SPLIB: A LIBRARY OF ITERATIVE METHODS FOR SPARSE LINEAR SYSTEMS

RANDALL BRAMLEY AND XIAOGE WANG*
DEPARTMENT OF COMPUTER SCIENCE
INDIANA UNIVERSITY - BLOOMINGTON

September 3, 1995

1. Introduction. Sparse linear systems are the primary consumer of computer time for most scientific and engineering computer programs. Analyzing them and developing faster and more accurate solution methods for the types of linear systems encountered in practice is difficult, however. Part of the difficulty is because those systems often fail to satisfy the hypotheses commonly assumed for detailed mathematical analysis. Difficulty in experimental testing of methods occurs because virtually every application program uses a different data structure for storing the coefficient matrix of the linear system. Testing different iterative solvers within the source program generally requires writing and tuning the solvers for the particular data structure, and also requires fitting the solver into the memory management scheme used by the source program. In practice, this means major recoding effort for each solver and preconditioner tried.

Because of this, a different methodology for collaborative work with scientists has proven useful: first, sample linear systems are extracted from the application program and written out to a file. Then using data structure conversion routines (which usually require writing a small routine customized for the application), the linear system is converted to a predetermined standard format. Once this is done, dozens of different solution methods and preconditioners can be tested on the linear system, and these tests can be run on workstations independently of the scientist's program. However, this approach is limited by space available for storing such extracted linear systems. Furthermore, for nonlinear solvers small differences in the solution of the linear system can cause large differences in the solution trajectory that the nonlinear problem follows. This necessitates a second phase of testing: directly calling a package of linear solvers from the application code. Again, a routine for converting from the application's native data structure to the one required by the linear solver package is required, but this generally involves only a small amount of programming. Only when a good solution strategy has been found should the extensive work required to implement it efficiently within the source program be carried out. The iterative library should also be relatively parsimonious with storage, so that it can reasonably be used as a production solver for small to medium sized applications.

SPLIB is a library of sparse iterative solvers, with preconditioners, for rapid prototyping of solvers for nonsymmetric linear systems of equations. SPLIB was developed with the intent that it be useful for comparing iterative methods and preconditioners in

* WORK SUPPORTED BY NSF GRANTS CDA 93-03189, CDA 93-09746, AND ASC-9502292

a uniformly coded implementation, while facilitating interdisciplinary collaboration. A second role is for research into the linear solvers themselves. The package was designed to satisfy these two purposes by being written in reasonably portable Fortran 77, and easily called as a subroutine from Fortran, C, or C++ code. The number of arguments that must be passed to SPLIB is kept as small as possible, while still allowing flexibility in the parameterization of the preconditioners and solvers. Common blocks are avoided except internally, and the code is heavily instrumented to provide valuable information about the convergence history, memory usage, and CPU time used.

This document is intended primarily to describe SPLIB. For information about the methods implemented, see the references or general sources such as [4, 6].

2. Comparison With Other Packages. Many libraries of iterative solvers have recently been made available, and SPLIB draws from some of them. For example, Yousef Saad's ILUT preconditioner is included as a possible preconditioner, and Templates [7] versions of GMRES(m) and biconjugate gradients stabilized are included as alternative implementations to ones already in SPLIB. However, SPLIB differs from other libraries in important ways. First of all, it is intended to play several roles. For scientists and engineers, the package can be used in a black-box fashion, without knowing the details of the methods used. Since it is written mainly in Fortran 77 and uses reasonably efficient data structures, its performance is within a small factor of that achievable by routines specially designed for the applications program. Furthermore, it can be easily used to test various methods, and switching between methods is possible when difficulties are encountered on a particular system.

Unlike Templates, this package computes and implements preconditioners, including incomplete factorizations with levels of fill-in allowed. Templates attempts to avoid this problem by counting on the user to supply routines for the matrix operations and the application of the preconditioners. In practice, scientists and engineers find the implementation of iterative methods easy, but the computation and application of preconditioners much more difficult. Furthermore, virtually all nonsymmetric linear system solvers fail on most problems they are applied to, unless an effective preconditioner is used. SPLIB basically performs much more of the difficult coding work for the user. The weakness of this approach is that a particular sparse matrix storage scheme is used, but with the availability of efficient data structure conversion routines the price seems worthwhile. A data structure conversion routine is invariably easier to write than an incomplete factorization routine.

Among existing libraries, NSPCG is closest to the current SPLIB, and implements many of the same methods. NSPCG has the advantage of allowing several different data input formats, not restricting the user to a single one. However, NSPCG primarily implements older iterative methods, and in particular does not include the more sophisticated preconditioners such as ILUT and ECIMGS which are available in SPLIB. Furthermore, SPLIB has outstanding instrumentation and provides more clues about its performance and nonconvergence cases.

Also unlike Templates and the solvers provided with Sparskit [21], this code uses a forward instead of a reverse communication model. In reverse communication, when

the iterative solver needs application of the matrix through a matrix–vector product, or when it needs the preconditioning matrix applied, it returns control to the application routine with a flag indicating the operation to perform. The invoking routine must then perform the requested operation and return control to the iterative solver. This allows the user to use the native data structures of the invoking program for the matrix–vector product, but requires him to handle the more difficult coding problem of creating the preconditioner. Most scientists and engineers seem to prefer a forward communication model, where the matrix, relevant vectors, and additional workspace for preconditioning are passed to the iterative solver package, and a solution vector or error flag is returned. The disadvantage of this is that a particular data structure (or selection of data structures as NSPCG allows) must be used by the scientist or engineer. Experience suggests that most applications researchers find converting their linear system to a particular data structure less cumbersome than, for example, computing an incomplete factorization with levels of fill and numerical dropping allowed. Furthermore, most codes written in imperative languages follow the model of passing data to a routine and having a solution returned. Although there are advantages to the reverse communication model, SPLIB uses what seems the more natural model for applications researchers.

Although SPLIB has an implementation of the transpose–free QMR method, QMRPACK provides far more sophisticated implementations of the many variants of QMR. In particular, it provides the recommended look–ahead Lanczos versions which are significantly more robust than the simple transpose–free version implemented here.

3. SPLIB Data Structure. SPLIB implements several incomplete factorization based preconditioning methods which must access the entries of the coefficient matrix A. That in turn means that some common data structure must be settled on. The data structure used in SPLIB is CSR, compressed sparse row. This is the transpose of the data structure used for the Harwell/Boeing (HB) collection of test matrices. CSR stores the nonzeros of the matrix, row by row, in a singly indexed array A of double precision numbers (real*8 in Fortran). Another array, COLIND, contains the column index of each corresponding entry in the A array. A third integer array, RWPTR, contains pointers to the beginning of each row, with an additional pointer to the first index following the nonzeros of the matrix A. If NNZ is the number of nonzeros and N is the order of the matrix, then A and COLIND have length NNZ and RWPTR has length N+1.

As an example, consider the matrix

$$A = \begin{bmatrix} 11 & 12 & & 14 & & & & \\ & 22 & 23 & & 25 & & & \\ 31 & & 33 & 34 & & & & \\ & 42 & & & 45 & 46 & & \\ & & & & 55 & & & \\ & & & & 65 & 66 & 67 & \\ & & & & 75 & & 77 & 78 \\ & & & & & & 87 & 88 \end{bmatrix} \tag{1}$$

INDEX	1	2	3	4	5	6	7	8	9	10	11
A	11	12	14	22	23	25	31	33	34	42	45
COLIND	1	2	4	2	3	5	1	3	4	2	5

RWPTR	1	4	7	10	13	14	17	20	22

INDEX	12	13	14	15	16	17	18	19	20	21	22
A	46	55	65	66	67	75	77	78	87	88	-
COLIND	6	5	5	6	7	5	7	8	7	8	-

TABLE 1
Compressed Sparse Row Data Structure

The CSR data structure containing A is shown in Table 1. Although CSR requires entries from a row to be consecutively located in data structure, the columns need not be ordered in any way (but see Section 8 for restrictions on ordering within a row for certain methods in SPLIB). The total storage required is NNZ floating point and NNZ + N + 2 integers, where the additional integer is for the storage of N. CSR makes accessing rows easy, but accessing columns is more difficult. This matches the intended use of SPLIB, where generally matrix-vector products are important and on most hierarchical memory machines that is done more efficiently using an inner product form of matrix-vector product. When efficient column access is needed, most sparse solver libraries set up an auxillary linked list data structure. The matrix vector products $y = A * x$ in CSR have the form

```
Initialize y(1:n) = 0
for i = 1 to n
   for k = rwptr(i) to rwptr(i+1)-1
       y(i) = y(i) + A(k)*x(colind(k))
   end for
end for
```

Note that a gather/scatter operation (i.e., indirect addressing) is required only on one vector in the innermost loop. Also, it is a read indirection, which does not stall pipelining as badly as write indirection. CSR requires four memory reads and one write per two flops. Tools are available in Yousef Saad's Sparskit [21] for converting from other sparse matrix data structures to and from CSR.

Research [1, 20, 15] has shown that a jagged diagonal data structure is often more efficient for matrix–vector products, especially for vector or heavily pipelined machines. However, jagged diagonals is generally difficult to implement correctly for people who do not customarily work with sparse matrix data structures. Furthermore, a permu-

tation of the product vector is needed after each matrix–vector product, and on many workstations and hierarchical memory machines this tends to cause jagged diagonals to lose their advantage.

Internally, SPLIB computes some preconditioners in a MSR data structure, which primarily differs from CSR by storing the diagonal entries of the matrix in the first N locations of the array A, and using the first N+1 entries of the COLIND array to store the row pointers. This will eventually be changed so that the preconditioners are also stored in a CSR type format because experiments have shown that the time to apply a preconditioner can be significantly smaller when using CSR format. However, the use of MSR is visible to applications scientists using the code since the preconditioners are computed and manipulated only internally.

Finally, in the description above "nonzero entries" should be interpreted as "interesting entries", since it is allowed to explicitly store zeros in the data structure. This might be done, for example, with a discretized PDE where some of the coefficients are zero on initial time steps but later become nonzero. In that case, it is desirable to set up a single CSR data structure that can handle all the linear systems arising from the single discretization mesh.

4. Methods in SPLIB. SPLIB implements 13 iterative solvers, primarily modern CG-like methods, and 7 preconditioning methods. The preconditioners include ones that allow multiple levels of fill-in through positional and numerical strategies. All of the preconditioners are parameterized, providing a rich variety of preconditioning strategies. The iterative methods currently are

- bi-conjugate gradients (BICG) [11],
- conjugate gradients for the normal equations $AA^Ty = b, x = A^Ty$ (CGNE) [9].
- conjugate gradients for the normal equations $A^TAx = A^Tb$ (CGNR) [16],
- conjugate gradients squared (CGS) [23],
- biconjugate gradients stabilized (CGSTAB) [10],
- restarted generalized minimum residual (GMRES) [22],
- Transpose-free QMR (QRMTF) [12],
- Templates version of biconjugate gradients stabilized (BICGS) [6],
- Templates version of GMRES (GMREST) [6],
- Jacobi iteration (JACOBI) [14],
- Gauss-Seidel (GAUSS) [14],
- Successive Over-relaxation (SOR) [14],
- Orthomin(m) (ORTHM) [24].

Note that two methods are repeated (GMRES(k) and biconjugage gradients stabilized), with the second versions modified from Templates [6]. The Templates GMRES uses left preconditioning, solving $M^{-1}Ax = M^{-1}b$, while the native SPLIB version uses right preconditioning, solving $AM^{-1}y = b$, $x = M^{-1}y$. The two versions of biconjugate gradients stabilized differ in the change of variables used to handle preconditioning, and the native SPLIB version implements double ($v = v + a * w + b * u$) vector updates in a single loop. For machines like the IBM RS/6000, this gives a more accurate update than using a sequence of scalings and vector updates via the BLAS because the combined

multiply/add unit retains the full result of a multiply that is then sent to the add unit. This provides more accurate computations, but does not satisfy the IEEE standard for floating point arithmetic

The preconditioners currently implemented in SPLIB are

- incomplete LU factorization with s levels of fill (ILU(s)) [18],
- modified incomplete LU factorization with s levels of fill and relaxation parameter ω (MILU(s,ω)) [13],
- incomplete LU factorization with threshold dropping (ILUT(s,ϵ)) [21],
- symmetric successive overrelaxation (SSOR(ω)) [14],
- block diagonal with tridiagonal blocks TRID(s),
- incomplete LU factorization without modification of off-diagonal elements (ILU0),
- extended compressed incomplete modified Gram-Schmidt (ECIMGS) [8].

All of the preconditioning methods are used as left preconditioners by the solvers, except for GMRES which uses right preconditioning (as noted above, GMREST is the same algorithm but with left preconditioning). The levels of fill in ILU(s) and MILU(s,ω) refer to the amount of fill-in allowed during the incomplete factorization. Using $s = 0$ gives upper and lower triangular factors L and U such that $L+U$ has the same sparsity pattern as A. Using $s = n$ gives the full LU factorization of A without pivoting. Usually, increasing s provides a better quality preconditioner (in terms of reducing the number of iterations required), but at the cost of increasing storage. However, a counterexample is provided later showing that increasing s can actually make the preconditioner worse. The recommended usage is to try a small value of s, and increase it if the iterative method fails, or if it requires an excessive number of iterations. MILU(s,ω) also takes a parameter ω. During the factorization, discarded fill elements are multiplied by ω and added to the diagonal. Choosing $\omega = 0$ gives ILU(s), while $\omega = 1$ gives the classical modified ILU factorization. For finite element problems, MILU(s,1) gives a preconditioner which is exact on constant functions. Although it can be shown to have better order of accuracy on model problems, on practical problems it generally gives a worse preconditioner; for an explanation of this phenemonen, see the results in [5].

ILUT is Yousef Saad's algorithm (and implementation) of thresholding as a dropping criterion. This allows dropping of elements during the incomplete factorization based on their relative size instead of their positions within the matrix. See the documentation in the routine PRECOND/ILUT.F for details, and suggestions of how to set the thresholding. Although for a given amount of storage ILUT generally outperforms ILU in quality of preconditioning, it takes longer to compute and examples exist where ILUT fails outright while ILU succeeds.

The method TRID(s) implements block diagonal preconditioning where each block except the last one is of dimension $s \times s$. When $s = 1$, this gives diagonal preconditioning. and when $s = n$ the entire tridiagonal part of A is used as a preconditioner. Gaussian elimination with pivoting is used to factor the tridiagonal blocks, so this method requires 4 vectors of length n in storage.

ILU0 is distinct from ILU(s) with $s = 0$ in that its factorization process only modifies the diagonal entries of the matrix being factored. For PDE problems arising

from quasi-uniform meshes, the two are equivalent, but for irregular meshes they differ (see [13], where the two different algorithms are distinguished by using an asterisk). ILU0 requires only one additional vector of storage.

SSOR(ω) is a preconditioner based on a matrix splitting, and so requires no additional floating point storage. The preconditioner is

$$M = (I + \omega L D^{-}1)(\omega^{-1} D + U),$$

where $\omega \in (0, 2)$ is the SSOR acceleration parameter. Note that the usual way of writing M is

$$M = (1/(2-\omega))(\omega^{-1} D + L)(\omega * D^{-1}) * (\omega^{-1} D + U)$$

and so the scaling factor of $1/(2-\omega)$ has been omitted and the last two factors have been combined. The second form shows that when A is symmetric, so is the SSOR preconditioner.

The ECIMGS preconditioner [8] is unique to SPLIB, and is derived from incomplete orthogonalization preconditioners [26, 25]. The parameter s provided to ECIMGS is a floating point number, which specifies the sparsity pattern to target. ECIMGS can provide a preconditioner with a less dense sparsity pattern than that of A, or it can provide a preconditioner with the same levels of fill allowed by ILU(s). For a given target sparsity pattern, however, ECIMGS requires much more intermediate storage than the corresponding ILU does. Like ILUT, it tends to provide a better quality preconditioner, but the high cost of computing it suggests it is more suitable for applications where the same preconditioner can be used to solve several linear systems, or where robustness of the iterative method is a problem.

Iterative methods and preconditioners which are based on some form of splitting of the matrix A into upper/lower triangular parts are more efficient if the data structure for A has each row sorted so that first the lower triangular entries occur, then the diagonal element, followed by the upper triangular entries. For that reason, SPLIB will rearrange the data structure to assure that this is the case. That in turn means that on return, although the same matrix A is still represented, the computer arrays to represent it may have been rearranged. This is the case for iterative methods Jacobi, Gauss-Seidel, and SOR(ω). It also applies to preconditioners ILU0 and SSOR(ω).

5. SPLIB Instrumentation. A major goal of SPLIB was to provide extensive instrumentation of the solution process. This is done by providing timings of the major phases of the computation a detailed listing of the memory usage, and graphical views of several measures of convergence.

The usual goal in iterative methods is to have the residual norm $\|r_k\| = \|Ax_k - b\|$ sufficiently reduced, but iterative methods often provide a measure of the preconditioned residual $\|M^{-1}(Ax_k - b)\|$ within the iteration at little or no extra cost. In SPLIB, this preconditioned residual is used as a tentative stopping test. First, a local tolerance is set as $\epsilon =$ TOL, where TOL is the user-supplied tolerance. If $\|M^{-1}(Ax_k - b)\| < \epsilon$ then the unpreconditioned residual is tested, and the iteration is stopped. However if

$\|r_k\| > \epsilon$, then ϵ is reset as $\epsilon = \epsilon/\|r_k\|$ * TOL, and the iterations resume. To prevent pointless cycling for a stalled iteration, this resetting is done at most 8 times and an error flag is returned.

When INSTLVL is greater than or equal to 1, Xgraph files showing the unpreconditioned residual versus iteration number and elapsed time are produced. If INSTLVL is greater than or equal to 2, other files showing the preconditioned residual versus iteration number and elapsed time are also produced. Another norm is important for understanding the convergence of iterative methods, one proposed by Oetli and Prager in [19] and used by Arioli *et al* in [3]. This norm is

$$\omega_k = \|Ax_k - b\|_\infty / (\|A\|_\infty \|x_k\|_1 + \|b\|_\infty,$$

and a value of ϵ for this Oetli/Prager measure means that x_k is the exact solution of a problem with coefficient matrix that differs from A in the infinity norm by less than ϵ. The Oetli/Prager measure thus provides an important check on the conditioning of the problem, as will be shown in Section 9. When INSTLVL is greater than or equal to 3, Xgraph files showing the Oetli/Prager measure versus iteration number and elapsed time are also produced.

Another set of Xgraph files can be produced whenever an exact solution x^* is available. The true error norm $\|x_k - x^*\|$ can then be computed, but because this is an extremely unusual case using this option requires passing the solution vector into SPLIB using a common block. If such a norm is important, check the code documentation for the common block TRUESOL.

Elapsed execution time is used for showing convergence history because preconditioned iterative methods can vary greatly in their cost of preprocessing (which primarily consists of computing a preconditioner), cost per iteration, and effective use of high performance computer architectural features such as caches and pipelining. On each iteration, the solvers call the instrumentation routine INSTR.F. Because of the cost of computing all the requisite information the instrumentation routine stops the elapsed time clock, computes and outputs the requested information, then restarts the clock and returns to the iterative solver. Using full instrumentation can more than double the wall clock time required by the solvers, but the elapsed time shown in the summary output file and Xgraph files excludes this perturbing cost. For large linear systems and machines with hierarchical memory the instrumentation routine can cause further perturbations by changing the number of cache misses that would have occured without instrumentation. However, tests on a variety of machines have never revealed any cases where the perturbation exceeded 3% of the time.

Another source of timing perturbation can occur when testing several methods with the same matrix. The first solver/preconditioner pair will induce the cost of paging in problem and code data and loading the caches with that data. This can be tested for by running the same solver/preconditioner pair two or more times in a single run. If the first run takes significantly longer than succeeding ones, it may be necessary to discard it for comparative purposes.

The summary file also prints out the time spent in

- computing the preconditioner
- iterating, excluding the instrumentation routine INSTR time
- matrix-vector products, including that from INSTR
- transposed matrix-vector products, including that from INSTR
- preconditioner application, including that from INSTR
- total time, excluding that spent in INSTR

6. Using SPLIB.

6.1. Structure of SPLIB Library. The directory structure of SPLIB is shown in Figure 1. The BLAS directory and routines are provided for systems without a native BLAS library. Generally it is more efficient to use the ones provided with your machine, if the vendor has supplied them. The directories SOLVERS, PRECOND, and TOOLS contain the iterative solvers, the preconditioners, and tools for basic matrix operations and instrumentation, respectivelly. Source code include files are in the directory INCLUDE.

6.2. Portability. "There are no portable codes; only codes that have been ported." Although for the most part SPLIB has been written with standard Fortran 77, it extensively uses a UNIX[1] environment. Following are the known non-portable or non-Fortran 77 standard features:

- Include files are used in the Fortran source. Although this is not in the Fortran 77 standard, we do not know of any existing compilers which do not recognized the "include" statement.
- "Implicit none" statements are used throughout.
- One C routine has been linked with the code: GETR.C, which reads the RUSAGE structure for system resource usage. The underscore convention has been followed: the Fortran code calls GETR, but the source code in GETR.C has the name GETR_. When using IBM RS/6000 systems, it is necessary to edit the source code to remove the underscore. On some systems the RUSAGE structure is not accessible, or has fields that are not implemented.
- A function called MYTIME is used for all timings. It in turn calls the standard UNIX function ETIME(). For systems like Cray or IBM, you will need to comment out the section of MYTIME that corresponds to ETIME and uncomment out the section that contains the timing facility on your machine.

Additionally, some compilation features will probably need to be customized for your machine. Machine dependent compiler options are in the file MAKE.INC, which gets included in each of the other makefiles provided. Particular features to be aware of are:

- Alignment of common blocks on double word boundaries is necessary.
- The makefiles test for the presence of the Unix utility RANLIB in /usr/bin/ranlib. If RANLIB exists but it is not present in that customary place, the makefiles should be edited to show the correct location.
- The makefiles test for the presence of native BLAS routines in /usr/lib/libblas.a; if they are not present the Fortran versions of the BLAS routines in subdirec-

[1] UNIX is a trademark of AT&T

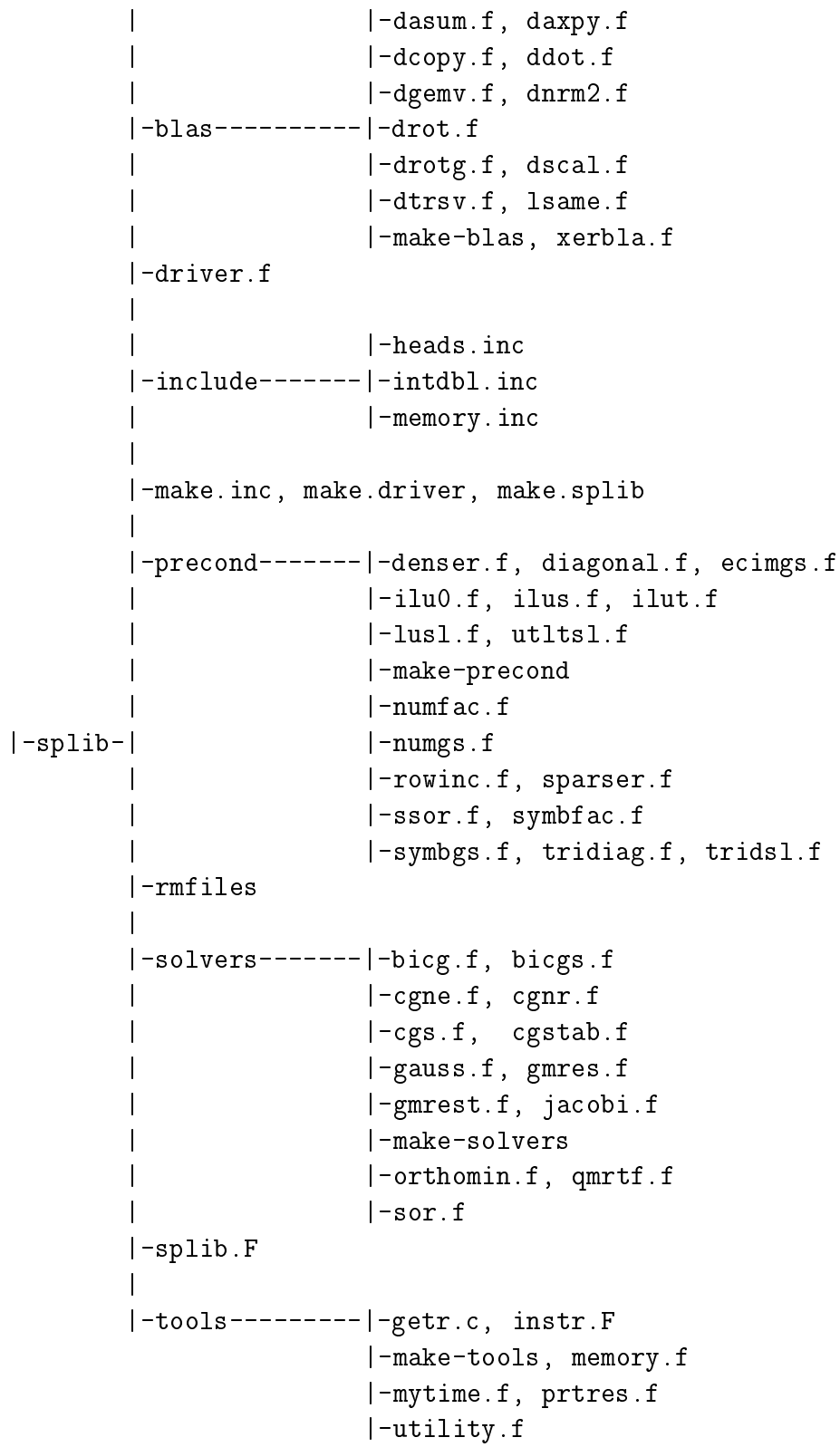

FIG. 1. *Directory Structure of SPLIB*

tory splib/blas will be compiled and linked instead. If the BLAS libraries are present but in a different location, edit the makefiles for that location. Using the Fortran versions will not cause an error, but using native BLAS libraries gives significantly faster code.

To make life easier, several commented out blocks are provided in MAKE.INC which correspond to a variety of computers; probably just uncommenting out one of the blocks will allow your compilation to succeed.

Finally, the function DLAMCH() from the LAPACK project is used to set machine dependent parameters. This function will trigger underflows and other IEEE floating point traps, so you will need to either disable them or provide your own trap handler. On most systems the IEEE traps are disabled by default, with perhaps a warning message given at the end of the run.

6.3. Running SPLIB. SPLIB is compiled as an archive library SPLIB.A, which should be linked with your own code. A sample driver that reads a transposed Harwell/Boeing formatted matrix and solves an artificial linear system with it as coefficient matrix is in the SPLIB directory in DRIVER.F, and can be compiled with the command

```
make -f make.driver
```

A sample Harwell/Boeing formatted matrix is provided in the file MATRIX.

The most onerous task on a user of SPLIB is setting up the matrix in a CSR data structure. However, it was necessary to select a data structure in order to allow the implementation of sophisticated incomplete factorization preconditioners, which require access to individual components of the coefficient matrix.

Secondly, the user must supply a vector of work storage for containing auxillary vectors and preconditioner data that SPLIB generates. In general, the iterative methods use only a few (3–8) additional real*8 vectors of length n. The restarted methods GMRES(m) and ORTHOMIN(m) also require an additional M and 2M vectors of length n, for storing bases of the underlying Krylov subspaces. The preconditioners are generally unpredictable in their storage requirements, depending on user controllable parameters such as levels of fill and drop tolerances as well as the nonzero pattern of the coefficient matrix A and the particular values in A. If SPLIB runs out of space, an error flag is returned along with a estimate of the amount of additional storage needed.

The current calling sequence for SPLIB is:

```
      call  splib(a, colind, rwptr, n,
     1             x, b, work, space,
     2             tol, reltv, maxits, pcmeth, solmeth,
     3             iparms, rparms, redo, errflg, iunit, instlvl)
```

The arguments are split into lines according to a grouping into categories.

6.3.1. Data Structure for A. The data structure for A is in A, COLIND, RWPTR, and N. As described earlier, this is a compressed sparse row (CSR) format for A. If you invoke the iterative methods Jacobi, Gauss-Seidel, or SOR(ω), or if you use the preconditioners ILU0 or SSOR(ω), the data structure for A on return will be rearranged to have the entries in each row in column index increasing order.

6.3.2. Required Vectors. The vector X on entry contains a user-supplied starting estimate of the solution, which can be a vector of zeros. On exit X contains the solution found by SPLIB. The vector B contains the right hand side of the system, and WORK is an integer array of length SPACE, used by SPLIB for containing preconditioning and other data. Because most preconditioners require storage at least equal to that used by A, be generous with the amount provided in WORK.

6.3.3. Control Parameters. The parameters TOL and RELTV are the stopping test tolerance information. SPLIB stops if $\|Ax - b\|_2 \leq tol$ (when RELTV = .false.) or if $\|Ax - b\|_2 \leq tol * \|b\|$ (when RELTV = .true.). MAXITS contains on entry the maximum allowed number of iterations and on exit the actual number of iterations taken. BEWARE: if you will be calling SPLIB for a sequence of systems, you will need to reset MAXITS on each call to SPLIB.

If the parameter REDO = .false, this indicates that SPLIB should use the preconditioner computed on earlier steps. Since SPLIB sets up the preconditioner in the array WORK, note that this means the user cannot overwrite WORK with his or her own data without risking segmentation fault. If it is necessary to retrieve some or all of the space in the array WORK but you wish to still reuse an earlier preconditioner, you can do so by examining the output from routine LISTBL, which is automatically called after each run when the instrumentation level INSTLVL ¿ 0. See the example in Section 9. If REDO = .true., SPLIB will compute the preconditioner as specified by the parameter PCMETH described below.

The integer parameter SOLMETH specifies which iterative solver to use, following the schema:

```
solmeth =  1 -->  Bi-Conjugate Gradients
solmeth =  2 -->  Conjugate Gradients for system AA'y = b, x = A'y.
solmeth =  3 -->  Conjugate Gradients for system A'A x = A'b
solmeth =  4 -->  Conjugate Gradients Squared
solmeth =  5 -->  Conjugate Gradients Stabilized
solmeth =  6 -->  GMRES(k)
solmeth =  7 -->  Transpose-free QMR
solmeth =  8 -->  Templates version of Conjugate Gradients Stabilized
solmeth =  9 -->  Templates version of GMRES
solmeth = 10 -->  Jacobi iteration
solmeth = 11 -->  Gauss-Seidel
solmeth = 12 -->  Successive Over-relaxation (SOR)
solmeth = 13 -->  Orthomin(m)
```

When a SOLMETH of zero or less is specified, CG-Stab is used by default. Methods such as ORTHOMIN(m) and GMRES(m) that require a Krylov subspace size to be specified obtain it from the second component of RPARMS: M = RPARMS(2). If that is zero or less, the Krylov subspace size is set to 10.

The integer parameter PCMETH specifies which preconditioner to use, following the schema:

```
pcmeth = 0 -->  no preconditioner.
```

```
pcmeth = 1 -->  ILU(s)
pcmeth = 2 -->  MILU(s,rpcprm)
pcmeth = 3 -->  ILUT(s,rpcprm)
pcmeth = 4 -->  SSOR(rpcprm)
pcmeth = 5 -->  TRID(s), where s is the block size.
pcmeth = 6 -->  ILU0, the space saver version of ILU(0)
pcmeth = 7 -->  ECIMGS(rpcprm).
```

Some preconditioners use one or both of an integer and a real*8 parameter. The integer parameter s is in IPARMS(1), and indicates the levels of fill-in to allow for ILU and MILU and the block size to use in the tridiagonal preconditioner TRID (see Section 4). For ILUT, s is the maximum additional entries to allow per row in the preconditioner, compared to the original matrix. So setting IPARMS(1) = 4 will allow four more nonzeros in each row of the preconditioner than is found in the corresponding row of A.

The real*8 parameter RPCPRM = RPARM(1) for MILU is the relaxation parameter, the amount to multiply discarded fill-in entries by before adding them to the diagonal. For SSOR, it is the relaxation parameter ω. For ILUT, it is the drop tolerance. ILUT is one of the more sophisticated preconditioners in SPLIB, and the user is encouraged to read the documentation in that code for the setting of the parameters it uses.

For ECIMGS, the integer parameter s is not used, and the real*8 parameter RPCPRM = RPARMS(1) specifies the sparsity pattern the preconditioner should have with the interpretation

```
rpcprm = 0.0       use the nonzero pattern of the matrix A
0.0 < rpcprm < 1.0 use a sparser pattern than that of A
rpcprm = 1.0       use a diagonal pattern
rpcprm > 1.0       use a denser pattern than that of A
                    with  (int(rpcprm) levels of fillin)
```

Although ECIMGS can produce preconditioners that have the same sparsity pattern (and hence the same amount of final storage) as ILU(s), it sometimes requires a great deal of intermediate storage, more than the corresponding ILU(s) does. This is only in the intermediate computations, however, and generally ECIMGS produces a better preconditioner than ILU(s) does. If the user specifies PCMETH < 0, then no preconditioner is used by default. For applications which solve a sequence of linear systems (e.g., ones that solve a nonlinear problem using Newton's method), it is often more efficient to compute a preconditioner once and then reuse it for several linear systems. The parameter REDO indicates whether SPLIB is to reuse the preconditioner computed from an earlier invocation. Because this requires preserving part or all of the array WORK, it is recommended initially to simply recompute the preconditioner until you are comfortable with using SPLIB.

6.3.4. Instrumentation Parameters. SPLIB differs from other packages primarily in its heavy use of instrumentation. The user can specify the output of Xgraph files for plotting various measures of error norms against iteration number or against elapsed CPU time. These are controlled by the parameter INSTLVL:

```
instlvl < 0  no output from splib
```

Iunit index	File Name	Error Norm and Abssica
4	RESI	$\|Ax-b\|$ vs. iter
5	MRESI	$\|M^{-1}(Ax-b)\|$ vs. iter
6	RELI	$\|Ax-b\|/\|r_0\|$ vs. iter
7	MRELI	$\|M^{-1}(Ax-b)\|/\|M^{-1}r_0\|$ vs. iter
8	OETLI	$\|Ax-b\|_\infty/(\|A\|_\infty\|x\|_1+\|b\|_\infty)$ vs. iter
9	ERRORI	$\|x-x^*\|$ vs. iter
10	–Not used –	
11	REST	$\|Ax-b\|$ vs. time
12	MREST	$\|M^{-1}(Ax-b)\|$ vs. time
13	RELT	$\|Ax-b\|/\|r_0\|$ vs. time
14	MRELT	$\|M^{-1}(Ax-b)\|/\|M^{-1}r_0\|$ vs. time
15	OETLT	$\|Ax-b\|_\infty/(\|A\|_\infty\|x\|_1+\|b\|_\infty)$ vs. time
16	ERRORT	$\|x-x^*\|$ vs. time

TABLE 2
Unit Numbers and File Names Opened in SPLIB

```
instlvl = 0  output only summary data
instlvl = 1  output residual norm data
instlvl = 2  output preconditioned residual norm
instlvl = 3  output relative residual norm
instlvl = 4  output relative preconditioned residual norm
instlvl = 5  output Oetli/Prager norm
instlvl = 6  output error norm
```

The level is cumulative, so specifying output of Oetli/Prager norms implies all lower level output such as residual norms, etc. All output is to files, and the parameter IUNIT is an integer vector of length 16 giving the unit numbers that SPLIB should use. The summary data is output to IUNIT(1), and the other instrumentation levels will cause output to IUNIT numbers as:

```
instlvl = 1  --> iunit(4) and iunit(11)
instlvl = 2  --> iunit(5) and iunit(12)
instlvl = 3  --> iunit(6) and iunit(13)
instlvl = 4  --> iunit(7) and iunit(14)
instlvl = 5  --> iunit(8) and iunit(15)
instlvl = 6  --> iunit(9) and iunit(16)
```

Unit numbers in IUNIT(2), IUNIT(3), and IUNIT(10) are not used. BEWARE: currently the memory manager in SPLIB writes some output when LISTBL() is called or when error conditions occur. So IUNIT(1) should be opened and connected to some file, even if that file is only /dev/null. BEWARE also that SPLIB opens up files for the graphics output, giving them mnemonic names. Table 2 shows the association and names used for the files. The last letter of a file is "i" or "t" indicating if the convergence history is plotted against iterations or elapsed time, respectivelly. Because so many files can be

created by SPLIB, a small script file RMFILES is included that simply removes all the output files. Note that by specifying INSTLVL = -1 the user can avoid the need to worry about the graphic output from SPLIB, but it is useful in many cases to find how SPLIB is performing. However, when the instrumentation level is greater than zero, SPLIB computes the necessary data by stopping the clock on every iteration and computing the requisite data (since norms such as the true residual norm are not readily available with all the methods). The clock is then restarted before returning to the iterative solver. This can double or even triple the wall clock execution time of the code, and should only be used for exploring the iterative solvers, either to compare them or to understand a lack of convergence.

SPLIB will also write out timing information to the summary results file. The times reported for computing the preconditioner and iterating the solver exclude the time spent in the instrumentation routine, but the breakdown of times for matrix-vector products $w = Ad$, transpose matrix vector products $W = A^T d$, and application of the preconditioner will include calls from the instrumentation routine. Set INSTLVL = 0 if you wish to find the times in those phases that exclude most of the instrumentation costs.

Warning: because of the cost of bringing in page tables and loading caches with both data and code, the first run of SPLIB will take more time than succeeding runs. If you wish to compare the time cost of different methods, make an initial run with an arbitrary method and preconditioner, and then discard its timing data.

7. Recommended Usage. The default values for SPLIB are :

- SOLMETH = 5 (biconjugate gradients stabilized)
- PCMETH = 0 (no preconditioning)
- TOL = 10^{-6}. (residual norm less than 1.0e-6).
- RELTV = .false. (test on absolute residual)
- INSTLVL = 0 (summary output only)

In general, start by using the iterative method Conjugate Gradients Stabilized (SOLMETH = 5) and ILUT preconditioning (PCMETH = 3). Within ILUT, the parameters to set are IPARMS(1) = 0 and RPARMS(1) = 0.0, which gives ILU(0) based on a drop tolerance. If that is not effective, try reading ILUT documentation to see how increasing the integer parameter IPARMS(1) and setting RPARMS(1) can give more robust preconditioning (at the cost of more time and space required by ILUT).

For stopping tests, try 1.0D-6 and RELTV = .true.; this requires six orders of magnitude reduction in the residual norm. MAXITS should be set relatively small; the default value is 100, but MAXITS = 200 should be safe if the system is not too ill-conditioned.

For instrumentation, set INSTLVL = 0 and be sure to provide a file with unit number given in IUNIT(1). That will provide you with a breakdown of where the time was spent in SPLIB, as well as the rusage information (page faults, context switches, etc). More importantly, it will provide a map of the memory used from the WORK array, so you can find out how much was used in the preconditioner.

SPLIB has been compiled and validated on Sparcstations, IBM RS/6000's, SGI Challenge and Power Challenge machines, and DEC Alpha based machines. The pack-

age has already been succesfully used on systems with over 240 000 unknowns and 4.5 million nonzeros. The computations were carried out on an IBM RS/6000 with 128 Mbytes of memory.

8. Gotchas in SPLIB. Here is a summary list of some potential problems in using SPLIB, which a user should beware of. Some have already been mentioned, but they are collected here for convenience.

1. Although CSR data structure allows the entries in a row to appear in any order, for some methods it is assumed that in each row the nonzeros are ordered so that first all the strictly lower triangular entries appear, then the diagonal entry, followed by all the strictly upper triangular entries. This applies to solution methods Gauss-Seidel, Jacobi, and SOR, and to preconditioners SSOR and ILU(-1). Those methods will explictly sort the entries in each row to be in ascending column index order, so the data structure for A (but not the underlying matrix) may be changed on return.
2. Not all combinations of solvers/preconditioners are allowed or even make sense. In particular, preconditioning is not allowed for the linear stationary methods Jacobi, Gauss-Seidel, and SOR because they require access to the matrix entries which in general is not possible for a preconditioned coefficient matrix.
3. The input variable MAXITS is changed by SPLIB, and must be reset if the user wants to call SPLIB more than once in a single execution.
4. Currently the memory manager in SPLIB performs some output, when LISTBL() is called or when error conditions occur. So IUNIT(1) should be opened and connected to some file.
5. SPLIB opens up files for the graphics output, giving them mnemonic names. Those names might collide with files you have already got in your directory, and this can overwrite them. Be sure to avoid using filenames RESI, REST, RELI, RELT, MRESI, MRELI, MREST, MRELI, OETLI, OETLT, ERRORI, and ERRORT, or reassign their values in the routine STRINGS in file UTILITY.F.
6. ECIMGS preconditioning can require, in the worst case, $\mathcal{O}(n^2)$ intermediate storage. When you specify the amount of storage using the input parameter SPACE, set it to the maximum amount you are willing to allow SPLIB to use, even if more is available.
7. Floating point traps: SPLIB computes quantities such as machine epsilon and safe invertible minimum by calls to the LAPACK [2] routines DLAMCH. If you have set floating point traps, SPLIB is almost certain to trigger them. This can be handled by explicitly setting the parameters SMALL and PDR in the codes.
8. Even if the preconditioners successfully complete their factorizations, an error condition is triggered if the initial preconditioned residual is 10^{20} times larger than the unpreconditioned residual (experience shows that the iteration never gives suitable answers when that happens). However, sometimes the preconditioning can still lead to overflows or IEEE NaN's. In that case, the code will continue running, but the solution vector will likely have garbage answers,

usually NaN's. Try a different preconditioner if that happens.

9. "What comes free comes with no guarantee".

9. Examples. The use of SPLIB is now illustrated. The first problem is SAYLR3, a matrix of order 1000 contributed by Paul Saylor to the Harwell/Boeing collection of test matrices. Because the test problem is not supplied with a right hand side vector b, we arbitrarily it to have components $b_i == 1 - 2/i$. SPLIB is run with the settings:

- solmeth = 5 (biconjugate gradients stabilized)
- tol = 10^{-10} (tolerance on residual norm)
- reltv = .true. (use relative residual for stopping test)
- maxits = 100 (allow up to 100 iterations)
- instlvl = 5 (full instrumentation level)
- redo = .true. (recompute preconditioner each time)

On the same run, tests were made with no preconditioning and ILU(s) with $s = 0, 1, 2$. Figure 2 shows the residual norm $\|Ax_k - b\|$ versus the iteration number k. Note that without preconditioning, the residual norm stalls around $\|r_k\| \approx 2.61$, but with preconditioning the residual norm increases quickly to around 10^{12}. Examining the Oetli/Prager measure in Figure 4 shows the preconditioned methods converging to machine tolerance, around 10^{-15} and the unpreconditioned measure converging towards 10^{-6}. The preconditioned residual norms in Figure 3 all start with an initial residual around 10^8. This suggests that the preconditioners are unstable, and that the matrix A itself has troublesome characteristics. Examining the matrix in more detail, rows 998 and 999 are exact negatives of each other and so A is actually singular. Note that simply examining the residual norms alone would not reveal this problem, but with a combination of the preconditioned residual and Oetli/Prager measure strong evidence was provided that the system was intrinsically difficult, and that the poor results are not a feature of the biconjugate gradient stabilized algorithm.

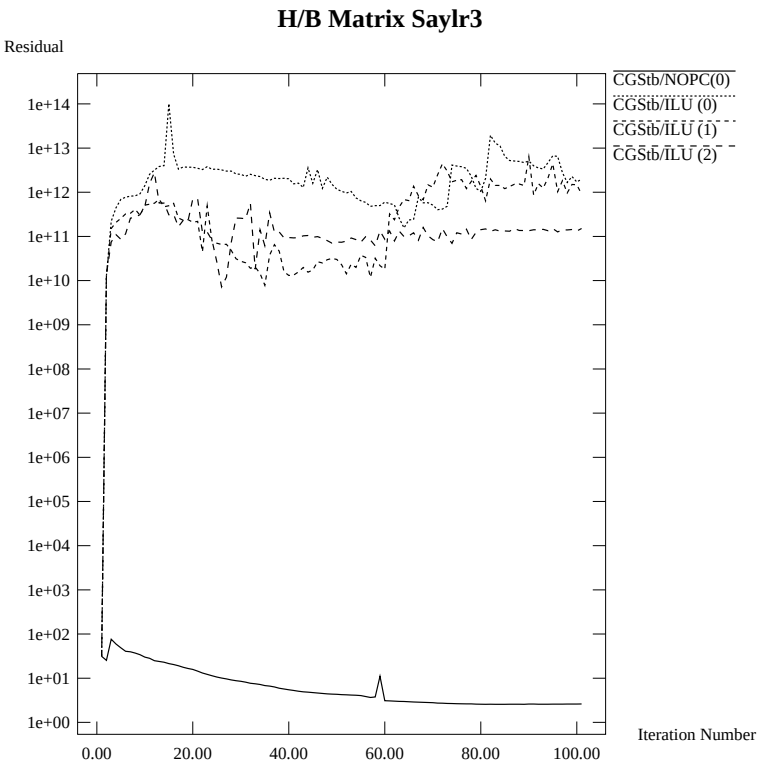

FIG. 2. *Residual Norm for* SAYLR3

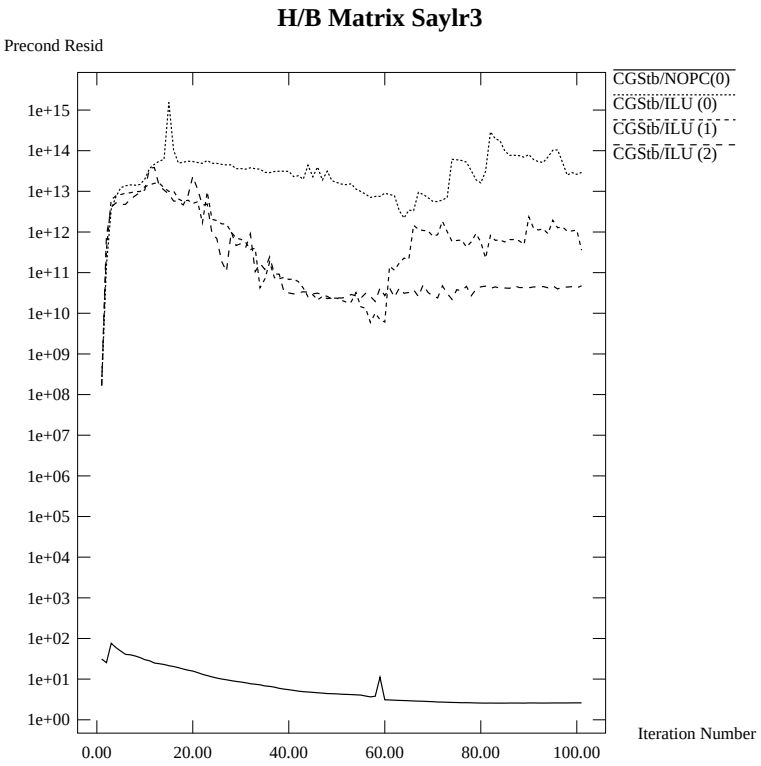

FIG. 3. *Preconditioned Residual Norm for* SAYLR3

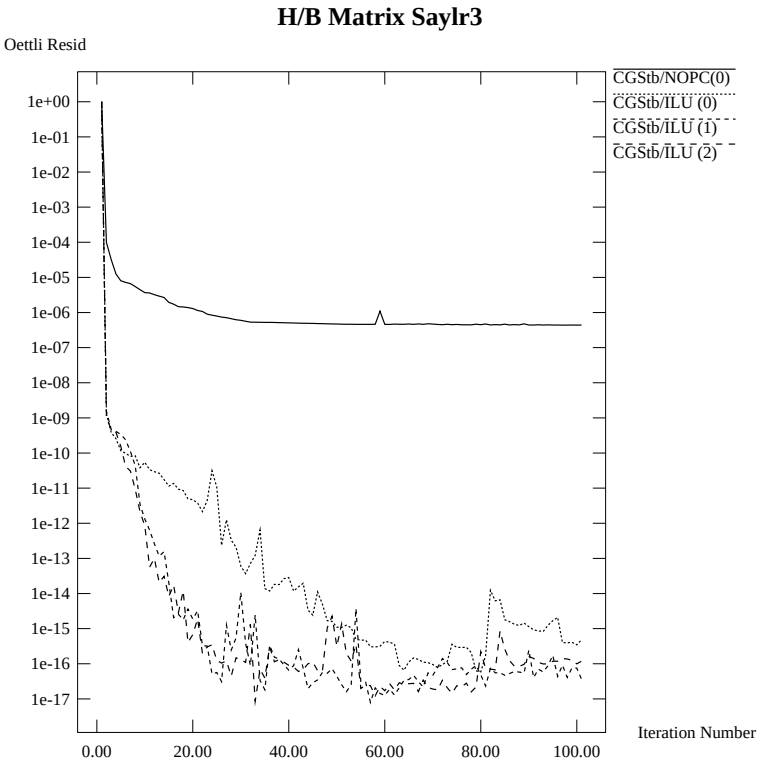

FIG. 4. *Oetli/Prager for* SAYLR3

The second example is a linear system with a coefficient matrix from the convection–diffusion operator on a finite element mesh in two dimensions. The system has 11 099 unknowns and 254 885 nonzeros, and SPLIB was run with ILU(0) and ILU0 preconditioners; recall that ILU0 creates a preconditioner that differs from the coefficient matrix in the diagonal of pivots, and so requires less storage. The summary output file from the run looks like:

```
>>>>>>>>>>>>>>>>>>>>>>>>> Output from SPLIB <<<<<<<<<<<<<<<<<<<<<<<<<<<<

Order of matrix =  11099
Number of nonzeros   254885
Iterative method used:    CGStb with     ILU  preconditioning
Levels of fill allowed in precond:   0
Acceleration parameter:     1.00
Tolerance on residual = 1.0000E-10
Max allowed iterations:    200
Right hand side norm:  1.05195E+02
Successful return; tolerance achieved
Iterations performed:   60
Final residual 2-norm:  4.78545E-10

   Time for A*x (incl. instrumentation):  6.66130E+00
```

```
   Time for At*x (incl. instrumentation):  0.00000E+00
   Time for LU-solve (incl. instrumentation):  9.97217E+00
   Time to compute preconditioner:  8.04046E-01
   Time to iterate solver:  1.15827E+01
   Total time:  1.23868E+01

------------------------------------------------------------------------
*** listbl ***  called from routine : "summar"
                list of all arrays allocated at this point
                ------------------------------------------

    no.      table        length type        begins at    ends at   origin

     1 -     rwptr         11100   integer           1     11100     readmt
     2 -     colind       254885   integer       11101    265986     readmt
     3 -     a            254885   real         265987    775756     readmt
     4 -     x             11099   real         775757    797954     driver
     5 -     b             11099   real         797955    820152     driver
     6 -     sptime            8   real         820153    820168     splib
     7 -     r             11099   real         820169    842366     splib
     8 -     uptr          11099   integer      842367    853466     (m)ilu
     9 -     jlu          254886   integer      853467   1108352     (m)ilu
    10 -     lu           254886   real        1108353   1618124     (m)ilu
    11 -     itvecs        66594   real        1618125   1751312     cgstab
------------------------------------------------------------------------

  User cpu time  :  0.18347D+02
  System cpu time:  0.15970D+00
  Total time     :  0.18507D+02
   Maximum resident set size         :         0
   Text memory in shared libraries   :      5280
   Unshared data space kb/s          :         0
   Unused quantity (not on IBM)      :         0
   Page faults without I/O activity  :         0
   Page faults with I/O activity     :         0
   Number of swaps out of main memory:         0
   Number of file system input       :         0
   Number of file system output      :         0
   Number of IPC messages sent       :         7
   Number of IPC messages received   :         0
   Number of signals delivered       :         0
   Voluntary context switches        :         0
   Involuntary context switches      :         0
```

```
>>>>>>>>>>>>>>>>>>>>>>>> End Output from SPLIB <<<<<<<<<<<<<<<<<<<<<<<<


>>>>>>>>>>>>>>>>>>>>>>>> Output from SPLIB <<<<<<<<<<<<<<<<<<<<<<<<

Order of matrix =  11099
Number of nonzeros   254885
Iterative method used:    CGStb with     ILU0 preconditioning
Levels of fill allowed in precond:   0
Acceleration parameter:     1.00
Tolerance on residual = 1.0000E-10
Max allowed iterations:    200
Right hand side norm:  1.05195E+02
*** Maximum allowed iterations reached
Iterations performed:  200
Final residual 2-norm:  1.27926E+01

   Time for A*x (incl. instrumentation):  1.30702E+01
   Time for At*x (incl. instrumentation):  0.00000E+00
   Time for LU-solve (incl. instrumentation):  1.77144E+01
   Time to compute preconditioner:  4.78518E-01
   Time to iterate solver:  2.14825E+01
   Total time:  2.19610E+01

-------------------------------------------------------------------------
*** listbl ***  called from routine : "summar"
                list of all arrays allocated at this point
                ------------------------------------------

    no.      table        length type        begins at    ends at   origin

     1 -     rwptr       11100   integer           1     11100     readmt
     2 -     colind     254885   integer       11101    265986     readmt
     3 -     a          254885   real         265987    775756     readmt
     4 -     x           11099   real         775757    797954     driver
     5 -     b           11099   real         797955    820152     driver
     6 -     sptime          8   real         820153    820168     splib
     7 -     r           11099   real         820169    842366     splib
     8 -     jlu         11099   integer      842367    853466     ilu0
     9 -     lu          11099   real         853467    875664     ilu0
    10 -     itvecs      66594   real         875665   1008852     cgstab
```

```
------------------------------------------------------------------------

  User cpu time   :  0.33546D+02
  System cpu time:  0.20035D+00
  Total time      :  0.33746D+02
   Maximum resident set size          :          0
   Text memory in shared libraries    :       1456
   Unshared data space kb/s           :          0
   Unused quantity (not on IBM)       :          0
   Page faults without I/O activity   :          0
   Page faults with I/O activity      :          0
   Number of swaps out of main memory:          0
   Number of file system input        :          0
   Number of file system output       :          0
   Number of IPC messages sent        :          0
   Number of IPC messages received    :          0
   Number of signals delivered        :          0
   Voluntary context switches         :          0
   Involuntary context switches       :          0

>>>>>>>>>>>>>>>>>>>>>>>> End Output from SPLIB <<<<<<<<<<<<<<<<<<<<<<<<
```

On the first run, the first 9 lines echo the input parameters and simple information about the system. Line 9 shows the number of nonzeros stored in the preconditioner array, which is 1 more than that required by the coefficient matrix A because the preconditioner is in MSR format. Lines 10-12 indicate the problem was solved to the required tolerance. Lines 14-19 give a summary of the timings (recall that the times shown for matrix-vector products and preconditioning include the calls from the instrumentation routine, but the other times do not.) 1 - rwptr 11100 integer 1 11100 readmt 2 - colind 254885 integer 11101 265986 readmt 3 - a 254885 real 265987 775756 readmt 4 - x 11099 real 775757 797954 driver 5 - b 11099 real 797955 820152 driver 6 - sptime 8 real 820153 820168 splib 7 - r 11099 real 820169 842366 splib 8 - uptr 11099 integer 842367 853466 (m)ilu 9 - jlu 254886 integer 853467 1108352 (m)ilu 10 - lu 254886 real 1108353 1618124 (m)ilu 11 - itvecs 66594 real 1618125 1751312 cgstab The block of output beginning with "listbl" show the memory usage. For completeness, this shows the storage required for the coefficient matrix and vectors x and b as well as the storage used by the solver. This is shown in lines for RWPTR, COLIND, A, X, and B. The array SPTIME is used internally in SPLIB to store execution times, R is the residual vector, and UPTR, JLU, and LU are arrays used for the preconditioner data structure. Finally, ITVECS is the array used for vectors required by the iterative solver, exclusive of any used by the instrumentation routine. Because the preconditioner requires as much storage as the matrix A itself, solving this linear system required 114% more storage than that required just to represent the linear system. The final block is the output from PRTRES, which indicates that there were no page faults or context switches during

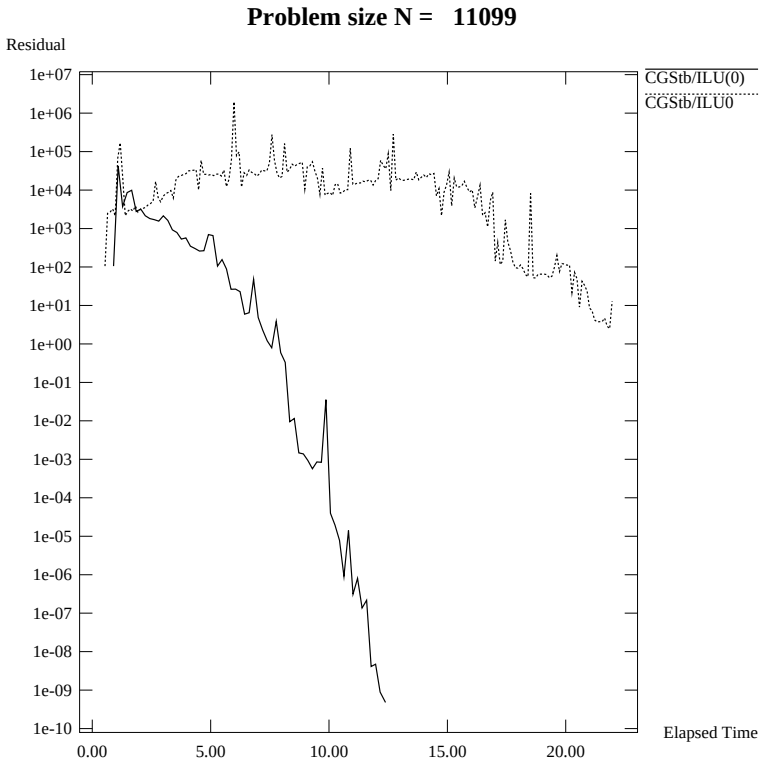

FIG. 5. *Residual History Norm for CFD Problem*

the run. A high number of either of those can indicate that the timings were seriously perturbed, perhaps by other jobs running on the system at the same time.

The second run shows similar information for the ILU0 preconditioner. Now the additional storage to solve is only 23% that required to represent the system, but the solver failed to converge in the allowed number of iterations. The convergence history of the residual norms against time is shown in Figure 5.

As these brief examples show, SPLIB can provide information about the solvers and preconditioners used, as well as their interaction with the machine architecture.

10. Future Work on SPLIB. Further work on the package will make it more valuable, and provide excellent training for graduate students starting to work on scientific computing problems. In particular, projects include:

1) Development of a pC++ version. Simultaneous development of the package in Fortran and pC++ potentially doubles the applications and research projects possible. Fortran is valuable because its compilers create efficient code that allows the package to be directly used in applications programs. C++ and parallel C++ will further extend the area of application.

2) Introducing parallelism. For most applications, effective parallelism comes from partitioning the data based on physical characteristics of the problem. For distributed memory machines, such a partitioning must be introduced from the start of the program. Nevertheless, having the linear solver package able to utilize various parallel schemes allows the user to experiment with parallelism in the most time-consuming part of the

code, with full introduction of partitioning deferred until a good approach is found. Note that the partitioning is more logical than physical in the case of shared memory multiprocessors, but is still important for reasons of data locality.

3) An automated data partitioning scheme. This will be different from most in that it will not just partition the matrix, but will seek to simultaneously find a conformal partitioning for the matrix and its preconditioner, with fill-in possible via structural or numerical strategies. The partitioning of the rest of the data will be conformal with that of the matrices. Until this is developed, the library will simply assume the user provides the partitioning for parallelism, and has explicitly reordered the matrix correspondingly.

4) Implementation of additional visualization tools for the matrices, their preconditioners, and the algorithms used. Among the research parts of this are ways of visualizing truly huge sparse systems, visualization of data structures for storing the matrices, and displays of the fill-in process . Furthermore, the current displays should be extended to more fully demonstrate the the performance of the algorithms. Ultimately this should be integrated with a parallel performance visualization method. A visualization tool called EMILY [17] has been developed, using color to indicate magnitudes of nonzero entries and providing zoom capabilities to expand mouse-selected principal submatrices.

5) Data structure conversion is needed, that is, "automatically" picking data structures for the matrices and their preconditioners based on characteristics of the systems (number of nonzeros per row, e.g.) and the solver/ preconditioner pairs chosen. This includes developing cost models of whether it is cheaper to convert or to just stick with a suboptimal data structure, etc.

6) Implementation of reorderings. It is well known that the performance of linear solvers depends on the ordering of the unknowns in the system, and the amount of fill-in and quality of preconditioners are especially sensitive to orderings. However, most work on reordering a linear system is targeted towards either reduced fill-in (for direct methods), or for parallelism. There is little research on developing reordering strategies based on improving the performance of the combination of iterative solver and preconditioner.

7) Addition of other methods. In particular, row projection methods have good potential for several applications areas, but the difficulty of of implementing them has daunted many users from experimenting with them. This library will next be extended to allow implemenation of (block) Kaczmarz and Cimmino methods, with CG acceleration.

REFERENCES

[1] E. ANDERSON, *Parallel implementation of preconditioned conjugate gradient methods for solving sparse systems of linear equations*, Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., 1988.

[2] E. ANDERSON ET AL., *LAPACK Users' Guide*, SIAM, Philadelphia, PA, 1992.

[3] M. ARIOLI, I. DUFF, J. NOAILLES, AND D. RUIZ, *A block projection method for general sparse matrices*, SIAM Journal of Scientific and Statistical Computing, 13 (1992), pp. 47–70.

[4] O. Axelsson, *Iterative Solution Methods*, Cambridge University Press, Cambridge, UK, 1 ed., 1994.

[5] O. Axelsson and G. Lindskog, *On the eigenvalue distribution of a class of preconditioning methods*, Numer. Math., 48 (1986), pp. 479–498.

[6] R. Barrett et al., *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1 ed., 1994.

[7] ———, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM, Philadelphia, 1 ed., 1994.

[8] R. Bramley, X. Wang, and D. Pelletier, *Orthogonalization based iterative methods for generalized Stokes problems*, in Solution Techniques for Large–Scale CFD Problems, W. G. Habashi, ed., Centre de Recherce en Calcul Applique, 1994.

[9] E. J. Craig, *The N-step iterations procedures*, J. Math. Physics, 34 (1955), pp. 64–73.

[10] H. V. der Vorst, *Bi–CGSTAB: A fast and smoothly converging variant of Bi–CG for the solution of nonsymmetric linear systems*, SIAM Journal of Scientific and Statistical Computing, 13 (1992), pp. 631–644.

[11] R. Fletcher, *Conjugate gradient methods for indefinite systems*, in Lecture Notes in Mathematics, No. 506, Springer-Verlag, Heidelberg, 1976, pp. 73–89.

[12] R. Freund, *A transpose–free quasi–minimum residual algorithm for non–Hermitian linear systems*, SIAM J. Sci. Stat. Comp.., 14 (1993), pp. 470–482.

[13] I. Gustafsson, *Modified incomplete Cholesky methods*, in Preconditioning Methods: Theory and Applications, D. Evans, ed., New York, 1983, Gordon and Breach, pp. 265–293.

[14] L. Hageman and D. Young, *Applied Iterative Methods*, Academic Press, New York, 1 ed., 1981.

[15] M. Heroux, P. Vu, and C. Yang, *A parallel preconditioned conjugate gradient package for solving sparse linear systems on a Cray Y-MP*, Applied Numerical Mathematics, 8 (1991), pp. 93–115.

[16] M. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, Journal of Research National Bureau of Standards, 49 (1952), pp. 409–436.

[17] T. Loos and R. Bramley, *Emily: A visualization utility for large matrices*, Tech. Rep. 412, Indiana University–Bloomington, Bloomington, IN 47405, 1994. Submitted for publication in IEEE Visualization, February 1995.

[18] J. Meijerink and H. van der Vorst, *An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix*, Math. Comp., 31 (1977), pp. 148–162.

[19] W. Oetli and W. Prager, *Compatibility of approximate solution of linear equations with given error bounds for coefficients and right hand sides*, Numer. Math, 6 (1964), pp. 405–409.

[20] G. V. Paolini and G. di Brozolo, *Data structures to vectorize CG algorithms for general sparsity patterns*, BIT, 29 (1989), pp. 703–718.

[21] Y. Saad, *SPARSKIT: a basic tool kit for sparse matrix computations*, tech. rep., Center for Supercomputing Research and Development, University of Illinois, Urbana, Illinois, 1990.

[22] Y. Saad and M. Schultz, *Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, SIAM J. Sci. Stat. Comput., 7 (1986), pp. 856–869.

[23] P. Sonneveld, *Cgs, a fast Lanczos–type solver for nonsymmetric linear systems*, SIAM J. Sci. Stat. Comp., 10 (1989), pp. 36–52.

[24] P. Vinsome, *Orthomin, an iterative method for solving sparse sets of simultaneous linear equations*, Proc. Fourth Symp. on Reservoir Simulation, (1976), pp. 149–159.

[25] X. Wang, *Incomplete Factorization Preconditioning for Linear Least Squares Problems*, PhD thesis, University of Illinois Urbana-Champaign, 1993. Also available as Tech. Rep. UIUCDCS-R-93-1834, Computer Science Department, University of Illinois – Urbana.

[26] X. Wang, K. A. Gallivan, and R. Bramley, *CIMGS: A incomplete orthogonalization preconditioner*, Tech. Rep. 393, Indiana University–Bloomington, Bloomington, IN 47405, 1994. Accepted for publication in SIAM J. Sci. Comp.