of symmetric positive definite matrices which are not M-matrices.

4.8 Journal Files.

The j command causes *atest* to read its command strings from the file $jrfile$, rather than accepting them interactively from the user. It is the only option available in batch mode. A journal file is an ASCII file containing a sequence of command strings as described in Section 4.2. The symbol # appearing as the first character in a line causes that line to be interpreted as a comment. When the end of the file is reached *atest* returns to terminal or web browser mode and again accepts commands interactively. If a q command is encountered in a journal file, *atest* will exit.

When reading a journal file in web browser mode, if a graphics command (g or m) is executed, for some devices *atest* will pause after the picture is drawn until the *click to continue* popup is pressed. This allows time for the user to view the picture before proceeding to the next command in the journal file.

4.9 Machine Dependent Routines.

During the initial installation of the package, the user must provide several machine dependent routines associated with timing and graphics. Default versions of these routines are provided with the package, which should work without modification in many environments, and in any event can serve as a model for a new implementation.

Fortran module *mthdef* is used throughout the package to specify the precision of the floating point arithmetic to be used. The graphics routines *mtxplt* and *gphplt* address the graphics output device through the routines *pltutl*, *pframe*, *pline*, and *pfill*. These routines are documented in detail below.

4.9.1 Arithmetic Specification.

atest uses module *mthdef* to specify the precision of arithmetic to be used. In particular, Below appears the default version of *mthdef*.

```
        module mthdef
c
        integer(kind=4), parameter :: isngl=4
        integer(kind=4), parameter :: idble=8
        integer(kind=4), parameter :: rsngl=4
        integer(kind=4), parameter :: rdble=8
        integer(kind=4), parameter :: rquad=16
c
        integer(kind=4), parameter :: iknd=isngl
        integer(kind=4), parameter :: rknd=rdble
c
        end module
```

The parameters *rsngl rdble* and *rquad* define single, double, and quadruple precision arithmetic, respectively. *isngl* and *idble* define standard and long integers. These definitions should work with no change on most systems. The parameter *rknd* can be set to *rsngl* for a single precision version of the code, to *rdble* for a double precision version, or to *rquad* for quadruple precision version. *iknd* can be set to *isngl* for standard or to *idble* for long integer arithmetic.

4.9.2 Timing Routine.

The timing routine *cpu_time* is used to compute the execution times for subroutines *mginit* and *mg*. If this routine is not available on a particular system, as suitable substitute is generally available. *cpu_time* is called only from the main program, and not from any internal subroutines.

4.9.3 Graphics Interface.

The four device dependent routines in the graphics package are

subroutine pltutl(ncolor, red, green, blue)
subroutine pframe(iframe)

subroutine pline(x, y, z, n, icolor)
subroutine pfill(x, y, z, n, icolor)

Subroutine *pltutl* takes various actions depending on the value of the integer *ncolor*. $ncolor > 0$ specifies initialization; in this case, *ncolor* denotes the number of colors to be used and satisfies $2 \leq ncolor \leq mxcolr$. *red*, *green* and *blue* are vectors of length *ncolor*. The entries $red(i)$, $green(i)$, and $blue(i)$, $1 \leq i \leq ncolor$, are floating point numbers on the interval $[0, 1]$, corresponding to *rgb* values for the ith color. Color number 1 is always white ($red(1) = green(1) = blue(1) = 1.0$), and color number 2 is always black ($red(2) = green(2) = blue(2) = 0.0$). The *rgb* values of the remaining entries depend on the picture to be drawn and the value of *mxcolr*. *pltutl* should create a color map with the required colors, as these will be referenced in future calls to *pline* and $pfill$. If *pltutl* is called with $ncolor < 0$, the drawing is complete and any necessary post processing should be carried out (e.g., close the plot file).

The drawing space used by the graphics routines is always assumed to be either the unit square $(0, 1) \times (0, 1)$ or the rectangle $(0, 1.5) \times (0, 1)$. For devices that have a so-called z-buffer, the drawing space is either the unit cube $(0, 1) \times (0, 1) \times (0, 1)$ or the brick $(0, 1.5) \times (0, 1) \times (0, 1)$. The graphics display itself is always viewed as rectangular with aspect ratio 3/2, which is either a single rectangular frame or three square frames. These frames are numbered 1 to 4 as illustrated in Figure 4.5. The graphics routines write their output to various *lists*. A list consists of a frame, and the attributes rotating/non-rotating. This attribute may not have realizations for all graphics devices. The five available lists are summarized in Table 4.3.

When graphics is initiated for a certain list, say list k, subroutine $pframe(k)$ is called to indicate that subsequent calls of *pline* and $pfill$ contain data to be written to list k. $pframe(-k)$ indicates that the output to the given list should be terminated. By convention, graphics routines are allowed only one open list at a time. Therefore, when $pframe$ is invoked with a positive argument, the given list should be opened and the mapping from the unit cube or brick to the actual device coordinates for the given list should be computed. If the rotation attribute is available, it should be set as specified in Table 4.3. When $pframe$ is invoked with a negative argument, the given list should be closed.

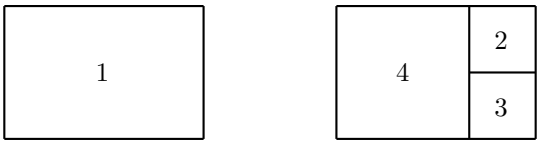

Figure 4.5. *Frame definitions.*

Subroutine *pline* has arguments x, y, z, n, and *icolor*. x, y, and z are vectors of length $n \geq 2$. The points $(x(i), y(i), z(i))$ lie in the unit cube or the brick $(0, 1.5) \times$

list	frame	rotating
1	1	no
2	2	no
3	3	no
4	4	no
5	4	yes

Table 4.3. *list specifications for pframe.*

$(0,1) \times (0,1)$. The z coordinate is useful only for devices that have a z-buffer, and can be ignored in other cases. *icolor* is an integer between 1 and *ncolor*, where *ncolor* was the argument that initialized *pltutl*, indicating the color to be used. *pline* should draw the given polyline $(x(i), y(i), z(i))$ to $(x(i+1), y(i+1), z(i+1))$, $1 \leq i \leq n-1$, with the specified color in the proper frame.

Subroutine *pfill* has arguments x, y, z, n, and *icolor*. x, y, and z are vectors of length $n \geq 3$. The points $(x(i), y(i), z(i))$ lie in the unit cube or the brick $(0, 1.5) \times (0,1) \times (0,1)$, and define an n-sided (planar) polygonal region with sides $(x(i), y(i), z(i))$ to $(x(i+1), y(i+1), z(i+1))$ for $1 \leq i \leq n-1$, and $(x(n), y(n), z(n))$ to $(x(1), y(1), z(1))$. *icolor* is an integer between 1 and *ncolor*, where *ncolor* was the argument that initialized *pltutl*, indicating the color to be used. *pfill* should color the specified polygon with the specified color in the proper frame.

idevce	output device
0	web browser
1	XPM file
2	PostScript file

Table 4.4. *Default graphics devices.*

The default installation of the *atest* package includes several output graphics devices. These are described in Table 4.4. PostScript and XPM are both ASCII files. The parameter *grfile* specifies the file name. To allow one file name variable to stand for multiple files, the file name *grfile* is scanned for the string *figXXX*. If it is found, for the first actual file name, it is replaced by *fig*001, for the second by *fig*002, and so on, allowing each actual file to receive a unique name. The file name is also scanned for the string *.ext* and if found, it is replaced by *.xpm* or *.ps* as appropriate. If the string *.ext* is not found, the appropriate extension is appended to the existing name.

The web browser interface provides both 2D and 3D graphics. The canvas and the type of graphics are specified through the parameter *ipane*, as described in Table 4.5. The parameters *mpane* and *gpane* are associated with graphics subroutines *mtxplt* and *gphplt*, respectively, and specify the value of *ipane* to be used

ipane	*webgui* canvas	graphics type
0	upper right	WebGL 3D
1	lower left	WebGL 3D
2	lower right	WebGL 3D
3	upper right	Pixmap 2D
4	lower left	Pixmap 2D
5	lower right	Pixmap 2D

Table 4.5. *ipane definitions.*

with that routine.

WebGL graphics written to $list = 5$ can be manipulated with the mouse. Mouse button one (left) can be used to rotate the image. Button two (middle) is used to pan the image, and button three (right) can be used to zoom. On Apple systems with a one button mouse, button two and three are simulated by pressing the OPTION and COMMAND keys, respectively, simultaneously with the mouse. On Windows one button systems the ALT and WINDOWS keys play a similar role. In WebGL windows, the OPTION + I toggle referenced in Table 4.2 can display the numerical values for the zoom, rotation matrix and translation.

Additionally, associated with each WebGL canvas are six buttons. These are described in Table 4.6.

button	effect
RESET	return a zoomed/panned/rotated image to its initial state
FREE	clear the image and free associated memory
SPIN	spin image
POP	place canvas in its own web browser tab
PUSH	return canvas to its original location in the web browser tab
SAVE	save image to a file
LOAD	read image from a file

Table 4.6. *WebGL graphics window buttons.*

When the POP button is pushed, the graphics window is removed from the array of graphics windows in the main web browser display, and appears in a separate tab. This is useful, as typically the size of the image increases. In the new tab, the POP button is replaced by one labeled PUSH. When the PUSH button is pressed, the canvas is returned to its previous location in the main display. The SAVE button saves a binary file containing the image, which can later be restored using the LOAD button. If the OPTION key is pressed simultaneously with the SAVE button as indicated in Table 4.2, then the image is saved as an ASCII file.

If $3 \leq ipane \leq 5$ then only the FREE button appears. These values of *ipane*

are useful to preview pixmaps, that can then later be saved as XPM files.

Bibliography

[1] Randolph E. Bank and Tony F. Chan, *An analysis of the composite step biconjugate gradient method*, Numerische Mathematik, 66 (1993), pp. 295–319.

[2] Randolph E. Bank and R. Kent Smith, *General sparse elimination requires no permanent integer storage*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. 574–584.

[3] ——, *An algebraic multilevel multigraph algorithm*, SIAM J. on Scientific Computing, (to appear).

[4] ——, *Multigraph algorithms based on sparse gaussian elimination*, in Thirteenth International Symposium on Domain Decomposition Methods for Partial Differential Equations, Domain Decomposition Press, Bergen, to appear.

[5] Christopher Deotte, *Webgui: A web browswer base graphical user interface for scientific software users' guide*, tech. report, Department of Mathematics, University of California at San Diego, 2017.

[6] S. C. Eisenstat, M. C. Gursky, M.H. Schultz, and A.H. Sherman, *Algorithms and data structures for sparse symmetric Gaussian elimination*, SIAM J. Sci. Statist. Comput., 2 (1982), pp. 225–237.

[7] Wolfgang Hackbusch, *Multigrid Methods and Applications*, Springer-Verlag, Berlin, 1985.