Algorithm 8xx: KLU, a direct sparse solver for circuit simulation problems

TIMOTHY A. DAVIS
University of Florida
and
EKANATHAN PALAMADAI NATARAJAN
Ansoft, Inc.

KLU is a software package for solving sparse unsymmetric linear systems of equations that arise in circuit simulation applications. It relies on a permutation to block triangular form (BTF), several methods for finding a fill-reducing ordering (variants of approximate minimum degree and nested dissection), and Gilbert/Peierls' sparse left-looking LU factorization algorithm to factorize each block. The package is written in C and includes a MATLAB interface. Performance results comparing KLU with SuperLU, Sparse 1.3, and UMFPACK on circuit simulation matrices are presented. KLU is the default sparse direct solver in the Xyce™circuit simulation package developed by Sandia National Laboratories.

Categories and Subject Descriptors: G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*linear systems (direct methods), sparse and very large systems*; G.4 [**Mathematics of Computing**]: Mathematical Software—*algorithm analysis, efficiency*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: LU factorization, sparse matrices, circuit simulation

1. OVERVIEW

KLU is a set of routines for solving a sequence of sparse unsymmetric linear system of equations $Ax = b$, where subsequent matrices A have the same nonzero pattern as the first. The package employs a pre-ordering algorithm that permutes the matrix to a block upper triangular form (BTF) [Duff 1981b; Duff and Reid 1978b]. This is followed by a fill-reducing pre-ordering of each diagonal block, and an asymptotically efficient left looking LU factorization algorithm with partial pivoting [Gilbert and Peierls 1988]. KLU is the default sparse direct solver in the Xyce circuit simulation package [Hutchinson et al. 2002].

Section 2 describes the characteristics of circuit matrices and why the BTF ordering is so helpful for these matrices. Section 3 gives a brief description of the algorithm. A more detailed discussion may be found in [Palamadai Natarajan 2005]. Performance results of KLU in comparison with SuperLU [Demmel et al. 1999], Sparse 1.3 [Kundert 1986; Kundert and Sangiovanni-Vincentelli 1988], and UMFPACK [Davis and Duff 1997; 1999; Davis 2002] are presented in Section 4. Sections 5 and 6 give an overview of how to use the package in MATLAB and in a stand-alone C program.

2. CHARACTERISTICS OF CIRCUIT MATRICES

Circuit matrices arise from Newton's method applied to the differential algebraic equations representing the underlying circuit [Nichols et al. 1994]. A modified nodal analysis is typically used, resulting in a sequence of linear systems with unsymmetric sparse coefficient matrices. The asymmetry arises from the presence of voltage sources in the circuit. Except for these devices, the nonzero pattern of the matrix is symmetric. Circuit matrices exhibit certain unique characteristics for which KLU is designed. They are very sparse since each node typically has few devices connected to it. When ordered properly, the factors remain sparse as well, in contrast to sparse matrices arising from finite-element discretizations, for example. Because they are extremely sparse, dense matrix kernels (the BLAS, [Dongarra et al. 1990]) are not applicable. Dense matrix kernels can be used in supernodal and multifrontal methods when groups of rows and/or columns have identical or similar nonzero pattern in the LU factors (see [Davis 2006] for an overview of these methods). A set of columns in L with identical or similar nonzero pattern is called a *supernode*; these allow the use of the BLAS. Circuit matrices typically do not have large supernodes since the interconnection among nodes is not similar across all the nodes in the circuit. The matrices are easily permutable to block upper triangular form, and have a zero-free diagonal unless voltage sources are present (in which case the diagonal is mostly zero-free). Another peculiar feature of circuit matrices is that the nonzero pattern of each block after permutation to block upper triangular form is more symmetric than the original matrix. In DC operating point analysis, capacitors are open and hence node connectivity is broken in the circuit. This helps in creating many small strongly connected components in the corresponding graph, and the resulting permuted matrix is block triangular with many small blocks. However in transient simulation, capacitors are not open and hence the nodes of the circuit are mostly reachable from each other. This often leads to a single large diagonal block when permuted to BTF form, but still with a large number of small blocks due to current and voltage sources.

3. ALGORITHM

KLU performs the following steps when solving the first linear system in a sequence.

(1) The matrix is permuted into block upper triangular form (BTF). This consists of two steps: an unsymmetric permutation to ensure a zero free diagonal using maximum transversal [Duff 1981b; 1981a], followed by a symmetric permutation to block upper triangular form by finding the strongly connected components of the graph [Duff and Reid 1978a; 1978b; Tarjan 1972]. A matrix with full rank, permuted to block upper triangular form looks as follows:

$$PAQ = \begin{bmatrix} A_{11} & A_{12} & \cdots & A_{1n} \\ & A_{22} & & \vdots \\ & & \ddots & \vdots \\ & & & A_{nn} \end{bmatrix}$$

(2) Each block A_{kk} is ordered to reduce fill. The Approximate Minimum Degree

(AMD) ordering [Amestoy et al. 1996; 2004] on $A_{kk} + A_{kk}^T$ is used by default. The user can alternatively choose COLAMD [Davis et al. 2004b; 2004a], an ordering provided by CHOLMOD (such as nested dissection based on METIS [Karypis and Kumar 1998]), or any user-defined ordering algorithm that can be passed as a function pointer to KLU. Alternatively, the user can provide a permutation to order each block.

(3) Each diagonal block is scaled and factorized using our implementation of Gilbert/ Peierls' left looking algorithm with partial pivoting. The same algorithm is used in the LU factorization method in the CSparse package, `cs_lu` [Davis 2006] (but without the pre-scaling and without a BTF permutation).

(4) The system is solved using block back substitution.

For subsequent factorizations, the first two steps above are skipped. The third step is replaced with a simpler left-looking method that does not perform partial pivoting (a *refactorization*). This allows the depth-first-search used in Gilbert/Peierls' method to be skipped, since the nonzero patterns of L and U are already known.

There are several advantages in permuting the matrix to BTF form. Entries outside the diagonal blocks do not need to be factorized, requiring no work and causing no fill-in. Only the diagonal blocks need to be factorized.

The final system of equations to be solved after ordering and factorization with partial pivoting can be represented as

$$(PRAQ)Q^T x = PRb \tag{1}$$

where P represents the row permutation due to the BTF and fill-reducing ordering and partial pivoting, and Q represents the column permutation due to just the BTF and fill-reducing ordering. The matrix R is a diagonal row scaling matrix. KLU provides row scaling with respect to either the maximum absolute value across each row or the sum of absolute values of elements in each row. Let $(PRAQ) = LU + F$ where LU represents the factors of all the blocks collectively and F represents the entire off diagonal region. Equation (1) can now be written as

$$x = Q(LU + F)^{-1}(PRb). \tag{2}$$

The block back substitution in (2) can be better visualized as follows. Consider a simple 3-by-3 block system

$$\begin{bmatrix} L_{11}U_{11} & F_{12} & F_{13} \\ 0 & L_{22}U_{22} & F_{23} \\ 0 & 0 & L_{33}U_{33} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix}. \tag{3}$$

The equations corresponding to the above system are

$$L_{11}U_{11}x_1 + F_{12}x_2 + F_{13}x_3 = b_1 \tag{4}$$
$$L_{22}U_{22}x_2 + F_{23}x_3 = b_2 \tag{5}$$
$$L_{33}U_{33}x_3 = b_3 \tag{6}$$

In block back substitution, we first solve (6) for x_3, and then eliminate x_3 from (5) and (4) using the off-diagonal entries. Next, we solve (5) for x_2 and eliminate x_2 from (4). Finally we solve (4) for x_1.

The core of the Gilbert/Peierls factorization algorithm used in KLU is solving a lower triangular system $Lx = b$ with partial pivoting where L, x and b are all sparse. It consists of a symbolic step to determine the non-zero pattern of x and a numerical step to compute the values of x. This lower triangular solution is repeated n times during the entire factorization (where n is the size of the matrix) and each solution step computes a column of the L and U factors. The importance of this factorization algorithm is that the time spent in factorization is proportional to the number of floating point operations performed. The entire left looking algorithm is described in the algorithm below.

Algorithm 1 LU factorization of a n-by-n unsymmetric matrix A

$L = I$
for $k = 1$ to n **do**
  solve the lower triangular system $Lx = A(:, k)$
  do partial pivoting on x
  $U(1 : k, k) = x(1 : k)$
  $L(k : n, k) = x(k : n)/U(k, k)$
end for

The lower triangular solve is the most expensive step and includes a symbolic and a numeric factorization step. Let $b = A(:, k)$, the kth column of A. Let $G = G_L$ be the directed graph of L with n nodes. The graph G_L has an edge $j \rightarrow i$ iff $L_{ij} \neq 0$. Let $\mathcal{B} = \{i | b_i \neq 0\}$ and $\mathcal{X} = \{i | x_i \neq 0\}$ represent the set of nonzero indices in b and x respectively. Now the elements of $\mathcal{X}$ are given by

$$\mathcal{X} = Reach_{G_L}(\mathcal{B}) \tag{7}$$

The nonzero pattern $\mathcal{X}$ is computed by the determining the vertices in G_L that are reachable from the vertices of the set $\mathcal{B}$. The reachability problem is solved using a depth-first-search. During the depth-first search, Gilbert/Peierls' algorithm computes the topological order of $\mathcal{X}$. This topological ordering is used in eliminating unknowns in the numerical factorization step. Sorting the nodes in $\mathcal{X}$ could take more time than the number of floating-point operations, so this is skipped. Instead, x can be computed in any topological order of G_L. That is, x_j must be computed before x_i if there is a path from j to i in G_L. Since the depth-first graph traversal naturally produces $\mathcal{X}$ in topological order, the solution of $Lx = b$ can be computed using the algorithm below.

Algorithm 2 Solve $Lx = b$ where L, x and b are sparse

$\mathcal{X} = Reach_{G_L}(\mathcal{B})$
$x = b$
for $j \in \mathcal{X}$ in any topological order **do**
  $x(j + 1 : n) = x(j + 1 : n) - L(j + 1 : n, j)x(j)$
end for

The computation of $\mathcal{X}$ and x both take time proportional to the floating-point operation count.

4. PERFORMANCE RESULTS

Five different sparse LU factorization techniques are compared:

(1) KLU with default parameter settings: BTF enabled, the AMD fill-reducing ordering applied to $A + A^T$, and a strong preference for pivots selected from the diagonal.

(2) KLU with default parameters, except that BTF is disabled. For most matrices, using BTF is preferred, but in a few cases the BTF pre-ordering can dramatically increase the fill-in in the LU factors.

(3) SuperLU 3.1 [Demmel et al. 1999], using non-default diagonal pivoting preference and ordering options identical to KLU (but without BTF).[1] These options typically give the best results for circuit matrices. SuperLU is a supernodal variant of the Gilbert/Peierls' left-looking algorithm used in KLU.

(4) UMFPACK [Davis and Duff 1997; 1999; Davis 2002] with default parameters. In this mode, UMFPACK evaluates the symmetry of the nonzero pattern and selects either the AMD ordering on $A + A^T$ and a strong diagonal preference, or it uses the COLAMD ordering with no preference for the diagonal. For most circuit simulation matrices, the AMD ordering is used. UMFPACK is a right-looking multifrontal algorithm that makes extensive use of BLAS kernels.

(5) Sparse 1.3 [Kundert 1986; Kundert and Sangiovanni-Vincentelli 1988], the sparse solver used in SPICE3f5, the latest version of SPICE.[2]

The University of Florida Sparse Matrix Collection [Davis 2008] includes 81 real square unsymmetric matrices or matrix sequences (only the first matrix in each sequence is considered here) arising from the differential algebraic equations used in SPICE-like circuit simulation problems, or from power network simulation. All five methods were tested an all 81 matrices, except for two matrices too large for any method on the computer used for these tests (a single-core 3.2 Ghz Pentium 4 with 4GB of RAM). The thirteen matrices requiring the most amount of time to analyze, factorize, and solve (as determined by the fastest method for each matrix) are shown in Table I. None of these thirteen matrices come from a DC analysis, since the run time for KLU is so low for those matrices. The table lists the matrix name followed by the size of the whole matrix and the largest block in the BTF form (the dimension and the number of nonzeros). The last two columns list the dimension of the second-largest block, and the number of 1-by-1 blocks, respectively.

The performance profiles of the methods are shown in Figures 1 and 2. Figure 1 is the total time for solving the first system in a sequence of linear systems arising from the nonlinear iteration. It includes any symbolic ordering and analysis needed. Figure 2 is the time for subsequent matrices in the sequence, where the pivot ordering and nonzero patterns of the prior LU factors are already known (the refactorization step). The x axis is the time relative to the fastest time for any given matrix (a log scale). The y axis is the number of problems. In a performance

[1] Threshold partial pivoting tolerance of 0.001 to give preference to the diagonal, the SuperLU "symmetric mode", and the AMD ordering on $A + A^T$.

[2] http://bwrc.eecs.berkeley.edu/Classes/icbook/SPICE/, SPICE3f5 last updated 1997, retrieved March 2009.

Matrix	Entire matrix rows $\times 10^3$	Entire matrix nonzeros $\times 10^3$	Largest block rows $\times 10^3$	Largest block nonzeros $\times 10^3$	Rows in 2nd largest block	singletons $\times 10^3$
Raj1	263.7	1300.3	263.6	1299.6	5	0.2
ASIC_680k	682.9	2639.0	98.8	526.3	2	583.8
rajat24	358.2	1947.0	354.3	1923.9	172	3.4
TSOPF_RS_b2383_c1	38.1	16171.2	4.8	31.8	654	0.0
TSOPF_RS_b2383	38.1	16171.2	4.8	31.8	654	0.0
rajat25	87.2	606.5	83.5	589.8	57	3.4
rajat28	87.2	606.5	83.5	589.8	57	3.4
rajat20	86.9	604.3	83.0	587.5	57	3.6
ASIC_320k	321.8	1931.8	320.9	1314.3	6	0.3
ASIC_320ks	321.7	1316.1	320.9	1314.3	6	0.1
rajat30	644.0	6175.2	632.2	6148.3	7	11.7
Freescale1	3428.8	17052.6	3408.8	16976.1	19	0.0

Table I. The thirteen largest test matrices

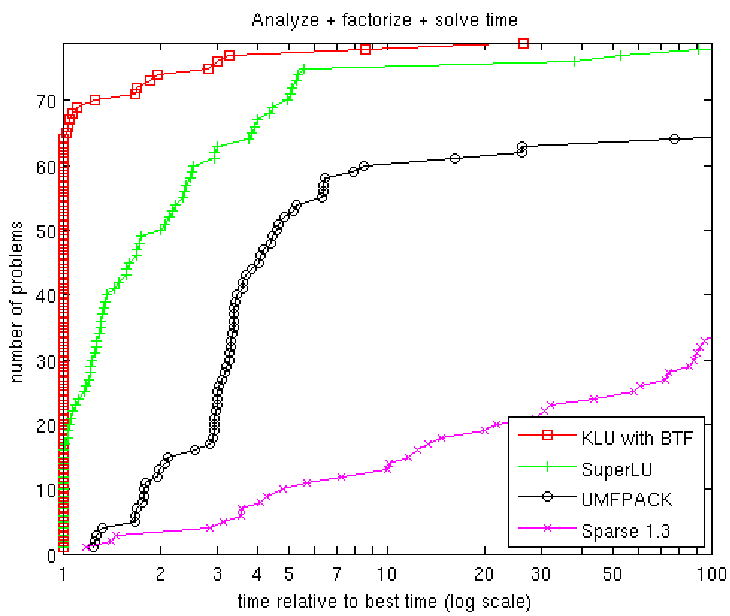

Fig. 1. Performance profile of analyze+factorize+solve time

profile, a point (x, y) is plotted if a method takes no more than x times the run time of the fastest method for y problems. For most matrices, KLU (with BTF) is the fastest method. In the worst case (the Raj1 matrix) it is 26 times slower than SuperLU, but this is because the permutation to BTF used by KLU causes fill-in to dramatically increase, as will be shown in Table II. KLU is particularly competitive for the refactorization step (Figure 2). This is a critical metric, since it accounts for the bulk of the time a circuit simulator spends solving linear systems.

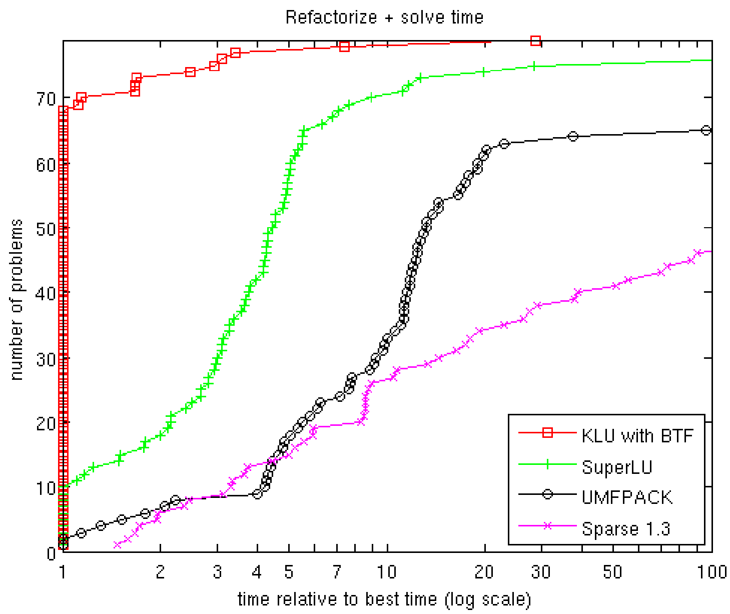

Fig. 2. Performance profile of refactorize+solve time

The total run time (analyze + factorize + solve) for the thirteen largest matrices is shown in Table II. Run times within 25% of the fastest are shown in bold. A dash is shown if the method ran out of memory. The two columns for KLU also include the relative fill-in, which is the number of entries in $L+U+F$ divided by the number of entries in A. KLU (with or without BTF) is the fastest method (or nearly so) for all but two matrices in the table, and no other solver could handle all 79 matrices in the test set. UMFPACK is more prone to pivot off the diagonal than KLU and SuperLU, and thus is not not well-suited for circuit simulation matrices. The time for the refactorization step for these thirteen matrices is shown in Table III.

The BTF pre-ordering could in principle be used with any method, and normally most of the time is spent factorizing the largest diagonal block. To account for this, in the next experiment, the largest block was extracted from the thirteen largest matrices and then analyzed, factorized, and solved by each method. The time to find the BTF ordering and to extract the largest block was not included. The results are shown in Table IV. KLU is fastest (or tied) for about half of the matrices; SuperLU is fastest for about the other half. The run times for UMFPACK are greatly improved when preceded by BTF, but even so, it is fastest (or nearly so) for only 4 of the 13 matrices.

5. USING KLU IN MATLAB

A simple MATLAB interface allows KLU to be used in place of sparse backslash or the sparse `LU` function in MATLAB. The LU factorization of a set of diagonal blocks of the block-triangular form is not representable as `L*U=P*A*Q` as it is in

Matrix	KLU+BTF		KLU no BTF		SuperLU	UMFPACK	Sparse 1.3
	fill	time	fill	time	time	time	time
Raj1	40.3	111.0	5.5	**4.6**	**4.2**	1690.0	3038.9
ASIC_680ks	2.6	**5.0**	2.7	7.2	**4.6**	8.3	818.1
ASIC_680k	2.1	**5.8**	2.1	7.4	**5.8**	11.5	8835.1
rajat24	28.7	119.0	3.3	**6.0**	13.9	-	-
TSOPF_RS_b2383_c1	1.3	**6.5**	2.1	71.8	34.9	-	-
TSOPF_RS_b2383	1.3	**6.5**	2.1	72.0	34.2	-	-
rajat25	6.7	**8.5**	35.2	31.7	37.2	-	2675.4
rajat28	6.9	**9.1**	28.4	25.4	50.0	-	3503.0
rajat20	7.0	**9.1**	35.2	31.3	40.5	704.3	4314.1
ASIC_320k	2.5	30.4	42.9	447.5	**18.1**	142.0	7908.2
ASIC_320ks	3.2	36.6	3.2	36.4	**21.5**	136.4	684.9
rajat30	5.1	73.0	3.2	**23.8**	**22.5**	-	-
Freescale1	3.9	**86.8**	3.9	**85.6**	-	-	-

Table II. Total run time (analyze+factorize+solve) in seconds, and relative fill-in for KLU

Matrix	KLU+BTF	KLU no BTF	SuperLU	UMFPACK	Sparse 1.3
	time	time	time	time	time
Raj1	94.4	**3.0**	**3.3**	1679.4	127.4
ASIC_680ks	**3.9**	5.4	**3.5**	6.3	256.7
ASIC_680k	**4.6**	**5.1**	**4.6**	9.4	835.8
rajat24	91.2	**3.7**	12.4	-	-
TSOPF_RS_b2383_c1	**5.2**	40.8	10.9	-	-
TSOPF_RS_b2383	**5.1**	41.0	10.9	-	-
rajat25	**6.7**	27.0	36.8	-	374.4
rajat28	**7.3**	21.8	49.6	-	512.7
rajat20	**7.3**	26.8	40.2	701.6	657.1
ASIC_320k	28.7	429.0	**17.1**	133.4	870.1
ASIC_320ks	35.0	35.0	**20.7**	129.0	182.0
rajat30	60.5	**18.6**	**19.6**	-	-
Freescale1	**70.5**	**70.6**	-	-	-

Table III. Refactorize+solve time in seconds

Matrix	KLU+BTF	KLU no BTF	SuperLU	UMFPACK	Sparse 1.3
	time	time	time	time	time
Raj1	**149.5**	**156.6**	-	-	-
ASIC_680ks	4.6	4.6	**2.6**	7.0	781.2
ASIC_680k	5.3	5.3	**3.3**	9.8	748.3
rajat24	117.9	117.8	**68.9**	**72.1**	-
TSOPF_RS_b2383_c1	**0.0**	**0.0**	**0.0**	0.1	2.3
TSOPF_RS_b2383	**0.0**	**0.0**	**0.0**	0.0	2.2
rajat25	**7.5**	**7.5**	138.3	**6.9**	8834.1
rajat28	**7.9**	**7.9**	146.9	**7.5**	6727.2
rajat20	11.1	11.1	13.5	**6.7**	5763.0
ASIC_320k	30.2	30.1	**16.8**	142.2	763.5
ASIC_320ks	36.5	36.4	**20.5**	136.8	790.1
rajat30	**64.2**	**63.8**	-	384.5	-
Freescale1	**86.8**	**85.7**	-	-	-

Table IV. Run time (analyze+factorize+solve) just for the largest block, in seconds

KLU usage	MATLAB equivalent
`x = klu (A,'\',b)`	`x = A\b`, using KLU instead of sparse backslash
`LU = klu (A)`	factorizes `R\A(p,q) = L*U+F`, returning a struct
`x = klu (LU,'\',b)`	`x = A\b`, where `LU = klu(A)`

Table V. Sample MATLAB interface for KLU

`klu_defaults`	set default parameters
`klu_analyze`	order and analyze a matrix
`klu_analyze_given`	order and analyze a matrix
`klu_factor`	numerical factorization
`klu_solve`	solve a linear system
`klu_tsolve`	solve a transposed linear system
`klu_refactor`	numerical refactorization
`klu_free_symbolic`	destroy the symbolic object
`klu_free_numeric`	destroy the numeric object
`klu_sort`	sort the row indices in the columns of L and U
`klu_flops`	determine the flop count
`klu_rgrowth`	determine the pivot growth
`klu_condest`	accurate condition number estimation
`klu_rcond`	cheap reciprocal condition number estimation
`klu_scale`	scale and check a sparse matrix
`klu_extract`	extract the LU factorization
`klu_malloc`, etc,	wrappers for `malloc`/`free`/etc
`btf_maxtrans`	maximum transversal
`btf_strongcomp`	strongly connected components
`btf_order`	permutation to block triangular form

Table VI. C-callable functions in KLU and BTF

MATLAB, so the LU factors are returned as a MATLAB struct. The user can then pass this struct back to the `klu` mexFunction to solve a sparse linear system. Since MATLAB drops numerically zero entries from its sparse matrices, the `klu` mexFunction does not support an interface to the refactorization phase of KLU. Both real and complex matrices are supported. Settings such as the pivot tolerance and ordering options can be modified via an optional input parameter. Examples are given in Table V.

6. USING KLU IN A C PROGRAM

There are four variants of KLU, with both `int` and `long` integers, and real and complex (double precision) numerical entries. Parameter settings give the user control over the partial pivoting tolerance (for giving preference to the diagonal), ordering options, pre-scaling, whether or not to use BTF, and an option to limit the work performed in the maximal matching phase of BTF (this phase can take $O(n|A|)$ time in the worst case where $|A|$ is the number of nonzeros in A, [Duff 1981b]). If the limit is not reached the result is the same, but if the limit is reached only a partial match is found, leading to fewer blocks in the BTF. The C-callable interfaces of KLU and BTF provide the functions listed in Table VI.

7. ACKNOWLEDGEMENTS

Mike Heroux coined the name and suggested that we tackle this project in support of the Xyce circuit simulation package developed at Sandia National Laboratories [Hutchinson et al. 2002; Sipics 2007]. KLU is based on the non-supernodal Gilbert/Peierls' method that was the precursor to the supernodal SuperLU. Thus, KLU is a "Clark Kent" LU factorization method. Portions of this work were supported by the Department of Energy (Sandia National Laboratories), and by the National Science Foundation under grants 0203270, 0620286, and 0619080.

REFERENCES

Amestoy, P. R., Davis, T. A., and Duff, I. S. 1996. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Appl. 17,* 4, 886–905.

Amestoy, P. R., Davis, T. A., and Duff, I. S. 2004. Algorithm 837: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw. 30,* 3, 381–388.

Davis, T. A. 2002. Algorithm 832: UMFPACK V4.3, an unsymmetric-pattern multifrontal method. *ACM Trans. Math. Softw. 30,* 2, 196–199.

Davis, T. A. 2006. *Direct Methods for Sparse Linear Systems.* SIAM, Philadelphia, PA.

Davis, T. A. 2008. University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*. under submission; see also http://www.cise.ufl.edu/sparse/matrices.

Davis, T. A. and Duff, I. S. 1997. An unsymmetric-pattern multifrontal method for sparse LU factorization. *SIAM J. Matrix Anal. Appl. 18,* 1, 140–158.

Davis, T. A. and Duff, I. S. 1999. A combined unifrontal/multifrontal method for unsymmetric sparse matrices. *ACM Trans. Math. Softw. 25,* 1, 1–19.

Davis, T. A., Gilbert, J. R., Larimore, S. I., and Ng, E. G. 2004a. Algorithm 836: COLAMD, a column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw. 30,* 3, 377–380.

Davis, T. A., Gilbert, J. R., Larimore, S. I., and Ng, E. G. 2004b. A column approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw. 30,* 3, 353–376.

Demmel, J. W., Eisenstat, S. C., Gilbert, J. R., Li, X. S., and Liu, J. W. H. 1999. A supernodal approach to sparse partial pivoting. *SIAM J. Matrix Anal. Appl. 20,* 3, 720–755.

Dongarra, J. J., Du Croz, J., Duff, I. S., and Hammarling, S. 1990. A set of level-3 basic linear algebra subprograms. *ACM Trans. Math. Softw. 16,* 1, 1–17.

Duff, I. S. 1981a. Algorithm 575: Permutations for a zero-free diagonal. *ACM Trans. Math. Softw. 7,* 1, 387–390.

Duff, I. S. 1981b. On algorithms for obtaining a maximum transversal. *ACM Trans. Math. Softw. 7,* 1, 315–330.

Duff, I. S. and Reid, J. K. 1978a. Algorithm 529: Permutations to block triangular form. *ACM Trans. Math. Softw. 4,* 2, 189–192.

Duff, I. S. and Reid, J. K. 1978b. An implementation of Tarjan's algorithm for the block triangularization of a matrix. *ACM Trans. Math. Softw. 4,* 2, 137–147.

Gilbert, J. R. and Peierls, T. 1988. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Statist. Comput. 9*, 862–874.

Hutchinson, S. A., Keiter, E. R., Hoekstra, R. J., Waters, L. J., Russo, T., Rankin, E., Wix, S. D., and Bogdan, C. 2002. The Xyce$^{\text{TM}}$parallel electronic simulator – an overview. In *Parallel Computing: Advances and Current Issues, Proc. ParCo 2001*, G. R. Joubert, A. Murli, F. J. Peters, and M. Vanneschi, Eds. Imperial College Press, 165–172.

Karypis, G. and Kumar, V. 1998. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput. 20*, 359–392.

Kundert, K. S. 1986. Sparse matrix techniques and their applications to circuit simulation. In *Circuit Analysis, Simulation and Design*, A. E. Ruehli, Ed. New York: North-Holland.

Kundert, K. S. and Sangiovanni-Vincentelli, A. 1988. User's guide: Sparse 1.3. Tech. rep., Dept. of EE and CS, UC Berkeley.

NICHOLS, K., KAZMIERSKI, T., ZWOLINSKI, M., AND BROWN, A. 1994. Overview of SPICE-like circuit simulation algorithms. *IEE Proc. Circuits, Devices & Sys. 141,* 4 (Aug), 242–250.

PALAMADAI NATARAJAN, E. 2005. KLU - a high performance sparse linear system solver for circuit simulation problems. M.S. Thesis, CISE Department, Univ. of Florida.

SIPICS, M. 2007. Sparse matrix algorithm drives SPICE performance gains. *SIAM News 40,* 4 (May).

TARJAN, R. E. 1972. Depth first search and linear graph algorithms. *SIAM J. Comput. 1*, 146–160.

Received Month Year; revised Month Year; accepted Month Year