OPTIMIZING THE SPARSE CHOLESKY FACTORIZATION

by

Evan Porter

A thesis submitted in partial fulfillment of the requirements
for graduation with Honors in Mathematics.

Whitman College
2025

Certificate of Approval

This is to certify that the accompanying thesis by Evan Porter has been accepted in partial fulfillment of the requirements for graduation with Honors in Mathematics.

_____

Albert Schueller, Ph.D.

Whitman College
May 17, 2025

ABSTRACT OF THESIS

OPTIMIZING THE SPARSE CHOLESKY FACTORIZATION

This thesis explores the structure and computation of Cholesky factorizations for sparse symmetric positive definite matrices. Beginning with a finite element discretization of a boundary value problem, we derive the stiffness matrix and its corresponding linear system and examine the cause and effect of fill-in during factorization.

To reduce fill-in, we investigate symbolic reordering strategies like the Minimum Degree heuristic. We then extend this framework to parallel computing by introducing the elimination tree, column supernodes, assembly trees, and the multifrontal method. These techniques enable efficient computationally sparse factorizations by exploiting the sparsity patterns of the matrix.

Evan Porter
Whitman College
May 2025

Contents

1 Introduction

In the study of linear algebra and numerical methods, matrices arise naturally as tools for representing systems of equations, transformations, and discretizations of differential equations. Among the many types of matrices encountered in practice, a fundamental distinction exists between dense and sparse matrices.

A dense matrix is one in which most entries are nonzero. These matrices are the most common, and algorithms designed for them typically operate without regard to zero structure. In contrast, a sparse matrix contains a significant number of zero entries. Such matrices commonly arise in applications like finite element analysis, graph theory, and machine learning, where the underlying systems model local interactions or constraints. Figure 1 illustrates the difference between a dense matrix and a sparse matrix.

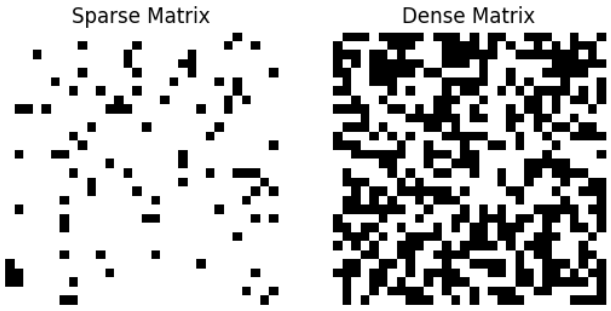


Figure 1: Comparison of a dense matrix (left) and a sparse matrix (right). Notice how the sparse matrix has significantly more zero entries.

However, exploiting sparsity requires more than just ignoring zeros. During matrix factorizations such as the Cholesky decomposition, zero entries may become nonzero due to *fill-in*, leading to increased memory and time requirements. Understanding, predicting, and controlling fill-in is another central challenge within sparse linear algebra.

In this thesis, we study sparse matrices through the lens of numerical linear algebra and graph theory, with a particular focus on the Cholesky decomposition of sparse symmetric positive definite matrices.

2 Motivation

As motivation for using sparse matrices, we will consider a simple example of a 1D boundary value problem. Starting with the following differential equation:

$$\frac{d}{dx}\left(k\frac{du}{dx}\right) + q = 0 \tag{1}$$

Assuming the following boundary conditions:

$$u(0) = u_0, \quad u(L) = u_L$$

We can find the exact solution easily using integration and assuming q and

k are constants:

$$u(x) = -\frac{q}{2k}x^2 + C_1 x + C_2 \tag{2}$$

But what happens when the q and k are variable? Perhaps:

$$k(x) = 1 + sin(x), \quad q(x) = \exp(x)$$

Then it becomes impossible to find the solution analytically. We must use a numerical method to approximate the solution. One such method is the Finite Element Method (hereafter abbreviated as FEM).

2.1 Finite Element Method

The first part of the FEM is to define a set of basis functions ϕ_i, where each phi_i is a continuous, differential, piecewise function defined over the interval $[0, L]$. These functions are constructed to satisfy the following property:

$$\phi_i(x) = \begin{cases} \frac{x - x_{i-1}}{h}, & x \in [x_{i-1}, x_i] \\ \frac{x_{i+1} - x}{h}, & x \in [x_i, x_{i+1}] \\ 0, & \text{otherwise.} \end{cases} \tag{3}$$

where h is the step size between nodes:

$$h = x_i - x_{i-1} = x_{i+1} - x_i \tag{4}$$

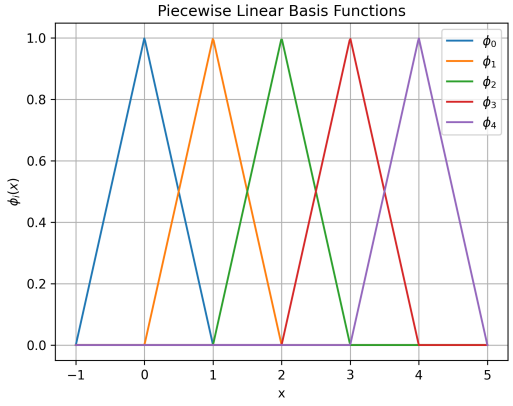


Figure 2: Graph of the piecewise linear basis function ϕ_i as given by Equation 3.

Figure 2 graphs ϕ_i for different values of i.

We can then approximate the solution to Equation 1, u as a linear combination of these basis functions:

$$u \approx \sum_{i=1}^{N-1} \alpha_i \phi_i(x) \tag{5}$$

where α_i are the constants we want to find.

We use these basis functions to help derive the weak form of the differential equation (this is the Galerkin Projection method). We multiply the differential equation by a test function ϕ_j and integrate over the domain $[0, L]$:

$$\int_0^L \phi_j \left[(k_0 u')' + q \right] dx = 0$$

k_0 is the property of the unknown solution (ie: like thermal diffusivity). q is the source term.

Expanding the integral we get:

$$k \sum_{i=1}^{N-1} \alpha_i \int_0^L \phi_i' \phi_j' dx = \int_0^L \phi_j q dx, \quad \forall j = 1, 2, \ldots, N-1. \tag{6}$$

Where the derivative of ϕ_i is:

$$\phi_i' = \begin{cases} \frac{1}{h}, & x \in (x_{i-1},\ x_i) \\ -\frac{1}{h}, & x \in (x_i,\ xi+1) \\ 0, & \text{otherwise} \end{cases}$$

Equation 6 can be represented with the equation:

$$\mathrm{K}\boldsymbol{\alpha} = \mathbf{d}$$

Where:

$$K_{ij} = \int_0^L k \phi_i' \phi_j' dx \tag{7}$$

$$\mathbf{d} = \int_0^L \phi_j q dx \tag{8}$$

First we examine the diagonal entries of K, in other words when $i = j$:

$$K_{ii} = \int_0^L k\phi_i'\phi_i' dx.$$

Since ϕ_i' is zero everywhere except in $(x_{i-1},\ x_i)$ and $(x_i,\ x_{i+1})$, we can reduce the limits of intergration and split the integral:

$$K_{ii} = \int_{x_{i-1}}^{x_i} k\left(\frac{1}{h}\right)^2 dx + \int_{x_i}^{x_{i+1}} k\left(-\frac{1}{h}\right)^2 dx.$$

Evaluating the integrals:

$$K_{ii} = k\left[\frac{1}{h^2}(x_i - x_{i-1}) + \frac{1}{h^2}(x_{i+1} - x_i)\right].$$

Thus we obtain,

$$K_{ii} = k\left[\frac{h}{h^2} + \frac{h}{h^2}\right] = \frac{2k}{h}.$$

Now we look at the off diagonal entries when $j = i + 1$:

$$K_{i,i+1} = \int_0^L k\phi_i'\phi_{i+1}' dx.$$

Since ϕ_i' and ϕ_{i+1}' overlap only in (x_i, x_{i+1}), we compute:

$$K_{i,i+1} = \int_{x_i}^{x_{i+1}} k\left(-\frac{1}{h}\cdot\frac{1}{h}\right) dx.$$

Evaluating the integral:

$$K_{i,i+1} = k\left(-\frac{1}{h^2}\right)(x_{i+1} - x_i).$$

From Equation 4, $x_{i+1} - x_i = h$, we get:

$$K_{i,i+1} = -\frac{k}{h} \tag{9}$$

Similarly, by symmetry:

$$K_{i-1,i} = -\frac{k}{h} \tag{10}$$

Thus we can put this together to get the stiffness K (which is a sparse matrix):

$$K = \frac{k}{h}\begin{bmatrix} 2 & -1 & 0 & 0 & \cdots & 0 \\ -1 & 2 & -1 & 0 & \cdots & 0 \\ 0 & -1 & 2 & -1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \cdots & 2 & -1 \\ 0 & 0 & 0 & \cdots & -1 & 2 \end{bmatrix} \tag{11}$$

In Equation 2 we gave the analytical solution given a set of boundary conditions. Assume now $q = 1$ and $k = 1$. We find then C_1 to be 1 and C_2 to be 0. Thus:

$$u(x) = -\frac{1}{2}x^2 + \frac{1}{2}x,$$

We can validate our analytical derivation, with the finite element method. Using the stiffness matrix from Equation 11 and assuming a constant source term $q = 1$ and constant material property $k = 1$, given $u(0) = 1$ and $u(1) = 0$, we can solve the linear system $K\boldsymbol{\alpha} = \mathbf{d}$ both analytically and numerically. The analytical solution is derived from Equation 2, while the numerical solution is obtained by solving Equation 11. The resulting solution can then be compared to the analytical solution. Figure 3 demonstrates this.

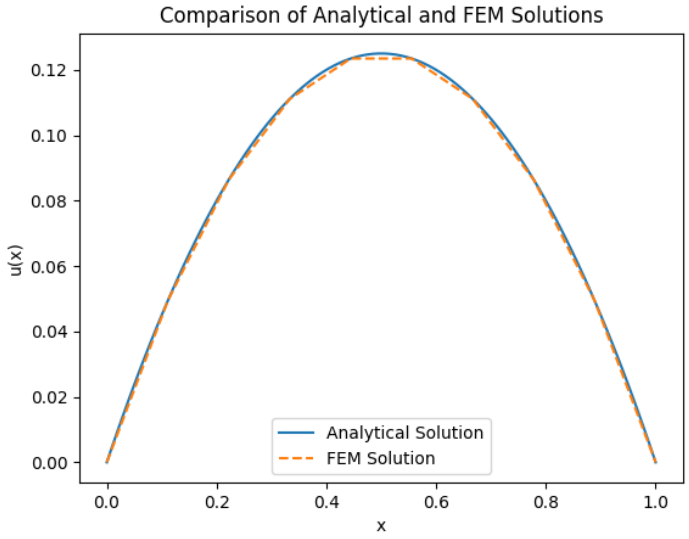


Figure 3: Comparison of the analytical solution and FEM approximation for $N = 10$ nodes.

3 Solving Sparse Linear Systems

As previously disussed, the finite element method above results in the following linear system:

$$\mathrm{K}\boldsymbol{\alpha} = \mathbf{d}$$

Where K is a sparse symmetric positive definite matrix. We can generalize this further into the following form:

$$Ax = b$$

In many cases and depending on the desired accuracy and mesh size, A can be extremely large (some problems require it to be several million row by several million columns [6, pg. 1]). Due to the size the problem quickly transitions from a straightforward modeling problem to a mathematical optimization problem. Thus the method with which you solve for x is important.

There are two common methods for solving for x. The first is gaussian elimination, and the second is the cholesky factorization.

3.1 Gaussian Elimination

When solving a system of equations $Ax = b$ with Gaussian Elimination, the goal is to reduce the matrix A to upper triangular form. This involves using each row to eliminate the entries below the leading coefficient (pivot)

by subtracting a suitable multiple of the pivot row from the rows beneath it. These operations continue from the top-left to the bottom-right of the matrix, ensuring that all entries below the main diagonal become zero. The resulting matrix is an upper triangular matrix, which can be represented as:

$$U = \begin{bmatrix} u_{11} & u_{12} & \dots & u_{1n} \\ 0 & u_{22} & \dots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & u_{nn} \end{bmatrix}$$

Once this is done, we can solve the system of equations by back substitution. Back substitution is the process of starting with $u_{nn}x_n = b_n$ and plugging this in backward (or upward) to solve for x. In Gaussian Elimination the upper triangular matrix is usually obtained by subtracting a multiple of one row from another.For back substitution, solving for each variable requires $\frac{n}{2}$ operations. Since there are n variables there are a total of $\frac{n^2}{2}$ operations done in back substitution. In big O notation this is $O(\frac{n^2}{2}) \approx O(n^2)$. For gaussian elimination, eliminating column j requires $(n-j)^2$ operations. The sum of all these columns is

$$\sum_{j=1}^{n}(n-j)^2 = \sum_{j=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6} \approx \frac{n^3}{3}$$

.

In big O notation this is $O(\frac{n^3}{3}) \approx O(n^3)$.

Thus the total guassian substitution has a complexity of $O(n^3+n^2) \approx O(n^3)$.

The large time complexity of Gaussian Elimination is due to it calculating every element in the matrix. This is not necessary for sparse matrices which has many nonzero elements.

3.2 Cholesky

The Cholesky is different from traditional linear solvers (such as the Gaussian) in that it only opererates on symmetric positive definite matrices.

Definition 3.1 (Symmetric Positive Definite)**.** A matrix that is both symmetric and positive definite if $A = A^T$ the determinant is greater than zero, and all eigenvalues of A are positive. [8, pg. 2]

The Cholesky decomposition factorizes a symmetric positive definite matrix A into:

$$A = LL^T \tag{12}$$

where L is a lower triangular matrix and L^T is a upper triangular matrix

To compute the entries of the lower triangular matrix L, we iterate over each row i and each column j of A, and apply the following formulas based on the relative positions of i and j:

Diagonal elements $(i = j)$

$$L_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} L_{ik}^2}$$

Off-diagonal elements $(i > j)$

$$L_{ij} = \frac{1}{L_{jj}} \left(A_{ij} - \sum_{k=1}^{j-1} L_{ik} L_{jk} \right)$$

Upper triangular entries $(i < j)$

$$L_{ij} = 0 \quad \forall (i < j)$$

Given a system of equations $Ax = b$ where A is a symmetric positive definite matrix, also given a lower triangular matrix L such that $LL^T = A$, we use forward substitution to solve for y in $Ly = b$. Then we use backward substitution to solve for x in $L^T x = y$.

Typically the sparse Cholesky factorization is done by only computing the entries of L that correspond to the nonzeros of A or the places where fill in happens. Because only the nonzero's are taken into account the sparse Cholesky factorization has a time complexity of $O(nc^2)$ where c is the average number of nonzeros per row.

4 Fill In

4.1 Sparsity Pattern

Sparse matrices, by definition, contain significantly more zero elements than nonzero ones. By taking advantage of this structure, we can greatly speed up computations by skipping over the zero entries. However, to do this efficiently, careful thought must be given to how the nonzero elements are stored and organized. To formalize this, we introduce the concept of the sparsity pattern.

Definition 4.1 (Sparsity Pattern)**.** Given a matrix $A \in \mathbb{R}^{n \times n}$, the sparsity pattern of A, denoted $\mathcal{S}(A)$, is the set of index pairs corresponding to non-zero elements in the matrix:

$$\mathcal{S}(A) = \{(i, j) \in \{1, \ldots, n\}^2 \mid A_{ij} \neq 0\}$$

In the case of symmetric matrices, it is sufficient to consider only the lower (or upper) triangular part:

$$\mathcal{S}_{\text{lower}}(A) = \{(i, j) \in \mathcal{S}(A) \mid i \geq j\}$$

When talking about the sparsity pattern as it relates to the cholesky we will hereafter be referring to the lower triangular part.

Example 1. Heres a concrete example to illustrate the concept of a sparsity pattern. On the left, we show a matrix with its nonzero entries (numbers) and blank spaces representing zeros. On the right, we represent the same matrix by marking nonzero entries with asterisks ($*$).

	1	2	3	4
1	7		4	2
2		2		
3	4		6	
4	2			4

$\rightarrow$

	1	2	3	4
1	$*$		$*$	$*$
2		$*$		
3	$*$		$*$	
4	$*$			$*$

The corresponding sparsity pattern $\mathcal{S}(A)$ is the set of index pairs (i, j) where the matrix entries are nonzero. In this case, we have:

$$\mathcal{S}(A) = \{(1,1), (1,3), (1,4), (2,2), (3,3), (4,1), (3,1), (4,4)\}.$$

Now, given the matrix in the previous example, we observe its original sparsity pattern $\mathcal{S}(A)$, where only certain entries are nonzero. However, when we perform Cholesky factorization on A to find $A = LL^T$, additional nonzero entries may appear in L that were originally zeros in A. This phenomenon is known as fill-in.

Example 2. In the diagram on the left we have the matrix from the previous example. In the diagram on the right, we see the sparsity pattern of the Cholesky factor L. Notice that new nonzero entries appear at positions $(3, 4)$ and $(4, 3)$, even though these entries were originally zero

in A. These new nonzero positions are examples of fill-in.

$$A = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & * & & * & * \\ 2 & & * & & \\ 3 & * & & * & \\ 4 & * & & & * \end{array} \qquad L + L^T = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & * & & * & * \\ 2 & & * & & \\ 3 & * & & * & \mathbf{f} \\ 4 & * & & \mathbf{f} & * \end{array}$$

Where **f** represents fill in.

We can formalize the above example of fill-in with the following definition:

Definition 4.2 (Fill-In)**.** Fill-in is when an entry in the Cholesky factor L that is zero in the original matrix A becomes nonzero during factorization due to interactions between existing nonzeros in the matrix and the recursive nature of Cholesky factorization. **??** illustrates the fill-in during a cholesky factorization.

When not managed properly it can lead to increased complexity and storage requirements. With regards to the storage cost, the system has to allocate space every time a new nonzero is created. Worst case scenario this will add a time complexity of $O(\#\text{ fill-ins} \times \#\text{ nonzeros})$ to the algorithm.

4.2 Why Fill-In Occurs

If both $L_{ik} \neq 0$ and $L_{jk} \neq 0$, then there may be fill-in in L_{ij}. As proof consider again the Cholesky for off-diagonal elements:

$$L_{ij} = \left(A_{ij} - \sum_{k=0}^{j-1} L_{ik} L_{jk} \right) / L_{jj}$$

Assume first that $A_{ij} = 0$. Assume second that $L_{ik} \neq 0$ and $L_{jk} \neq 0$ for some $k < j$. Then the subtraction of the sum introduces a non-zero L_{ij}. This leads us to the following heuristic.

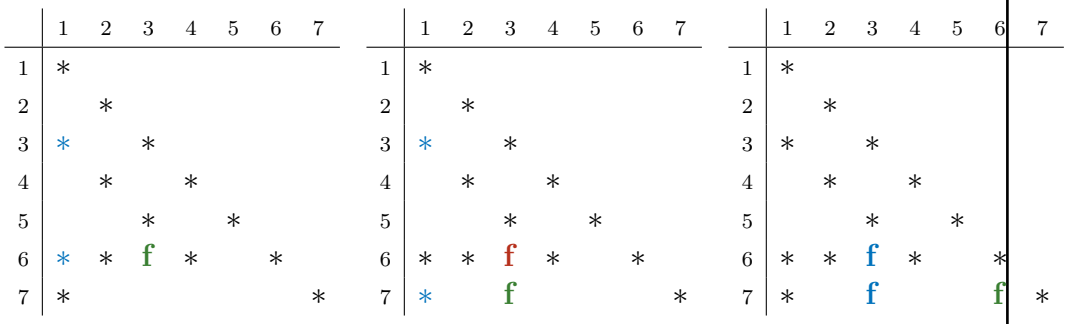
Theorem 4.1 (Fill-In Heuristic)**.** *Let A be a symmetric positive definite matrix and L its Cholesky factor. If $L_{ik} \neq 0$ and $L_{jk} \neq 0$ for some $k < j$, then L_{ij} may be nonzero, even if $A_{ij} = 0$.*

Corollary 4.1.1. *The sparsity pattern of L satisfies:*

$$\mathcal{S}(A) \subseteq \mathcal{S}(L)$$

These are heuristics because they are not always true. There are some edges cases where $A_{ij} = \sum_{k=0}^{j-1} L_{ik} L_{jk}$ in which case $L_{ij} = 0$, but when we are only trying to find the sparsity pattern of L they suffice.

Example 3. Here we demonstrate with a more in depth example of why fill in occurs within the Cholesky factorization.



In the left most matrix, even though the the position at $(6,3)$ (marked in green) is zero in A, the nonzeros at positions $(6,1)$ and $(3,1)$ (marked in blue) mean that $(6,3)$ becomes nonzero in L

In the middle most matrix the nonzeros at positions $(7,1)$ and $(3,1)$ lead to fill in at position $(7,3)$.

In the right most matrix we see that fill in can lead to more fill in as we see here where the previously filled in positions $(6,3)$ and $(7,3)$ create fill in at $(7,6)$. This demonstrates the dependency relation between the columns of L.

In Section 5.4 we will discuss how to reduce fill-in.

5 Symbolic Analysis

5.1 Sparse Matrices as Graphs

A graph $G = (V, E)$ consists of a finite set of vertices $V = (1, 2, \ldots, n)$ and a set of edges $E = ((v_i, v_j) \mid v_i \rightarrow v_j)$. The graph can be represented as the sparse matrix A where $A_{ij} \neq 0$ if there is an edge between i and j. Otherwise $A_{ij} = 0$. This can be represented mathematically as:

$$A = \begin{cases} 1 & \text{if } (i,j) \in E \\ 0 & \text{otherwise} \end{cases}$$

Note this assumes that A is a symmetric matrix and the graph is undirected. Put simply the rows and columns of the sparse matrix correspond to the nodes of the graph. Non zero elements in the matrix represent edges between nodes. On a graph fill in looks like additional edges that are created between nodes.

Example 4. To better understand how sparse matrices relate to graphs, let us consider a concrete example based on the matrix A below:

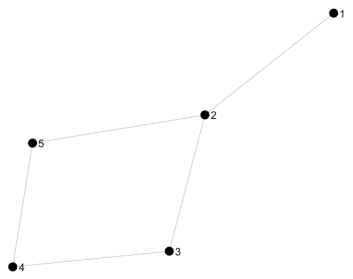
	1	2	3	4	5
1	$*$				
2	$*$	$*$			
3		$*$	$*$		
4			$*$	$*$	
5		$*$		$*$	$*$

In this matrix, nonzero entries are indicated by the $*$ symbol. The corresponding graph G_A has vertices $V = \{1, 2, 3, 4, 5\}$, where an undirected edge exists between two vertices if the corresponding matrix entry is nonzero.

The set of edges E is:

$$E = \{(1, 2), (2, 3), (3, 4), (2, 5), (4, 5)\}.$$

We can visualize this graph as:

$G_A =$ 

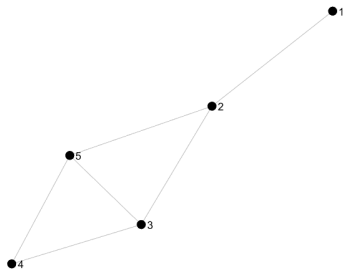
When performing Cholesky factorization $A = LL^T$, fill-in may occur. That is, new edges may be added to the graph corresponding to nonzero entries that arise during elimination, even though they were originally zeros in A.

In particular, in this example, eliminating node 2 introduces a new connection between nodes 3 and 5. This is because node 2 connects to both nodes 3 and 5, and eliminating node 2 creates a new interaction between them.

The updated set of edges after fill-in would be:

$$E_{\text{after fill-in}} = \{(1,2),(2,3),(3,4),(2,5),(4,5),(3,5)\}.$$

The corresponding graph G_L after fill-in would look like this:

$G_L =$ 

5.2 Intermatrix Dependencies

In Cholesky factorization, we process the matrix block by block. At each step, we factor a portion of the matrix and then use that information to update the remaining submatrix. To make this structure clearer, we divide the matrix into vertical blocks. For the lower triangular part of A, each block can be split into two parts: a diagonal block (a square submatrix of size *width* by *width*) and a rectangular block below it. This allows us to express the

matrix in block form. Consider:

$$A_{col} = \begin{bmatrix} A_{\text{diag}} \\ A_{\text{rect}} \end{bmatrix} \quad L_{col} = \begin{bmatrix} L_{\text{diag}} \\ L_{\text{rect}} \end{bmatrix} \tag{13}$$

From here the full matrices A and L would look like:

$$A = \begin{bmatrix} A_{\text{diag}} & A_{\text{rect}}^T \\ A_{\text{rect}} & A_{\text{off}} \end{bmatrix} \quad L = \begin{bmatrix} L_{\text{diag}} & 0 \\ L_{\text{rect}} & L_{\text{off}} \end{bmatrix}$$

Where A_{diag} is a symmetric positive definite submatrix. A_{rect} and A_{off} are appropriately sized blocks.

Example 5. Consider the following matrix A:

$$A = \begin{array}{c|cccc} & 1 & 2 & 3 & 4 \\ \hline 1 & * & & & \\ 2 & * & * & & \\ 3 & * & * & * & \\ 4 & * & * & & * \end{array}$$

We can break this into blocks as follows:

$$A = \begin{bmatrix} \begin{bmatrix} * & \\ * & * \end{bmatrix} & \\ \begin{bmatrix} * & * \\ * & * \end{bmatrix} & \begin{bmatrix} * & \\ & * \end{bmatrix} \end{bmatrix}$$

Since $A = LL^T$ we have:

$$\begin{bmatrix} A_{\text{diag}} & A_{\text{rect}}^T \\ A_{\text{rect}} & A_{\text{off}} \end{bmatrix} = \begin{bmatrix} L_{\text{diag}} & 0 \\ L_{\text{rect}} & L_{\text{off}} \end{bmatrix} \begin{bmatrix} L_{\text{diag}}^T & L_{\text{rect}}^T \\ 0 & L_{\text{off}}^T \end{bmatrix}$$

We can multiply this out to get:

$$A = LL^T = \begin{bmatrix} A_{\text{diag}} & A_{\text{rect}}^T \\ A_{\text{rect}} & A_{\text{off}} \end{bmatrix} = \begin{bmatrix} L_{\text{diag}}L_{\text{diag}}^T & L_{\text{diag}}L_{\text{rect}}^T \\ L_{\text{rect}}L_{\text{diag}}^T & L_{\text{rect}}L_{\text{rect}}^T + L_{\text{off}}L_{\text{off}}^T \end{bmatrix} \tag{14}$$

There are three important things to note here. First is that $A_{\text{diag}} = L_{\text{diag}}L_{\text{diag}}^T$. This is a much smaller symmetric positive definite matrix. Thus we can find L_{diag} by computing the cholesky of A_{diag}.

Second, the bottom-left block, A_{rect}, is the product of L_{rect} and L_{diag}^T. The implications of this is that since A_{rect} and L_{diag} are both known, we can compute L_{rect} using backward substitution.

$$A_{\text{rect}} = L_{\text{rect}}L_{\text{diag}}^T \implies L_{\text{rect}} = A_{\text{rect}}L_{\text{diag}}^{-T}$$

Third, the bottom-right block, A_{off}, is composed of two terms: the contribution from the current column block via $L_{\text{rect}}L_{\text{rect}}^T$, and the remaining part $L_{\text{off}}L_{\text{off}}^T$. There is something interesting to notice about $L_{\text{off}}L_{\text{off}}^T$; that it matches the form of Equation 12. Thus we notice that it represents the

Cholesky factorization of the yet-unprocessed portion of the matrix. To isolate the part of the matrix we still need to factor, we subtract the contribution from the current block:

$$A_{\text{off}}^{\text{updated}} = A_{\text{off}} - L_{\text{rect}} L_{\text{rect}}^T$$

This update is known as the Schur complement [1, pg. 1] [2, pg. 62].

$$S = A_{\text{off}} - L_{\text{rect}} L_{\text{rect}}^T \tag{15}$$

Importantly, this is how Cholesky proceeds recursively: at each step, we factor a diagonal block, solve for the off-diagonal contributions, and update the trailing matrix using the Schur complement. This process continues until the entire matrix is factorized.

This recursive update structure, where each block depends on the factorization and update from earlier blocks, is precisely what gives rise to the elimination tree. The elimination tree encodes these dependencies between columns or supernodes: if the factorization of one block depends on the results of another, there is a parent-child relationship between them in the tree.

Theorem 5.1 (Update Heuristic)**.** *The Schur Complement update (Equation 15) introduces fill-in between every pair of neighbors of the eliminated node.*

Proof. For any pair of neighbors (i, j) within the node k, the (i, j) entry of $L_{\text{rect}}L_{\text{rect}}^T$ is:

$$(L_{\text{rect}}L_{\text{rect}}^T)_{ij} = \sum_k L_{ik}L_{jk}$$

If both L_{ik} and L_{jk} are nonzero for some k, then (i, j) becomes a nonzero in S regardless of whether A_{off} originally contained a zero at that position.

This implies that every pair of neighbors becomes connected in the updated matrix, and thus the neighbors of the eliminated node form a clique. □

Definition 5.1 (Clique)**.** A clique is a set of nodes in a graph such that every node is connected to every other node in the clique. Mathematically, a clique is a set of nodes $C \subseteq V$ such that

$$\forall u, v \in C, u \neq v \rightarrow (u, v) \in E$$

Our Update Heuristic becomes very important as we develop methods to reduce fill in later on.

5.3 Column Replication

While Theorem 5.1 explains why new nonzeros may appear in L, it struggles to capture the pattern in which they appear. Consider Figure 4 which illustrates the progression of the cholesky factorization within a 7x7 sparsity pattern. Notice how the nonzeros in column 1 are repeated in column 2, and they are repeated again in row 5.

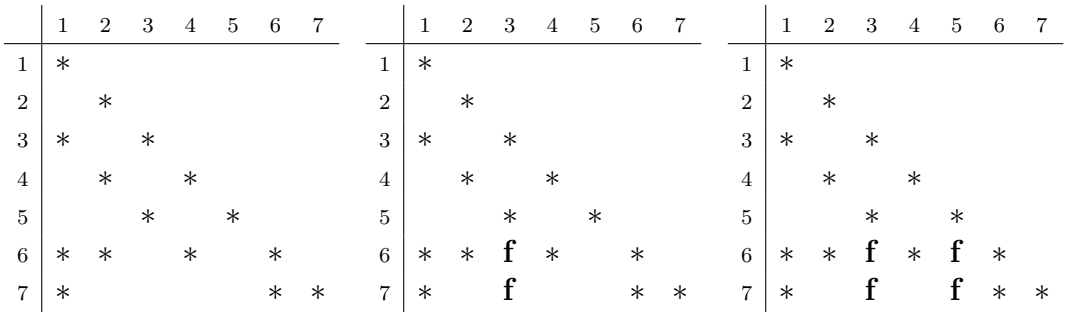


Figure 4: Progression of Cholesky factorization demonstrating column replication

We can make a fundamental observation from this matrix.

Observation 5.1 (Column Replication Principle)**.** *The nonzeros in column j of L are replicated in the sparsity pattern of column i of L for all i such that $L_{ij} \neq 0$ and $i > j$:*

$$\mathcal{S}(L_{:,j}) \subseteq \mathcal{S}(L_{i:n,i}) \quad \textit{if } L_{ij} \neq 0$$

In other words the for each nonzero at row i in column j, the sparsity pattern of column j is replicated in column i such that $i > j$. [7, pg. 54]

We can refer to the first nonzero entry in column j that occurs in a row $i > j$ i as the parent of column j; that is column i is the parent of column j. If column i is the parent of column j then column i will replicate the structure of column j in its own column sparsity pattern. This is the basis

of the elimination tree.

Definition 5.2 (Elimination Tree)**.** The **elimination tree** (or **etree**) of A is a directed acyclic graph such that node j is the parent of node i if $i > j$ and $L_{ij} \neq 0$ and $L_{ik} = 0$ for all $j < k < i$. Equivalently, for each column j, its parent is the smallest index $i > j$ such that $L_{ij} \neq 0$. If no such i exists, then parent$(j) = 0$ and node j is a root. [3, pg. 131] [5, pg. 139]

The elimination tree captures the structure of column replication: the nonzero pattern of column j of L is replicated into each of its ancestors along the path in the elimination tree from j to the root. In particular,

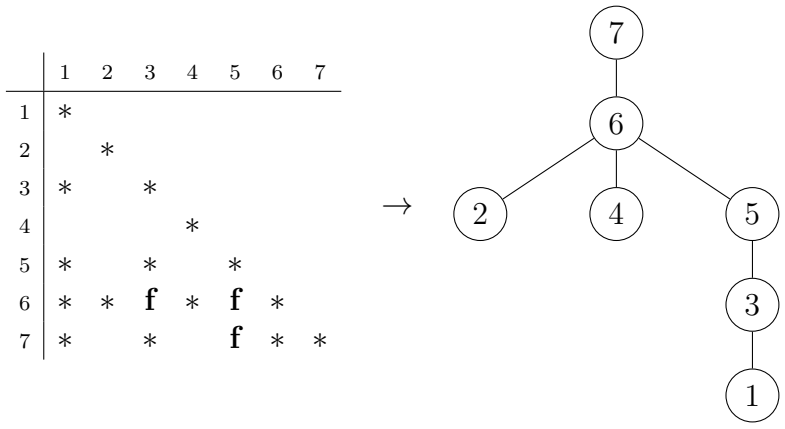
$$\mathcal{S}(L_{:,j}) \subseteq \mathcal{S}(L_{:,\,\mathrm{parent}(j)}) \subseteq \mathcal{S}(L_{:,\,\mathrm{parent}^2(j)}) \subseteq \cdots$$

Example 6. Consider the following matrix A. A has an elimination tree with 7 nodes the structure of the elimination tree is as follows:

$$[3 \quad 6 \quad 5 \quad 6 \quad 6 \quad 7 \quad 0]$$

The matrix A and its corresponding elimination tree represented as a

graph are shown below:



The importance of the elimination tree is that it allows us to understand the dependencies between columns in the Cholesky factorization. By understanding the dependencies, we are able to compute certains blocks of the cholesky simultaneously with other blocks. Each independent subtree in the elimination tree can be processed in parallel with other independent subtrees.

5.4 Reducing Fill In

As established by Theorem 5.1, every time a node is eliminated during Cholesky factorization, fill-in may be introduced between all of its neighbors. This corresponds to forming a clique among the neighbors in the graph representation of the matrix. Thus, the order in which nodes are eliminated directly influences how much fill-in is created.

We then use reordering as a tool to minimumuze the fill in and preserve the sparsity of A in L. The goal is to choose an elimination order that reduces

the creation of new edges, ideally keeping the Cholesky factor as sparse as possible. One of the main tools for doing this is the use of permutation matrices.

5.4.1 Permutations

Permutation matrices allow sparse matrices to be strategically reordered to reduce fill in. Premultiplying a matrix A by P reorders the matrix rows, while postmultiplying by P^T reorders the columns. [7, p. 25]. A permutation matrix can also be represented as a permutation vector of the form $p = \{p_1, p_2, \ldots, p_n\}$ where p_i is the index of the ith row in the original matrix. For example:

$$P = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix} \quad \text{and} \quad p = \begin{pmatrix} 2 \\ 3 \\ 1 \end{pmatrix}.$$

The proof behind the fact that we can reorder a matrix using a permutation matrix is as follows:

Proof. Let P be an orthogonal matrix, so that $P^T = P^{-1}$ and $PP^T = I$. We then start with the original system of equations:

$$Ax = b$$

To reorder this system using P, define the transformed variables:

$$x' = Px, \quad b' = Pb, \quad A' = PAP^T$$

Substituting $x = P^T x'$ into the original system gives:

$$A(P^T x') = b$$

Multiply both sides by P to eliminate P^T:

$$PAP^T x' = Pb$$

Now use the definitions $A' = PAP^T$ and $b' = Pb$ to write:

$$A'x' = b'$$

This gives a reordered system that is structurally equivalent but potentially sparser. Once we solve for x', the original solution can be recovered via:

$$x = P^T x'$$

□

Permutation matrices are not useful on their own. They must be guided by a strategy for choosing a good ordering. This is where fill-reducing heuristics

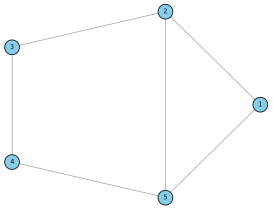
come into play. These heuristics aim to find a permutation that minimizes fill-in during factorization. The most basic of these strategies is the Minimum Degree algorithm.

5.4.2 Minimum Degree

The Minimum Degree algorithm is one of the most fundamental strategies for reducing fill-in during sparse matrix factorization. At each step, it selects the node with the fewest connections (smallest degree) and eliminates it from the graph. This approach minimizes the number of new edges that must be created during elimination.

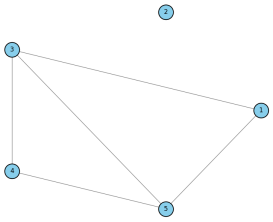
Example 7. Consider the following diagram. The first image shows the original matrix. The second image shows the reordered matrix after eliminating node 2. The third image shows the reordered matrix after eliminating node 1.

Original Matrix



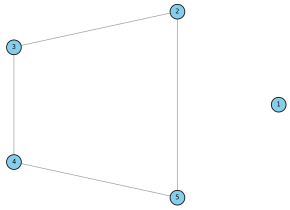
	1	2	3	4	5
1	∗				
2	∗	∗			
3		∗	∗		
4			∗	∗	
5	∗	∗		∗	∗

Eliminating Node 2



	1	2	3	4	5
1	∗				
2		∗			
3	∗	∗	∗		
4			∗	∗	
5	∗		∗	∗	∗

Eliminating Node 1



	1	2	3	4	5
1	∗				
2		∗			
3		∗	∗		
4			∗	∗	
5		∗		∗	∗

Eliminating node 2 results in additional fill-in edges being created, forming a clique among its neighbors (nodes 3 and 5). In contrast eliminating node 1, which has fewer connections, does not introduce any fill-in. This illustrates why the minimum degree algorithm consistently chooses nodes with the smallest degree: to minimize the formation of new edges during elimination.

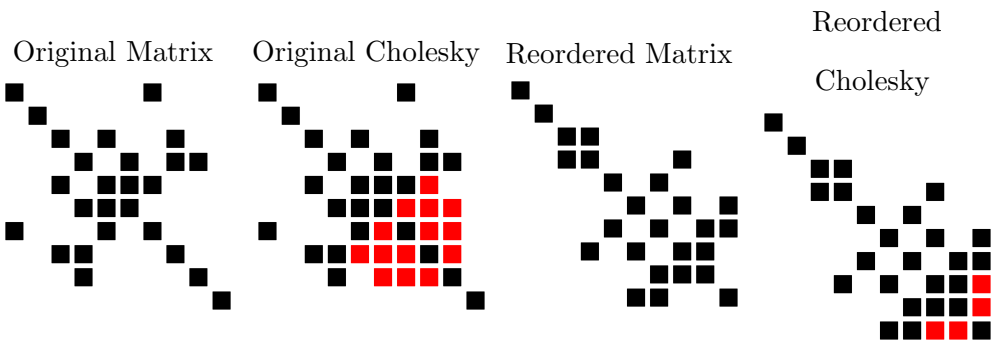
The effectiveness of the minimum degree heuristic relies on a simple observation: eliminating a node of degree d can create up to $\binom{d}{2}$ new edges among its neighbors, assuming none of the neighbors were already connected. By always picking the node with the lowest degree, the algorithm attempts to minimize this fill-in at each step.

The following figure compares the original matrix, its Cholesky factor, and their reordered versions after applying the minimum degree algorithm. In

the Cholesky factor of the unreordered matrix, significant fill-in occurs (highlighted in red). After applying the minimum degree heuristic, the reordered matrix and its Cholesky factor exhibit much less fill-in, leading to a sparser factorization. Note that the Cholesky factors shown are symmetrized ($L+L^T$) to better illustrate the sparsity structure and the effects of fill-in.

Example 8. The following matrices provide an illustrative example of how the minimum degree reordering algorithm can be used to reduce fill in. The permutation vector used is:

$$[1, 9, 0, 6, 8, 2, 3, 4, 5, 7]$$



In the matrices, black represents a nonzero entry, while red represents a fill-in entry. We see that the amount of fill in $L + L^T$ is significantly reduced after reordering the original matrix and calculating the cholesky of the reordered matrix.

Other strategies to reduce fill-in include the Advanced Minimum Degree algorithm, which refines the basic Minimum Degree approach by considering

additional factors such as the structure of the matrix and the potential for future fill-in.

Along side the methods of symbolic analysis we've there exists an equally powerful method of quickly computing the Cholesky factorization of a matrix; that computing the matrix in parallel. This is the focus of the next section.

6 Parallelism

In the previous sections, we explored how the Cholesky factorization unfolds column-by-column, using symbolic tools such as the elimination tree and the Schur complement update to understand the dependencies and fill-in behavior. However, in high-performance computing settings, operating on single columns is often too fine-grained to exploit modern hardware efficiently. Instead, we group adjacent columns with identical sparsity structure into supernodes. Supernodes are the basis of the next section.

6.1 Supernodes

Definition 6.1 (Supernode)**.** [4, pg. 4] A column subset $S = \{s, s+1, \ldots, s+t-1\}$ is a supernode of the matrix L if and only if it is a maximal contiguous column subset satisfying

$$\mathcal{S}(L_{*,s}) = \mathcal{S}(L_{*,s+t-1}) \cup \{s, \ldots, s+t-2\}.$$

In other words, all columns in the supernode share the same sparsity pattern below the diagonal.

Each supernode forms a trapezoidal matrix within L. The upper triangular portion of this trapezoid is zero by defintion of L, the diagonal block is dense and lower triangular, and the rectangular portion below the diagonal is replicated across the supernode via the Column Replication Principle (Observation 5.1). In particular for the rectangular portion the only parts needing to be stored are the lower triangular component and the nonzero rows in the rectangular component.

This block structure allows us to reinterpret the Cholesky factorization of A as operating on supernodes rather than individual columns.

$$A_{block} = \begin{bmatrix} A_{\text{diag}} \\ A_{\text{rect}} \end{bmatrix} \quad L_{block} = \begin{bmatrix} L_{\text{diag}} \\ L_{\text{rect}} \end{bmatrix} \tag{16}$$

It follows then that we can represent the matrix A as Equation 14. It is the same process to update the matrix as before, using the Schur complement. The only difference is that we are now working with blocks rather than individual columns. This allows us to work with larger blocks of data at a time, which can be more efficient in parallel computing environments.

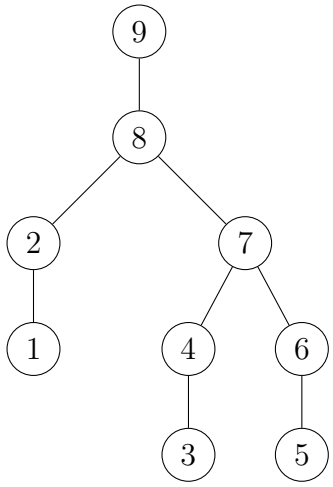
Likewise, the depenendency structure of the matrix is also preserved, so it can be represented with a elimination tree. However because we are working

with blocks of contiguous columns rather than individual columns, we refer to this tree as a Assembly tree.

Definition 6.2 (Assembly Tree)**.** [5, pg. 9] The Assembly tree or the order in which supernodes must be compiled in is found by compressing the elimination tree. Each super node represents a node in the Assembly tree. The parent child relationship represents an edge in the graph. In essence the parent of a supernode is the first supernode that contains any parent of any column in the current supernode.

Example 9. To illustrate this idea of supernodes and the assembly tree consider the following matrix and its corresponding elimination tree.

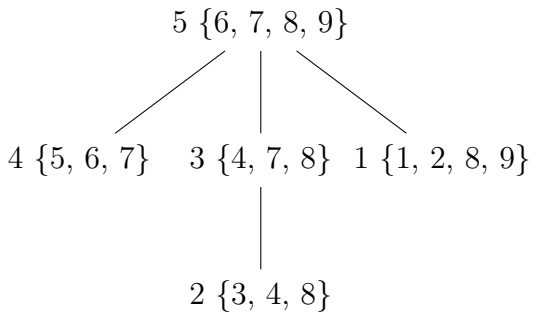
	1	2	3	4	5	6	7	8	9
1	*	*						*	*
2	*	*						*	*
3			*	*				*	
4			*	*			*	*	
5					*	*	*		
6					*	*	**f**	*	*
7				*	*	**f**	*	**f**	*
8	*	*	*	*		*	**f**	*	**f**
9	*	*				*	*	**f**	*



We notice that there are five supernodes within the matrix. The first supernode is composed of columns 1 and 2, the second supernode is composed of columns 3, the third supernode is composed of columns 4, the

fourth supernode is composed of columns 5, and the fifth supernode is composed of column 6, 7, 8 and 9. We can group up the supernodes into the following matrix, we are also able to now display the assembly tree for the matrix as shown on the right.

	1	2	3	4	5	6	7	8	9
1	∗	∗						∗	∗
2	∗	∗						∗	∗
3			∗	∗				∗	
4			∗	∗			∗	∗	
5					∗	∗	∗		
6					∗	∗	**f**	∗	∗
7				∗	∗	**f**	∗	**f**	∗
8	∗	∗	∗	∗		∗	**f**	∗	**f**
9	∗	∗				∗	∗	**f**	∗
	1		2	3	4	5			



We can now zoom in on the first supernode, which consists of columns 1 and 2. Recall that within a supernode, the matrix naturally splits into two parts, a dense lower triangular block corresponding to the diagonal part A_{diag} and a rectangular block corresponding to the off-diagonal part A_{rect}.

$$A^{(1)}_{\text{diag}} = \begin{array}{c|cc} & 1 & 2 \\ \hline 1 & * & \\ 2 & * & * \end{array} \quad A^{(1)}_{\text{rect}} = \begin{array}{c|cc} & 1 & 2 \\ \hline 8 & * & * \\ 9 & * & * \end{array} \quad A^{(1)} = \begin{bmatrix} A_{\text{diag}} \\ A_{\text{rect}} \end{bmatrix} = \begin{array}{c|cc} & 1 & 2 \\ \hline 1 & * & \\ 2 & * & * \\ 8 & * & * \\ 9 & * & * \end{array}$$

This block-based view of factorization lays the groundwork for more advanced algorithms. In particular, it sets the stage for the multifrontal method, where computations are organized around dense submatrices called frontal matrices, and updates from child nodes are delayed and assembled recursively.

6.2 Multifrontal Cholesky

The multifrontal method offers an alternative way of using the supernodes to calculate the cholesky. It does so by dividing A into seperate blocks called frontal matrices. Each frontal matrix is a dense matrix that is formed by the intersection of the supernodes. Recall from Equation 15 that the trailing submatrix is updated by subtracting the outer product of the current column block:

$$S = A_{\text{off}} - L_{\text{rect}} L_{\text{rect}}^T$$

For clarity we can individually define each contibution into a structure known as the contribution matrix V_k.

$$V_k = L_{\text{rect}}^{(k)} L_{\text{rect}}^{(k)T} \tag{17}$$

$L_{\text{rect}}^{(k)}$ refers to the specific rectangular block in the kth node. With a *node* referring to either a column or a supernode.

The contribution matrix holds the Schur complement update from a previously eliminated node k. Unlike V_k which contains only a single update from

node k, the update matrix U_j accumulates all contributions for node j.

$$U_j = - \sum_{k \in \text{children[j]}} L_{\text{rect}}^{(k)} L_{\text{rect}}^{(k)T} \tag{18}$$

Where children[j] is the set of all direct children of node j in the assembly tree. Also note that since each block L_{rect} has different dimensions, summing up the product of $L_{\text{rect}}^{(k)} L_{\text{rect}}^{(k)T}$ will mean the matrix's don't line up one for one. For that we introduce the extend add on operator $\boxplus$.

Definition 6.3 (Extend-Add Operator)**.** Let F_j be a frontal matrix defined over the index set I_j, and let V_k be a contribution matrix defined over a subset $I_k \subseteq I_j$. The extend-add operator is defined as:

$$F_j = F_j \boxplus V_k$$

where $\boxplus$ expands V_k to the full index space I_j by zero-padding the missing rows and columns and then adds it to F_j.

Example 10. As an example consider the following equation:

	1	2	3
1	*	*	*
2	*		
3	*		

$\boxplus$

	3	4
3	*	
4		*

The operation could be rewritten as:

	1	2	3	4
1	*	*	*	
2	*			
3	*			
4				

+

	1	2	3	4
1				
2				
3			*	
4				*

=

	1	2	3	4
1	*	*	*	
2	*			
3	*		*	
4				*

Notice how each matrix is padded with zeros so that the rows and columns line up. The extend add operator is a simple way to combine two matrices of different sizes into one larger matrix.

Now given a supernode j we can define the frontal matrix F_j as the sum of the assembled matrix from A and the update matrix U_j. The frontal matrix is defined as:

$$F_j = \begin{bmatrix} A_{\text{diag}}^{(j)} & A_{\text{rect}}^{(j)T} \\ A_{\text{rect}}^{(j)} & 0 \end{bmatrix} + U_j$$

Note that A_{rect} contains only the nonzero rows. Note that the contribution matrices are assembled from the frontal matrix using Equation 17.

Where $a_{j,j}$ is the diagonal entry of the matrix A and $a_{i,j}$ is the off-diagonal entry. Each i_r is the index of the nonzero row in the rectangular portion of the matrix. The U_j matrix is the update matrix that contains all the updates from previously eliminated nodes.

Example 11. As an illustration of the multifrontal method, recall Example 9. There are five supernodes in the assembly tree. The first supernode is composed of columns 1 and 2 and has non zeroes in row 8 and 9. The frontal matrice and the contribution matrix for this supernode are as follows:

$$F_1 = \begin{array}{c|cccc} & 1 & 2 & 8 & 9 \\ \hline 1 & * & * & * & * \\ 2 & * & * & * & * \\ 8 & * & * & & \\ 9 & * & * & & \end{array} \qquad V_1 = \begin{array}{c|cc} & 8 & 9 \\ \hline 8 & * & \mathbf{f} \\ 9 & \mathbf{f} & * \end{array}$$

The second supernode is composed of column 3 and has non zeros in row 4 and 8. The frontal matrice and the contribution matrix for this supernode are as follows:

$$F_2 = \begin{array}{c|ccc} & 3 & 4 & 8 \\ \hline 3 & * & * & * \\ 4 & * & & \\ 8 & * & & \end{array} \qquad V_2 = \begin{array}{c|cc} & 4 & 8 \\ \hline 4 & * & * \\ 8 & * & * \end{array}$$

The third supernode is composed of column 4 and has non zeroes in row 7 and 8. Furthermore it has a single child in node 2. The frontal matrice and the contribution matrix for this supernode are as follows:

$$F_3 = \begin{array}{c|ccc} & 4 & 7 & 8 \\ \hline 4 & * & * & * \\ 7 & * & & \\ 8 & * & & \end{array} \boxplus V_2 \quad V_3 = \begin{array}{c|cc} & 7 & 8 \\ \hline 7 & * & \mathbf{f} \\ 8 & \mathbf{f} & * \end{array}$$

The fourth supernode is composed of column 5 and has non zeroes in row 6 and 7. The frontal matrice and the contribution matrix for this supernode are as follows:

$$F_4 = \begin{array}{c|ccc} & 5 & 7 & 8 \\ \hline 5 & * & * & * \\ 7 & * & & \\ 8 & * & & \end{array} \quad V_4 = \begin{array}{c|cc} & 7 & 8 \\ \hline 7 & * & \mathbf{f} \\ 8 & \mathbf{f} & * \end{array}$$

The fifth supernode is composed of columns 6, 7, 8, and 9 and has non zeroes in every row. Furthermore it has three children in $\{1, 3, 4\}$ The frontal matrice and the contribution matrix for this supernode are as follows:

$$F_5 = \begin{array}{c|cccc} & 6 & 7 & 8 & 9 \\ \hline 6 & * & & * & * \\ 7 & & * & & * \\ 8 & * & & * & \\ 9 & * & * & & * \end{array} \boxplus V_4 \boxplus V_3 \boxplus V_1$$

In summary, the update matrix U_j extends the Schur complement idea by recursively accumulating all contributions from descendants of node j in the elimination tree. Rather than computing each update at the moment of elimination, the multifrontal method defers these updates and stores them as generated elements to be assembled into future frontal matrices. This enables

more efficient memory access and greater opportunities for parallelism in sparse factorization algorithms.

7 Conclusion

In this thesis, we examined the numerical, algebraic, and symbolic structures underpinning sparse matrix factorization, with a particular focus on Cholesky decomposition. Beginning with a motivating boundary value problem, we used the finite element method to discretize the continuous system, resulting in a large, sparse, symmetric positive definite matrix. From this context, we explored the computational challenges of solving sparse linear systems efficiently.

We demonstrated how naive approaches such as Gaussian elimination lead to unnecessary fill-in and high computational complexity, motivating more specialized techniques like Cholesky factorization. We then introduced the concept of fill-in, provided heuristics for predicting its occurrence, and formalized its behavior through graph-theoretic tools such as the sparsity pattern, elimination tree, and cliques.

To control fill-in, we examined reordering strategies, particularly the Minimum Degree heuristic, and showed how symbolic analysis can dramatically reduce computational cost. Finally, we introduced modern techniques that leverage block structure and parallelism, most notably the multifrontal method and the assembly treeenabling scalable and efficient factorization on

large systems.

Sparse matrix factorization lies at the intersection of numerical linear algebra, graph theory, and high-performance computing. Its applications extend well beyond boundary value problems, appearing in optimization, machine learning, scientific simulations, and more. As these fields continue to evolve, the need for efficient algorithms and data structures will only grow, making the study of sparse matrices and their factorizations increasingly relevant.

References

[1] David Bindel. Cholesky. `https://www.cs.cornell.edu/courses/cs4220/2022sp/lec/2022-02-16.pdf`, 2022. Lecture notes for CS 4220: Numerical Analysis, Spring 2022.

[2] Timothy A. Davis. *Direct Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, 2006.

[3] Joseph W. Liu. A compact row storage scheme for cholesky factors using elimination trees. *ACM Trans. Math. Softw.*, 12(2):127148, June 1986.

[4] Joseph W. H. Liu, Esmond G. Ng, and Barry W. Peyton. On finding supernodes for sparse matrix computations. *SIAM Journal on Matrix Analysis and Applications*, 14(1):242–252, 1993.

[5] Joseph W.H. Liu. The role of elimination trees in sparse factorization. *SIAM Journal on Matrix Analysis and Applications*, 11(1):134–172, 1990.

[6] Pramod Rustagi and Ilya Sharapov. Projecting performance of ls-dyna implicit for large multiprocessor systems. In *Proceedings of the 9th International LS-DYNA Users Conference*. Livermore Software Technology Corporation, 2006.

[7] Jennifer Scott and Miroslav Tma. *Algorithms for Sparse Linear Systems*. Springer International Publishing, 2023.

[8] Patrick Dylan Zwick. Positive definite matrices. `https://www.math.utah.edu/~zwick/Classes/Fall2012_2270/Lectures/Lecture33_with_Examples.pdf`, 2012. Lecture 33 for Math 2270.