An Investigation of Incomplete Multifrontal LDL^T for the Finite Differences Laplacian

Cameron Holland

Aug 2024

Abstract

We implement and study an incomplete sparse multifrontal LDL^T matrix factorization. We discuss the sparse factorization methods that form the foundation for this factorization and investigate how using the factorization to generate a preconditioner for a conjugate gradient method affects the rate of convergence of the method. We use a five-point finite differences approximation to Laplace's equation on a square grid with Dirichlet boundary conditions to do so.

1 Background Information

Sparse matrices are matrices where enough of the elements are zero that storing only the non-zeros is more efficient than storing the whole matrix. When using classical methods on sparse matrices, there are often many additions and multiplications that involve elements with zero value. Performing these calculations is inefficient since they do not affect the final solution. Sparse solution methods aim to cut this inefficiency and are often able to improve on both time and space complexity.

In his survey of sparse matrix research, Duff lists a myriad of applications where sparse matrices play a significant role [Duf77]. Numerically solving Laplace's equation is one such application where a particular sparse matrix emerges as an incredibly useful tool.

Laplace's equation generally describes physical situations that are time-independent and is the special case of Poisson's equation when there are no point sources within the region. For example, the steady state distribution of heat within a domain can be modeled by the solution of Laplace's equation. Iterative methods are well suited to solve Laplace's equation since approximate solutions are sufficient and iterative methods can take full advantage of the sparsity pattern of the matrix used to solve the equation [AG11].

Laplace's equation can be written as:

$$\nabla^2 f = 0 \tag{1}$$

where f is the function we are studying and ∇^2 is the Laplacian.

Numerically solving Laplace's equation on a square domain can be done effectively by discretizing the domain using a five-point finite differences approximation [Sau06]. To explain this we will use the examples of a 2x2 and 3x3 square grid:

$$\begin{array}{cc} 1 & 3 \\ 2 & 4 \end{array} \qquad\qquad \begin{array}{ccc} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{array} \tag{2}$$

The following is the five-point finite differences approximation used:

$$\nabla^2 f \approx \frac{f(x-h,y) - 2f(x,y) + f(x+h,y)}{h^2} + \frac{f(x,y-h) - 2f(x,y) + f(x,y+h)}{h^2}. \tag{3}$$

The matrix representations of (3) (really of $-\nabla^2$) on the 2x2 and 3x3 square grids shown in (2) respectively are:

$$\begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & 4 & -1 & -1 & 0 \\ 2 & -1 & 4 & 0 & -1 \\ 3 & -1 & 0 & 4 & -1 \\ 4 & 0 & -1 & -1 & 4 \end{array} \qquad \begin{array}{c|ccccccccc} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\ \hline 1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 2 & -1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 3 & 0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\ 4 & -1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\ 5 & 0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\ 6 & 0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\ 7 & 0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\ 8 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\ 9 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 \end{array} \tag{4}$$

The matrices shown in (4) are built from the locations of the spots on the square grids shown in (2). For example, in the 3x3 square grid, the spot 1 is adjacent to itself, 2, and 4. As a result, row 1 in the corresponding five-point finite differences matrix has non-zeros in columns 1, 2, and 4. The 4 at (1,1) means that spot 1 is positively affected by four times the value at spot 1. The -1 at (1,2) means that spot 1 is affected negatively by the value of spot 2. The zeros are where there is no interaction between two spots. A lexicographic arrangement of columns and rows (as shown in (4)), results in a sparse symmetric positive definite (SPD) matrix with a very simple structure. A similarly simple matrix can be found using a five-point finite differences approximation regardless of the grid size.

As the grid size (N) increases, the number of non-zeros (nnz) in the five-point finite differences matrix grows according to this formula:

$$\text{nnz} = 12 + 16(N-2) + 5(N-2)^2. \tag{5}$$

Equation (5) is derived from analyzing an NxN square grid. We note that for each spot in the grid the number of non-zeros present in the matrix is equal to the number of spots that spot interacts with. Every corner spot interacts with themselves and only two other spots. Every edge spot interacts with themselves and three other spots. Every internal spot interacts with themselves and four adjacent spots. Using the fact that there are four corner spots, $4(N-2)$ edge spots, and $(N-2)^2$ internal spots and adding all those interactions together gives (5). The number of nonzeros in the matrix is thus $O(N^2)$. Since the number of total elements in the matrix grows by $O(N^4)$, these matrices have a number of zeros that grows by $O(N^4) - O(N^2) = O(N^2)$. This large growth rate in number of zeros as the grid size and matrix size increases motivates the use of sparse algorithms for these matrices.

In our analysis of how our multifrontal LDL^T factorization performs, we use it to solve Laplace's equation on a square domain. In order to get non-trivial solutions, we impose the following asymmetrical Dirichlet boundary conditions.

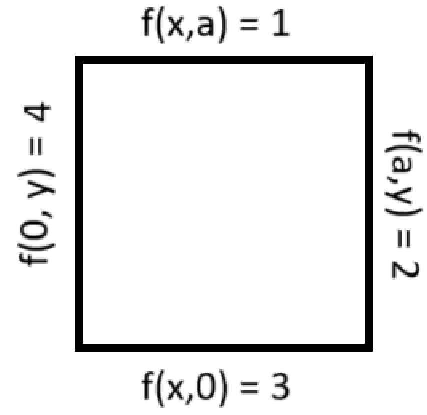

Figure 1: The boundary conditions used to solve Laplace's equation. The square domain has side length a and the origin is located at the bottom-left corner of the domain.

These boundary conditions are chosen because they result in a non-trivial solution and are implemented without disturbing the finite differences matrix so that it stays SPD.

2 Outline

The aim of this thesis is to investigate the characteristics of an incomplete sparse LDL^T factorization, which is a variant of the sparse Cholesky factorization. We start by discussing the sparse Cholesky factorization method ($A = LL^T$) because the symbolic analysis can be reused in the LDL^T variant. Throughout our investigation we will use the matrix that represents the five-point finite differences approximation of Laplace's equation because it is SPD, sparse, and physically significant.

First, a Cholesky factorization will be described following Davis et al.'s survey of direct methods for sparse linear systems [DRSL16]. We will then discuss how the sparse multifrontal LDL^T factorization method builds upon the sparse Cholesky factorization. Finally, we will characterize how dropping small values to create an incomplete multifrontal LDL^T factorization affects the rate of convergence of an iterative method.

We choose a sparse multifrontal LDL^T factorization method to investigate because it efficiently factors the matrices we are concerned with. An LDL^T factorization is the process of breaking a matrix, A, into three matrices which, when multiplied together, reform A. The L factor is lower triangular, the D factor is diagonal, and the L^T factor is a transpose of the L factor. Multifrontal methods are very efficient on multi-core platforms because they are highly parallelizable [GKK97] and they allow algorithms to leverage dense factorization methods[DR83]. LDL^T has the slight advantage over Cholesky in that it allows for factoring indefinite matrices [Fle76] and does not need to perform a square root on every iteration [KM13]. Note that an LDL^T factorization for an indefinite matrix cannot always be formed with a diagonal D factor, instead, the factorization must then be modified to form a block diagonal D factor [BK77]. Additionally, where most other factorizations need to store multiple sparse matrix factors, the LDL^T factorization is more space efficient by storing the L factor as a sparse matrix and the D factor as a dense vector.

The algorithms are implemented in C++. This choice is motivated by the fact that C++ manages memory explicitly when executing algorithms, meaning we have the tools to keep our algorithm running efficiently.

3 Cholesky

The algorithm for a sparse Cholesky factorization can be broken into four steps: compressing the input matrix, building the elimination tree and row subtrees of the matrix, performing a symbolic factorization to generate the non-zero structure of the Cholesky factor, and performing the numerical calculations to fill the Cholesky factor with values.

1. In the first step, the matrix is read and stored in the Compressed Sparse Column (CSC) format that is used generally in sparse solvers [DRSL16].

$$C = \begin{bmatrix} q & 0 & u & w \\ 0 & 0 & v & 0 \\ r & t & 0 & 0 \\ s & 0 & 0 & x \end{bmatrix} \tag{6}$$

$$p[] = [1, \qquad 4, 5, \quad 7, \quad 9] \tag{7}$$

$$i[] = [1, 3, 4, 3, 1, 2, 1, 5] \tag{8}$$

$$a[] = [q, r, s, t, u, v, w, x] \tag{9}$$

   To explain the CSC format, (6) shows an example matrix C. Equations (7), (8), and (9) are the three arrays that represent C in the CSC format. The lowercase letters q-x are arbitrary non-zero numerical values. When using this format there is a hidden index. The p array keeps track of that hidden index's value at the start of each column of the matrix. The i and a arrays keep track of the row and value of each non-zero respectively. For example, to access the first non-zero in the fourth column, start by retrieving the hidden index's starting value by calling $p[4] = 7$. Then extract the row and value by calling $i[7 + 0] = 1$ and $a[7 + 0] = w$. We get the result $C_{1,4} = w$. A second example is to access the second nonzero in the third column, first call $p[3] = 5$. Then $i[5 + 1] = 2$ and $a[5 + 1] = v$, so $C_{2,3} = v$. Note that a sparse matrix's column count is always equal to the length of p minus 1.

2. In the next step, the elimination tree is built. Its structure comes from the first non-zero in each column of the input matrix A. The nodes in the elimination tree are the columns of A. In the Cholesky algorithm, row sub-trees are built from the elimination tree. We leave these out as they are not a part of understanding the multifrontal LDL^T algorithm. See Figure 2.

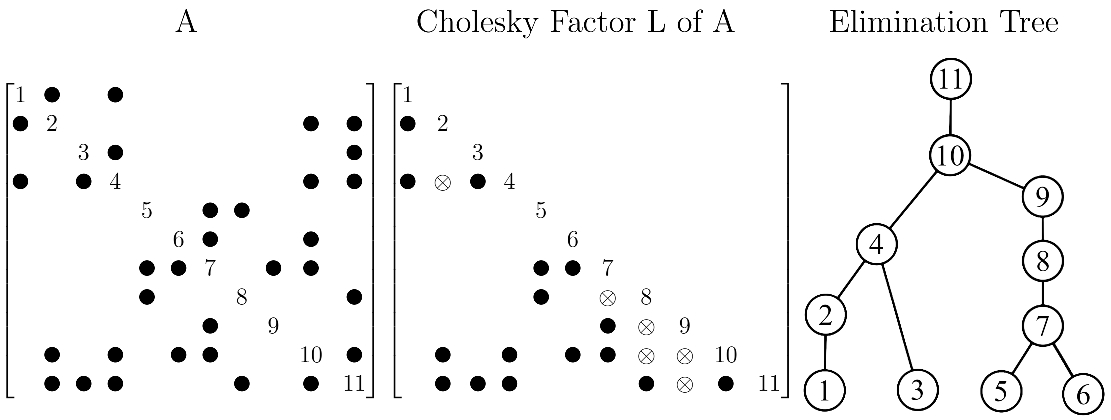

Figure 2: From the survey by Davis et al. Depicted are an example matrix (A), its Cholesky factor (L) and the elimination tree of A. The numbers on the diagonal, the dots, and the circled Xs are non-zero elements. In the elimination tree, the edge between nodes 5 and 7 corresponds to the non-zero at row 7 and column 5 in L. Similarly, all edges in the elimination tree connect the row and column number of the first non-zero in each column of L.

Algorithm 1: Elimination Tree Construction

```
input: A is nxn
output: Elimination Tree

tree ← int array of size n filled with head flag (== −1)
foreach column (k) in A do
    foreach nonzero in k above the diagonal (j) do
        idx ← row of j in A
        while tree[idx] != head flag do
            idx ← tree[idx];
        if idx != k then
            tree[idx] ← k;
return tree
```

Algorithm 1 gives the pseudo-code of our algorithm to fill a one dimensional array with the elimination tree. This algorithm is derived from the explanation of elimination tree construction in Davis et al. [DRSL16]. For the matrix in Figure 2, the generated elimination tree is:

$$\text{tree} = [\underset{1}{2}, \underset{2}{4}, \underset{3}{4}, \underset{4}{10}, \underset{5}{7}, \underset{6}{7}, \underset{7}{8}, \underset{8}{9}, \underset{9}{10}, \underset{10}{11}, \underset{11}{-1}].$$

The algorithm relies on three properties of the elimination tree: no node points to a node with a smaller number, no node points to more than one node, and every nonzero in A guarantees a path between two nodes. The algorithm builds the elimination tree by moving across A from left to right, ensuring that every path that is guaranteed by a non-zero can be followed. Below is a trace of the algorithm for a sample column.

(a) Looking at column 4 of A in Figure 2 there are non-zeros above the diagonal in rows 1 and 3.

(b) The tree when the algorithm is working on column 4 of A would look like $[2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1]$ because the tree is initially filled with head$= -1$ and tree[1] was set to 2 from the non-zero at $A_{1,2}$.

(c) The algorithm would ensure that 1 has a path to 4. It follows 1 to 2 by calling tree[1], then since tree[2] $= -1$, it sets tree[2] $= 4$. 1 now has the path $1 \rightarrow 2 \rightarrow 4$.

(d) The algorithm would then ensure that 3 has a path to 4. Since tree[3] = -1, it sets tree[3] = 4. 3 now has the path $3 \rightarrow 4$.

3. After building the elimination tree, a symbolic analysis is performed on the initial matrix. This analysis produces the non-zero pattern of each column in the Cholesky factor. In Figure 2 the circled Xs in L are what is produced by this analysis. The analysis is a symbolic analysis since only the locations of the non-zeros in the input matrix A are used, not the values themselves. There are many symbolic factorization algorithms available. After implementing the up-looking symbolic factorization [RTL76] and the symbolic factorization using the quotient graph [ADD96], we found a slightly better performance with the quotient graph as expected by theory, so we use this algorithm for the symbolic analysis.

Algorithm 2: Symbolic Factoriztion using the Quotient Graph

```
input: the nonzero structure of the lower triangular part of A as sets
output: nonzero structure of L in sets

$\mathcal{A} \leftarrow$ the nonzero structure of the lower triangular part of A as sets;
$\mathcal{E}$ is empty and the size of $\mathcal{A}$ = the size of A
foreach column (j) in A do
    SetL = $\mathcal{A}$[j];
    foreach SetE in $\mathcal{E}$ do
      if j in SetE then
        SetL $\cup$ SetE;
        delete SetE;
    $\mathcal{E}$[j] $\leftarrow$ (SetL except j)
    foreach $l_1$ in SetL do
      if ($l_1$ == j) continue;
      foreach $l_2$ in SetL do
        $\mathcal{A}[l_1] \leftarrow (\mathcal{A}[l_1]$ except $(l_2))$;
    $\mathcal{A}$[j] = SetL;
  return $\mathcal{A}$;
```

Algorithm 2, an explicit implementation of Davis et al.'s algorithm for symbolic factorization using the quotient graph [DRSL16], starts with the non-zero structure of A in an array of sets. An example input using the matrix A in Figure 2 would be:

$$\mathcal{A} = [\begin{Bmatrix}1\\2\\4\end{Bmatrix}, \begin{Bmatrix}2\\10\\11\end{Bmatrix}, \begin{Bmatrix}3\\4\\11\end{Bmatrix}, \begin{Bmatrix}4\\10\\11\end{Bmatrix}, \begin{Bmatrix}5\\7\\8\end{Bmatrix}, \begin{Bmatrix}6\\7\\10\end{Bmatrix}, \begin{Bmatrix}7\\9\\10\end{Bmatrix}, \begin{Bmatrix}8\\11\end{Bmatrix}, \{9\}, \begin{Bmatrix}10\\11\end{Bmatrix}, \{11\}] \tag{10}$$

Where $\mathcal{A}[1] = \{1, 2, 4\}$ in (10) comes from the fact that the first column of A has non-zeros at and below the diagonal in rows 1, 2, and 4, $\mathcal{A}[2] = \{2, 10, 11\}$ comes from the non-zeros in rows 2, 10, and 11 in the second column of A, and so on. Algorithm 2 uses a quotient graph constructed by creating a number of nodes equal to the number of columns in A and making two nodes adjacent if there is a non-zero connecting them. For example using the matrix in Figure 2, node 1 would be adjacent to nodes 2 and 4 since column one has non-zeros in rows 2 and 4.

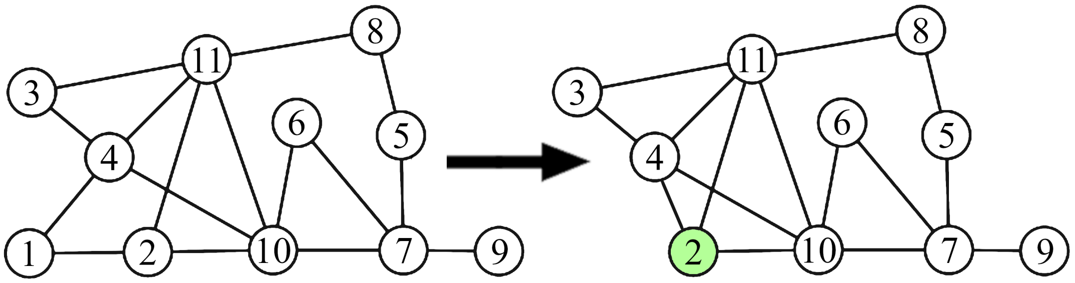

Figure 3: The starting quotient graph for the matrix in Figure 2 and the resulting quotient graph after node 2 has been removed from the graph (the resulting quotient graph after the explanation below). Node 2 has been removed and replaced with the green element 2.

Algorithm 2 then removes the nodes from the quotient graph sequentially, replacing each one with its own element. Whenever a node is removed that is adjacent to an element, it consumes the element. In doing so, the element is deleted and the nonzero pattern of the column in $\mathcal{A}$ is modified. In Figure 3, the algorithm starts by removing node 1 replacing it with element 1, so $\mathcal{E}[1] = \{2, 4\} = (\mathcal{A}[1]$ except 1). Then node 2 is removed, but since node 2 is adjacent to the newly created element 1, node 2 consumes element 1 and 4 is added in the union on line 10 of the code; $\mathcal{A}[2] = \{2, 4, 10, 11\}$. Then element 2 is added to the graph, $\mathcal{E}[2] = \{4, 10, 11\} = (\mathcal{A}[2]$ except 2). The algorithm continues until there are no more columns in A. Note that $\mathcal{A}[2]$ has correctly identified that there must be a non-zero in the Cholesky Factor at row 4 that was not in the non-zero structure of A, see Figure 2.

4. The factorization algorithm finally uses the row sub-trees and symbolic factorization to perform the necessary numerical calculations, appropriately skipping most of the multiplications and additions of zero that would be present in a classical algorithm. We leave the explanation for this step out as it does not play a part in our LDL^T factorization.

4 Multifrontal

The multifrontal factorization begins by building an assembly tree from the elimination tree (see Figure 2). The assembly tree is an ordering of matrices that are formed using the non-zeros from individual columns of A. These dense frontal matrices are factored independently and their results modify both A and L. The multifrontal factorization then starts from the bottom of the assembly tree and crawls up to the top. See Figure 4.

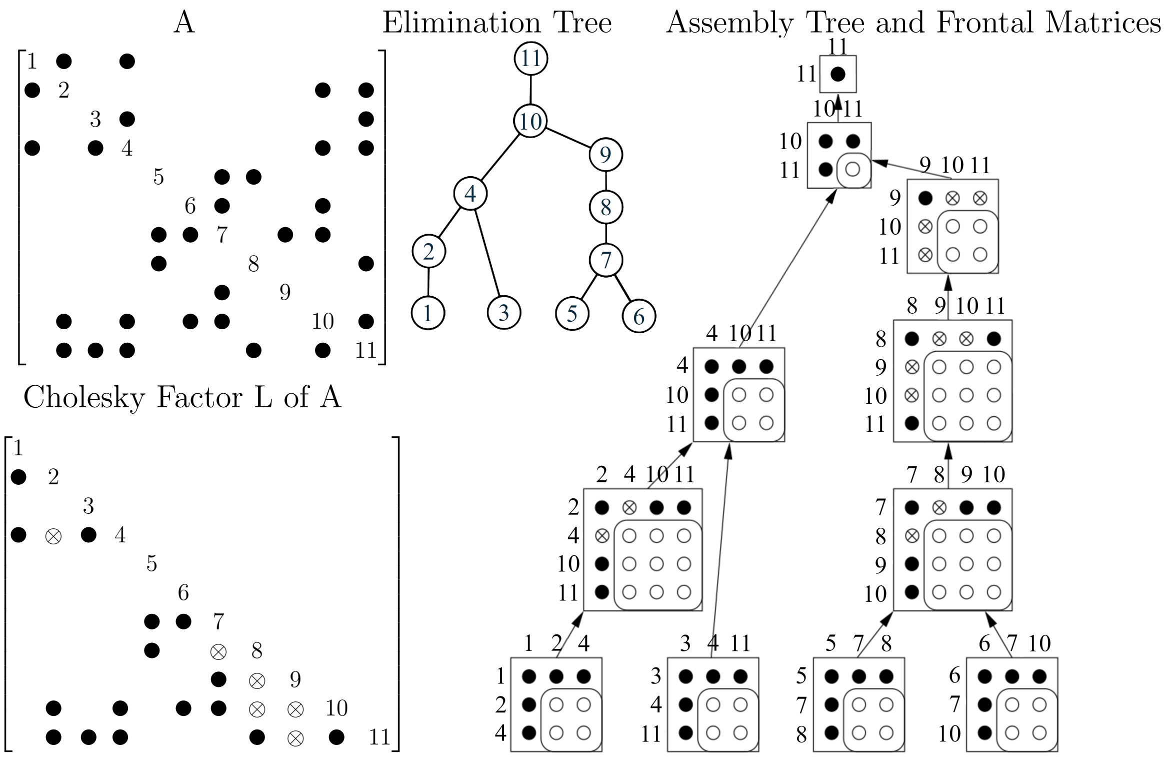

Figure 4: From the survey by Davis et al. Depicted are an example matrix A, the Cholesky factor L of A, the elimination tree of A, and an assembly tree of A. The assembly tree consists of dense matrices, called frontal matrices, formed from the non-zero structure of A. The rounded squares within each frontal matrix represent the values that will be used in later factorizations up the assembly tree.

A frontal matrix can be built for any node in the elimination tree, as an example consider node 3. Examining column 3 of the original matrix we see it has non-zeros at and below the diagonal in row 3, 4, and 11. The frontal matrix for node 3 is therefore formed from the values in A at each permutation of those 3 row numbers: $A_{3,3}, A_{3,4}, A_{3,11}, A_{4,3}, A_{4,4}, A_{4,11}, A_{11,3}, A_{11,4}$, and $A_{11,11}$. While a frontal matrix can be built for any node, in order to preserve numerical accuracy, the factorization must start at the bottom of the elimination tree. In our example this means we begin at nodes 1, 3, 5 and 6.

For a multifrontal LDL^T factorization, the frontal matrices are factored using our variation of Algorithm 1 in Davis's User Guide for LDL [Dav06]. Because we are working under the restriction that all input matrices are SPD and we are using the factorization in a multifrontal scheme, we were able to simplify this algorithm, producing only one column of L per frontal matrix.

Algorithm 3: Sparse Frontal LDL Factorization

```
input: the frontal matrix A and col, the column being worked on (= the node from the elimination tree)
output: a new column inside factors L and D and the values that will be used later factorizations

//build L and D factors
diag ← $A_{col,col}$
column col of L ← $\frac{\text{column col of A}}{diag}$
$D_{col,col}$ ← diag
//multiply back LDL and modify A
foreach nonzero in the new column of L (nzA) do
    foreach nonzero in the new column of L starting at nzA (nzB) do
        AColIdx ← the row index of nzA
        ARowIdx ← the row index of nzB
        $A_{\text{AColIdx,ARowIdx}} \leftarrow (A_{\text{AColIdx,ARowIdx}}) - (\text{nzA}\cdot\text{nzB}\cdot\text{diag})$
```

After each frontal matrix factorization, the L factor has only one more column filled and the D factor has only one new element. The values in the rounded box in Figure 4, which are sent up the assembly tree, are built by lines 9-13 in Algorithm 3, which multiply the frontal factors back out with LDL^T and subtracts those values from A.

The algorithm repeats building frontal matrices and modifying A with the values within the rounded squares, until the head of the tree is reached and the multifrontal factorization is complete. The most important aspect of this multifrontal scheme is that separate branches of the assembly tree do not rely on the modifications from other paths. These separate branches can all be computed in parallel. In our example, each of the leaf matrices can be factored in parallel. The frontal matrix corresponding to node 2 in the elimination tree only has to wait for the 1 leaf to finish factoring before it is free to begin being factored.

5 Incomplete Multifrontal LDL^T

In order to use our multifrontal factorization as a preconditioner, we must compute approximate factors with fewer non-zeros than the exact factors. Common incomplete LDL^T factorization strategies include choosing a sparsity pattern and keeping values within that pattern [MvdV77], dropping values based on their relative magnitude in the original matrix [AJ84], and dropping values based on their magnitude in the computed factor [Tis91]. We chose an approach similar to Tismenetsky's strategy by augmenting our multifrontal LDL^T factorization to set small numbers to zero in the computed L factors of each dense frontal matrix. We use a predetermined drop value, dropping values less than the drop value during factorization.

6 Reorderings

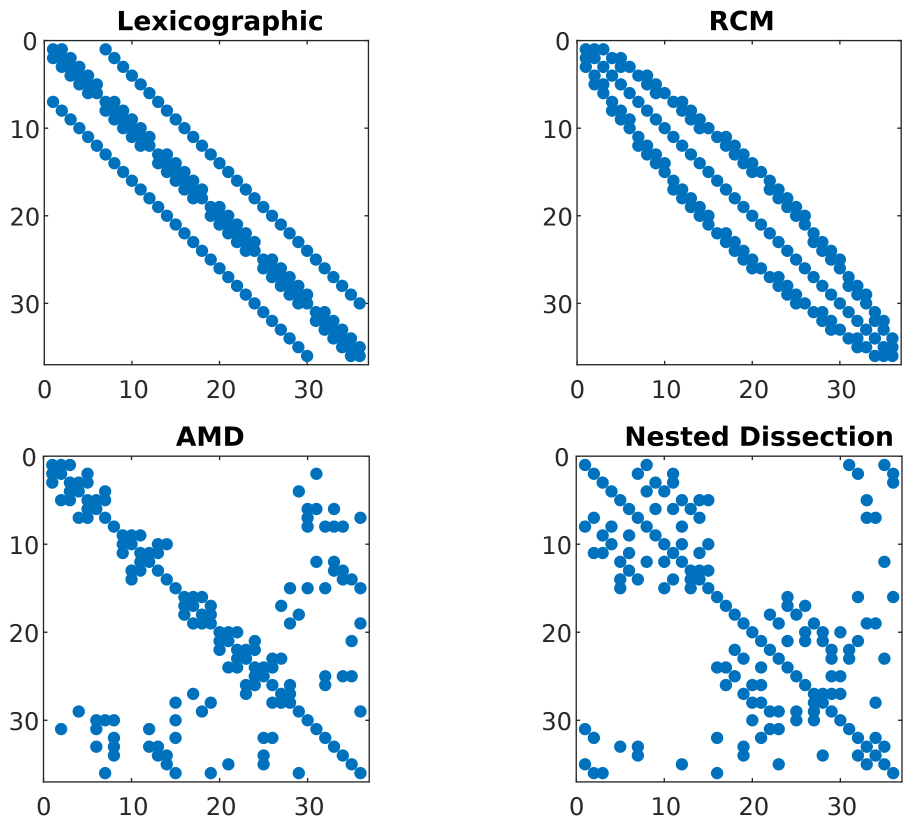

Figure 5: This figure depicts the sparsity pattern of the finite differences matrix for a 6x6 square grid under different reordering strategies. Dots are drawn where there are non-zeros in the matrix. For example, the dots at (1,1) in all the orderings mean that $A_{1,1} \neq 0$.

Certain orderings of sparse matrices can yield better schemes for different algorithms [Saa03]. For a sparse matrix factorization, different choices of orderings can result in matrix factors with vastly different numbers of non-zeros. In iterative solution methods, matrix factors with more non-zeros are more expensive to use but can make more progress per iteration. In this thesis, we investigate how the reversed Cuthill McKee (RCM) [CM69], approximate minimum degree (AMD) [ADD96], and nested dissection [Geo73] ordering techniques affect the performance of our incomplete factorization as a preconditioner. These orderings are chosen because they are effective for reducing fill-in (the increase of non-zeros in the factors) for SPD matrices.

For the incomplete multifrontal LDL^T factorization of a finite differences Laplacian matrix, the lexicographic elimination tree has almost no branching because the first non-zero in almost every column is in the row below the diagonal ($A_{k+1,k} \neq 0$). This makes reorderings particularly interesting because each ordering strategy results in a unique elimination tree structure, potentially presenting unique results/behaviours. Note that the RCM ordering also has a simple elimination tree structure for finite differences Laplacian matrices.

7 Preconditioned Conjugate Gradient

We use *Matlab*'s preconditioned conjugate gradient (PCG) algorithm to measure the effectiveness of our incomplete multifrontal factorization by measuring the number of iterations that PCG takes to converge to the solution of Laplace's equation within a tolerance.

The (non-preconditioned) conjugate gradient method is an iterative method that aims to solve a linear system ($Ax = b$) [HS52]. Before entering the iterative loop, the algorithm uses an initial guess solution (x_0) to compute an initial search direction (p_0) and residual ($r_0 = b - Ax_0$). On each iteration, the algorithm:

1. Travels a computed amount along the current search direction to the new estimate solution and computes a new residual.

2. Computes a new search direction ensuring that it is conjugate with every previous search direction ($p_i \cdot Ap_j = 0 \quad (i \neq j)$).

The algorithm continues iterating until the relative residual is below a given tolerance or it reaches a given iteration limit. PCG enhances this process by using a preconditioner matrix to choose more intelligent search directions on every iteration. The effectiveness of PCG is therefore dependent on the choice of preconditioner matrix.

The algorithm for PCG cited by *Matlab* is in Templates for the Solution of Linear Systems by Barrett, R., M. Berry, T. F. Chan, et al. [BMC+94]. The step in each iteration where the preconditioned matrix is used is the following: **solve** $Mz^{(i-1)} = r^{(i-1)}$, where M is the preconditioning matrix being used, r is a given vector, and z is the desired vector. If $M = LDL^T$, then a solve would use the following steps:

	$LDL^Tz = r$	Given LDL^T and r, solve for z
1.	$Lx = r$	Use Forward Substitution to determine x
2.	$y = D^{-1}x$	Multiplication of x by D^{-1}
3.	$L^Tz = y$	Use Backward Substitution to determine the solution, z

However, if $DL^T = U$, then $M = LU$, a solve would instead use the following steps:

	$LUz = r$	Given LU and r, solve for z
i.	$Ly = r$	Use Forward Substitution to determine y
ii.	$Uz = y$	Use Backward Substitution to determine the solution, z

It follows from studying the steps above that when using an LDL^T factorization as a preconditioner for PCG, it is mathematically equivalent and can be faster to first multiply D into L^T to create an U factor. Creating the U factor involves multiplying one sparse upper triangular matrix by a diagonal matrix in order to skip the vector multiplication in step 2 of every iteration. In this thesis, we create the U factor before running PCG.

8 Results

In this section we present results characterizing our incomplete multifrontal LDL factorization method. We found that the runtime for the (complete) sparse multifrontal LDL^T algorithm was $O(n^2)$, which is slightly worse than the theoretical $O(n^{\frac{3}{2}})$ [GT87]. We numerically solve Laplace's equation on a square domain with Dirichlet boundary conditions using PCG to show how drop values, different sizes of matrices, and reordering strategies affect iteration counts.

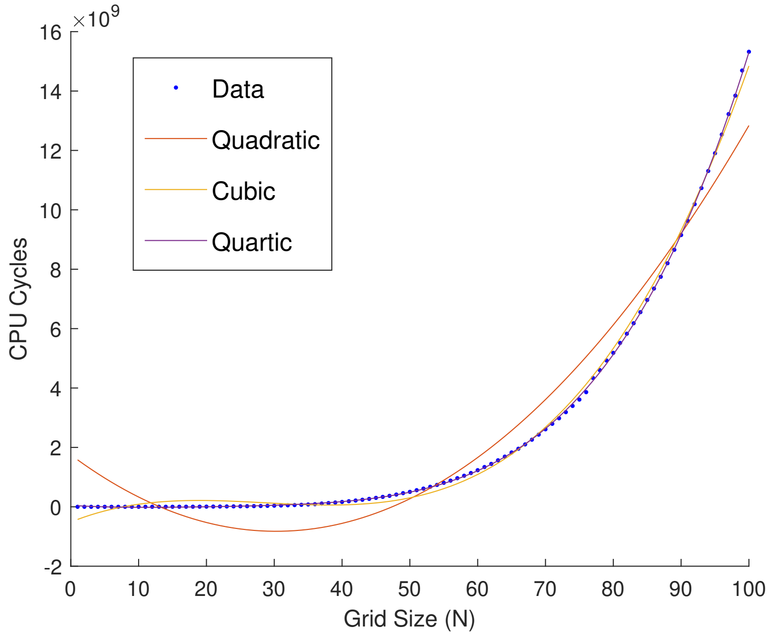

Figure 6: Depicted is the CPU cycle count for our multifrontal LDL^T factorization algorithm to factor the matrices that result from different sized square grids. A lexicographic ordering and a drop value of 0 was used for this figure (the factors produced are complete). Least-squares fits of increasing degree have been applied to the data until a good fit is achieved (quartic). For the quartic the coefficients are: [356, -29.3e3, 1.03e6, -14.2e6, 48.0e6]

In order to measure the performance of our factorization, we factored finite differences matrices for varied grid sizes and recorded the CPU cycles required to complete the factorization. We hope to show that our algorithm is faster than $O(n^3)$, the runtime of a dense LDL^T factorization algorithm on any ordering of a regular finite mesh [Geo73]. Figure 6 shows that a quartic ($O(N^4)$) is the lowest degree polynomial that provides a proper fit to the data. Since the size of the matrix, n, and the number of nonzeros, nnz, both grow by $O(N^2)$, the time complexity of our algorithm is $O(N^4) = O(n^2) = O(\text{nnz}^2)$ where N is the size of the grid. Since $O(n^2) < O(n^3)$, our algorithm is indeed taking advantage of the sparsity of the matrix. With a grid size of 100, the number of CPU cycles is 15.3×10^9. The processor we use runs at 2.8×10^9 Hz which means that we should measure this factorization algorithm taking 5.5 seconds. A test run using the *time* command in bash returns a user time of 5.187 seconds in good agreement with what is expected.

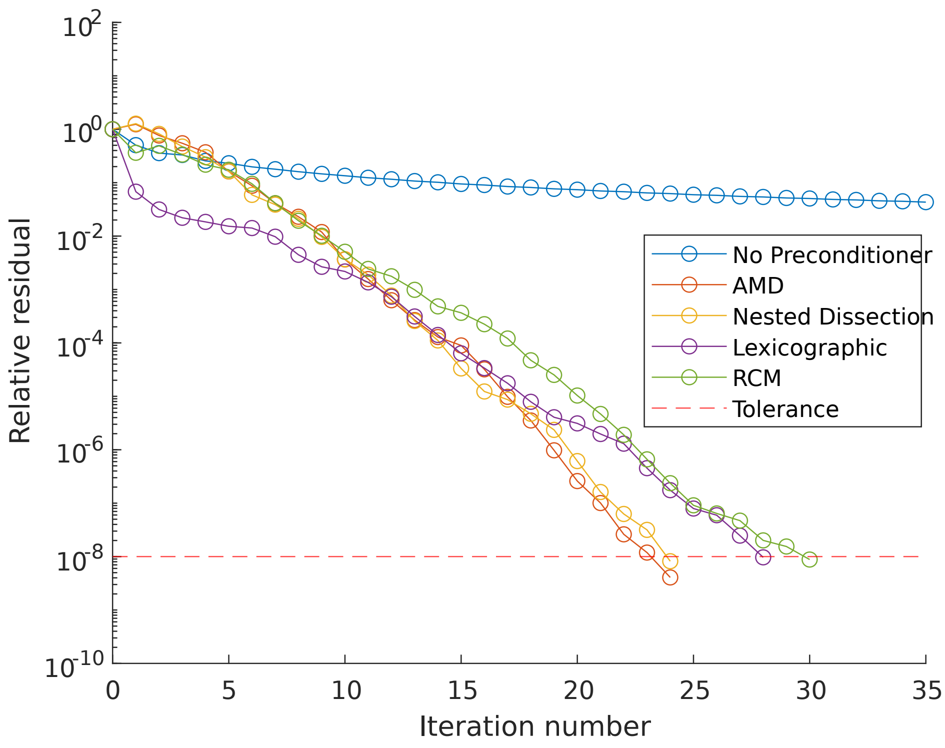

Figure 7: This figure depicts the relative residual at each iteration of *Matlab*'s PCG algorithm for different pre-conditioners. The red-dotted line is the tolerance at which we decided the algorithm is close enough to solution to be considered correct. For this figure matrices are built using a grid size of 100 and a drop value of 0.005.

To characterize our incomplete multifrontal LDL^T factorization, we use it to generate preconditioner matrices and solve a linear system using *Matlab*'s PCG. In Figure 7 we investigate how different orderings affect the convergence rate of PCG and in Figure 8 we explore how different drop values and matrix size affect the convergence rate of PCG. From Figure 7, it is clear that using our preconditioner increases the rate of convergence of PCG. In our small-scale test, the AMD ordering converged within the fewest iterations, followed by nested dissection, lexicographic, and then RCM. While none of these orderings show a significant improvement over any other ordering, the AMD and nested dissection orderings seem to have a marginally faster rate of convergence than the other two orderings.

As mentioned in the Reorderings section, reorderings affect the shape of the elimination and assembly tree because they affect the locations of the nonzeros below the diagonal from which the trees are built. We find that the non-zero pattern of the lexicographic and RCM orderings result in trees with a lower number of branches compared to the AMD and nested dissection orderings for the finite differences approximation matrix.

Since multifrontal LDL^T algorithms start from the bottom of the assembly tree and work upwards, any approximation in a frontal matrix affects the factorization in every frontal matrix that is a direct or indirect parent of that frontal matrix. For the lexicographic ordering of the finite differences matrix, we find that setting a value to zero in a child frontal matrix results in a decrease in magnitude of some values in parent frontal matrices. Assuming this pattern, we would find that dropping values in child matrices increases the chance of dropping values during the parent matrix factorization. This would explain the results of Figure 7 because matrices with more dropped values tend to converge slower and require less work per iteration since there are less non-zero values to calculate with.

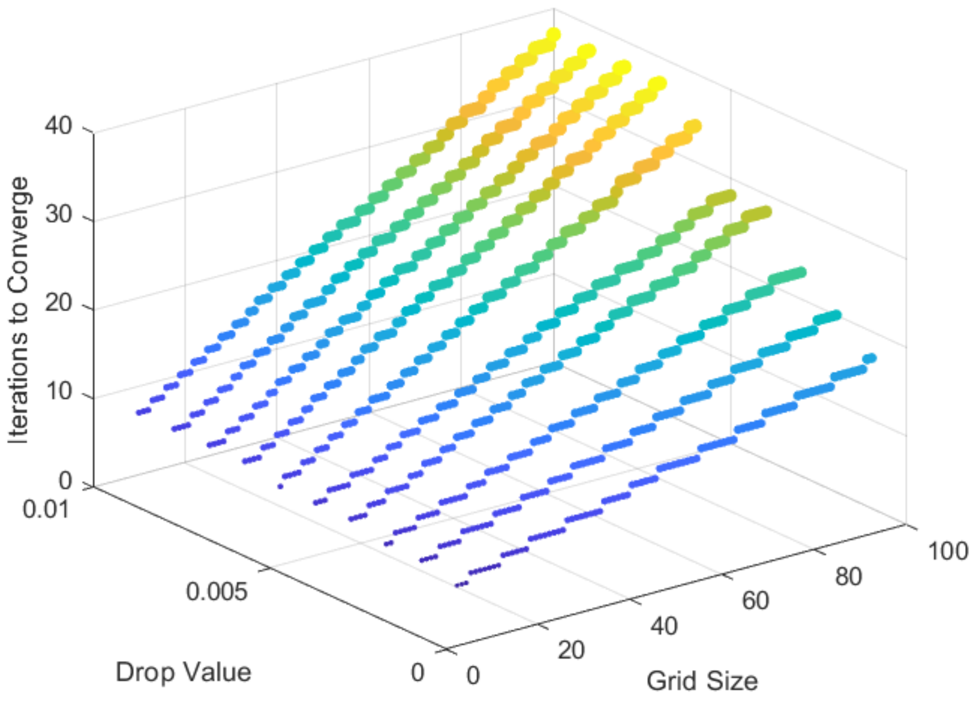

Figure 8: This figure displays how the number of iterations PCG completes in depends on the grid size and drop value used to create the matrix and preconditioner. The z-axis and colour scale both depict the number of iterations the PCG algorithm took to reach the tolerance value given in Figure 7. The x-axis is the grid size and the y-axis is the drop value for the incomplete preconditioner. A lexicographic ordering is used to create this graph and the drop tolerance is varied between 10^{-2} and 10^{-3}.

From Figure 8, we conclude that as the drop value increases more iterations are needed for the algorithm to converge. This is consistent with our conclusion from Figure 7 that matrices with fewer non-zeros take more iterations to converge. We also see that an increase in grid size increases the number of iterations needed for PCG to converge. We would expect PCG to take more iterations to converge for a larger grid size since there are more variables to solve for and more variables contributing to the norm of the residual. It is notable however that the trend lines for fixed drop value are fairly linear (iterations to converge is $O(N)$). This once again stresses that we find it is the number of nonzeros ($O(N)$), not the size of the matrix ($O(N^2)$), that is important when predicting the number of iterations to converge.

9 Summary

In this thesis, we have investigated two algorithms for sparse matrix factorizations and seen that our incomplete sparse mutifrontal LDL^T factorization method has a time complexity of $O(n^2)$, where n is the size of the input matrix. We then applied our own algorithm to solve an instance of Laplace's equation by using a five-point finite differences approximation of a square domain and Matlab's preconditioned conjugate gradient algorithm. We found that our incomplete multifrontal strategy of using a drop value is sensitive to the ordering of the input matrix. Orderings with a higher number of branches resulted in more non-zero values in the resulting matrix factors. We also found that PCG converges at a faster rate when the preconditioner matrix has more non-zeros and when the grid size is smaller.

In future work, one could try to incorporate the symbolic element-based discard rule developed by Vannieuwenhoven and Meerbergen [VM13]. This rule uses an integer parameter, κ, to decide the level at which the algorithm begins approximating the Schur complement. Higher values of κ result in more non-zeros and a more accurate factorization, while $\kappa = 0$ results in no non-zero values outside the sparsity pattern defined by the *element structure* of the intial matrix. A second improvement would be to increase the robustness and efficiency of our algorithm by utilizing dense matrix routines in each frontal matrix factorization. These routines are maximally efficient and use partial pivoting to help reduce rounding error. Additionally, when using these routines we would merge certain nodes in the assembly tree to reduce the number of frontal factorizations [DRSL16].

Currently, our algorithm only supports SPD matrices and is completely single-threaded. Our multifrontal LDL^T algorithm could be extended to allow a block diagonal D factor to be able to factor semi-definite and indefinite matrices and it could be multithreaded to leverage multifrontal's high parallelizability.

References

[ADD96] Patrick R. Amestoy, Timothy A. Davis, and Iain S. Duff. An Approximate Minimum Degree Ordering Algorithm. *SIAM Journal on Matrix Analysis and Applications*, 17(4):886–905, 1996.

[AG11] Uri M. Ascher and Chen Greif. *A First Course in Numerical Methods.* Society for Industrial and Applied Mathematics, Philadelphia, PA, 2011.

[AJ84] M. A. Ajiz and A. Jennings. A robust incomplete Choleski-conjugate gradient algorithm. *International Journal for Numerical Methods in Engineering*, 20(5):949–966, 1984.

[BK77] James Bunch and Linda Kaufman. Some Stable Methods for Calculating Inertia and Solving Symmetric Linear Systems. *Mathematics of Computation*, 31:163–163, 01 1977.

[BMC$^+$94] R.M. Barrett, Berry MW, T. Chan, Donato JM, Jack Dongarra, Victor Eijkhout, Roldan Pozo, Chris Romine, and Henk Van der Vorst. *Templates for the Solution* *of Linear Systems: Building Blocks for Iterative Methods*, volume 43. SIAM, 01 1994.

[CM69] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th National Conference*, ACM '69, page 157–172. Association for Computing Machinery, 1969.

[Dav06] Tim Davis. User Guide for LDL, a concise sparse Cholesky package. *Acta Numerica*, 01 2006.

[DR83] I. S. Duff and J. K. Reid. The Multifrontal Solution of Indefinite Sparse Symmetric Linear. *ACM Trans. Math. Softw.*, 9(3):302–325, sep 1983.

[DRSL16] Tim Davis, Siva Rajamanickam, and Wissam Sid-Lakhdar. A survey of direct methods for sparse linear systems. *Acta Numerica*, 25:383–566, 05 2016.

[Duf77] I.S. Duff. A survey of sparse matrix research. *Proceedings of the IEEE*, 65(4):500–535, 1977.

[Fle76] R. Fletcher. Factorizing symmetric indefinite matrices. *Linear Algebra and its Applications*, 14(3):257–272, 1976.

[Geo73] Alan George. Nested Dissection of a Regular Finite Element Mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363, 1973.

[GKK97] A. Gupta, G. Karypis, and V. Kumar. Highly scalable parallel algorithms for sparse matrix factorization. *IEEE Transactions on Parallel and Distributed Systems*, 8(5):502–520, 1997.

[GT87] J. R. Gilbert and R. E. Tarjan. The analysis of a nested dissection algorithm. *Numer. Math.*, 50(4):377–404, feb 1987.

[HS52] Magnus R. Hestenes and Eduard Stiefel. Methods of conjugate gradients for solving linear systems. *Journal of research of the National Bureau of Standards*, 49:409–435, 1952.

[KM13] Aravindh Krishnamoorthy and Deepak Menon. Matrix inversion using Cholesky decomposition. In *2013 Signal Processing: Algorithms, Architectures, Arrangements, and Applications (SPA)*, pages 70–72, 2013.

[MvdV77] J. A. Meijerink and H. A. van der Vorst. An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-Matrix. *Mathematics of Computation*, 31(137):148–162, 1977.

[RTL76] Donald J. Rose, R. Endre Tarjan, and George S. Lueker. Algorithmic Aspects of Vertex Elimination on Graphs. *SIAM Journal on Computing*, 5(2):266–283, 1976.

[Saa03] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, second edition, 2003.

[Sau06] T. Sauer. *Numerical Analysis*. Pearson Addison Wesley, 2006.

[Tis91] M. Tismenetsky. A new preconditioning technique for solving large sparse linear systems. *Linear Algebra and its Applications*, 154-156:331–353, 1991.

[VM13] Nick Vannieuwenhoven and Karl Meerbergen. IMF: An Incomplete Multifrontal LU-Factorization for Element-Structured Sparse Linear Systems. *SIAM Journal on Scientific Computing*, 35(1):A270–A293, 2013.

10 Appendix

Appendix of Code
.\incomplete.cpp

```
#include <iostream>
#include <vector>
#include <x86intrin.h>

#include "DenseMatrix.hpp"
#include "CSCMatrix.hpp"
#include "Symbolic.hpp"
#include "Incomplete.hpp"
#include "Name.hpp"
#include "CSCOperations.hpp"
#include "SPDTester.hpp"

using namespace std;
typedef uint64_t cpucycle;

int main(int argc, char* argv[])
{
  cpucycle begin = __rdtsc();
  double alpha = 0;
  if (argc > 1)
    alpha = stod(argv[1]); //read in alpha from cmd line if available

  string name = "name here";
  const int x = 100; //chose size of matrix (1-100 produced by convert.m)
  // matlab files are created by convert.m
  TripletMatrix tmat = readMatlabFile(matrixDir + to_string(x) + suffix);

  // Dense matrix support for smaller hand made matrices see DenseMatrix.hpp
//  DenseMatrix dmat = matrixLDLTest();

  // Compresses the matrix into CSC format
  CSCMatrix cmat = convertTripletToCSC(tmat);
//  CSCMatrix cmat = convertDenseToCSC(dmat);

  if (!isSPD(cmat)) //unfinished SPD check
    cout << "input matrix was not symmetric positive definite" << endl;

  CSCMatrix cmat3(cmat); //copy for later error checking

  // Symbolic Analysis
  vector<Set> scriptL = symbolicWithQuotient(cmat);

  // LDL Factorization
  cpucycle begin2 = __rdtsc();
  CSCMatrix lmat;
  vector<double> dvec;
  tie(lmat, dvec) = incompleteLDL(scriptL, cmat, alpha);
  cpucycle end2 = __rdtsc();

  // output matrix to file
//  TripletMatrix outMat = convertCSCToTriplet(lmat);
//  writeMatlabFile(outMat, dvec, lFactorDir + name + suffix, dFactorDir + name +
    suffix);

  // Checking error
  CSCMatrix cmat2 = multiplyLDL(lmat, dvec);
  subtractMatrix(cmat3, cmat2);
  cout << "subtracting matrices: " << endl << cmat3 << endl;
  cout << "magnitude of original matrix: " << matrix2NormMag(cmat) << endl;
  cout << "magnitude of back-multiplied matrix: " << matrix2NormMag(cmat2) <<
      endl;
  cout << "magnitude of subtracted matrix: " << matrix2NormMag(cmat3) << endl;
  cout << "program complete at " << __rdtsc()-begin << " with factor taking " <<
      end2-begin2 << endl;
  cout << getCounters() << endl;
  exit(0);
}
```

.\Incomplete.hpp

```
#pragma once

#include <tuple>
#include <vector>
#include "Symbolic.hpp"

using namespace std;

// Input: Full CSCMatrix
// Output: Lower triangle of CSCMatrix
```

```cpp
// separating this allows quick access to diagonal
CSCMatrix lowerDiagonal(const CSCMatrix& cmat)
{
  CSCMatrix lowerDiag;
  int dim = cmat.p.size() - 1;
  int ctr = 0;
  for (int col = 0; col < dim; col++) {
    for (int idx = cmat.p[col]; idx < cmat.p[col + 1]; idx++) {
      int row = cmat.i[idx];
      if (col > row)
        continue;
      lowerDiag.i.push_back(row);
      lowerDiag.a.push_back(cmat.a[idx]);
      ctr++;
    }
    lowerDiag.p.push_back(ctr);
  }
  return lowerDiag;
}

// Input: Symbolic Analysis Sets
// Output: Inverted Symbolic Analysis Sets
// Used to figure out with nodes on the assembly tree are leaves (not dependent
    on another frontal factorization)
vector<Set> findDependent(const vector<Set>& scriptL)
{
  int dim = scriptL.size();
  vector<Set> scriptQ(dim);
  for (int i = 0; i < dim; i++)
    for (auto it = scriptL[i].begin(); it != scriptL[i].end(); ++it)
      scriptQ[*it].insert(i);
  return scriptQ;
}

// Inserts the next column into the l factor the algorithm is building column by
    column (the workhorse of the algorithm)
void insertColumnToCSC(const int col, const Set& scriptL,
  const CSCMatrix& mmat, CSCMatrix& lmat, double alpha)
{
  int dim = lmat.p.size() - 1;
  int nNonZero = lmat.p[dim];
  for (int i = dim; i < col + 1; i++) { // resize lmat as needed
    lmat.p.push_back(nNonZero);
  }
  int lStart = lmat.p[col];
  int lCtr = 0;
  int mIndex = mmat.p[col];
  int tooFar = mmat.p[col+1];
  double diag = mmat.a[mIndex];
  for (auto it = scriptL.begin(); it != scriptL.end(); ++it, ++mIndex) {
    if (mIndex >= tooFar) {
      break;
    }
    if (mmat.i[mIndex] != *it){
      --mIndex;
      continue;
    }
    double updateVal = mmat.a[mIndex] / diag;
    if (abs(updateVal) > alpha) {
      lmat.i.insert(lmat.i.begin() + lStart + lCtr, *it);
      lmat.a.insert(lmat.a.begin() + lStart + lCtr, updateVal);
      lCtr++;
    }
  }
  dim = lmat.p.size() - 1;
  for (int i = col + 1; i < dim + 1; i++) {
    lmat.p[i] += lCtr;
  }
}

// inserts the next diagonal value into the d vector
void insertDValueToVec(const int col, const double d, vector<double>& dvec)
{
  int dim = dvec.size();
  for (int i = dim; i < col + 1; i++) // resize dvec as needed
    dvec.push_back(0);
  dvec[col] = d;
}

// Updates the matrix based on the factors from the previous frontal matrix
// modifies if the matrix had a value there before
// inserts if the matrix had a 0 there before
void modifyOrInsert(const int row, const int col, CSCMatrix& mmat,
  const double delta)
{
  int idx = mmat.p[col];
  while (mmat.i[idx] < row && idx != mmat.p[col + 1]){
    idx++;
  }
  if (mmat.i[idx] > row || idx == mmat.p[col + 1]) {
    mmat.i.insert(mmat.i.begin() + idx, row);
    mmat.a.insert(mmat.a.begin() + idx, -delta);
    int dim = mmat.p.size() - 1;
    for (int i = col + 1; i < dim + 1; i++)
```

```
      mmat.p[i]++;
  } else {
    mmat.a[idx] -= delta;
  }
}

// Updates the matrix based on the factors from the previous frontal matrix
// does (M - frontal LDL) one spot at a time
void modifyMMatrixByLDL(const int col, CSCMatrix& mmat, const CSCMatrix& lmat,
    const double diag)
{
  for (int colIdx = lmat.p[col]; colIdx < lmat.p[col + 1]; colIdx++) {
    for (int rowIdx = colIdx; rowIdx < lmat.p[col + 1]; rowIdx++) {
      double delta = lmat.a[colIdx] * diag * lmat.a[rowIdx];
      modifyOrInsert(lmat.i[rowIdx], lmat.i[colIdx], mmat, delta);
    }
  }
}

// Performs a single frontal matrix factorization
void singlefrontal(const int col, const Set& scriptL, CSCMatrix& mmat,
  CSCMatrix& lmat, vector<double>& dvec, double alpha)
{
  insertColumnToCSC(col, scriptL, mmat, lmat, alpha);
  double diag = mmat.a[mmat.p[col]];
  insertDValueToVec(col, diag, dvec);
  modifyMMatrixByLDL(col, mmat, lmat, diag);
}

// Erases the column as a dependent from the dependency list
(makes new leaves the algorithm can choose from)
// See findDependent
void eraseColumnFromSet(const int col, vector<Set>& scriptQ)
{
  int dim = scriptQ.size();
  for (int i = 0; i < dim; i++)
    scriptQ[i].erase(col);
}

//  Picks the next node to perform frontal factorization on using the dependency
    list
void singleNode(const vector<Set>& scriptL, vector<Set>& scriptQ,
  CSCMatrix& mmat, CSCMatrix& lmat, vector<double>& dvec, double alpha)
{
  int col = 0;
  while (scriptQ[col].size() != 1) // column node is not a leaf
    col++;
  singlefrontal(col, scriptL[col], mmat, lmat, dvec, alpha);
  eraseColumnFromSet(col, scriptQ);
}

// Performs an incomplete LDL factorization given symbolic analysis and the
   matrix
tuple<CSCMatrix, vector<double>> incompleteLDL(const vector<Set>& scriptL,
  const CSCMatrix& cmat, double alpha)
{
  int dim = cmat.p.size() - 1;
  CSCMatrix mmat = lowerDiagonal(cmat);
  CSCMatrix lmat;
  vector<double> dvec;
  vector<Set> scriptQ = findDependent(scriptL);
  for (int nodeCtr = 0; nodeCtr < dim; nodeCtr++) {
    singleNode(scriptL, scriptQ, mmat, lmat, dvec, alpha);
  }
  return make_tuple(lmat, dvec);
}
```

.\Symbolic.hpp

```
#pragma once

#include <set>
#include <vector>

using namespace std;

typedef set<int> Set;

ostream& operator<< (ostream& o, const Set& s)
{
  for (auto i = s.begin(); i != s.end(); ++i)
    o << *i << " ";
  return o;
}

vector<Set> initializeSet(const CSCMatrix& cmat)
{
  const int dim = cmat.p.size() - 1;
  vector<Set> result(dim);
  for (int k = 0; k < dim; k++) {
    for (int j = cmat.p[k]; j < cmat.p[k+1]; j++) {
      if (cmat.i[j] < k) continue;
      int idx = cmat.i[j];
      result[k].insert(idx);
    }
  }
  return result;
}

vector<Set> symbolicWithQuotient(const CSCMatrix& cmat)
{
  vector<Set> scriptA = initializeSet(cmat);
  const int dim = scriptA.size();
  vector<Set> scriptE(dim);
  for (int j = 0; j < dim; j++) {
    Set scriptL = scriptA[j];
    for (auto it = scriptE.begin(); it != scriptE.end(); ++it) {
      if (it->find(j) != it->end()) {
        scriptL.insert(it->begin(), it->end());
        it->clear();
      }
    }
    scriptE[j].insert(scriptL.begin(), scriptL.end());
    scriptE[j].erase(j);
    for (auto itA = scriptL.begin(); itA != scriptL.end(); ++itA) {
      if ((*itA) == j) continue;
      for (auto itB = scriptL.begin(); itB != scriptL.end(); ++itB)
        scriptA[*itA].erase(*itB);
    }
    scriptA[j] = scriptL;
  }
  return scriptA;
}
```

.\CSCOperations.hpp

```
#pragma once

#include <vector>
#include "CSCMatrix.hpp"
#include <cmath>

using namespace std;

CSCMatrix buildLowerTriangle(const CSCMatrix& lmat, const vector<double>& dvec)
{
  CSCMatrix result;
  int dim = lmat.p.size() - 1;
  for (int i = 0; i < dim; i++)
    result.p.push_back(0);
  for (int i = 0; i < dim; i++) {
    for (int j = lmat.p[i]; j < lmat.p[i+1]; j++) {
      for (int k = j; k < lmat.p[i+1]; k++) {
        bool isFound = false;
        int indexOfBeyond = result.p[lmat.i[j]+1];
        double newValue = lmat.a[k]*lmat.a[j]*dvec[i];
        for (int m = result.p[lmat.i[j]]; m < indexOfBeyond; m++) {
          if (result.i[m] == lmat.i[k]) {
            isFound = true;
            result.a[m] += newValue;
          } else if (result.i[m] > lmat.i[k]) {
            indexOfBeyond = m;
            break;
          }
        }
        if (!isFound) {
          result.i.insert(result.i.begin() + indexOfBeyond, lmat.i[k]);
          result.a.insert(result.a.begin() + indexOfBeyond, newValue);
          for (int n = lmat.i[j]+1; n <= dim; n++)
            result.p[n] += 1;
        }
      }
    }
  }
  return result;
}

void fillWithTranspose(CSCMatrix& lmat) {
  int dim = lmat.p.size() - 1;
  for (int i = 0; i < dim; i++) {
    for (int j = lmat.p[i] + 1; j < lmat.p[i+1]; j++) {
      if (lmat.i[j] <= i)
        continue;
      int k = lmat.p[lmat.i[j]];
      while (k < lmat.p[lmat.i[j]+1] && lmat.i[k] < i)
        k++;
      lmat.i.insert(lmat.i.begin() + k, i);
      lmat.a.insert(lmat.a.begin() + k, lmat.a[j]);
      for (int n = lmat.i[j] + 1; n <= dim; n++)
        lmat.p[n] += 1;
    }
  }
}

CSCMatrix multiplyLDL(const CSCMatrix& lmat, const vector<double>& dvec)
{
  CSCMatrix result = buildLowerTriangle(lmat, dvec);
  fillWithTranspose(result);
  return result;
}
```

```

void subtractMatrix(CSCMatrix& amat, const CSCMatrix& bmat) {
  if(amat.p.size() != bmat.p.size()) {
    cout << "subtractMatrix given matrices of different size" << endl;
    exit(EXIT_FAILURE);
  }
  int dim = amat.p.size() - 1;
  for (int i = 0; i < dim; i++) {
    for (int j = bmat.p[i]; j < bmat.p[i+1]; j++) {
      bool isFound = false;
      int indexOfBeyond = amat.p[i+1];
      for (int k = amat.p[i]; k < indexOfBeyond; k++) {
        if (amat.i[k] == bmat.i[j]) {
          isFound = true;
          amat.a[k] -= bmat.a[j];
        } else if (amat.i[k] > bmat.i[j]) {
          indexOfBeyond = k;
          break;
        }
      }
      if (!isFound) {
        amat.i.insert(amat.i.begin() + indexOfBeyond, bmat.i[j]);
        amat.a.insert(amat.a.begin() + indexOfBeyond, -1*bmat.a[j]);
        for (int n = i+1; n <= dim; n++)
          amat.p[n] += 1;
      }
    }
  }
}

double matrix2NormMag(const CSCMatrix& cmat) {
  double count = 0;
  for (size_t i = 0; i < cmat.a.size(); i++)
    count += pow(cmat.a[i], 2);
  return sqrt(count);
}
```

.\CSCMatrix.hpp

```
#pragma once

#include <iostream>
#include <fstream>
#include <vector>

#include "DenseMatrix.hpp"
#include "TripletMatrix.hpp"

```

```
using namespace std;

struct CSCMatrix {
  vector<int> p;
  vector<int> i;
  vector<double> a;

  CSCMatrix() {
    p.push_back(0);
  }

  CSCMatrix(const CSCMatrix& cmat) {
    p = cmat.p;
    i = cmat.i;
    a = cmat.a;
  }
};

CSCMatrix convertDenseToCSC(const DenseMatrix& dmat)
{
  if(dmat.size() == 0 || dmat.size() != dmat[0].size()) {
    cout << "0 size and non-square matrices are not supported" << endl;
    exit(1);
  }
  CSCMatrix cmat;
  int count = 0;
  int dim = dmat.size();
  for (int col = 0; col < dim; col++) {
    for (int row = 0; row < dim; row++) {
      if (dmat[row][col] != 0) {
        cmat.i.push_back(row);
        cmat.a.push_back(dmat[row][col]);
        count++;
      }
    }
    cmat.p.push_back(count);
  }
  return cmat;
}

CSCMatrix convertTripletToCSC(const TripletMatrix& tmat)
{
  CSCMatrix cmat;

  const int dim = tmat.a.size();
  int count = 0;
  int col = 0;
  for (int idx = 0; idx < dim;) {
    for(; col == tmat.j[idx] && idx < dim; idx++) {
      cmat.i.push_back(tmat.i[idx]);
      cmat.a.push_back(tmat.a[idx]);
      count++;
    }
    col++;
    cmat.p.push_back(count);
  }
  return cmat;
}

TripletMatrix convertCSCToTriplet(const CSCMatrix& cmat)
{
  TripletMatrix tmat;

  const int dim = cmat.a.size();
  int col = 0;
  for (int idx = 0; idx < dim; idx++) {
    while (idx >= cmat.p[col + 1])
      col++;
    tmat.i.push_back(cmat.i[idx]);
    tmat.j.push_back(col);
    tmat.a.push_back(cmat.a[idx]);
  }
  return tmat;
}

DenseMatrix convertCSCToDense(const CSCMatrix cmat) {
  DenseMatrix dmat;
  int dim = cmat.p.size() - 1;
  dmat.resize(dim);
  for(int i = 0; i < dim; i++) {
    dmat[i].resize(dim);
    fill(dmat[i].begin(), dmat[i].end(), 0);
  }
  for(int k = 0; k < dim; k++) {
    for(int j = cmat.p[k]; j < cmat.p[k+1]; j++)
      dmat[cmat.i[j]][k] = cmat.a[j];
  }
  return dmat;
}

ostream& operator<< (ostream& o, const CSCMatrix& cmat)
{
  o << cmat.p << endl;
  o << cmat.i << endl;
  o << cmat.a;
  return o;
}
```

.\TripletMatrix.hpp

```
#pragma once

#include <iostream>
#include <fstream>
#include <vector>
#include <string>

using namespace std;

struct TripletMatrix {
  vector<int> i;
  vector<int> j;
  vector<double> a;
};

TripletMatrix readMatlabFile(const string& filename)
{
  TripletMatrix tmat;
  ifstream inFile(filename);
  while (!inFile.eof()) {
    int row;
    inFile >> row;
    if (inFile.eof()) break;
    if (!inFile.good()) {
      cout << "Bad read in input file" << endl;
      exit(EXIT_FAILURE);
    }
    tmat.i.push_back(row - 1);
    int col;
    double value;
    inFile >> col >> value;
    tmat.j.push_back(col - 1);
    tmat.a.push_back(value);
  }
  inFile.close();
  return tmat;
}

void writeMatlabFile(const TripletMatrix& tmat, const vector<double>& dvec,
    const string& tripletFile, const string& dFile)
{
  cout << "writing out to " << tripletFile << " and " << dFile << endl;
  ofstream f(tripletFile);
  for (size_t k = 0; k < tmat.i.size(); k++) {
    f << tmat.i[k] + 1 << " " << tmat.j[k] + 1 << " " << tmat.a[k] << endl;
  }
  ofstream g(dFile);
  for (size_t k = 0; k < dvec.size() - 1; k++) {
    g << dvec[k] << " ";
  }
  g << dvec[dvec.size()-1];
}

ostream& operator<< (ostream& o, const TripletMatrix& t)
{
  const int dim = t.i.size();
  for (int row = 0; row < dim; row++) {
    o << "( " << t.i[row] << ", " << t.j[row] << ", " << t.a[row] << ")";
    if (row != dim - 1)
      o << endl;
  }
  return o;
}
```

.\Name.hpp

```
#pragma once

#include <string>
using namespace std;

const string matrixDir("../convertedMats/");
const string lFactorDir("../lFactors/");
const string dFactorDir("../dFactors/");
const string suffix(".txt");
```

.\SPDTester.hpp

```
#pragma once

#include <vector>
#include "CSCMatrix.hpp"
#include <cmath>

using namespace std;


bool isSymmetric(const CSCMatrix& cmat) // also checks for positive diagonal
    elements
{
```

```cpp
 int dim = cmat.p.size() - 1;
 for (int i = 0; i < dim; i++) {
  for (int j = cmat.p[i]; j < cmat.p[i+1]; j++) {
   if (cmat.i[j] < i) { // above diagonal, skip b/c redundant
    continue;
   }
   else if (cmat.i[j] == i) { // on the diagonal, check that it’s positive for
     free
    if (cmat.a[j] < 0)
     return false;
   }
   else { // below diagonal, check that swapping x and y gives same value
  if (cmat.i[j] >= (int)cmat.p.size()) { // matrix square check
     // cout << "non-square matrix found" << endl;
     return false;
    }
    for (int k = cmat.p[cmat.i[j]]; k < cmat.p[cmat.i[j] + 1]; k++) {
     if (cmat.i[k] == i) {
      if (cmat.a[k] != cmat.a[j])
       return false;
     }
    }
   }
  }
 }
 return true;
}

bool passesVectorDiagRule(const CSCMatrix& cmat)
{
 int dim = cmat.p.size() - 1;
 for (int i = 0; i < dim; i++) {
  int counter = 0;
  for (int j = cmat.p[i]; j < cmat.p[i+1]; j++) {
   if (i == cmat.i[j])
    counter += abs(cmat.a[j]);
   else
    counter -= abs(cmat.a[j]);
  }
  if (counter <= 0)
   return false;
 }
 return true;
}

bool passesFullSPDTest() //const CSCMatrix& cmat)
{
  return true;
}

bool isSPD(const CSCMatrix& cmat)
{
  if (!isSymmetric(cmat)) {
    cout << "Not symmetric, ";
    return false;
  }
  if (passesVectorDiagRule(cmat)) {
    cout << "Skips full check, ";
    return true;
  }
  return passesFullSPDTest(); //cmat);
}
```

.\DenseMatrix.hpp

```
#pragma once
#include <vector>
using namespace std;
typedef vector<vector<double>> DenseMatrix;

DenseMatrix matrixSix()
{
  DenseMatrix m {
    { 1., 0., 3., 4., 0., 0. },
    { 0., 1., 0., 0., 0., 7. },
    { 3., 0., 1., 0., 5., 2. },
    { 4., 0., 0., 1., 6., 0. },
    { 0., 0., 5., 6., 1., 9. },
    { 0., 7., 2., 0., 9., 1. }
  };
  return m;
}

DenseMatrix matrixFour()
{
  DenseMatrix m {
    { 1.8, 0.9, 0.0, 0.5 },
    { 0.9, 1.1, 0.0, 0.0 },
    { 0.0, 0.0, 1.2, 0.6 },
    { 0.5, 0.0, 0.6, 1.1 }
  };
  return m;
}

DenseMatrix matrixLDLTest()
```

```
{
  DenseMatrix m {
    { 1. , 0. , 0. , 0. , 0. , 0.2, 0.2, 0. , 0. , 0. , 0. },
    { 0. , 1. , 0.2, 0. , 0. , 0. , 0. , 0.2, 0. , 0. , 0. },
    { 0. , 0.2, 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0.2, 0.2 },
    { 0. , 0. , 0. , 1. , 0. , 0.2, 0. , 0. , 0. , 0.2, 0. },
    { 0. , 0. , 0. , 0. , 1. , 0. , 0. , 0.2, 0. , 0. , 0.2 },
    { 0.2, 0. , 0. , 0.2, 0. , 1. , 0. , 0. , 0.2, 0.2, 0. },
    { 0.2, 0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. , 0. , 0.2 },
    { 0. , 0.2, 0. , 0. , 0.2, 0. , 0. , 1. , 0. , 0.2, 0.2 },
    { 0. , 0. , 0. , 0. , 0. , 0.2, 0. , 0. , 1. , 0. , 0. },
    { 0. , 0. , 0.2, 0.2, 0. , 0.2, 0. , 0.2, 0. , 1. , 0.2 },
    { 0. , 0. , 0.2, 0. , 0.2, 0. , 0.2, 0.2, 0. , 0.2, 1. }
  };
  return m;
}

DenseMatrix matrixThree()
{
  DenseMatrix m {
    { 25., 15., -5. },
    { 15., 18., 0. },
    { -5., 0., 11. }
  };
  return m;
}

template <typename T>
ostream& operator<< (ostream& o, const vector<T>& t)
{
  if (!t.size()) {
    o << "Size zero";
  } else {
    size_t i;
    for (i = 0; i < t.size() - 1; i++)
      o << t[i] << " ";
    o << t[i];
  }
  return o;
}

ostream& operator<< (ostream& o, const DenseMatrix& m)
{
  for (size_t i = 1; i < m.size(); i++) {
    o << m[i];
    if (i != m.size() - 1)
      o << endl;
  }
  return o;
}
```

.\convert.m

```
for x=1:100
  A = delsq(numgrid('S', x + 2));
  [i,j,val] = find(A);
  data_dump = [i,j,val];
  fid = fopen(['convertedMats/' num2str(x) '.txt'],'w');
  fprintf( fid,'%d %d %f\n', transpose(data_dump) );
  fclose(fid);
end
```

.\read.m

```
name = ['name here'];
lFolder = './lFactors.';
dFolder = './dFactors.';
extension = '.txt';

d = readmatrix(strcat(dFolder, name, extension));
d = sparse(diag(d));
tempmat = readmatrix(strcat(lFolder, name, extension));
i = tempmat(:, 1);
j = tempmat(:, 2);
val = tempmat(:, 3);
l = sparse(i, j, val);

clear lFolder dFolder name extension tempmat i j val
```