Algorithm 907: KLU, A Direct Sparse Solver for Circuit Simulation Problems

TIMOTHY A. DAVIS
University of Florida
and
EKANATHAN PALAMADAI NATARAJAN
ANSYS, 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$^{\text{TM}}$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 design and analysis; efficiency*

General Terms: Algorithms, Experimentation, Performance

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

ACM Reference Format:
Davis, T. A. and Palamadai Natarajan, E. 2010. Algorithm 907: KLU, A direct sparse solver for circuit simulation problems. ACM Trans. Math. Softw. 37, 3, Article 36 (September 2010), 17 pages. DOI = 10.1145/1824801.1824814. http://doi.acm.org/10.1145/1824801.1824814.

36

1. OVERVIEW

The KLU software package is specifically designed for solving sequences of unsymmetric sparse linear systems that arise from the differential-algebraic

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.
Authors' addresses: T. A. Davis, University of Florida, Gainsville, FL 32611; email: davis@cise.ufl.edu; E. Palamadai Natarajan, Ansys, Inc.

equations used to simulate electronic circuits. Two aspects of KLU are essential for these problems: (1) a permutation to block upper triangular form [Duff 1981b; Duff and Reid 1978b], and (2) an asymptotically efficient left-looking LU factorization algorithm with partial pivoting [Gilbert and Peierls 1988]. KLU does not exploit supernodes, since the factors of circuit simulation matrices are far too sparse as compared to matrices arising in other applications (such as finite-element methods).

Circuit simulation involves many different tasks for which KLU is useful:

(1) DC operating point analysis, where BTF ordering is often helpful. Convergence in DC analysis is critical in that it is typically the first step of a higher-level analysis like transient analysis.
(2) Transient analysis, which requires a fast and accurate sparse LU factorization. The sparse linear factorization/solve stages typically dominate the runtime of transient analyses of postlayout circuits with a large number of parasitic devices.
(3) Harmonic balance analysis, which is typically solved using Krylov-based iterative methods, since the Jacobian representing all the harmonics is huge and cannot be solved with a direct method. KLU is useful in factor/solve stages involving the preconditioner.

KLU with the BTF ordering is the default sparse direct solver in the Xyce circuit simulation package [Hutchinson et al. 2002].

Section 2 describes the characteristics of circuit matrices which motivate the design of the KLU algorithm. 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, respectively. A synopsis of this article will appear in Davis and Palamadai Natarajan [to appear].

In this article, $|A|$ denotes the number of nonzeros in the matrix A.

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 with identical nonzero pattern (ignoring numerical cancellation). In modified nodal analysis, all the devices whose linearized constitutive equations are of the form $i - f(v) = 0$ (where i is the current through the device and $f(v)$ is a function of the voltage across the device) lead to a structurally symmetric matrix. This applies to many devices such as resistors, capacitors, diodes, and inductors. For inductors, the linearized time-discretized constitutive equation satisfies $i - f(v) = 0$.

Sources, both controlled and independent, cause asymmetry in the matrix. Circuit matrices exhibit certain unique characteristics for which KLU is designed, which are not generally true of matrices from other applications.

(1) Circuit matrices are extremely sparse and remain so when factorized. The ratio of floating-point operation (flop) count over $|L + U|$ is much smaller than matrices from other applications (even for comparable values of $|L + U|$). A set of columns in L with identical or similar nonzero pattern is called a *supernode*. Supernodal and multifrontal methods obtain high performance by exploiting supernodes via dense matrix kernels (the BLAS [Dongarra et al. 1990]). Because their nodal interconnection is highly dissimilar and their fill-in is so low, circuit matrices typically do not have large supernodes.

(2) Nearly all circuit matrices are permutable to a block triangular form. 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 large and dominant diagonal block when permuted to BTF form, but still a large number of small blocks due to the presence of independent and controlled sources.

The following experiment illustrates the low fill-in properties of circuit matrices. As of March 2010, the University of Florida Sparse Matrix Collection [Davis and Hu to appear] contains 491 matrices that are real, square, unsymmetric, and have full structural rank[1] (excluding matrices tagged as subsequent matrices in sequences of matrices with the same size and pattern). Of these 491 matrices, 81 are from circuit or power network simulation. Figure 1 plots the fill-in factor ($|L + U|/|A|$ versus $|A|$) for each matrix, using `lu` in MATLAB (R2010a). If the matrix is reducible to block triangular form, only the largest block is factorized for this experiment (found via `dmperm` [Davis 2006]).

Only the largest block is factorized since the purpose is to show why supernodes are not as useful for circuit simulation as compared to matrices from other applications. The `lu` function in MATLAB is based on UMFPACK [Davis and Duff 1997; 1999; Davis 2002], and uses AMD [Amestoy et al. 1996; 2004] or COLAMD [Davis et al. 2004b; 2004a] as its fill-reducing ordering (chosen automatically). It does not exploit a permutation to block triangular form, except to permute 1-by-1 blocks to the front of the factorization, whenever possible.

Not only do circuit matrices stay sparse when factorized, they also require fewer floating-point operations per entry in their LU factors as compared to noncircuit matrices. This is illustrated in Figure 2. For sparse Cholesky factorization, a low ratio of flops per $|L + U|$ indicates that a nonsupernodal method will be faster than a supernodal method (`chol` in MATLAB) [Chen et al. 2008]. Similar results for LU factorization are shown below in Section 4.

[1]A matrix has full structural rank if a permutation exists so that the diagonal is zero-free.

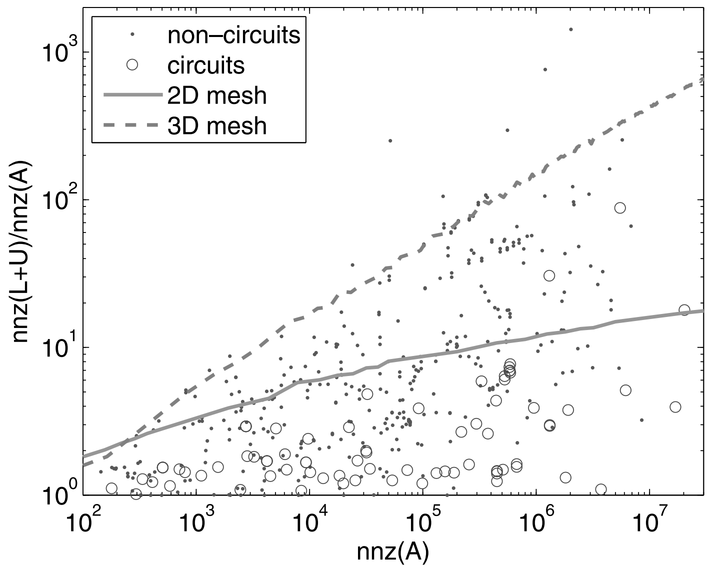

Fig. 1. Fill-in factor versus the number of nonzeros in the largest irreducible block. Each circle is a circuit or power network matrix. Each dot is a matrix from another application.

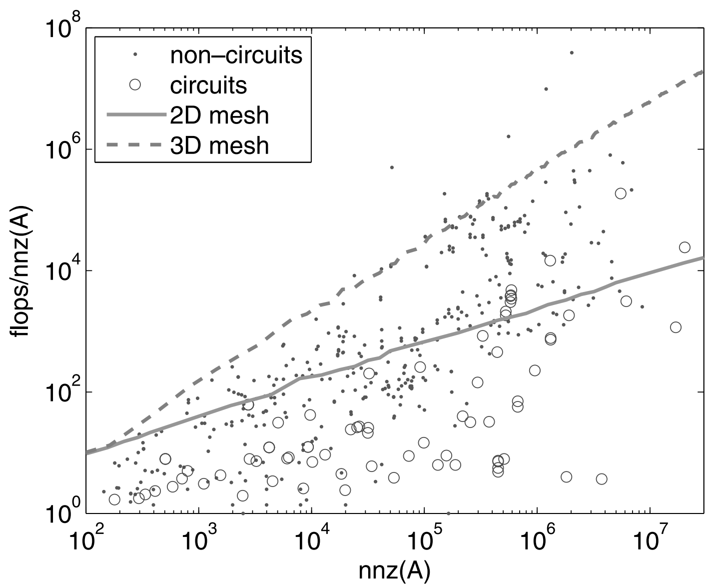

Fig. 2. Flops per $|L+U|$ versus the number of nonzeros in the largest irreducible block.

For comparison, the two lines in Figures 1 and 2 are 2D and 3D square meshes as ordered by METIS [Karypis and Kumar 1998], which obtains the asymptotically optimal ordering for regular meshes.

The fill-in factor for circuit matrices stays remarkably low as compared to matrices from other applications, even though the BTF form is exploited for all matrices in this experiment. Very few circuit matrices experience as much fill-in as 2D or 3D meshes.

Of the 81 circuit matrices in this test set, nearly all (76) are reducible to block triangular form. This makes a BTF ordering essential for obtaining good performance on circuit matrices, as will be seen in Section 4. For the 410 non-circuit matrices, only 262 are reducible.

The properties of circuit matrices demonstrated here indicate that they should be factorized via an asymptotically efficient nonsupernodal sparse LU method, which motivates the KLU algorithm discussed in the next Section.

3. KLU ALGORITHM

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

(1) The matrix is permuted into Block 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 triangular form by finding the strongly connected components of the graph [Duff 1978a; 1978b; Tarjan 1972]. A matrix with full rank permuted to block 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 [Gilbert and Peierls 1988]. A simpler version of the same algorithm is used in the LU factorization method in the CSparse package, `cs_lu` [Davis 2006] (but without the prescaling and without a BTF permutation). Pivoting is constrained to within each diagonal block, since the factorization method factors each block as an independent problem. No pivots can ever be selected from the off-diagonal blocks.

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

For subsequent factorizations for matrices with the same nonzero pattern, the first two steps given before 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.

When the BTF form is exploited, 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 (discussed shortly). Let $(PRAQ) = LU + F$ where LU represents the factors of all the blocks collectively and F represents the entire off-diagonal region. Eq. (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 preceding system are as follows.

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

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 nonzero 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 that follows.

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_L be the directed graph of L with n nodes. The graph G_L has an edge $j \rightarrow i$ iff

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
```

$l_{ij} \neq 0$. 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 nonzero pattern $\mathcal{X}$ is given by

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

$Reach_G(i)$ denotes all nodes in a graph G reachable via paths starting at node i. $Reach(S)$ applied to a set S is the union of $Reach(i)$ for all nodes $i \in S$. Eq. (7) states that the nonzero pattern $\mathcal{X}$ is computed by 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 $\mathcal{X}$. If the nodes of a directed acyclic graph are written out in topological order from left to right, then all edges in the graph would all point to the right. If $Lx = b$ is solved in topological order, all numerical dependencies are satisfied. The natural order $1, 2, \ldots, n$ is one such ordering (since the matrix L is lower triangular), but any topological ordering will suffice. 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 produces $\mathcal{X}$ in topological order as an intrinsic by-product, the solution of $Lx = b$ can be computed using the algorithm that follows. Sorting the nodes in $\mathcal{X}$ to obtain the natural ordering could take more time than the number of floating-point operations, so this is skipped. The computation of $\mathcal{X}$ and x both take time proportional to the floating-point operation count.

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

```
X = Reach_{G_L}(B)
x = b
for j ∈ X in any topological order do
    x(j+1 : n) = x(j+1 : n) − L(j+1 : n, j)x(j)
end for
```

An example from Davis [2006] is shown in Figure 3.

3.1 The Effect of Scaling in KLU

KLU provides three scaling options: no scaling at all, or row scaling with respect to either the maximum absolute value across each row or the sum of absolute values of elements in each row. The default is to apply max-scaling. Scaling tends to decrease the amount of off-diagonal pivoting performed. If all entries in the matrix are scaled so that they have comparable magnitude, then any entry can be selected as a pivot. In particular, this makes the diagonal

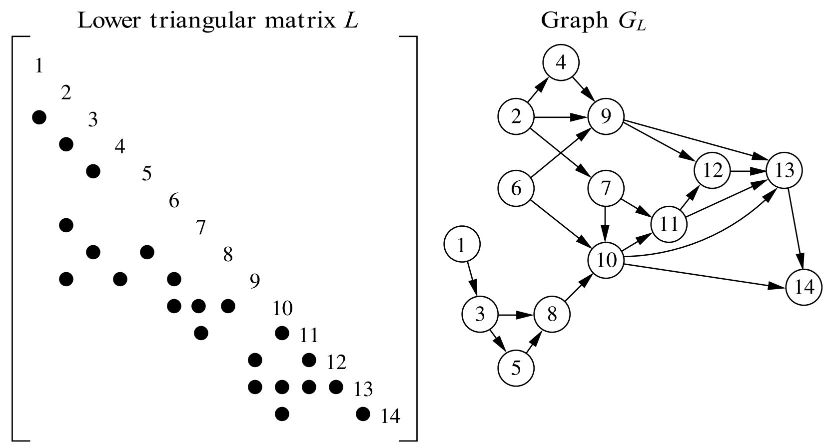

Fig. 3. Solving $Lx = b$ where L, x, and b are sparse, from [Davis 2006]. Suppose $\mathcal{B} = \{4, 6\}$. Then starting a depth-first search at node 4 gives Reach(4) = {4, 9, 12, 13, 14} in topological order. Next, Reach(6) = {6, 9, 10, 11, 12, 13, 14}, so $\mathcal{X} = \{6, 10, 11, 4, 9, 12, 13, 14\}$, also in topological order. The forward solve traverses the columns of L in this order.

entries acceptable. KLU has a strong preference for selecting the diagonal as the pivot entry, which ensures the fill-reducing ordering from the symbolic phase is maintained. Off-diagonal pivoting can increase fill-in dramatically.

The benefits of scaling are highly problem-dependent, however, and it can even make the matrix harder to factorize. There is no single best method for scaling a matrix prior to factorization [Higham 2002]. Scaling can have a large impact on partial-pivoting, which can dramatically affect fill-in (the discussion by Higham [2002] does not consider sparsity).

Figure 4 gives a histogram of the 1-norm condition number estimates of all but one of the 81 circuit matrices. KLU optionally computes this estimate using the method of Hager [1984]. Also shown in this figure is a histogram of the relative residuals, $||Ax - b||/(||A||||x|| + ||b||)$, also using the 1-norm. In spite of the ill-conditioning of these problems, KLU is typically able to find a low relative residual (except for one matrix for which KLU failed when scaling was disabled). No iterative refinement was used for this experiment.

Whether or not scaling is used, or which method of scaling is used, has no effect on 67 of the 80 matrices. Scaling (or not) has an appreciable effect on the remaining 13 matrices. Of those 13, max-scaling is best for 8 and no-scaling is best for 5. Sum-scaling is typically identical to max-scaling, except for a few matrices for which it is worse than max-scaling. Table I lists the results for the three largest of these 13 (as measured by the lowest KLU runtime), and three others of interest, with different scaling options. Fastest times are in bold.

4. PERFORMANCE COMPARISONS WITH OTHER SOLVERS

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.

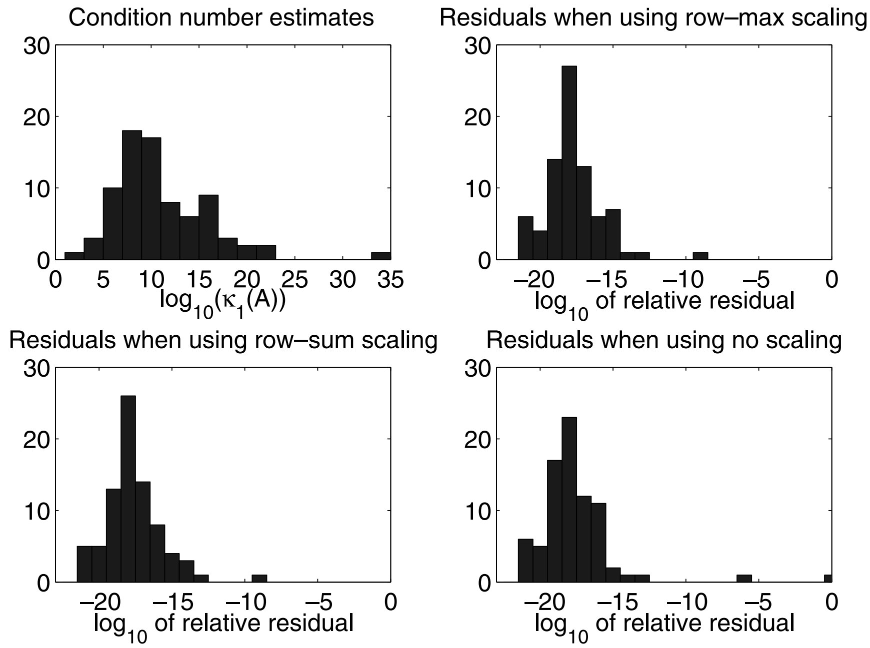

Fig. 4. Condition numbers and the effect of scaling in KLU.

Table I. Effects of Scaling in KLU

matrix: $\kappa_1(A)$ est.	ckt11752_dc_1 10^{15}	ASIC_100ks 9×10^9	rajat25 2×10^{15}	rajat24 2×10^{15}	rajat30 3×10^{14}	Raj1 9×10^{11}
max-scaling:						
time (sec)	**0.54**	10.0	**10.7**	152.2	**97.6**	**142.2**
residual	10^{-21}	10^{-21}	10^{-15}	10^{-16}	10^{-9}	10^{-14}
$\|L+U\| \times 10^6$	1.1	4.3	4.1	56.0	31.7	52.4
sum-scaling:						
time (sec)	1.75	10.0	12.1	153.2	**97.6**	**141.2**
residual	10^{-21}	10^{-21}	10^{-15}	10^{-16}	10^{-9}	10^{-14}
$\|L+U\| \times 10^6$	2.1	4.3	4.3	56.3	31.7	52.4
no scaling:						
time (sec)	2.73	**6.8**	119.9	**42.9**	(failed)	584.4
residual	10^{-22}	10^{-21}	10^{-13}	10^{-17}	-	10^{-7}
$\|L+U\| \times 10^6$	2.6	3.6	42.0	15.6	-	153.2

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

(3) SuperLU 3.1 [Demmel et al. 1999], using nondefault diagonal pivoting preference and ordering options identical to KLU (but without BTF).[2] 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.

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

Table II. The Thirteen Largest Test Matrices

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

(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.[3]

The University of Florida Sparse Matrix Collection [Davis and Hu to appear] 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 II. None of these thirteen matrices comes from a DC analysis, since the runtime 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 5 and 6. Figure 5 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 6 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

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

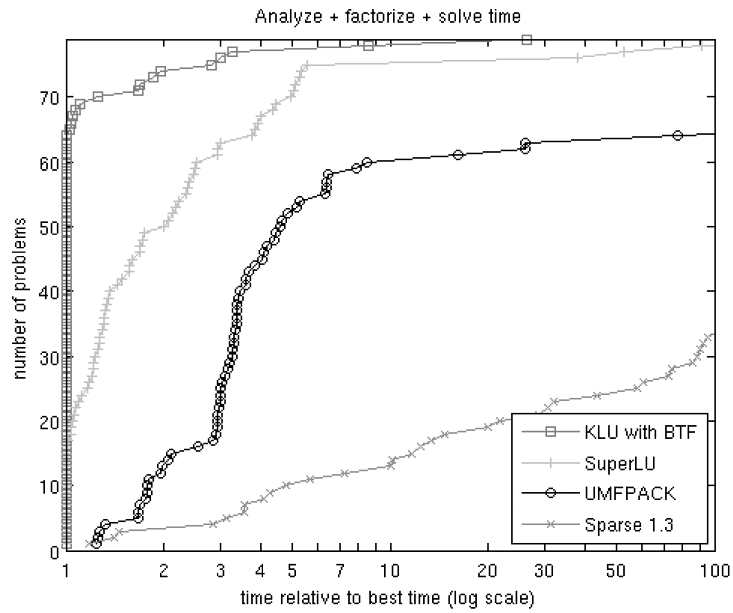

Fig. 5. Performance profile of analyze+factorize+solve time.

the fastest time for any given matrix (a log scale). The y axis is the number of problems. In a performance profile, a point (x, y) is plotted if a method takes no more than x times the runtime 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 III. 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.

The total runtime (analyze + factorize + solve) for the thirteen largest matrices is shown in Table III. Runtimes 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 IV.

The BTF preordering 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 V. KLU is fastest (or tied) for about half of the matrices; SuperLU is fastest for about the other half. The

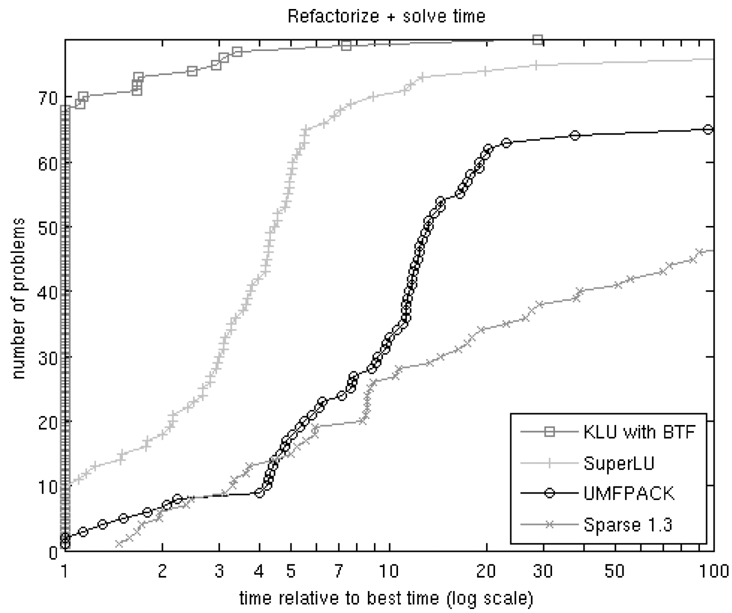

Fig. 6. Performance profile of refactorize+solve time.

Table III. Total Runtime (analyze+factorize+solve) in Seconds, and Relative Fill-In for KLU

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**	-	-	-

runtimes 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.

For sparse Cholesky factorization, the flops per $|L|$ ratio is an accurate predictor of the relative performance of a BLAS-based supernodal method versus a nonsupernodal method. If this ratio is 40 or higher, `chol` in MATLAB (and `x=A\b` for sparse symmetric positive definite matrices) automatically selects a supernodal solver. Otherwise, a nonsupernodal solver is used [Chen et al. 2008]. A similar comparison is shown in Figure 7 between KLU and UMFPACK. The x axis in this figure is the same as the y axis in Figure 2. If the matrix is reducible, only the largest block is factorized. Figure 8 shows the results for sparse Cholesky factorization from Chen et al. [2008].

Table IV. Refactorize+solve Time in Seconds

Matrix	KLU+BTF time	KLU no BTF time	SuperLU time	UMFPACK time	Sparse 1.3 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 V. Runtime (analyze+factorize+solve) Just for the Largest Block, in Seconds

Matrix	KLU+BTF time	KLU no BTF time	SuperLU time	UMFPACK time	Sparse 1.3 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**	-	-	-

These results are remarkable for three reasons.

(1) Circuit matrices tend to have a low flop/$|L + U|$ ratio as compared to other matrices (this is also clear in Figure 2).

(2) Even when the flop/$|L+U|$ ratio is high enough (200 or more) to justify using the BLAS, the relative performance of a BLAS-based method (UMFPACK) versus KLU is much less than what would be expected if only noncircuit matrices were considered. Thus, circuits not only remain sparse when factorized, even large circuit matrices with higher flops/$|L + U|$ ratios hardly justify the use of the BLAS.

(3) The flops/$|L + U|$ ratio for LU factorization (Figure 7) is not a very accurate predictor of the relative performance of BLAS-based sparse methods as compared to non-BLAS-based methods, as it is for sparse Cholesky factorization (Figure 8).

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

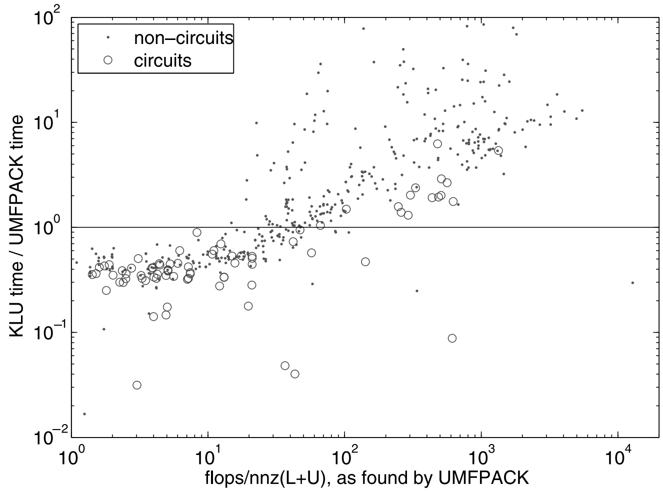

Fig. 7. Relative performance of KLU versus UMFPACK as a function of flops/$|L + U|$, on a single core 64-bit AMD Opteron system with 64GB of RAM.

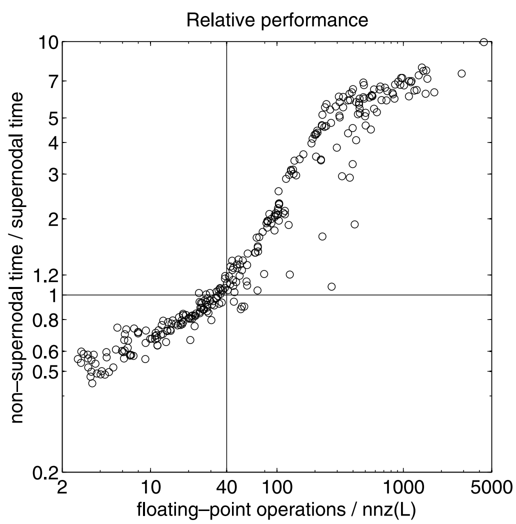

Fig. 8. Relative supernodal and nonsupernodal performance for the CHOLMOD sparse Cholesky factorization. From Chen et al. [2008]. Note that most matrices lie along a smooth curve, in contrast to Figure 7.

blocks of the block triangular form is not representable as `L*U=P*A*Q` as it is in 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 VI.

Table VI. Sample MATLAB Interface for KLU

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 VII. C-Callable Functions in KLU and BTF

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

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, prescaling, 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 VII.

7. SUMMARY

KLU has been shown an effective solver for the sequences of sparse matrices that arise when solving differential algebraic equations for circuit simulation problems. It is the default sparse solver in Xyce, a circuit simulation package developed by Sandia National Laboratories [Hutchinson et al. 2002], for which it has been proven a robust and reliable solver [Sipics 2007].

ACKNOWLEDGMENTS

We would like to thank Mike Heroux for coining the name "KLU" and suggesting 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 nonsupernodal Gilbert/Peierls' method that was the precursor to the supernodal SuperLU. Thus, KLU is a "Clark Kent" LU factorization method.

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.

CHEN, Y., DAVIS, T. A., HAGER, W. W., AND RAJAMANICKAM, S. 2008. Algorithm 887: CHOLMOD, supernodal sparse Cholesky factorization and update/downdate. *ACM Trans. Math. Softw. 35*, 3, 1–14.

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. 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.

DAVIS, T. A. AND HU, Y. To appear. University of Florida sparse matrix collection. *ACM Trans. Math. Softw.* To appear; (see also http://www.cise.ufl.edu/sparse/matrices).

DAVIS, T. A. AND PALAMADAI NATARAJAN, E. To appear. Sparse matrix methods for circuit simulation problems. In *Proceedings of the Conference on Scientific Computing in Electrical Engineering (SCEE'10)*.

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.

HAGER, W. W. 1984. Condition estimates. *SIAM J. Sci. Statist. Comput. 5*, 2, 311–316.

HIGHAM, N. J. 2002. *Accuracy and Stability of Numerical Algorithms*, 2nd Ed. SIAM, Philadelphia, PA.

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™parallel electronic simulator – An overview. In *Proceedings of the Parallel Computing: Advances and Current Issues*, G. R. Joubert, A. Murli, F. J. Peters, and M. Vanneschi Eds., Imperial College Press, London, 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. North-Holland, New York.

KUNDERT, K. S. AND SANGIOVANNI-VINCENTELLI, A. 1988. User's guide: Sparse 1.3. Tech. rep., Department of Electrical Engineering and Computer Science. University of California, Berkeley.

NICHOLS, K., KAZMIERSKI, T., ZWOLINSKI, M., AND BROWN, A. 1994. Overview of SPICE-like circuit simulation algorithms. *IEEE Proc. Circ. Devices Syst. 141*, 4, 242–250.

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

SIPICS, M. 2007. Sparse matrix algorithm drives SPICE performance gains. *SIAM News 40*, 4, 1.

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

Received April 2009; revised April 2010; accepted May 2010