

Algorithm 8xx: a concise sparse Cholesky factorization package

Timothy A. Davis*

January 5, 2004

Abstract

The LDL software package is a set of short, concise routines for factorizing symmetric positive-definite sparse matrices, with some applicability to symmetric indefinite matrices. Its primary purpose is to illustrate much of the basic theory of sparse matrix algorithms in as concise a code as possible, including an elegant new method of sparse symmetric factorization that computes the factorization row-by-row but stores it column-by-column. The entire symbolic and numeric factorization consists of a total of only 53 lines of code. The package is written in C, and includes a MATLAB interface.

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.

Keywords: sparse matrices, linear equations, Cholesky factorization

1 Overview

LDL is a set of short, concise routines that compute the \mathbf{LDL}^T factorization of a sparse symmetric matrix \mathbf{A} . Their primary purpose is to illustrate much of the basic theory of sparse matrix algorithms in as compact a code as possible, including an elegant new method of sparse symmetric factorization (related to [8, 10]). The lower triangular factor \mathbf{L} is computed row-by-row, in contrast to the conventional column-by-column method. This row-oriented algorithm allows for simpler access to the data structure for \mathbf{L} during the factorization. Although it does not achieve the same level of performance as methods based on dense matrix kernels (such as [11, 12]), its performance is competitive with column-by-column

*Dept. of Computer and Information Science and Engineering, Univ. of Florida, Gainesville, FL, USA. email: davis@cise.ufl.edu. <http://www.cise.ufl.edu/~davis>. This work was supported by the National Science Foundation, under grant DMS-0203270. Portions of the work were done while on sabbatical at Stanford University and Lawrence Berkeley National Laboratory (with funding from Stanford University and the SciDAC program).

methods that do not use dense kernels [3, 4, 5]. The symbolic factorization is typically an order of magnitude faster than the corresponding method used in MATLAB Version 6.5. The numeric factorization achieves a peak performance that is about 50% higher than the peak performance of `chol` in MATLAB Version 6.5, which is based on the column-by-column algorithm, and does not use dense matrix kernels [5].

Section 2 gives a brief description of the algorithm used in the symbolic and numeric factorization. A more detailed tutorial-level discussion may be found in [13]. You may find it helpful to have the code in front of you while you read this paper. Some of the details of the concise implementation of this method are given in Section 3. Sections 4 and 5 give an overview of how to use the package in MATLAB and in a stand-alone C program.

2 Algorithm

The underlying numerical algorithm is described below. The k th step solves a lower triangular system of dimension $k - 1$ to compute the k th row of \mathbf{L} and the d_{kk} entry of the diagonal matrix \mathbf{D} .

Algorithm 1 (LDL^T factorization of a n -by- n symmetric matrix \mathbf{A})

```

for  $k = 1$  to  $n$ 
    (step 1) Solve  $\mathbf{L}_{1:k-1,1:k-1}\mathbf{y} = \mathbf{A}_{1:k-1,k}$  for  $\mathbf{y}$ 
    (step 2)  $\mathbf{L}_{k,1:k-1} = (\mathbf{D}_{1:k-1,1:k-1}^{-1}\mathbf{y})^T$ 
    (step 3)  $l_{kk} = 1$ 
    (step 4)  $d_{kk} = a_{kk} - \mathbf{L}_{k,1:k-1}\mathbf{y}$ 
end for

```

When \mathbf{A} and \mathbf{L} are sparse, step 1 of Algorithm 1 requires a triangular solve of the form $\mathbf{L}\mathbf{x} = \mathbf{b}$, where all three terms in the equation are sparse. This is the most costly step of the Algorithm. Steps 2 through 4 are fairly straightforward.

Let \mathcal{X} and \mathcal{B} refer to the set of indices of nonzero entries in \mathbf{x} and \mathbf{b} , respectively, in the lower triangular system $\mathbf{L}\mathbf{x} = \mathbf{b}$. To compute \mathbf{x} efficiently the nonzero pattern \mathcal{X} must be found first. In the general case when \mathbf{L} is arbitrary [7], the nonzero pattern \mathcal{X} is the set of nodes reachable via paths in the graph G_L from all nodes in the set \mathcal{B} , and where the graph G_L has n nodes and a directed edge (j, i) if and only if l_{ij} is nonzero. To compute the numerical solution to $\mathbf{L}\mathbf{x} = \mathbf{b}$ by accessing the columns of \mathbf{L} one at a time, \mathcal{X} can be traversed in any topological order of the subgraph of G_L consisting of nodes in \mathcal{X} . That is, x_j must be computed before x_i if there is a path from j to i in G_L . The natural order $(1, 2, \dots, n)$ is one such ordering, but that requires a costly sort of \mathcal{X} . With a graph traversal and topological sort, the solution of $\mathbf{L}\mathbf{x} = \mathbf{b}$ can be computed using Algorithm 2 below. The computation of \mathcal{X} and \mathbf{x} both take time proportional to the floating-point operation count.

Algorithm 2 (Solve $\mathbf{L}\mathbf{x} = \mathbf{b}$, where \mathbf{L} is lower triangular with unit diagonal)

```

 $\mathcal{X} = \text{Reach}_{G_L}(\mathcal{B})$ 
 $\mathbf{x} = \mathbf{b}$ 
for  $i \in \mathcal{X}$  in any topological order
     $\mathbf{x}_{i+1:n} = \mathbf{x}_{i+1:n} - \mathbf{L}_{i+1:n,i}x_i$ 
end for

```

The general result also governs the pattern of \mathbf{y} in Algorithm 1. However, in this case \mathbf{L} arises from a sparse Cholesky factorization, and is governed by the elimination tree [9]. A general graph traversal is not required. In the elimination tree, the parent of node i is the smallest $j > i$ such that l_{ji} is nonzero. Node i has no parent if column i of \mathbf{L} is completely zero below the diagonal; i is a root of the elimination tree in this case. The nonzero pattern of \mathbf{x} is the union of all paths from any node i (where b_i is nonzero) to the root of the elimination tree. It is referred to here as a tree, but in general it can be a forest.

Rather than a general topological sort of the subgraph of G_L consisting nodes reachable from nodes in \mathcal{B} , a simpler tree traversal can be used. First, select any nonzero entry b_i and follow the path from i to the root of tree, marking the nodes along the way. Place this path on a stack, in order, with i at the top of the stack and the root at the bottom. Repeat for every other nonzero entry in b_i , in arbitrary order, but stop just before reaching a marked node (the result can be empty if i is already in the stack). The stack now contains \mathcal{X} , a topological ordering of the nonzero pattern of \mathbf{x} , which can be used in Algorithm 2 to solve $\mathbf{L}\mathbf{x} = \mathbf{b}$. The time to compute \mathcal{X} using an elimination tree traversal is much faster than the general graph traversal, taking time proportional to the size of \mathcal{X} rather than the number of floating-point operations required to compute \mathbf{x} .

In the k th step of the factorization, the set \mathcal{X} becomes the nonzero pattern of row k of \mathbf{L} . This step requires the elimination tree of $\mathbf{L}_{1:k-1,1:k-1}$, and must construct the elimination tree of $\mathbf{L}_{1:k,1:k}$ for step $k + 1$. Recall that the parent of i in the tree is the smallest j such that $i < j$ and $l_{ji} \neq 0$. Thus, if any node i already has a parent j , then j will remain the parent of i in the elimination trees of all other larger leading submatrices of \mathbf{L} , and in the elimination tree of \mathbf{L} itself. If $l_{ki} \neq 0$ and i does not have a parent in the elimination tree of $\mathbf{L}_{1:k-1,1:k-1}$, then the parent of i is k in the elimination tree of $\mathbf{L}_{1:k,1:k}$. Node k becomes the parent of any node $i \in \mathcal{X}$ that does not yet have a parent.

Since Algorithm 2 traverses \mathbf{L} in column order, \mathbf{L} is stored in a conventional sparse column representation. Each column j is stored as a list of nonzero values and their corresponding row indices. When row k is computed, the new entries can be placed at the end of each list. As a by-product of computing \mathbf{L} one row at a time, the columns of \mathbf{L} are computed in a sorted manner. This is a convenient form of the output. MATLAB requires the columns of its sparse matrices to be sorted, for example. Sorted columns improve the speed of Algorithm 2, since the memory access pattern is more regular. The conventional column-by-column algorithm [3, 4] does not produce columns of \mathbf{L} with sorted row indices.

If the size of each column of \mathbf{L} could be incrementally increased as they are computed, no symbolic pre-analysis would be required. The elimination tree, nonzero pattern of \mathbf{L} , and numerical values of \mathbf{L} could all be computed in a single step. However, to allow for a simple static data structure, the above algorithm can be repeated, but without the numerical computation. All that is required to compute the nonzero pattern of the k th row of \mathbf{L} is the partially constructed elimination tree and the nonzero pattern of the k th column of \mathbf{A} . This is computed in time proportional to the size of this set, using the elimination tree traversal. Once constructed, the number of nonzeros in each column of \mathbf{L} is incremented, for each entry in \mathcal{X} , and then \mathcal{X} is discarded. The set \mathcal{X} need not be constructed in topological order, so no stack is required. The run time of the symbolic analysis algorithm is thus proportional to the number of nonzeros in \mathbf{L} , and the memory requirements are just the matrix \mathbf{A} and a few size- n integer arrays. The result of the algorithm is the elimination tree, a count of the

number of nonzeros in each column of \mathbf{L} , and the cumulative sum of the column counts.

3 Implementation

Because of its simplicity, the implementation of this algorithm leads to a very short, concise code. The symbolic analysis routine `ldl_symbolic` consists of only 20 lines of executable C code. This includes 5 lines of code to allow for a sparsity-preserving ordering \mathbf{P} so that either \mathbf{A} or \mathbf{PAP}^T can be analyzed, 3 lines of code to compute the cumulative sum of the column counts, and one line of code to speed up a `for` loop. An additional line of code allows for a more general form of the input sparse matrix \mathbf{A} . A shorter 9-line synopsis of `ldl_symbolic` is shown below. The `ldl_symbolic` routine ignores entries in the lower triangular part of \mathbf{A} ; the following synopsis requires the upper triangular part of \mathbf{A} only.

The n -by- n sparse matrix \mathbf{A} is provided in compressed column form as an `int` array `Ap` of length `n+1`, an `int` array `Ai` of length `nz`, and a `double` array `Ax` also of length `nz`, where `nz` is the number of entries in the matrix. The numerical values of entries in column j are stored in `Ax [Ap[j] ... Ap[j+1]-1]` and the corresponding row indices are in `Ai [Ap[j] ... Ap[j+1]-1]`. With `Ap[0] = 0`, the number of entries in the matrix is `nz = Ap[n]`.

The outputs of the following code are two size- n arrays: `Parent` holds the elimination tree, and `Lnz` holds the counts of the number of entries in each column of \mathbf{L} . The size- n array `Flag` is used as workspace. None of the output or workspace arrays need to be initialized.

```
for (k = 0 ; k < n ; k++)
{
    Parent [k] = -1 ;           /* parent of k is not yet known */
    Flag [k] = k ;             /* mark node k as visited */
    Lnz [k] = 0 ;              /* count of nonzeros in column k of L */
    for (p = Ap [k] ; p < Ap [k+1] ; p++)
    {
        /* follow path from i to root of etree, stop at flagged node */
        for (i = Ai [p] ; Flag [i] != k ; i = Parent [i])
        {
            /* find parent of i if not yet determined */
            if (Parent [i] == -1) Parent [i] = k ;
            Lnz [i]++ ;          /* L (k,i) is nonzero */
            Flag [i] = k ;      /* mark i as visited */
        }
    }
}
```

The above code is roughly an order of magnitude faster than the MATLAB equivalent, below, on a wide range of sparse symmetric matrices, although faster methods are available [6].

```
Lnz = symbfact (A) ;
Parent = etree (A) ;
```

The numeric factorization in `ldl_numeric` includes this same kernel, except that each path is placed on a stack that represents \mathcal{X} , the nonzero pattern of the k th row of \mathbf{L} . Next, a sparse forward solve is performed using this pattern \mathcal{X} , and all of the nonzero entries in the resulting k th row of \mathbf{L} are appended to their respective columns in the data structure of \mathbf{L} . Only a real (`double`) version is provided. A complex version could easily be generated. In addition to appearing as a Collected Algorithm of the ACM, LDL Version 1.0 is available at <http://www.cise.ufl.edu/research/sparse>.

The following table illustrates a few performance results on a Pentium 4 laptop with 1GB of memory. Each matrix is permuted with AMD [1, 2]. MATLAB's `chol` computes \mathbf{L} column-by-column using the conventional method, but as a result the columns are not sorted. It then sorts the columns by transposing the matrix and returning \mathbf{L}^T instead. It thus uses twice the memory of the LDL package, which by design constructs \mathbf{L} in place with sorted columns. Memory limitations severely reduced `chol`'s performance on the largest matrix in the table.

Matrix	nonzeros in \mathbf{L} (10^6)	flops (10^9)	symbolic time (sec)		total time (sec)		Mflops	
			LDL	MATLAB	LDL	MATLAB	LDL	MATLAB
Boeing/bcsstk34	0.04	0.004	0.0004	0.005	0.012	0.036	314	108
Boeing/ct20stif	10.68	7.1	0.22	0.86	35.7	41.2	200	173
Nasa/nasasrb	11.90	4.8	0.14	0.91	26.1	31.0	183	154
Boeing/pwtk	59.84	46.8	0.70	4.13	246.2	1728.4	190	27

4 Using LDL in MATLAB

The simplest way to use LDL is within MATLAB. Once the LDL mexFunction is compiled and installed, the MATLAB statement `[L, D, Parent, f1] = ldl (A)` returns the sparse factorization $\mathbf{A} = (\mathbf{L}+\mathbf{I})\mathbf{D}(\mathbf{L}+\mathbf{I})'$, where \mathbf{L} is lower triangular, \mathbf{D} is a diagonal matrix, and \mathbf{I} is the n -by- n identity matrix (LDL does not return the unit diagonal of \mathbf{L}). The elimination tree is returned in `Parent`. If no zero on the diagonal of \mathbf{D} is encountered, `f1` is the floating-point operation count. Otherwise, `D(-f1,-f1)` is the first zero entry encountered. Let `d=-f1`. The function returns the factorization of $\mathbf{A}(1:d,1:d)$, where rows `d+1` to `n` of \mathbf{L} and \mathbf{D} are all zero. If a sparsity preserving permutation \mathbf{P} is passed, `[L, D, Parent, f1] = ldl (A,P)` operates on $\mathbf{A}(\mathbf{P},\mathbf{P})$ without forming it explicitly.

The statement `x = ldl (A, [], b)` is roughly equivalent to $\mathbf{x} = \mathbf{A}\backslash\mathbf{b}$, when \mathbf{A} is sparse, real, and symmetric. The \mathbf{LDL}^T factorization of \mathbf{A} is performed. If \mathbf{P} is provided, `x = ldl (A, P, b)` still performs $\mathbf{x} = \mathbf{A}\backslash\mathbf{b}$, except that $\mathbf{A}(\mathbf{P},\mathbf{P})$ is factorized instead.

5 Using LDL in a C program

The C-callable LDL library consists of nine user-callable routines and one include file.

- `ldl_symbolic`: given the nonzero pattern of a sparse symmetric matrix \mathbf{A} and an optional permutation \mathbf{P} , analyzes either \mathbf{A} or \mathbf{PAP}^T , and returns the elimination tree, the number of nonzeros in each column of \mathbf{L} , and the `Lp` array for the sparse matrix data structure for \mathbf{L} . Duplicate entries are allowed in the columns of \mathbf{A} , and the row

indices in each column need not be sorted. Providing a sparsity-preserving ordering is important for obtaining good performance. A minimum degree ordering (such as AMD [1, 2]) or a graph-partitioning based ordering are appropriate.

- `ldl_numeric`: given L_p and the elimination tree computed by `ldl_symbolic`, and an optional permutation \mathbf{P} , returns the numerical factorization of \mathbf{A} or \mathbf{PAP}^T . Duplicate entries are allowed in the columns of \mathbf{A} (any duplicate entries are summed), and the row indices in each column need not be sorted. The data structure for \mathbf{L} is the same as \mathbf{A} , except that no duplicates appear, and each column has sorted row indices.
- `ldl_solve`: given the factor \mathbf{L} computed by `ldl_numeric`, solves the linear system $\mathbf{Lx} = \mathbf{b}$, where \mathbf{x} and \mathbf{b} are full n -by-1 vectors.
- `ldl_dsolve`: given the factor \mathbf{D} computed by `ldl_numeric`, solves the linear system $\mathbf{Dx} = \mathbf{b}$.
- `ldl_ltsolve`: given the factor \mathbf{L} computed by `ldl_numeric`, solves the linear system $\mathbf{L}^T\mathbf{x} = \mathbf{b}$.
- `ldl_perm`: given a vector \mathbf{b} and a permutation \mathbf{P} , returns $\mathbf{x} = \mathbf{Pb}$.
- `ldl_permt`: given a vector \mathbf{b} and a permutation \mathbf{P} , returns $\mathbf{x} = \mathbf{P}^T\mathbf{b}$.
- `ldl_valid_perm`: Except for checking if the diagonal of \mathbf{D} is zero, none of the above routines check their inputs for errors. This routine checks the validity of a permutation \mathbf{P} .
- `ldl_valid_matrix`: checks if a matrix \mathbf{A} is valid as input to `ldl_symbolic` and `ldl_numeric`.

Note that the primary input to the `ldl_symbolic` and `ldl_numeric` is the sparse matrix \mathbf{A} . It is provided in column-oriented form, and only the upper triangular part is accessed. This is slightly different than the primary output: the matrix \mathbf{L} , which is lower triangular in column-oriented form. If you wish to factorize a symmetric matrix \mathbf{A} for which only the lower triangular part is supplied, you would need to transpose \mathbf{A} before passing it `ldl_symbolic` and `ldl_numeric`.

The follow program illustrates the basic usage of the LDL routines. It analyzes and factorizes the sparse symmetric positive-definite matrix

$$\mathbf{A} = \begin{bmatrix} 1.7 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & .13 & 0 \\ 0 & 1. & 0 & 0 & .02 & 0 & 0 & 0 & 0 & .01 \\ 0 & 0 & 1.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & .02 & 0 & 0 & 2.6 & 0 & .16 & .09 & .52 & .53 \\ 0 & 0 & 0 & 0 & 0 & 1.2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & .16 & 0 & 1.3 & 0 & 0 & .56 \\ 0 & 0 & 0 & 0 & .09 & 0 & 0 & 1.6 & .11 & 0 \\ .13 & 0 & 0 & 0 & .52 & 0 & 0 & .11 & 1.4 & 0 \\ 0 & .01 & 0 & 0 & .53 & 0 & .56 & 0 & 0 & 3.1 \end{bmatrix}$$

and then solves a system $\mathbf{Ax} = \mathbf{b}$ whose true solution is $x_i = i/10$. Note that \mathbf{Li} and \mathbf{Lx} are statically allocated. Normally they would be allocated after their size, $\mathbf{Lp}[n]$, is determined by `ldl_symbolic`. This simple example does not check the return value of `ldl_numeric`, which is n if the factorization is successful, or less than n otherwise.

```
#include <stdio.h>
#include "ldl.h"
#define N 10      /* A is 10-by-10 */
#define ANZ 19   /* # of nonzeros on diagonal and upper triangular part of A */
#define LNZ 13   /* # of nonzeros below the diagonal of L */
int main (int argc, int **argv)
{
    int    Ap [N+1] = {0, 1, 2, 3, 4, 6, 7, 9, 11, 15, ANZ},
           Ai [ANZ] = {0, 1, 2, 3, 1,4, 5, 4,6, 4,7, 0,4,7,8, 1,4,6,9 } ;
    double Ax [ANZ] = {1.7, 1., 1.5, 1.1, .02,2.6, 1.2, .16,1.3, .09,1.6,
                       .13,.52,.11,1.4, .01,.53,.56,3.1},
           b [N] = {.287, .22, .45, .44, 2.486, .72, 1.55, 1.424, 1.621, 3.759};
    double Lx [LNZ], D [N], Y [N] ;
    int Li [LNZ], Lp [N+1], Parent [N], Lnz [N], Flag [N], Pattern [N], i ;
    ldl_symbolic (N, Ap, Ai, Lp, Parent, Lnz, Flag, (int *) NULL, (int *) NULL);
    (void) ldl_numeric (N, Ap, Ai, Ax, Lp, Parent, Lnz, Li, Lx, D, Y, Pattern,
                       Flag, (int *) NULL, (int *) NULL) ;
    ldl_lsolve (N, b, Lp, Li, Lx) ;
    ldl_dsolve (N, b, D) ;
    ldl_ltsolve (N, b, Lp, Li, Lx) ;
    for (i = 0 ; i < N ; i++) printf ("x [%d] = %g\n", i, b [i]) ;
}
```

More example programs are included with the LDL package.

References

- [1] P. R. Amestoy, T. A. Davis, and I. S. Duff. An approximate minimum degree ordering algorithm. *SIAM J. Matrix Anal. Applic.*, 17(4):886–905, 1996.
- [2] P. R. Amestoy, T. A. Davis, and I. S. Duff. Algorithm 8xx: AMD, an approximate minimum degree ordering algorithm. *ACM Trans. Math. Softw.*, 2003 (under submission). Also TR-03-010 at www.cise.ufl.edu.
- [3] A. George and J. W. H. Liu. The design of a user interface for a sparse matrix package. *ACM Trans. Math. Softw.*, 5(2):139–162, Jun. 1979.
- [4] A. George and J. W. H. Liu. *Computer Solution of Large Sparse Positive Definite Systems*. Englewood Cliffs, New Jersey: Prentice-Hall, 1981.
- [5] J. R. Gilbert, C. Moler, and R. Schreiber. Sparse matrices in MATLAB: design and implementation. *SIAM J. Matrix Anal. Applic.*, 13(1):333–356, 1992.
- [6] J. R. Gilbert, E. G. Ng, and B. W. Peyton. An efficient algorithm to compute row and column counts for sparse Cholesky factorization. *SIAM J. Matrix Anal. Applic.*, 15(4):1075–1091, 1994.

- [7] J. R. Gilbert and T. Peierls. Sparse partial pivoting in time proportional to arithmetic operations. *SIAM J. Sci. Statist. Comput.*, 9:862–874, 1988.
- [8] J. W. H. Liu. A compact row storage scheme for Cholesky factors using elimination trees. *ACM Trans. Math. Softw.*, 12(2):127–148, Jun. 1986.
- [9] J. W. H. Liu. The role of elimination trees in sparse factorization. *SIAM J. Matrix Anal. Applic.*, 11(1):134–172, 1990.
- [10] J. W. H. Liu. A generalized envelope method for sparse factorization by rows. *ACM Trans. Math. Softw.*, 17(1):112–129, 1991.
- [11] E. G. Ng and B. W. Peyton. A supernodal Cholesky factorization algorithm for shared-memory multiprocessors. *SIAM J. Sci. Comput.*, 14:761–769, 1993.
- [12] E. Rothberg and A. Gupta. Efficient sparse matrix factorization on high-performance workstations - Exploiting the memory hierarchy. *ACM Trans. Math. Softw.*, 17(3):313–334, 1991.
- [13] G. W. Stewart. Building an old-fashioned sparse solver. Technical report, Univ. Maryland (www.umd.cs.edu/~stewart), 2003.