Sparse User's Guide
A Sparse Linear Equation Solver
Version 1.4

Ken Kundert
kundert@users.sourceforge.net

June 2003

1 INTRODUCTION

Sparse1.4 is a flexible package of subroutines written in C that are used to quickly and accurately solve large sparse systems of linear equations. The package is able to handle arbitrary real and complex square matrix equations. Besides being able to solve linear systems, it is also able to quickly solve transposed systems, find determinants, and estimate errors due to ill-conditioning in the system of equations and instability in the computations. *Sparse* also provides a test program that is able read matrix equations from a file, solve them, and print useful information about the equation and its solution.

Sparse1.4 is generally competitive to other popular direct method sparse matrix packages when solving many matrices of similar structure. *Sparse* does not require or assume symmetry and is able to perform numerical pivoting to avoid unnecessary error in the solution. It handles its own memory allocation, which allows the user to forgo the hassle of providing adequate memory. It also has a natural, flexible, and efficient interface to the calling program.

Sparse was originally written for use in circuit simulators and is particularly apt at handling node- and modified-node admittance matrices. The systems of linear generated in a circuit simulator stem from solving large systems of nonlinear equations using Newton's method and integrating large stiff systems of ordinary differential equations. As iterative approaches are employed, *Sparse* is optimized for repeated solving matrices with the same structure. However, *Sparse* is also suitable for other uses as well.

1.1 Features of Sparse1.4

Beyond the basic capability of being able to create, factor and solve systems of equations, this package features several other capabilities that enhance its utility. These features are:

- Ability to handle both real and complex systems of equations. Both types may resident and active at the same time. In fact, the same matrix may alternate between being real and complex.
- Ability to quickly solve the transposed system. This feature is useful when computing the sensitivity of a circuit using the adjoint method.
- Memory for elements in the matrix is allocated dynamically, so the size of the matrix is only limited by the amount of memory available to *Sparse* and the range of the integer data type, which is used to hold matrix indices.
- Ability to efficiently compute the condition number of the matrix and an a posteriori estimate of the error caused by growth in the size of the elements during the factorization.
- Much of the matrix initialization can be performed by *Sparse*, providing advantages in speed and simplified coding of the calling program.
- Ability to preorder modified node admittance matrices to enhance accuracy and speed.
- Ability to exploit sparsity in the right-hand side vector to reduce unnecessary computation.
- Ability to scale matrices prior to factoring to reduce uncertainty in the solution.
- The ability to create and build a matrix without knowing its final size.
- The ability to add elements, and rows and columns, to a matrix after the matrix has been reordered.
- The ability to delete rows and columns from a matrix.
- The ability to strip the fill-ins from a matrix. This can improve the efficiency of a subsequent reordering.
- The ability to handle matrices that have rows and columns missing from their input description.
- Ability to output the matrix in forms readable by either by people or by the *Sparse* package. Basic statistics on the matrix can also be output.
- By default all arithmetic operations and number storage use double precision. Thus, *Sparse* usually gives accurate results, even on highly ill-conditioned systems. If so desired, *Sparse* can be easily configured to use single precision arithmetic.

1.2 Enhancements of Sparse1.4 over Sparse1.3

Sparse1.3 has been used for 18 years, particularly in the EDA (electronics design aids) community. During this time it has worked well with very few problems. With *Sparse1.4* the emphasis is not to make a big enhancement, rather to simply update it. In 18 years the languages have matured and things have become more standard. The changes are intended to make *Sparse1.4* conform to those standards. One set of standards is ANSI C and C++. At the time *Sparse* was written, C++ was very immature and ANSI C was brand new and not widely adopted. With *Sparse1.4* much of the cruft that was present to support non-standard compilers has been cleaned out. In addition, the files meant to be used in other programs (all files except the test program) have been checked to assure that they can be compiled by a C++ compiler.

Finally, *Sparse1.4* has formally been made an open-source project. You can find the source code on `sourcforge.net` and from there you can coolaborate on the development of *Sparse*.

1.3 License Information

2 PRIMER

2.1 Solving Matrix Equations

Sparse contains a collection of C subprograms that can be used to solve linear algebraic systems of equations. These systems are of the form:

$$Ax = b \tag{1}$$

where A is an $n \times n$ matrix, x is the vector of n unknowns and b is the vector of n right-hand side terms. Through out this package A is denoted *Matrix*, x is denoted *Solution* and b is denoted *RHS* (for right-hand side). The system is solved using LU factorization, so the actual solution process is broken into two steps, the factorization or decomposition of the matrix, performed by *spFactor*(), and the forward and backward substitution, performed by *spSolve*(). *spFactor*() factors the given matrix into upper and lower triangular matrices independent of the right-hand side. Once this is done, the solution vector can be determined efficiently for any number of right-hand sides without refactoring the matrix.

This package exploits the fact that large matrices usually are sparse by not storing or operating on elements in the matrix that are zero. Storing zero elements is avoided by organizing the matrix into an orthogonal linked-list. Thus, to access an element if only its indices are known requires stepping through the list, which is slow. This function is performed by the routine *spGetElement*(). It is used to initially enter data into a matrix and to build the linked-list. Because it is common to repeatedly solve matrices with identical zero/nonzero structure, it is possible to reuse the linked-list. Thus, the linked list is left in memory and the element values are simply cleared by *spClear*() before the linked-list is reused. To speed the entering of the element values into successive matrices, *spGetElement*() returns a pointer to the element in the matrix. This pointer can then be used to place data directly into the matrix without having to traverse through the linked-list.

The order in which the rows and columns of the matrix are factored is very important. It directly affects the amount of time required for the factorization and the forward and backward substitution. It also affects the accuracy of the result. The process of choosing this order is time consuming, but fortunately it usually only has to be done once for each particular matrix structure encountered. When a matrix with a new zero/nonzero structure is to be factored, it is done by using *spOrderAndFactor*(). Subsequent matrices of the same structure are factored with *spFactor*(). The latter routine does not have the ability to reorder matrix, but it is considerably faster. It may be that a order chosen may be unsuitable for subsequent factorizations. If this is known to be true a priori, it is possible to use *spOrderAndFactor*() for the subsequent factorizations, with a noticeable speed penalty. *spOrderAndFactor*() monitors the numerical stability of the factorization and will modify an existing ordering to maintain stability. Otherwise, an a posteriori measure of the numerical stability of the factorization can be computed, and the matrix reordered if necessary.

The *Sparse* routines allow several matrices of different structures to be resident at once. When a matrix of a new structure is encountered, the user calls *spCreate*(). This routine creates the basic frame for the linked-list and returns a pointer to this frame. This pointer is then passed as an argument to the other *Sparse* routines to indicate which matrix is to be operated on. The number of matrices that can be kept in memory at once is only limited by the amount of memory available to the user and the size of the matrices. When a matrix frame is no longer needed, the memory can be reclaimed by calling *spDestroy*().

A more complete discussion of sparse systems of equations, methods for solving them, their error mechanisms, and the algorithms used in *Sparse* can be found in Kundert[kundert86]. A particular emphasis is placed on matrices resulting from circuit simulators.

2.2 Error Control

There are two separate mechanisms that can cause errors during the factoring and solution of a system of equations. The first is ill-conditioning in the system. A system of equations is ill-conditioned if the solution is excessively sensitive to disturbances in the input data, which occurs when the system is nearly singular. If a system is ill-conditioned then uncertainty in the result is unavoidable, even if A is accurately factored into L and U. When ill-conditioning is a problem, the problem as stated is probably ill-posed and the system should be reformulated such that it is not so ill-conditioned. It is possible to measure the ill-conditioning of matrix using *spCondition*(). This function returns an estimate of the reciprocal of the condition number of the matrix ($\kappa(A)$)[strang80]. The condition number can be used when computing a bound on the error in the solution using the following inequality[golub86].

$$\frac{\| \delta x \|}{\| x \|} \leq \kappa(A) \left(\frac{\| \delta A \|}{\| A \|} + \frac{\| \delta b \|}{\| b \|} \right) + \text{higher order terms} \tag{2}$$

where δA and δb are the uncertainties in the matrix and right-hand side vector and are assumed small.

The second mechanism that causes uncertainty is the build up of roundoff error. Roundoff error can become excessive if there is sufficient growth in the size of the elements during the factorization. Growth is controlled by careful pivoting. In *Sparse*, the pivoting is controlled by the relative threshold parameter. In conventional full matrix techniques the pivot is chosen to be the largest element in a column. When working with sparse matrices it is important to choose pivots to minimize the reduction in sparsity. The best pivot to retain sparsity is often not the best pivot to retain accuracy. Thus, some compromise must be made. In threshold pivoting, as used in this package, the best pivot to retain sparsity is used unless it is smaller than the relative threshold times the largest element in the column. Thus, a relative threshold close to one emphasizes accuracy so it will produce a minimum amount of growth, unfortunately it also slows the factorization. A very small relative threshold emphasizes maintenance of sparsity and so speeds the factorization, but can result in a large amount of growth. In our experience, we have

found that a relative threshold of 0.001 seems to result in a satisfactory compromise between speed and accuracy, though other authors suggest a more conservative value of 0.1[duff86].

The growth that occurred during a factorization can be computed by taking the ratio of the largest matrix element in any stage of the factorization to the largest matrix element before factorization. The two numbers are estimated using *spLargestElement*(). If the growth is found to be excessive after *spOrderAndFactor*(), then the relative threshold should be increased and the matrix reconstructed and refactored. Once the matrix has been ordered and factored without suffering too much growth, the amount of growth that occurred should be recorded. If, on subsequent factorizations, as performed by *spFactor*(), the amount of growth becomes significantly larger, then the matrix should be reconstructed and reordered using the same relative threshold with *spOrderAndFactor*(). If the growth is still excessive, then the relative threshold should be raised again.

2.3 Building the Matrix

It is not necessary to specify the size of the matrix before beginning to add elements to it. When the compiler option EXPANDABLE is turned on it is possible to initially specify the size of the matrix to any size equal to or smaller than the final size of the matrix. Specifically, the matrix size may be initially specified as zero. If this is done then, as the elements are entered into the matrix, the matrix is enlarged as needed. This feature is particularly useful in circuit simulators because it allows the building of the matrix as the circuit description is parsed. Note that once the matrix has been reordered by the routines *spMNA_Preorder*(), *spFactor*() or *spOrderAndFactor*() the size of the matrix becomes fixed and may no longer be enlarged unless the compiler option TRANSLATE is enabled.

The TRANSLATE option allows *Sparse* to translate a non-packed set of row and column numbers to an internal packed set. In other words, there may be rows and columns missing from the external description of the matrix. This feature provides two benefits. First, if two matrices are identical in structure, except for a few missing rows and columns in one, then the TRANSLATE option allows them to be treated identically. Similarly, rows and columns may be deleted from a matrix after it has been built and operated upon. Deletion of rows and columns is performed by the function *spDeleteRowAndCol*(). Second, it allows the use of the functions *spGetElement*(), *spGetAdmittance*(), *spGetQuad*(), and *spGetOnes*() after the matrix has been reordered. These functions access the matrix by using row and column indices, which have to be translated to internal indices once the matrix is reordered. Thus, when TRANSLATE is used in conjunction with the EXPANDABLE option, rows and columns may be added to a matrix after it has been reordered.

Another provided feature that is useful with circuit simulators is the ability to add elements to the matrix in row zero or column zero. These elements will have no affect on the matrix or the results. The benefit of this is that when working with a nodal formulation, grounded components do not have to be treated special when building the matrix.

2.4 Initializing the Matrix

Once a matrix has been factored, it is necessary to clear the matrix before it can be reloaded with new values. The straight forward way of doing that is to call *spClear*(), which sets the value of every element in the matrix to zero. *Sparse* also provides a more flexible way to clear the matrix. Using *spInitialize*(), it is possible to clear and reload at least part of the matrix in one step.

Sparse allows the user to keep initialization information with each structurally nonzero matrix element. Each element has a pointer that is set and used by the user. The user can set this pointer using *spInstallInitInfo*() and may read it using *spGetInitInfo*(). The function *spInitialize*() is a user customizable way to initialize the matrix. Passed to this routine is a function pointer. *spInitialize*() sweeps through every element in the matrix and checks the *pInitInfo* pointer (the user supplied pointer). If the *pInitInfo* is NULL, which is true unless the user changes it (always true for fill-ins), then the element is zeroed. Otherwise, the function pointer is called and passed the *pInitInfo* pointer as well as the element pointer and the external row and column numbers, allowing the user to initialize the matrix element and the right-hand side.

Why *spInitialize*() would be used over *spClear*() can be illustrated by way of an example. Consider a circuit simulator that handles linear and nonlinear resistors and capacitors performing a transient analysis. For the linear resistors, a constant value is loaded into the matrix at each time step and for each Newton iteration. For the linear capacitor, a value is loaded into the matrix that is constant over Newton iterations, but is a function of the time step and the integration method. The nonlinear components contribute values to the matrix that change on every time step and Newton iteration.

Sparse allows the user to attach a data structure to each element in the matrix. For this example, the user might attach a structure that held several pieces of information, such as the conductance of the linear resistor, the capacitance of the linear capacitor, the capacitance of the nonlinear capacitor, and perhaps past values of capacitances. The user also provides a subroutine to *spInitialize*() that is called for each user-created element in the matrix. This routine would, using the information in the attached data structure, initialize the matrix element and perhaps the right-hand side vector.

In this example, the user supplied routine might load the linear conductance into the matrix and multiply it by some voltage to find a current that could be loaded into the right-hand side vector. For the capacitors, the routine would first apply an integration method and then load the matrix and the right-hand side.

This approach is useful for two reasons. First, much of the work of the device code in the simulator can be off-loaded onto the matrix package. Since there are usually many devices, this usually results overall in a simpler system. Second, the integration method can be hidden from the simulator device code. Thus the integration method can be changed simply by changing the routine handed to *spInitialize*(), resulting in a much cleaner and more easily maintained simulator.

2.5 Indices

By far the most common errors made when using *Sparse* are related to array indices. *Sparse* itself contributes to the problem by having several different indexing schemes. There are three different options that affect index bounds or the way indices are interpreted. The first is ARRAY_OFFSET, which only affects array indices. ARRAY_OFFSET is a compiler flag that selects whether arrays start at index zero or index one. Note that if ARRAY_OFFSET is zero then *RHS*[0] corresponds to row one in the matrix and *Solution*[0] corresponds to column one. Further note that when ARRAY_OFFSET is set to one, then the allocated length of the arrays handed to the *Sparse* routines should be at least the external size of the matrix plus one. The main utility of ARRAY_OFFSET is that it allows natural array indexing when *Sparse* is coupled to programs in other languages. For example; in FORTRAN arrays always start at one whereas in C array always start at zero. Thus the first entry in a FORTRAN array corresponds to the zero'th entry in a C array. Setting ARRAY_OFFSET to zero allows the arrays in FORTRAN to start at one rather than two. For the rest of this discussion, assume that ARRAY_OFFSET is set so that arrays start at one in the program that calls *Sparse*.

The second option that affects indices is EXPANDABLE. When EXPANDABLE is set false the upper bound on array and matrix indices is *Size*, where *Size* is a parameter handed to *spCreate*(). When EXPANDABLE set true, then there is essentially no upper bound on array indices. Indeed, the size of the matrix is determined by the largest row or column number handed to *Sparse*. The upper bound on the array indices then equals the final size determined by *Sparse*. This size can be determined by calling *spGetSize*().

The final option that affects indices is TRANSLATE. This option was provided to allow row and columns to be deleted, but it also allows row and column numbers to be missing from the input description for a matrix. This means that the size of the matrix is not determined by the largest row or column number entered into the matrix. Rather, the size is determined by the total number of rows or column entered. For example, if the elements [2,3], [5,3], and [7,2] are entered into the matrix, the internal size of the matrix becomes four while the external size is seven. The internal size equals the number of rows and columns in the matrix while the external size equals the largest row or column number entered into the matrix. Note that if a row is entered into the matrix, then its corresponding column is also entered, and vice versa. The indices used in the *RHS* and *Solution* vectors correspond to the row and column indices in the matrix. Thus, for this example, valid data is expected in *RHS* at locations 2, 3, 5 and 7. Data at other locations is ignored. Similarly, valid data is returned in *Solution* at locations 2, 3, 5, and 7. The other locations are left unmolested. This shows that the length of the arrays correspond to the external size of the matrix. Again, this value can be determined by *spGetSize*().

2.6 Configuring Sparse

It is possible at compile-time to customize *Sparse* for your particular application. This is done by changing the compiler options, which are kept in the personality file, **spConfig.h**. There are three classes of choices available. First are the *Sparse* options, which specify the dominant personality characteristics, such as if real and/or complex systems of equations are to be handled. The second class is the *Sparse* constants, such as the default pivot threshold and the amount of memory initially allocated per matrix. The last class is the machine constants. These numbers must be updated when *Sparse* is ported to another machine.

As an aid in the setup and testing of *Sparse* a test routine and several test matrices and their solutions have been provided. The test routine is capable of reading files generated by *spFileMatrix*() and *spFileVector*().

By default *Sparse* stores all real numbers and performs all computations using double precision arithmetic. This can be changed by changing the definition of *spREAL* from **double** to **float**. *spREAL* is defined in **spMatrix.h**.

3 INTRODUCTION TO THE SPARSE ROUTINES

In this section the routines are grouped by function and briefly described.

3.1 Creating the Matrix

spCreate()
Allocates and initializes the data structure for a matrix. Necessarily the first routine run for any particular matrix.

spDestroy()
Destroys the data structure for a matrix and frees the memory.

spSetReal()
spSetComplex()
These routines toggle a flag internal to *Sparse* that indicates that the matrix is either real or complex. This is useful if both real and complex matrices of identical structure are expected.

3.2 Building the Matrix

spGetElement()
Assures that the specified element exists in the matrix data structure and returns a pointer to it.

spGetAdmittance()
spGetQuad()
spGetOnes()
These routines add a group of four related elements to the matrix. *spGetAdmittance*() adds the four elements associated with a two terminal admittance. *spGetQuad*() is a more general routine that is useful for entering controlled sources to the matrix. *spGetOnes*() adds the four structural ones to the matrix that are often encountered with elements that do not have admittance representations.

spDeleteRowAndCol()
This function is used to delete a row and column from the matrix.

3.3 Clearing the Matrix

spClear()
Sets every element in the matrix to zero.

spInitialize()
Runs a user provided initialization routine on each element in the matrix. This routine would be used in lieu of *spClear*().

spGetInitInfo()
spInstallInitInfo()
These routines allow the user to install and read a user-provided pointer to initialization data for a particular matrix element.

spStripFills()
This routine returns a matrix to a semi-virgin state by removing all fill-ins. This can be useful if a matrix is to be reordered and it has changed significantly since it was previously ordered. This may be the case if a few rows and columns have been added or deleted or if the previous ordering was done on a matrix that was numerically quite different than the matrix currently being factored. Stripping and reordering a matrix may speed subsequent factorization if the current ordering is inferior, whereas simply reordering will generally only enhance accuracy and not speed.

3.4 Placing Data in the Matrix

spADD_REAL_ELEMENT()
spADD_IMAG_ELEMENT()
spADD_COMPLEX_ELEMENT()
Adds a value to a particular matrix element.

spADD_REAL_QUAD()
spADD_IMAG_QUAD()
spADD_COMPLEX_QUAD()
Adds a value to a group of four matrix elements.

3.5 Influencing the Factorization

spMNA_Preorder()
This routine preorders modified node admittance matrices so that *Sparse* can take full advantage of their structure. In particular, this routine tries to remove zeros from the diagonal so that diagonal pivoting can be used more successfully.

spPartition()
Sparse partitions the matrix in an attempt to make *spFactor*() run as fast as possible. The partitioning is a relatively expensive operation that is not needed in all cases. *spPartition*() allows the user specify a simpler and faster partitioning.

spScale()
It is sometimes desirable to scale the rows and columns of a matrix in to achieve a better pivoting order. This is particularly true in modified node admittance matrices, where the size of the elements in a matrix can easily vary through ten to twelve orders of magnitude. This routine performs scaling on a matrix.

3.6 Factoring the Matrix

spOrderAndFactor()
This routine chooses a pivot order for the matrix and factors it into LU form. It handles both the initial factorization and subsequent factorizations when a reordering is desired.

spFactor()
Factors a matrix that has already been ordered by *spOrderAndFactor*(). If *spFactor*() is passed a matrix that needs ordering, it will automatically pass the matrix to *spOrderAndFactor*().

3.7 Solving the Matrix Equation

spSolve()
Solves the matrix equation

$$Ax = b \tag{3}$$

given the matrix A factored into LU form and b.

spSolveTransposed()
When working with adjoint systems, such as in sensitivity analysis, it is desirable to quickly solve

$$A^T x = b \tag{4}$$

Once A has been factored into LU form, this routine can be used to solve the transposed system without having to suffer the cost of factoring the matrix again.

3.8 Numerical Error Estimation

spCondition()
Estimates the L-infinity condition number of the matrix. This number is a measure of the ill-conditioning in the matrix equation. It is also useful for making estimates of the error in the solution.

spNorm()
Returns the L-infinity norm (the maximum absolute row sum) of an unfactored matrix.

spPseudoCondition()
Returns the ratio of the largest pivot to the smallest pivot of a factored matrix. This is a rough indicator of ill-conditioning in the matrix.

spLargestElement()
If passed an unfactored matrix, this routine returns the absolute value of the largest element in the matrix. If passed a factored matrix, it returns an estimate of the largest element that occurred in any of the reduced submatrices during the factorization. The ratio of these two numbers (factored/unfactored) is the growth, which is used to determine if the pivoting order is numerically satisfactory.

spRoundoff()
Returns a bound on the magnitude of the largest element in $E = A - LU$, where E represents error in the matrix resulting from roundoff error during the factorization.

3.9 Matrix Operations

spDeterminant()
This routine simply calculates and returns the determinant of the factored matrix.

spMultiply()
This routine multiplys the matrix by a vector on the right. This is useful for forming the product $Ax = b$ in order to determine if a calculated solution is correct.

spMultTransposed()
Multiplys the transposed matrix by a vector on the right. This is useful for forming the product $A^T x = b$ in order to determine if a calculated solution is correct.

Matrix Statistics and Documentation

spErrorState()
Determines the error status of a particular matrix. While most of the *Sparse* routines do return an indication that an error has occurred, some do not and so *spErrorState*() provides the only way of uncovering these errors.

spErrorMessage()
Prints a *Sparse* error message.

spWhereSingular()
Returns the row and column number where the matrix was detected as singular or where a zero pivot was found.

spGetSize()
A function that returns the size of the matrix. Either the internal or external size of the matrix is returned. The internal size of the matrix is the actual size of the matrix whereas the external size is the value of the largest row or column number. These two numbers may differ if the TRANSLATE option is used.

spElementCount()
spFillinCount()
Functions that return the total number of elements in the matrix, and the number of fill-ins in the matrix. These functions are useful for gathering statistics on matrices.

spPrint()
This routine outputs the matrix as well as some statistics to standard output in a format that is readable by people. The matrix can be printed in either a compressed or standard format. In the standard format, a numeric value is given for each structurally nonzero element, whereas in the compressed format, only

the existence or nonexistence of an element is indicated. This routine is not suitable for use on large matrices.

spFileMatrix()
spFileVector()
These two routines send a copy of the matrix and its right-hand side vector to a file. This file can then be read by the test program that is included with *Sparse*. Only those elements of the matrix that are structurally nonzero are output, so very large matrices can be sent to a file.

spFileStats()
This routine calculates and sends some useful statistics concerning a matrix to a file.

4 SPARSE ROUTINES

For detailed information on the interface to Sparse and on its various configuration options, see the *Sparse Reference Manual*.

4.1 Example

Here is a simple example that uses *Sparse* to perform AC analysis on a simple RC ladder filter. It is meant to demonstrate the use of *Sparse* at a basic level.

```
/*
 * Simple example for Sparse
 *
 * This test builds a simple ladder network and then performs an AC analysis
 * to compute is transfer characteristics versus frequency.
 *
 * Assumes Sparse is configured for complex matrices and that
 * spSEPARATED_COMPLEX_VECTORS is NO.
 */

#include <stdio.h>
#include <math.h>
#include "spMatrix.h"

int
main( int argc, char **argv )
```

```
{
spMatrix A;
struct spTemplate Stamp[3];
spError err;
struct complex { double re; double im; } x[3], b[3];
double f, omega;

/* Create and build the matrix. */
    A = spCreate( 2, 1, &err );
    if (err >= spFATAL) {
        spErrorMessage( A, stderr, argv[0] );
        return 1;
    }
    spGetAdmittance( A, 1, 0, &Stamp[0] );
    spGetAdmittance( A, 1, 2, &Stamp[1] );
    spGetAdmittance( A, 2, 0, &Stamp[2] );
    if (spErrorState( A ) >= spFATAL) {
        spErrorMessage( A, stderr, argv[0] );
        return 1;
    }

/* Drive the circuit at node 1. */
    b[1].re = 1.0;      b[1].im = 0.0;
    b[2].re = 0.0;      b[2].im = 0.0;

/* Perform AC analysis over a range of frequencies. */
    for (f = 0.0; f <= 100000.0; f += 1000.0) {
        omega = 2.0 * M_PI * f;

/* Load the matrix. */
        spClear( A );
        spADD_COMPLEX_QUAD( Stamp[0], 1/50.0, 1e-6*omega );
        spADD_REAL_QUAD( Stamp[1], 1/200.0 );
        spADD_COMPLEX_QUAD( Stamp[2], 1/50.0, 1e-6*omega );

/* Solve the matrix equations Ax = b for x. */
        err = spFactor( A );
        if (err >= spFATAL) {
            spErrorMessage( A, stderr, argv[0] );
            return 1;
```

```
        }
        spSolve( A, (spREAL *)b, (spREAL *)x );
        printf( "f = %f, h = %f\n", f, sqrt(x[2].re*x[2].re + x[2].im*x[2].im) );
    }
    return 0;
}
```

5 FORTRAN COMPATIBILITY

The *Sparse1.4* package contains routines that interface to a calling program written in FORTRAN. Almost every externally available *Sparse1.4* routine has a counterpart defined with the same name except that the 'sp' prefix is changed to 'sf'. The *spADD_ELEMENT*() and *spADD_QUAD*() macros are also replaced with the *sfAdd1*() and *sfAdd4*() functions.

Any interface between two languages is going to have portibility problems, this one is no exception. To ease porting the FORTRAN interface file to different operating systems, the names of the interface functions can be easily redefined (search for 'Routine Renaming' in **spFortran.c**). When interfacing to a FORTRAN program, the FORTRAN option should be set to YES and the ARRAY_OFFSET option should be set to NO (see **spConfig.h**). For details on the return value and argument list of a particular interface routine, see the file **spFortran.c**.

A simple example of a FORTRAN program that calls *Sparse* follows.

5.1 Example

```
      integer matrix, error, sfCreate, sfGetElement, spFactor
      integer element(10)
      double precision rhs(4), solution(4)
c
c create matrix
      matrix = sfCreate(4,0,error)
c
c reserve elements
      element(1) = sfGetElement(matrix,1,1)
      element(2) = sfGetElement(matrix,1,2)
      element(3) = sfGetElement(matrix,2,1)
      element(4) = sfGetElement(matrix,2,2)
      element(5) = sfGetElement(matrix,2,3)
```

```
      element(6) = sfGetElement(matrix,3,2)
      element(7) = sfGetElement(matrix,3,3)
      element(8) = sfGetElement(matrix,3,4)
      element(9) = sfGetElement(matrix,4,3)
      element(10) = sfGetElement(matrix,4,4)
c
c clear matrix
      call sfClear(matrix)
c
c load matrix
      call sfAdd1Real(element(1), 2d0)
      call sfAdd1Real(element(2), \-1d0)
      call sfAdd1Real(element(3), \-1d0)
      call sfAdd1Real(element(4), 3d0)
      call sfAdd1Real(element(5), \-1d0)
      call sfAdd1Real(element(6), \-1d0)
      call sfAdd1Real(element(7), 3d0)
      call sfAdd1Real(element(8), \-1d0)
      call sfAdd1Real(element(9), \-1d0)
      call sfAdd1Real(element(10), 3d0)
      call sfPrint(matrix, .false., .false., .true.)
      rhs(1) = 34d0
      rhs(2) = 0d0
      rhs(3) = 0d0
      rhs(4) = 0d0
c
c factor matrix
      error = sfFactor(matrix)
c
c solve matrix
      call sfSolve(matrix, rhs, solution)
      write (6, 10) solution(1), solution(2), solution(3), solution(4)
   10 format (f 10.2)
      end
```

6 SPARSE TEST PROGRAM

The *Sparse* package includes a test program that is able to read matrix equations from text files and print their solution along with matrix statistics and timing information. The program can also generate files containing stripped versions of the unfactored and factored matrix suitable for plotting using standard plotting programs, such as the UNIX graph and plot commands.

The *Sparse* test program is invoked using the following syntax.

sparse [*options*] [*file1*] [*file2*] ...

Options:

-**s** Print solution only.

-**r x** Use *x* as relative threshold.

-**a x** Use *x* as absolute threshold.

-**n n** Print first *n* terms of solution vector.

-**i n** Repeat build/factor/solve *n* times for better timing results.

-**b n** Use column *n* of matrix as right-hand side vector.

-**p** Create plot files "*filename*.bef" and "*filename*.aft".

-**c** Use complete (as opposed to diagonal) pivoting.

-**x** Treat real matrix as complex with imaginary part zero.

-**t** Solve transposed system.

-**u** Print usage message.

The presence of certain options is dependent on whether the appropriate *Sparse* option has been enabled.

6.1 Matrix Files

If no input files are specified, *Sparse* reads from the standard input. The syntax of the input file is as follows. The matrix begins with one line of arbitrary text that acts as the label, followed by a line with the integer size of the matrix and either the **real** or **complex** keywords. After the header is an arbitrary number of lines that describe the structural nonzeros in the matrix. These lines have the

form *row column data*, where *row* and *column* are integers and *data* is either one real number for real matrices or a real/imaginary pair of numbers for complex matrices. Only one structural nonzero is described per line and the section ends when either *row* or *column* are zero. Following the matrix, an optional right-hand side vector can be described. The vector is given one element per line, the number of element must equal the size of the matrix. Only one matrix and one vector are allowed per file, and the vector, if given, must follow the matrix.

Real Example:

```
mat0  ---  Simple matrix.
4 real
1 1 2.0
1 2 -1.0
2 1 -1.0
2 2 3.0
2 3 -1.0
3 2 -1.0
3 3 3.0
3 4 -1.0
4 3 -1.0
4 4 3.0
0 0 0.0
34.0
0.0
0.0
0.0
```

Complex Example:

```
mat5  --  Test of complete pivoting capability.
3       complex
1       2       1       0
2       3       0       1
3       1       1       0
0       0       0       0
2       0
0       3
1       0
```

7 SPARSE FILES

The following is a list of the files contained in the *Sparse* package and a brief description of their contents. Of the files, only **spConfig.h** is expected to be modified by the user and only **spMatrix.h** need be imported into the program that calls *Sparse*.

spAlloc.c
This file contains the routines for allocating and deallocating objects associated with the matrices, including the matrices themselves.

User accessible functions contained in this module:
spCreate()
spDestroy()
spErrorState()
spWhereSingular()
spGetSize()
spSetReal()
spSetComplex()
spFillinCount()
spElementCount()

spBuild.c
This file contains the routines for clearing and loading the matrix.

User accessible functions contained in this module:
spClear()
spGetAdmittance()
spGetElement()
spGetInitInfo()
spGetOnes()
spGetQuad()
spInitialize()
spInstallInitInfo()

spConfig.h
This file contains the options that are used to customize the package. For example, it is possible to specify whether only real or complex systems of equations are to be solved. Also included in this file are the various constants used by the *Sparse* package, such as the amount of memory initially allocated for each matrix and the largest real number represented by the machine. The user is expected to modify this file to maximize the performance of the routines with his/her matrices.

spDefs.h

This module contains common data structure definitions and macros for the sparse matrix routines. These definitions are meant to remain hidden from the program that calls the sparse matrix routines.

spDoc
This reference manual. **spDoc.tex** contains the manual in LaTeX.

spFactor.c
This file contains the routines for factoring matrices into LU form.

User accessible functions contained in this module:
spFactor()
spOrderAndFactor()
spPartition()

spFortran.c
This file contains the routines for interfacing *Sparse1.4* to a program written in FORTRAN. The function and argument lists of the routines in this file are almost identical to their C equivalents except that they are suitable for calling from a FORTRAN program. The names of these routines use the 'sf' prefix to distinguish them from their C counterparts.

User accessible functions contained in this module:
sfAdd1Complex()
sfAdd1Imag()
sfAdd1Real()
sfAdd4Complex()
sfAdd4Imag()
sfAdd4Real()
sfClear()
sfCondition()
sfCreate()
sfDeleteRowAndCol()
sfDestroy()
sfDeterminant()
sfElementCount()
sfError()
sfErrorMessage()
sfFactor()
sfFileMatrix()
sfFileStats()
sfFileVector()
sfFillinCount()

sfGetAdmittance()
sfGetElement()
sfGetOnes()
sfGetQuad()
sfGetSize()
sfLargestElement()
sfMNA_Preorder()
sfMultTransposed()
sfMultiply()
sfNorm()
sfOrderAndFactor()
sfPartition()
sfPrint()
sfPseudoCondition()
sfRoundoff()
sfScale()
sfSetComplex()
sfSetReal()
sfSolve()
sfSolveTransposed()
sfStripFills()
sfWhereSingular()

spMatrix.h
This file contains definitions that are useful to the calling program. In particular, this file contains error keyword definitions, some macro functions that are used to quickly enter data into the matrix, the definition of a data structure that acts as a template for entering admittances into the matrix, and the type declarations of the various *Sparse* functions.

spOutput.c
This file contains the output-to-file and output-to-screen routines for the matrix package. They are capable of outputting the matrix in either a form readable by people or a form readable by the *Sparse* test program.

User accessible functions contained in this module:
spFileMatrix()
spFileStats()
spFileVector()
spPrint()

spRevision

The history of updates for the program.

spSolve.c
This module contains the forward and backward substitution routines.

User accessible functions contained in this module:
spSolve()
spSolveTransposed()

spTest.c
This module contains a test program for the sparse matrix routines. It is able to read matrices from files and solve them. Because of the large number of options and capabilities built into *Sparse*, it is impossible to have one test routine thoroughly exercise *Sparse*. Thus, emphasis is on exercising as many capabilities as is reasonable while also providing a useful tool.

spUtil.c
This module contains various optional utility routines.

User accessible functions contained in this module:
spCondition()
spDeleteRowAndCol()
spDeterminant()
spErrorMessage()
spLargestElement()
spMNA_Preorder()
spMultiply()
spMultTransposed()
spNorm()
spPseudoCondition()
spRoundoff()
spScale()
spStripFills()

Makefile
This file is used in conjunction with the UNIX program **make** to compile the matrix routines and their test program.

8 More Information

Detailed information on the functions, types, and options that make up the *Sparse* API can be found in the reference manual. Both the reference manual and this manual can be downloaded from `sparse.sourceforge.net`. Please report bugs to `kundert@users.sourceforge.net`.

Acknowledgements

We would like to acknowledge and thank the those people that contributed ideas that were incorporated into *Sparse*. In particular, Jacob White, Kartikeya Mayaram, Don Webber, Tom Quarles, Howard Ko and Beresford Parlett.

References

[duff86] I. S. Duff, A. M. Erisman, J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 1986.

[golub86] G. H. Golub, C. F. V. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1983.

[kundert86] Kenneth S. Kundert. Sparse matrix techniques. In *Circuit Analysis, Simulation and Design*, Albert Ruehli (editor). North-Holland, 1986.

[strang80] Gilbert Strang. *Linear Algebra and Its Applications*. Academic Press, 1980.