An Experimental Study of the Incomplete Cholesky Factorization for the Anisotropic Diffusion Equation

by

Austin Beauchamp

B.Sc., The University of Victoria, 2020

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

THE FACULTY OF GRADUATE AND POSTDOCTORAL STUDIES
(Computer Science)

The University of British Columbia
(Vancouver)

August 2022

© Austin Beauchamp, 2022
The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

**An Experimental Study of the Incomplete Cholesky Factorization for the Anisotropic Diffusion Equation**

submitted by **Austin Beauchamp** in partial fulfillment of the requirements for the degree of **Master of Science in Computer Science**.

**Examing Committee:**

Chen Greif, Professor, Computer Science, UBC  
*Supervisor*

Eldad Haber, Professor, Earth, Ocean and Atmospheric Sciences, UBC  
*Supervisory Committee Member*
Abstract

We consider the numerical solution of large and sparse linear systems arising from a finite difference discretization of the anisotropic diffusion equation. We study preconditioned conjugate gradient as an iterative solver, with the incomplete Cholesky factorization as a preconditioner. The incomplete Cholesky factor is created using two different dynamic dropping parameters: a drop tolerance to control the magnitude of included values, and a fill factor to control the memory allocation/usage of the factor. For large matrices, we show that certain drop tolerances will achieve optimal results with a controllable amount of memory usage. Our numerical experiments are based on a fast C code that has been written as part of this research project, and which uses an asymptotically optimal version of the Cholesky factorization as its base. We illustrate our findings on linear systems of dimensions up to ten million degrees of freedom.
Lay Summary

Differential equations are a type of equation that models physical processes in the real world. For example, a fluid dynamics simulation used for studying vehicle aerodynamics may be modelled as a differential equation. These equations are often very difficult to solve, so we create a system of equations to find an approximate solution.

Solving a system of equations typically consists of having \( n \) equations and \( n \) unknown variables, for some integer number \( n \). When \( n \) is relatively small (order of hundreds, thousands), the system can be solved directly. However, when the system becomes very large (hundreds of thousands, millions, and beyond) the direct approach is no longer doable within a reasonable amount of time. As such, we explore methods for solving these large systems using approximation techniques, along with ways to optimize the run-time to allow for quick solutions.
Preface

At the moment of submission, there have been no publications resulting from this work. I have done all the research for this thesis individually, with the guidance of my research supervisor. I have written the thesis individually, including all coding.
# Table of Contents

Abstract ................................................................. iii

Lay Summary .............................................................. iv

Preface ................................................................. v

Table of Contents ...................................................... vi

List of Tables ........................................................ viii

List of Figures ........................................................ ix

Acknowledgments ...................................................... xi

1 The Anisotropic Diffusion Equation ................................. 1
  1.1 Introduction ...................................................... 1
     1.1.1 Sparse Matrix Storage .................................. 2
  1.2 The Model Problem ............................................. 3
  1.3 Finite Difference Discretization ............................... 4
  1.4 The Linear System .............................................. 5

2 The Cholesky Factorization .......................................... 9
  2.1 Symbolic Factorization ......................................... 11
     2.1.1 Elimination Tree ........................................ 11
     2.1.2 Postordering ............................................ 14
     2.1.3 Column Counts .......................................... 16
2.2 Numerical Computation ........................................... 19
   2.2.1 Triangular Solve ........................................ 19
2.3 Example ........................................................ 21

3 Incomplete Cholesky for Conjugate Gradient ............... 27
   3.1 Conjugate Gradient ........................................ 28
   3.2 Preconditioned Conjugate Gradient ...................... 29
   3.3 Incomplete Cholesky ....................................... 30
       3.3.1 Drop Tolerance and Fill Factor .................... 32
       3.3.2 Factorization Strategies ............................ 34
       3.3.3 Experimental Decisions ............................ 40

4 Numerical Experiments ............................................ 41
   4.1 Evaluation Metrics ....................................... 42
   4.2 Cholesky Factorization .................................... 43
   4.3 ICHOLTP .................................................. 44
       4.3.1 Full Cholesky, Modified Optimal .................. 46
       4.3.2 Effect of Drop Tolerance ............................ 46
       4.3.3 Effect of Fill Factor ............................... 48
       4.3.4 Combined Drop Tolerance and Fill Factor ........ 50

5 Conclusions ....................................................... 61

Bibliography ...................................................... 63
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 2.1</td>
<td>Computation of column counts.</td>
<td>26</td>
</tr>
<tr>
<td>Table 4.1</td>
<td>Solution CPU time breakdown for $n^3 = 27000$.</td>
<td>47</td>
</tr>
<tr>
<td>Table 4.2</td>
<td>Drop tolerance and number of nonzeros in the incomplete factor for the matrix in Figure 4.8 with $n^3 = 15625$.</td>
<td>49</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Square discretization grid</td>
<td>6</td>
</tr>
<tr>
<td>2.1</td>
<td>Example $5 \times 5$ symmetric matrix $A$</td>
<td>12</td>
</tr>
<tr>
<td>2.2</td>
<td>Example Gaussian elimination step</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Elimination tree for the $5 \times 5$ example matrix</td>
<td>14</td>
</tr>
<tr>
<td>2.4</td>
<td>Example Cholesky factor nonzero pattern</td>
<td>14</td>
</tr>
<tr>
<td>2.5</td>
<td>Row subtrees from the elimination tree $T$</td>
<td>15</td>
</tr>
<tr>
<td>2.6</td>
<td>Example $8 \times 8$ matrix</td>
<td>22</td>
</tr>
<tr>
<td>2.7</td>
<td>Connecting node 3 to node 4</td>
<td>22</td>
</tr>
<tr>
<td>2.8</td>
<td>Connecting nodes in column 3</td>
<td>23</td>
</tr>
<tr>
<td>2.9</td>
<td>Fill values implied from column 3 for Figure 2.6</td>
<td>23</td>
</tr>
<tr>
<td>2.10</td>
<td>Connecting nodes 4, 5, and 6 to parent node 7</td>
<td>24</td>
</tr>
<tr>
<td>2.11</td>
<td>Generating row subtree for $k = 7$</td>
<td>25</td>
</tr>
<tr>
<td>2.12</td>
<td>All eight row subtrees for $A$</td>
<td>25</td>
</tr>
<tr>
<td>2.13</td>
<td>$8 \times 8$ Skeleton matrix</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>Histogram of nonzero values in $L$</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>Shift and insertion example for inserting</td>
<td>37</td>
</tr>
<tr>
<td>3.3</td>
<td>Example left-looking Cholesky for a $5 \times 5$ matrix</td>
<td>39</td>
</tr>
<tr>
<td>4.1</td>
<td>Example sparsity patterns for an anisotropic diffusion system at three various sizes</td>
<td>43</td>
</tr>
<tr>
<td>4.2</td>
<td>Cholesky factor sparsity pattern</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>Cholesky factor image</td>
<td>44</td>
</tr>
<tr>
<td>4.4</td>
<td>Cholesky factor histogram</td>
<td>45</td>
</tr>
</tbody>
</table>
Figure 4.5  PCG iterations versus different drop tolerances . . . . . . . . 48
Figure 4.6  Total CPU time for the ICHOLT . . . . . . . . . . . . . . 49
Figure 4.7  Nonzeros in the incomplete Cholesky factor at different drop
tolerances. . . . . . . . . . . . . . . . . . . . . . . . . . . . 50
Figure 4.8  Nonzeros in incomplete factor for the same system at different
drop tolerances . . . . . . . . . . . . . . . . . . . . . . . . . 53
Figure 4.9  Number of PCG iterations versus fill-factor . . . . . . . . . . 54
Figure 4.10  Total CPU time for the ICHOLP factorization plus the PCG solve 54
Figure 4.11  Maximum number of nonzeros and memory size allocated for
the fill-factor incomplete Cholesky factor. . . . . . . . . . . . 55
Figure 4.12  Number of PCG iterations when building preconditioner using
different $t, p$ parameter combinations. $n^3 = 1,728,000, \varepsilon = 0.01$. 55
Figure 4.13  Number of nonzeros in the preconditioner. $n^3 = 1,728,000,$
$\varepsilon = 0.01$. . . . . . . . . . . . . . . . . . . . . . . . 56
Figure 4.14  CPU time taken for constructing the preconditioner and solv-
ing the system with PCG. $n^3 = 1,728,000, \varepsilon = 0.01$. . . . . . 56
Figure 4.15  log10 CPU time, better emphasizing the differences between
parameters. $n^3 = 1,728,000, \varepsilon = 0.01$. . . . . . . . . . . 57
Figure 4.16  Allocated memory in GB, only dependent on $p$. $n^3 = 1,728,000,$
$\varepsilon = 0.01$. . . . . . . . . . . . . . . . . . . . . . . . 57
Figure 4.17  log10 CPU time. $n^3 = 1,728,000, \varepsilon = 0.1$. . . . . . . . 58
Figure 4.18  PCG Iterations. $n^3 = 9,938,375, \varepsilon = 0.01$ . . . . . . . . 58
Figure 4.19  Number of nonzeros in the preconditioner. $n^3 = 9,938,375,$
$\varepsilon = 0.01$ . . . . . . . . . . . . . . . . . . . . . . . . 59
Figure 4.20  CPU time for incomplete factorization and PCG. $n^3 = 9,938,375,$
$\varepsilon = 0.01$ . . . . . . . . . . . . . . . . . . . . . . . . 59
Figure 4.21  Logarithmic scale for CPU time. $n^3 = 9,938,375, \varepsilon = 0.01$ . 60
Figure 4.22  Initial memory allocation for the incomplete factor. $n^3 = 9,938,375,$
$\varepsilon = 0.01$ . . . . . . . . . . . . . . . . . . . . . . . . 60
Acknowledgments

First and foremost, I would like to give a huge thank you to my supervisor, Dr. Chen Greif who was nothing other than entirely supportive, understanding, patient, and encouraging throughout the last year and a half of working together. I would also like to thank the NLA group, our small but dedicated scientific computing group, for their helpful collaboration and our always-fun social gathers.

I would also like to thank my friends who have always encouraged me throughout this long and arduous journey.

I have to thank my loving girlfriend, Morgan. On top of continually supporting me through the program, she’s my motivation, inspiration, and confidence to even pursue graduate studies in the beginning.

I thank my parents for their unconditional love and support in academia and in life.

And lastly, I thank my brother.
Chapter 1

The Anisotropic Diffusion Equation

1.1 Introduction

Solving large and sparse linear systems of the form $Au = f$ has many use cases for solving physical systems that are modelled by differential equations [4, 32]. These differential equations may arise from applications such as electromagnetism [16, 26], fluid/aerodynamics [21], semiconductor device simulation [3, 28], and many more. Solving the linear system is often the main computational bottleneck for the simulation environment. Direct Gaussian elimination is the standard solver for a relatively small matrix or one with no special nonzero structure. Gaussian elimination complexity is $O(n^3)$ [14]. Matrix factorization techniques may introduce improvements, three popular methods being LU [29], QR [11], and Cholesky factorization [6]. However, for very large and sparse systems, Gaussian elimination is far too slow. We will study the optimal Cholesky factorization [8], giving an approachable explanation of why the algorithm is optimal, along with a high-level explanation behind the mechanisms in the algorithm.

When the systems become so large that direct solvers become infeasible, iterative methods can offer approximate solutions much faster [9]. One of the most successful and popular methods for solving symmetric positive definite (SPD) systems is the preconditioned conjugate gradient method (PCG/CG) [17]. The conju-
gate gradient method minimizes the energy norm of the error in an optimal fashion, with a guaranteed convergence in exact arithmetic, but more importantly will often significantly reduce the error in the first few iterations. In cases where the matrix is very difficult to solve, a preconditioner can be applied to the system to improve the performance of CG [27]. Two common preconditioners based on factorization techniques are incomplete-LU (ILU) and incomplete-Cholesky (IC or ICHOL). These incomplete factors are created using the same algorithms as their complete factorization counterparts, but may ignore (or “drop”) certain elements. For example, the incomplete factor may only include nonzeros in certain locations, or ignore elements which do not meet a set of numerical requirements. The incomplete factor takes significantly less time to create compared to the full factorization and can improve PCG performance compared to unpreconditioned CG.

ILU(0) only keeps nonzeros in the $L$, $U$ matrices where values are nonzero in $A$. This is referred to as a static pattern, as the nonzero pattern is determined before factorization. Common incomplete LU factorizations include ILU(0) and ILU($t$, $p$) (or ILUTP) [27]. ILU($t$, $p$) drops elements dynamically, depending on a drop tolerance, $t$, to ignore small elements, and a fill-factor, $p$, to only keep a certain amount of additional information per each row/column.

This thesis will study the anisotropic diffusion equation with constant coefficients. The matrix generated from this differential equation will be sparse, symmetric, and positive definite. Since the matrix is SPD, the Cholesky factorization becomes a candidate technique for factoring the matrix and solving the system efficiently. We aim to implement and study different incomplete Cholesky factorizations with dynamic dropping, creating IC($t$, $p$) (or ICHOLTP) for large and hard problems.

1.1.1 Sparse Matrix Storage

We store all sparse matrices in compressed-column format. For an $n \times n$ matrix with $m$ nonzero entries, a sparse matrix in compressed-column form can be stored using three arrays:

- data, $d = (d_0, d_1, \ldots, d_{m-1})$,
- row indices, $i = (i_0, i_1, \ldots, i_{m-1})$. 


column pointers, \( p = (p_0, p_1, \cdots, p_{n-1}) \).

The data array, \( d \), contains numerical values of the matrix listed column-wise. The index array, \( i \), contains the row index for each element in \( d \). The pointer array, \( p \), contains the index of each column’s starting element in \( d \). For example, the data and index of the value starting column \( j \) will be \( d_{pj} \) and \( i_{pj} \), respectively.

1.2 The Model Problem

One of the simplest, yet most fundamental partial differential equations (PDEs) in science and engineering is the Poisson equation \([7, 24, 33]\), which is given by

\[
-\nabla^2 u = f, \quad x \in \Omega \\
u = u_D, \quad x \in \partial\Omega
\]

where \( \nabla^2 \) is the Laplacian operator, \( u \) and \( f \) are real or complex-valued functions given on a domain \( \Omega \), with boundary \( \partial\Omega \). The function \( f \) is typically defined as the prescribed or source function which is known, and \( u \) is the unknown function. Boundary conditions are required for \( u \) to have a uniquely defined solution. A common boundary value is of the form \( \alpha u = g \), known as a Dirichlet boundary condition. A standard two-dimensional domain is a square centred at zero, \( \Omega = (-1, 1) \times (-1, 1) \). In two dimensions, Poisson’s equation can be represented as

\[
-\left( \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = f(x, y). \tag{1.1}
\]

A slight departure from Poisson’s equation results in the more complex anisotropic diffusion equation. An example equation can be characterized by scaling one of the partial terms in (1.1) by a factor \( \varepsilon \). Again in two dimensions, this will look like

\[
-\left( \varepsilon \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} \right) = f(x, y). \tag{1.2}
\]

There are other possibilities for \( \varepsilon \). For example, a variable coefficient instead of a scalar.

A common situation in physical applications of interest occurs when \( \varepsilon \in [0, 1] \). Note that towards either boundary as \( \varepsilon \to 0 \) or \( \varepsilon \to 1 \), the equation tends to a
lower dimensional Poisson equation, or the standard Poisson equation, respectively. Equations (1.1) and (1.2) can easily be extended into higher dimensions, typically into three dimensions, by adding another partial derivative with respect to \( z \). Throughout our investigation, the scaling factor will remain on a single term. For the numerical experiences run in this paper, we mainly use the 3D anisotropic diffusion equation, given by

\[
-\left( \varepsilon \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2} \right) = f(x, y, z).
\] (1.3)

### 1.3 Finite Difference Discretization

A standard way of solving differential equations is by using finite-difference methods (FDM) [31, 34]. Finite difference methods approximate derivatives by discretizing the domain into a finite number of points on a grid. A system of these points forms a linear system, which can be solved as any matrix system of the form \( Au = f \).

To discretize the continuous function \( u \), a uniform discretization scheme may space out each point by a discrete unit \( \Delta u \). Each discrete point is given as \( u_i = u_0 + i\Delta u \), with \( i \in [0, n + 1] \) to have a total of \( n \) points plus boundary conditions. The step (or mesh) size, \( \Delta u \), is typically referred to as \( h \), with \( h = \frac{1}{n+1} \).

To formulate our discrete system of equations, and as with all things in applied mathematics, we look to the Taylor series expansion for our function. In 1D:

\[
u(x_0 + h) = u(x_0) + hu'(x_0) + \frac{h^2}{2}u''(x_0) + \ldots + \frac{h^k}{k!}u^{(k)}(\xi)
\]

where \( x_0 \leq \xi \leq x_0 + h \). In 1D, the Laplacian operator is approximated as

\[
\frac{d^2u(x)}{dx^2} \approx \frac{u(x+h) + u(x-h) - 2u(x)}{h^2}
\]

which has an error term \( O(h^2) \). In 2D, the operator becomes more complex as each
second partial derivative needs an approximation. Combining all terms together:

\[
\nabla^2 u(x, y) = u_{xx}(x, y) + u_{yy}(x, y) \\
\approx \frac{u(x + h, y) + u(x - h, y) - 4u(x, y) + u(x, y + h) + u(x, y - h)}{h^2}.
\]

This expression is commonly represented as the five-point stencil, representing the points on the Cartesian grid required to solve for the value of the derivative at \((x, y)\), given by

\[
\frac{1}{h^2} \begin{bmatrix} 1 & 1 \\ 1 & -4 & 1 \\ 1 \end{bmatrix}.
\]

The equation and stencil for the anisotropic diffusion equation are derived the same, but now with the additional scaling term

\[
\frac{u_{xx}(x, y) + u_{yy}(x, y)}{\varepsilon} \approx \frac{\varepsilon u(x + h, y) + \varepsilon u(x - h, y) - 2(\varepsilon + 1)u(x, y) + u(x, y + h) + u(x, y - h)}{h^2},
\]

and the stencil

\[
\frac{1}{h^2} \begin{bmatrix} \varepsilon & 1 \\ -2(\varepsilon + 1) & \varepsilon \end{bmatrix}.
\]

### 1.4 The Linear System

Once we have a discretizing scheme, we need a method for solving the system of equations. A visual representation of the discretization is given in Figure 1.1. Here we move from expressing the \(\pm h\) terms from Equation (1.1) to use \(i\) and \(j\).

The grid domain is given as \(\Omega_h = \{(ih, jh) : i, j = 1, \ldots, n\}\), with \(h = h_x = h_y\) to have a uniform grid along both dimensions. The function \(u\) will be computed on grid points \(x_{i,j}\) such that \(u_{i,j} \approx u(x_{i,j})\). Using this notation, the linear system of
The simplest and most common boundary conditions are Dirichlet, where the boundaries are set to zero: $u_{0,j} = u_{n,j} = u_{i,0} = u_{i,n} = 0$ for $i, j = 0, \ldots, n$. Using the boundary conditions gives sufficient equations to solve for all the unknowns.

A common method for organizing the solution vector $u$ and source vector $f$ is with a lexicographic ordering:

$$u = (u_{1,1}, u_{2,1}, \ldots, u_{n,1}, u_{1,2}, u_{2,2}, \ldots, u_{n,n})^T.$$

These vectors have length $n^2$. With $u$ and $f$ aligned as column vectors, the coeffi-
cient matrix $A$ for the anisotropic diffusion equation has the block-structure

$$
A = \begin{pmatrix}
J & -\epsilon I \\
-\epsilon I & J & -\epsilon I \\
& \ddots & \ddots & \ddots \\
& & -\epsilon I & J & -\epsilon I \\
& & & -\epsilon I & J
\end{pmatrix},
$$

where $I$ is the $n \times n$ identity matrix and $J$ is the $n \times n$ tridiagonal matrix

$$
J = \begin{pmatrix}
2(\epsilon + 1) & -1 \\
-1 & 2(\epsilon + 1) & -1 \\
& \ddots & \ddots & \ddots \\
& & -1 & 2(\epsilon + 1) & -1 \\
& & & -1 & 2(\epsilon + 1)
\end{pmatrix}.
$$

Matrix $A$ is sparse, banded, symmetric, diagonally dominant, and positive definite. The matrix has dimension $n^2 \times n^2$, as each row of the matrix corresponds to the entire set of grid points for the physical system, which we need to solve for a single value $u(x_{i,j})$. With a bandwidth of $n$, there are $n - 2$ zero diagonals between the main inner diagonal, and the outer diagonal. The vectors $u$ and $f$ are length $n^2$, as each vector has effectively been flattened from the original $n \times n$ grid into column vectors. For the 3D problem, the matrix will have a dimension $n^3 \times n^3$.

As this matrix is SPD, one potential solving technique is to use the Cholesky factorization [6], $A = LL^T$. This turns the matrix into the product of two triangular matrices which can be successively solved much faster due to a triangular solver in $\mathcal{O}(n^2)$ time [10]. For 2D problems, Cholesky is possible and a highly effective method for solving the system.

We will discuss the asymptotically optimal full Cholesky factorization in Chapter 2, along with a worked example illustrating the most important concepts. Chapter 3 will discuss iterative methods and incomplete factorization techniques, detailing the incomplete Cholesky factorization methods implemented. Numerical
experiments using the incomplete factorizations will be examined in Chapter 4.
Chapter 2

The Cholesky Factorization

This chapter will give an explanation of the optimal Cholesky factorization [8]. Some rigorous mathematical details are omitted for the sake of clarity and approachability and will be noted accordingly.

The Cholesky factorization is the breakdown of a symmetric positive definite matrix $A$ into a lower triangular matrix $L$, such that $A = LL^T$. This factorization is a modified form of Gaussian elimination and has positive elements on its diagonal. Matrix $L$ often contains more nonzeros than $A$. Fill-in is defined as any value at index $(i, j)$ which is nonzero in $L$, but zero in $A$: $L_{ij} \neq 0$ and $A_{ij} = 0$. There are multiple different ways to compute the Cholesky factorization, the three main ones being up-looking, left-looking, and right-looking. Each of these methods is characterized by which direction the Cholesky factor is built, and which parts of the $A$ and the under-construction factor of $L$ are accessed/modified. The method we will focus on is up-looking Cholesky, which aims to build $L$ from the top down in a row-wise manner. This method will access previously computed rows above, hence “looking up.” Consider a block factorization for $A = LL^T$,

$$
\begin{bmatrix}
A_{k,k} & a_{k,k+1} \\
\mathbf{a}_{k+1,k}^T & a_{k+1,k+1}
\end{bmatrix} =
\begin{bmatrix}
L_{k,k} & \mathbf{l}_{k,k+1} \\
\mathbf{l}_{k+1,k}^T & \mathbf{l}_{k+1,k+1}
\end{bmatrix}
\begin{bmatrix}
L_{k,k} & \mathbf{l}_{k,k+1} \\
\mathbf{l}_{k+1,k}^T & \mathbf{l}_{k+1,k+1}
\end{bmatrix}
$$

where $L_{k,k}$ and $A_{k,k}$ are $k \times k$ matrices, and $\mathbf{l}$, $\mathbf{a}$ are vectors. Simply using matrix multiplication to solve for the components in $A$, for the $k$th step, the equations are:
From Equations (2.1a)-(2.1c), we can solve for the unknowns. Equation (2.1a) can be solved recursively to obtain $L_{k,k}$. Equation (2.1b) is simply a matrix system of the form $Au = f$ that can be solved with a triangular solve to compute $l_{k,k+1}$. Most simply, Equation (2.1c) is just an inner product and a square root, $l_{k+1,k+1}^2 = a_{k+1,k+1} - l_{k,k+1}^T l_{k,k+1}$. If $A$ is SPD, then $a_{k+1,k+1} > l_{k,k+1}^T l_{k,k+1}$, and the Cholesky factorization can continue. If $A$ is not SPD, then a real root will not exist and there is no real Cholesky factorization for $A$.

However, this approach becomes computationally heavy very fast. The difficulty with this recursive approach is that we must solve a system of equations at each step in Equation (2.1b). Solving a triangular system has a run time complexity of $O(n^2)$, and since we solve it $n$ times, that results in a Cholesky factorization runtime of $O(n^3)$. This is clearly prohibitive at even a modest $n$. If a sparse problem (such as our anisotropic diffusion problem) is represented in a dense storage structure while computing the triangular solve at each iteration, then a large majority of the work is potentially spent on numbers which are zero, which is wasted work. If the nonzero pattern of the Cholesky factor was known, then it would be possible to only do computations related to each nonzero, skipping all unnecessary floating point operations (flops). For a large and sparse problem, ignoring zeros can offer massive speedups. Discovering the nonzero pattern of $L$ is a nontrivial task, as fill-in values cascade down the factorization (one fill value may imply more future fill values). The following subsection is dedicated to the symbolic factorization of the Cholesky factor: finding the nonzero pattern of $L$ without doing any calculations.

\[
A_{k,k} = L_{k,k} L_{k,k}^T, \quad \text{Equation (2.1a)}
\]
\[
a_{k+1,k+1} = L_{k,k} l_{k,k+1}, \quad \text{Equation (2.1b)}
\]
\[
a_{k+1,k+1} = l_{k,k+1}^T l_{k,k+1} + l_{k+1,k+1}^2, \quad \text{Equation (2.1c)}
\]
2.1 Symbolic Factorization

The symbolic factorization of a matrix $A$ defines the nonzero pattern of $A$’s Cholesky factorization, $L$, without doing any of the numerical computation (find row indices $i$ and column pointers $p$, but not the data $d$). The nonzero pattern can be determined purely from the structure of $A$. The symbolic factorization is much more efficient than the numerical computation, which is where the triangular solve and flops will take place. To compute the nonzero pattern of $L$, the elimination tree must be computed, following its postordering, and finally, that can be used to compute the column counts. Once the structure of $L$ is computed, the amount of memory allocation required is known exactly.

2.1.1 Elimination Tree

The elimination tree is a very important structure for sparse matrix factorization [2, 5, 20]. The tree represents relationships between nonzero entries in the matrix $A$ and how they are connected. The nonzero pattern of $L$ is completely encapsulated within the elimination tree of $A$. The elimination tree is not necessarily a binary tree, as one parent can have more than two children. To begin, we define the associated graph of $A$ as $G(A) = (V, E)$. Each vertex $v \in V$ corresponds to the $v$th row of $A$, and also the $v$th column since $A$ is symmetric. Similarly, an edge $(v, w) \in E$ corresponds to the nonzero $a_{vw}$, and also $a_{wv}$ due to symmetry. The graph representing the full Cholesky structure is represented as $G(L + L^T)$, which we refer to as simple $G(L)$ for brevity. Note that $G(L) = (V, E_L)$.

The full nonzero pattern of the Cholesky factor can be determined from $G(A)$. The nonzero pattern is given by the reach of the graph: $\mathcal{L}_k = \text{Reach}_{G_k}(A_k)$, where $\mathcal{L}_k$ is the nonzero pattern of the $k$th row of $L$, $G_k$ is the directed graph of the $k \times k$ submatrix of $A$, and $A_k$ is the nonzero pattern of the upper triangular portion of the $k$th column of $A$. The two theorems which define the nonzero pattern are below.

**Theorem 2.1.1 (Davis [8])** For a Cholesky factorization $LL^T = A$, nonzero entries in $A$ imply nonzero entries in $L$: $a_{ij} \neq 0 \implies l_{ij} \neq 0$.

**Theorem 2.1.2 (Parter [25])** For a Cholesky factorization $LL^T = A$, for $i < j < k$ if both $l_{ji}$ and $l_{ki}$ are nonzero, then $l_{kj}$ will be nonzero too: $l_{ji} \neq 0 \land l_{ki} \neq 0 \implies l_{kj} \neq 0$.
\[ A = \begin{pmatrix} 1 & \cdot & \cdot & \cdot & \cdot \\ \cdot & 2 & \cdot & \cdot & \cdot \\ \cdot & \cdot & 3 & \cdot & \cdot \\ \cdot & \cdot & \cdot & 4 & \cdot \\ \cdot & \cdot & \cdot & \cdot & 5 \end{pmatrix} \]

**Figure 2.1:** Example 5 × 5 symmetric matrix A.

\[ l_{kj} \neq 0. \]

A depth-first search on \( G_k \) is an application of 2.1.2. Consider the computation for a row \( k \) where \( i < j < k \), with \( a_{ik} \neq 0 \) (which also implies \( a_{ji} \neq 0 \) and \( l_{ki} \neq 0 \)), and \( l_{ji} \neq 0 \). A graph traversal of \( G_k \) will start at \( i \), therefore \( i \in \mathcal{L}_k \) and \( l_{ki} \neq 0 \). The tree traversal will visit node \( j \) because of the edge \((i, j)\) from the nonzero \( l_{ji} \). Therefore, \( j \in \mathcal{L}_k \), so \( l_{kj} \neq 0 \). Edges \((i, j)\) and \((i, k)\) imply edge \((j, k)\).

**Example 2.1.1** A simple example is helpful to illustrate how the two nonzeros above imply the third nonzero. Consider the 5 × 5 matrix A with a nonzero pattern shown in Figure 2.1. As Cholesky is an adapted version of Gaussian elimination for SPD matrices, think of the first step of zeroing out the first column. In Gaussian elimination, this would require using row 1 to remove the elements \( a_{21} \) and \( a_{41} \), as these are the only nonzero entries in column 1. Ignoring the required constants as we are only interested in the nonzero pattern, this will result in two operations: \( R_2 \leftarrow R_2 - R_1 \) (subtract row 1 from row 2 in order to zero out the column) and similarly \( R_4 \leftarrow R_4 - R_1 \). Since matrix A is symmetric, having two row-subtractions implies that there will potentially be an area of fill. As we compute \( R_4 - R_1 \), there exists an element \( a_{12} \), denoted as the red dot in the upper triangular portion of Figure 2.2. This value exists from \( a_{21} \) being nonzero. This element will influence the row element-wise subtraction which modifies row 4, shown in Figure 2.2. When the row subtraction is computed, this nonzero value will contribute to the newly computed row in the Gaussian elimination. Since this is a subtraction, a value which was originally zero will now be non-zero. Since this new location, \( a_{42} \), was originally zero in A, this is a fill-in value.

In the graph of A, \( G(A) \), since \( k \) is reachable from \( i \) by passing through \( j \), the edge \((i, k)\) becomes redundant in \( G_k \), as is it not required to compute \( \text{Reach}(i) \).
Figure 2.2: Example Gaussian elimination step, row 4 minus row 1. Both the red nonzero and blue nonzero entries in A imply the existence of the new fill-in entry at $a_{42}$ after zeroing out the first column.

If this edge is removed, Reach$(x)$ for any $x < i$ with a path $x \rightarrow i$ is not affected. Removing these edges results in the elimination tree, which contains all necessary information about Reach$(i)$ with no redundant edges.

In the elimination tree $T$ of $G$, vertex $v_j$ is the parent of vertex $v_i$ in $T$ is $j = \min \{k > i \mid (v_i,v_k) \in E_L\}$. In other words, vertex $v_j$ is the parent of vertex $v_i$ if $j$ is the first nonzero index in columns $i$. The final tree may actually be a forest if there are multiple connected components in $G(A)$. The elimination tree provides all necessary information about the nonzero pattern of the Cholesky factor [2].

Consider computing the vector $l_{k,k+1}$ via a triangular solve in $a_{k,k+1} = L_{k,k} l_{k,k+1}$, where its transpose become the $k$th row of $L$. The nonzero pattern of this vector is given by $\mathcal{L}_k = \text{Reach}_{T_k}(A_k)$ where $T_k$ is the elimination subtree up to element $k$ [8]. This means that we can traverse the elimination tree to compute the nonzero pattern of the Cholesky factor of every row by doing a reach from the nodes given by the nonzero entries in $A$. Since we traverse only the elements that will exist in the row, the run-time for computing $\mathcal{L}_k$ is proportional to the number of elements in row $k$, $O(|\mathcal{L}_k|)$. A more rigorous mathematical proof of the above can be found in [8].

**Example 2.1.2** Figure 2.3 shows the elimination tree for Figure 2.1, and the nonzero pattern of its Cholesky factor in 2.4. There are two fill entries, $(4,2)$ and $(5,4)$. $(4,2)$ is nonzero from the existence of $(2,1)$ and $(4,1)$; and $(5,4)$ exists from $(4,2)$ and $(5,2)$ being nonzero. Using the reach notation along with the elimination tree to predict the nonzero pattern of row 4, we take the reach from nodes 1 and 3 to node 4 (since those are the nonzeros in column 4 in the original $A$).
Reach\(_{T_k}(1, 3) = \{1, 2, 3, 4\}\) which includes all the original nonzero entries, but also the fill-in values at 2. The tree created from traversing Reach\(_{T_k}(A_k)\) is called the row subtree, denoted for row j as \(\mathcal{T}^j\). The full set of row subtrees is shown in Figure 2.5.

**Elimination tree of \(A, T\)**

![Elimination tree of \(A, T\)](image)

**Figure 2.3:** Example elimination tree for Figure 2.1. Edges between (1,4) and (2,5) have been pruned while constructing the elimination tree, as they are not required for computing the reach.

### 2.1.2 Postordering

A postorder is a topological ordering where vertices of a subtree are numbered in decreasing order. Once the elimination tree is found, a postordering can be obtained. The postordered tree is required for finding the number of nonzeros in every column in the Cholesky factor, something necessary to properly allocate the

\[
L = \begin{pmatrix}
1 & 2 & 3 \\
\bullet & \bullet & \bullet & \bullet & \bullet \\
& & \star & 4 & \\
& & & \star & 5
\end{pmatrix}
\]

**Figure 2.4:** Example Cholesky factor nonzero pattern for Figure 2.1. Fill in values are represented as stars at locations (4,2) and (5,4).
Figure 2.5: Row subtrees from the elimination tree $T$ for Figure 2.1. The subtrees are found from the reach on the elimination tree between nonzeros in the upper triangular $j$th column in $A$ to node $j$ in the elimination tree, and contain the nonzero entries in the $j$th column of $L$.

compress-column form structure. Outside of computing the number of nonzeros per column, the postordered elimination tree can also be used to permute $A \rightarrow C = A(p, p)$ such that the nonzero pattern of $L$ will be more structured and faster to compute while having the same number of nonzeros [8]. A recursive approach to computing the postorder is given in Algorithm 1.

Algorithm 1 Postorder Elimination Tree

1: function POSTORDER($T$)  
2:     $k = 0$  
3:     for each root node $j \in T$ do  
4:         DFSTREE($j$)  
5:     end for  
6: end function  

7: function DFSTREE($j$)  
8:     for each child $i$ of $j$ do  
9:         DFSTREE($i$)  
10:     end for  
11: post[$k$] = $j$  
12:     $k = k + 1$  
13: end function
2.1.3 Column Counts

Since the row subtrees define the nonzero pattern of each row, the full matrix structure can be obtained and the number of nonzeros in each column of $L$ can be computed. Let $A_j$ denote the nonzero pattern of the upper triangular $j$th column of $A$, $L_j$ denote the nonzero pattern in the $j$th column of $L$, and $c_j = |L_j|$ represent the number of nonzeros present in column $j$, including the diagonal. Theorem 2.1.3 shows a brute-force approach for computing the nonzero pattern by looking at the children of node $j$ in the elimination tree. Only knowledge about node $j$’s children is required to compute the fill-in for $L_j$.

**Theorem 2.1.3 (George and Liu [12])** If $A_j$ and $L_j$ denote the nonzero pattern of the jth column of $A$ and $L$, respectively, then

$$L_j = A_j \cup \{j\} \cup \left( \bigcup_{s=\text{children}(j)} L_s \setminus \{s\} \right)$$

(2.2)

**Example 2.1.3** Applying 2.1.3 to the matrix in 2.4 for column $j = 4$. From $T$ in Figure 2.3, $\text{children}(4) = \{2, 3\}$. $L_2 \setminus \{2\} = \{4, 5\}$, $L_3 \setminus \{3\} = \{4\}$. Entering this all into (2.2),

$$L_j = \emptyset \cup \{4\} \cup (\{4, 5\} \cup \{4\}) = \{4, 5\}$$

and we see that the fill-in below the diagonal has been properly computed, coming from $L_2$.

This approach of using Theorem 2.1.3, or by traversing each row subtree explicitly and keeping track of how many times $j$ appears in any row subtree both run in $O(|L|)$ time, as the traversals will iterate over every single fill value. This approach effectively iterates over the same elimination tree multiple times, an operation which can be optimized to reduce the time to nearly $O(|A|)$ [8]. The number of nonzeros in $L$ is often much greater than the number of nonzeros in $A$, so reducing the run-time to be proportional to $A$ can offer large speedups.

In order to optimize the traversal, a few concepts must be introduced: the *skeleton matrix* and *least common ancestor*. In the skeleton matrix, denoted by $\tilde{A}$, an
entry $a_{ij}$ is present if node $j$ is a leaf of the $i$th row subtree. Knowing the skeleton matrix is enough information to fully construct the nonzero Cholesky factorization pattern, as the path from the leaf node to the row node will include all other nonzero entries. The least common ancestor of two nodes $a$ and $b$ is the lowest numbered node that is an ancestor (appears above) of both $a$ and $b$ in the tree.

Consider the case where $j$ is a leaf of the elimination tree. In Figure 2.3, this corresponds to node 1 and 3. We know the column count $c_j = |A_j| + 1 = |	ilde{A}_j| + 1$. Being a leaf of the elimination tree means the node has no children, therefore its row subtrees are only a single root node (see $T^1$, $T^3$ in Figure 2.5). From Equation (2.2), having no children removes the union operation from the equation. Being a leaf of the elimination tree also guarantees to be a leaf of any row subtree (note that the converse is not true: it is possible to be an internal node of the elimination tree and a leaf of a subtree). Being a subtree leaf ensures those nodes exist in the skeleton matrix, which is why $c_j = |	ilde{A}_j| + 1$, and only the skeleton matrix is required for the count of columns who are leaves of the elimination tree.

The more complex case is where $j$ is not a leaf of the elimination tree. Equation (2.2) states that we must compute the union of the node’s children. Since $\tilde{A}_j$ is disjoint from each child $L_s$, and $s \in L_s$,

$$c_j = |	ilde{A}_j| + \bigg| \bigcup_{s=\text{child}(j)} L_s \setminus \{s\} \bigg| = |	ilde{A}_j| - e_j + \bigg| \bigcup_{s=\text{child}(j)} L_s \bigg|,$$

where $e_j$ is the number of children of $j$ in the elimination tree $T$. Effectively, this is counting the leaf nodes $|\tilde{A}_j|$ of $j$ as above, but now we also count the length of all non-overlapping entries from the children. The number of children of $j$ is subtracted to remove the set difference earlier, ignoring the diagonal elements. Notice that this is almost an easy computation, with only the notion of “non-overlapping entries” from the children being nontrivial, where we have to account for the set union. Suppose that there was an easy operation to quantify this amount, call it $o_j$. Then the equation becomes

$$c_j = |	ilde{A}_j| - e_j - o_j + \sum_{s=\text{child}(j)} c_s$$  \hspace{1cm} (2.3)
and now there are no union operations. Simply count the nonzeros of the children’s columns, and subtract any overlap.

**Example 2.1.4** As an example, consider Figure 2.1 for \( j = 4 \). Note that from the subtrees in 2.5, the only two candidate entries that could be removed from the skeleton matrix are elements 2 \( \in T^4 \) and 4 \( \in T^5 \), as these nodes are not leaves of the respective subtree. To generate the skeleton matrix, we would drop these values from \( A \), however, they only exist as fill values and are not in \( A \) originally. Therefore, our original matrix is the skeleton matrix. Node 4 has two children, 2 and 3. Equation (2.3) becomes

\[
c_4 = |\tilde{A}_4| - e_4 + o_4 + c_2 + c_3 = 0 - 2 - 1 + 3 + 2 = 2
\]

which is shown in 2.4. The overlap \( o_4 = 1 \) because row 4 appears in both children, \( L_2 \setminus \{2\} = \{4, 5\}, L_3 \setminus \{3\} = \{4\} \) as before.

Overlap needs only to be considered in the case of nodes in row subtrees with more than one child. If node \( j \notin T^i \) then row \( i \) is not in \( L_j \) and does not contribute to the overlap. If node \( j \) is a leaf of \( T^i \), then \( a_{ij} \) is in the skeleton matrix by definition and will contribute exactly 1 to the \( |\tilde{A}| \) term, but not to the overlap because it does not appear in any children of \( j \). In Figure 2.5, only in subtree \( T^4 \) has node 4 with multiple children, so in this example, it is the only location to consider overlap.

The most important thing to notice for finding this overlap of node \( j \) is that for every successive pair of row subtree leaves whose least common ancestor is \( j \), the overlap will be increased by one. In Figure 2.5, node 4 has an overlap of one because it is the least common ancestor between 1 and 3. Only the least common ancestor between leaves needs consideration even though \( L_4 \) is computed using \( L_2 \) and \( L_3 \), because \( L_2 \) by definition includes the information from \( L_1 \). It is possible to combine the overlap cases into a single correction term, \( \delta_j \). If \( j \) is a leaf of the elimination tree, \( c_j = \delta_j = |\tilde{A}_j| + 1 \). Otherwise, \( \delta_j = |\tilde{A}_j| - e_j - o_j \), turning Equation (2.3) into

\[
c_j = \delta_j + \sum_{s=\text{child}(j)} c_s. \tag{2.4}
\]
It is possible to compute $\delta_j$ by traversing the elimination tree, initializing $\delta_j = 1$ if $j$ is a leaf and 0 otherwise. While traversing the tree, the following updates are applied to $\delta_j$:

- increment by one for every entry in the skeleton matrix in column $j$ (counting the skeleton matrix entries, $|\tilde{A}_j|$),
- decrement by one for each child of $j$ (counting children, $e_j$),
- decrement by one each time $j$ becomes the least common ancestor of a successive pair of row subtree leaves (counting overlap, $o_j$).

Once $\delta_j$ is computed for all $j$, then it simply becomes a summation using Equation (2.4). Some extra information is required to track subtree node leaves and ancestors, but all of the skeleton matrix entries, number of children, and overlap can be computed efficiently while traversing the elimination tree. This removes the need to explicitly build each row subtree and significantly reduces the computational complexity. Most of the implementation details and traversal optimizations are left to [8], but Section 2.3 will walk through an example using Equation (2.4).

The symbolic factorization is now complete, with methods to count the number of nonzeros elements in each column, $c_j$, which is required for compressed-column format allocation, and the ability to know each row nonzero pattern from the elimination tree’s reach.

## 2.2 Numerical Computation

Following the symbolic factorization, the Cholesky factor can be computed in asymptotically optimal time, using the minimum amount of floating-point operations and minimizing the number of tree traversals. All that is left now is to conduct a triangular solve for each row using only the necessary information to compute nonzero entries.

### 2.2.1 Triangular Solve

Since the Cholesky factor is lower triangular, the solve at each step can be done using a simple forward-substitution algorithm. As mentioned previously, up-looking
Cholesky uses the sub-matrix $L_{k-1,k-1}$ when computing row $k$, as the solver “looks up” for the numerical computation. The algorithm for solving a dense system $Lx = b$ is given in Algorithm 2. There are two major problems with a dense triangular solve, one in each for-loop. The first, and easiest to solve, is the inner for-loop on line 4. For any $l_{ij} = 0$, the operation is simply wasted. There can be an added conditional to only perform the operation when $l_{ij} \neq 0$. The second optimization enters the first for-loop on line 2. Using the symbolic factorization, where we can loop over only columns we know are going to be nonzero, that is $x_i \neq 0$, which is given by the reach on the elimination tree: $i \in \text{Reach}_{T_i}(A_i)$. We compute the reach for node $i$ from the elimination tree to find the nonzero entries. Both of these optimizations ensure that only required work is performed, entirely skipping over any indices which would result in a zero outcome.

Algorithm 3 incorporates these features. The returned solution vector $x$ becomes row $L_k$. The full Cholesky algorithm is given in Algorithm 4, which makes $k$ calls to the optimal lower triangular solve, using the elimination tree. Although only the elimination tree is required for solving the system efficiently, lines 5 and 6 require the full nonzero pattern to properly store each row in a compressed-column format. A copy of the column count vector is kept, $c = (c_0, c_1, \cdots, c_{n-1})$, and the data and row vectors are modified at location $c_i$. Every time element $L_{ki}$ is inserted at $c_i$, the copied vector increments at that index, $c_i = c_i + 1$. This keeps a record of which index to insert new elements in the sparse matrix for each column, effectively building the matrix row-wise.

**Algorithm 2 Dense Lower Triangular Solve**

<table>
<thead>
<tr>
<th>Input</th>
<th>Matrix $L$, vector $b$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>$x = b$</td>
</tr>
<tr>
<td>2:</td>
<td>for $i = 0 \rightarrow n - 1$ do</td>
</tr>
<tr>
<td>3:</td>
<td>$\alpha = b_i$</td>
</tr>
<tr>
<td>4:</td>
<td>for $j = 0 \rightarrow i - 1$ do</td>
</tr>
<tr>
<td>5:</td>
<td>$\alpha = \alpha - l_{ij}x_j$</td>
</tr>
<tr>
<td>6:</td>
<td>end for</td>
</tr>
<tr>
<td>7:</td>
<td>$x_i = \frac{\alpha}{l_{ii}}$</td>
</tr>
<tr>
<td>8:</td>
<td>end for</td>
</tr>
</tbody>
</table>
Algorithm 3 Sparse Optimal Lower Triangular Solve

**Input** Matrix $A$, elimination tree $T$, sparse matrix $L$, vector $b$

1: $x = b$
2: for $i \in \text{Reach}_k(A_k)$ do
3: \hspace{1em} $\alpha = b_i$
4: \hspace{2em} for $j = 0 \rightarrow i - 1$ where $l_{ij} \neq 0$ do
5: \hspace{3em} $\alpha = \alpha - l_{ij}x_j$
6: \hspace{2em} end for
7: \hspace{1em} $x_i = \alpha / l_{ii}$
8: end for

Algorithm 4 Optimal Cholesky factorization

**Input** Sparse positive-definite matrix $A$

1: $T = \text{etree}(A)$
2: for $k = 0 \rightarrow n - 1$ do
3: \hspace{1em} $b = A(:,k,k)$ \quad $\triangleright$ Get $k$th upper triangular column of $A$
4: \hspace{2em} $d = b[k]$ \quad $\triangleright$ Store diagonal entry
5: \hspace{2em} $L_k = \text{triangular\_solve}(A, T, L, b)$ \quad $\triangleright$ Solve for $k$th row of $L$
6: \hspace{2em} $L_{k,k} = \sqrt{d}$ \quad $\triangleright$ Store diagonal
7: end for

2.3 Example

This section will work through decomposing an $8 \times 8$ matrix which will demonstrate key concepts from the above sections. Figure 2.6 shows the nonzero pattern of the matrix we will decompose.

To begin, we must create the elimination tree for $A$. Figure 2.7 shows the connection of the first two nodes to their first off-diagonal nonzeros ($1 \rightarrow 2$, and $2 \rightarrow 3$). Because there exist additional off-diagonal entries, there is potential for fill-in. However, since all nonzeros for columns 1 and 2 exist in a fully dense block, there will be no fill created. For example, in column 1 we may consider the case where $(2,1)$ and $(3,1)$ will imply a nonzero at $(3,2)$, however, this entry already exists. The more interesting case happens in column 3, where there are additional nonzeros entries outside of the dense block that will introduce fill.

Node 3 is connected to its first off-diagonal entry, which is 4 (shown in red). However, all other pairs of nodes must be considered, shown with blue and green
Figure 2.6: Sparsity pattern of $8 \times 8$ example symmetric matrix $A$.

Figure 2.7: Connecting nodes 1 and 2 to their first off-diagonal entries. In the subsequent elimination tree, node $i$’s parent is node $j$ when $j$ is the first off-diagonal element.

arrows in Figure 2.8. All three nonzero nodes have coordinates $(4,3)$, $(7,3)$, and $(8,3)$. The three combinations of potential fill are then given by

- $(4,3) \wedge (7,3) \Rightarrow (7,4),$
- $(4,3) \wedge (8,3) \Rightarrow (8,4),$
- $(7,3) \wedge (8,3) \Rightarrow (8,7).$

All three coordinates are zero in $A$, and therefore will exist as fill, shown in Figure 2.9 as stars. An important distinction is that once an area of fill is generated, it becomes part of the column in regards to generating the elimination tree. In Figure 2.10, column 4 connects to its first nonzero elements in row 7, which was just filled.
Columns 5, 6, and 7 can also be connected as there is no more fill-in for this matrix (entries (7,4) and (8,4) imply (8,7) which is already filled) and they each only have a single off-diagonal element, meaning they will not contribute any more fill areas. Figure 2.10 shows the final elimination tree.

Once the elimination tree is found, it is necessary to postorder it for the column counts algorithm. Postordering changes the structure of the matrix but does not change the column counts. Postordering is necessary to ensure a topological ordering, such that when performing column count operations, any child node’s computation will come before the parent node, ensuring that all calculations are done in order. The tree in Figure 2.10 is already postordered as all node descendents are decreasing for any node $k$.

With the elimination tree constructed, the row subtrees can be obtained. For
Figure 2.10: Connecting nodes 4, 5, and 6 to parent node 7. The two nonzeros in column 4 do not imply a new area of fill. Columns 5 and 6 only have one off-diagonal element each, so they will also not introduce any fill.

a row $j$, the row subtree, $\mathcal{T}_j$ is found by walking the elimination tree from the nonzero elements in row $j$ for $A$. The row subtree represents the nonzero pattern of the Cholesky factor for that given row. We demonstrate the path tracing for subtree at $k = 7$, $\mathcal{T}_7$ in Figure 2.11. Green nodes represent the starting values: the nonzero positions in row 7. Red arrows represent the path walked while moving up towards 7. All row subtrees are generated in the same manner and plotted in Figure 2.12.

After finding the row subtrees, the skeleton matrix $\tilde{A}$ can be plotted by only including row elements which are leaves of the subtree. Figure 2.13 shows the skeleton matrix for $A$. Take row $k = 4$ as an example. Row subtree $\mathcal{T}_4$ from 2.12 only has a single leaf node, 1. Therefore the nodes in columns 2 and 3 are dropped. The entire elimination tree can be generated from just the skeleton matrix, and will also generate the same row subtrees.

Finally, with the completion of the skeleton matrix, Equation (2.4) has all the necessary information. The successive steps are listed in Table (2.1). The order of operations follows: initializing $\delta_j$ depending on if $j$ is a leaf of the elimination tree, counting the nonzeros in each column of the skeleton matrix $\tilde{A}_j$, decrementing for each child of $j$, and decrementing each time $j$ becomes the least common ancestor between successive nodes in a subtree. We present this in a way structured to separate the steps, however, it is possible to compute $\delta_j$ in order of the postordered tree, only visiting each node once (with updates to previous nodes when a new
Walk up $T$ for $k = 7$

Figure 2.11: Generating row subtree for $k = 7$. Staring nodes 3, 5, and 6 (denoted in green) come from the nonzero pattern of the 7th row (or column) of $A$. The walk from 3 to 7 includes 4, which is included in the subtree and is a fill value.

Figure 2.12: All eight row subtrees for $A$. 

25
Figure 2.13: $8 \times 8$ Skeleton matrix. Any entry which is not a leaf node of a subtree is dropped from the skeleton matrix.

![Skeleton matrix diagram](image)

Table 2.1: Computation of column counts. Column 1 the column/node index. Columns 2-5 are the numbers contributing to the correction term. Column 7 shows the set of children for node $j$, taken from the elimination tree. Final column represents the final column counts.

<table>
<thead>
<tr>
<th>j</th>
<th>is leaf</th>
<th>$\tilde{A}_j$</th>
<th>$e_j$</th>
<th>$o_j$</th>
<th>$\delta_j$</th>
<th>children</th>
<th>$\delta_j + \sum c_s$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>4</td>
<td>${}$</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>${1}$</td>
<td>$-1 + 4 = 3$</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>${2}$</td>
<td>$1 + 3 = 4$</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>${3}$</td>
<td>$-1 + 4 = 3$</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>${}$</td>
<td>2</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>${}$</td>
<td>2</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>-3</td>
<td>-2</td>
<td>-5</td>
<td>${4,5,6}$</td>
<td>$-5 + 2 + 2 + 3 = 2$</td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>-1</td>
<td>${7}$</td>
<td>$-1 + 2 = 1$</td>
</tr>
</tbody>
</table>

least common ancestor is discovered). Only node 7 has contributions in the overlap column, $o_j$. As discussed earlier, the only scenario where overlap occurs is where nodes in a row subtree have more than one child. In Figure 2.12, only subtree 7 has multiple children. In this case, there would be overlap in consecutive leaf nodes (3, 5) and (5, 6), resulting in an overlap of two. The final matrix structure is present in Figure 2.9 since only column 3 contributed to the fill (denoted with stars). The column counts from Table 2.1 are consistent with the nonzero pattern of $L$ in Figure 2.9.
Chapter 3

Incomplete Cholesky for Conjugate Gradient

For large and sparse linear systems of equations, direct solvers are often inefficient even when running in asymptotically optimal time [27]. When direct solvers fail, iterative methods can offer an efficient procedure that allows for convergence to a prescribed, small residual norm or related error metric. Contrary to direct methods, which are based on matrix factorizations, iterative methods are primarily based on matrix-vector products, and as such are capable of keeping the memory requirements modest.

Using Cholesky on the anisotropic diffusion matrix causes an almost complete fill-in of the zero diagonals within the band. The Cholesky factorization costs $O(n^3)$ flops and $O(n^2)$ storage [1]. It does not take long for an increasing $n$ to break the Cholesky factorization, suggesting that iterative methods may be required.

We will discuss the conjugate gradient (CG) method [17], a popular choice for solving SPD linear systems. It is typically necessary for CG to be aided by preconditioning, which is equivalent to pre-multiplying the linear system on both sides by a matrix that makes the spectral distribution more favourable or reduces the condition number of the matrix, and then applying CG to the preconditioned system. The resulting method, called preconditioned conjugate gradient, is expected to converge faster than CG applied to the original (unpreconditioned) system. In this thesis, we consider incomplete matrix factorizations as a method of precondi-
tioning. For PCG, we will discuss incomplete Cholesky factorizations.

3.1 Conjugate Gradient

As before, we consider the linear system given by

\[ Au = f. \]

In this section, we consider iterative methods which are defined as

\[ u_{k+1} = u_k + \alpha_k p_k \]

where \( u_k \) is the current approximation at the \( k \)th step, \( p_k \) is the search direction, and \( \alpha_k \) is the step size. The iteration starts with an initial guess \( u_0 \). The error of the \( k \)th iteration is defined as \( e_k = u - u_k \), and the residual by \( r_k = f - Au_k \). The conjugate gradient algorithm is given by Algorithm 5. The method was introduced in [17]. In essence, conjugate gradient minimizes the \( A \)-norm of the error over the Krylov subspace

\[ \mathcal{H}_k(A, f) = \text{span}\{f, Af, A^2f, \ldots, A^{k-1}f\}. \]

This causes each search direction to be \( A \)-orthogonal to the previous direction, and the convergence of an \( n \) dimensional \( A \) will happen in at most \( n \) iterations. Since \( n \) is typically quite large, the more important feature of CG is that the residual norm shrinks very quickly within the first few initial iterations. Instead of searching for the exact solution which will take \( n \) steps, we can instead find a solution within a reasonable error bound much faster. Convergence also depends on the eigenvectors/eigenvalues. CG converges only as well as the convergence of the worst eigenvector [30]. Conversely, convergence is faster if there are eigenvalues of a high multiplicity. CG will converge quicker if eigenvalues exist within tight clusters, as opposed to slower convergence with irregularly distributed eigenvalues. If a system has eigenvalues all within a handful of tight clusters, then CG will only need a handful of iterations to converge.
Algorithm 5 Conjugate Gradient

Input $u_0, \delta$

Output $u_k$

1: $r_0 = f - Au_0$
2: $\delta_0 = \langle r_0, r_0 \rangle$
3: $p_0 = r_0$
4: $k = 0$
5: while $\delta_k > \delta$ do
6: $\alpha_k = \frac{\delta_k}{\langle p_k, Ap_k \rangle}$
7: $u_{k+1} = u_k + \alpha_k p_k$
8: $r_{k+1} = r_k - \alpha_k Ap_k$
9: $\delta_{k+1} = \langle r_{k+1}, r_{k+1} \rangle$
10: $p_{k+1} = r_{k+1} + \delta_{k+1} p_k$
11: $k = k + 1$
12: end while

3.2 Preconditioned Conjugate Gradient

With preconditioning, we look to transform the system into something more favourable for solving, while keeping the same solution. Let $M^{-1}$ by a symmetric positive definite approximate inverse of $A$: $M^{-1} \approx A^{-1}$. We hope that $M^{-1}A \approx I$. The matrix $M^{-1}A$ hopefully has more clustered eigenvalues than $A$, resulting in quicker convergence. We can transform $Au = f$ into

$$M^{-1}Au = M^{-1}f.$$  \hspace{1cm} (3.1)

If $A$ and $M$ are both positive definite, then (3.1) can be reformulated into

$$(M^{-1/2}AM^{-1/2})(M^{-1/2}u) = M^{-1/2}f.$$  

The matrix $M^{-1}A$ is typically unsymmetric, but it does not matter, since CG is applied to Equation 3.1 in practise. Adding the preconditioner matrix modifies Algorithm 5 into preconditioned conjugate gradient, given in Algorithm 6.
Algorithm 6 Preconditioned Conjugate Gradient

Input $u_0$, tolerance $tol$

Output $u_k$

1: $r_0 = f - Au_0$
2: $h_0 = M^{-1}r_0$
3: $\delta_0 = r_0^T h_0$
4: $f_\delta = f^T M^{-1}b$
5: $p_0 = r_0$
6: $k = 0$
7: while $\delta_k > tol^2 f_\delta$ do
8: \[ \alpha_k = \frac{\delta_k}{f_\delta} \]
9: \[ u_{k+1} = u_k + \alpha_k p_k \]
10: \[ r_{k+1} = r_k - \alpha_k A p_k \]
11: \[ h_{k+1} = M^{-1} r_{k+1} \]
12: \[ \delta_{k+1} = (r_{k+1}, h_{k+1}) \]
13: \[ p_{k+1} = h_{k+1} + \frac{\delta_{k+1}}{\delta_k}p_k \]
14: \[ k = k + 1 \]
15: end while

3.3 Incomplete Cholesky

Symmetric positive definite systems can be solved using Incomplete Cholesky factorizations [22] for PCG. Given a symmetric matrix $A$ and a symmetric sparsity pattern $S$, an incomplete Cholesky factorization is defined as

\[ A = \bar{L}L^T + R \quad l_{ij} = 0 \text{ if } (i, j) \notin S \quad r_{ij} = 0 \text{ if } (i, j) \in S \]

where $R$ is the remainder terms that are truncated from the full Cholesky, and $\bar{L}$ is the “incomplete” Cholesky factor [19]. Breaking $A$ into triangular components allows for fast system solves during PCG. Lines 2 and 11, with $M = LL^T$ now become $h = (LL^T)^{-1} r$. Since there are two triangular matrices, the system can be solved directly through two simpler triangular solves. We can solve two systems:

- $\bar{L}x = r$
- $\bar{L}^T h = x$.

Using MATLAB notation, this may look like
\[ h = \tilde{L}^T \backslash (\tilde{L} \backslash r) \]

where the backslash operator is an efficient triangular solve.

There are many different ways to define the sparsity pattern, \( S \), of the incomplete factorization. For example, IC(0) only keeps nonzero entries where \( A \) is also nonzero (zero fill-in) [27]. In general, there are two classes of techniques when finding the nonzero pattern of the preconditioner matrix: static and dynamic. IC(0) is an example of a static pattern, where the sparsity pattern is computed a priori. On the other hand, dynamic approaches determine which values to drop during computation. A simple example of this, and one which will be explored in great detail, is a drop tolerance: define a threshold, \( t \), and set any value \( |d| < t \) to zero. The main idea is that small values are less important to the incomplete factorization, and ignoring them will benefit with a smaller memory footprint and faster matrix operations while remaining an effective preconditioner, improving CG convergence.

Another common method for dynamically controlling the sparsity pattern is by allowing a fill parameter, \( p \), which specifies the maximum number of fill entries per row (or column) [15, 19]. The original implementation by [15] only took the \( p \) largest elements at each step, ignoring any information from the original pattern of the matrix. The current definition keeps all entries where \( a_{ij} \neq 0 \), and allows for an additional \( p \) elements (see Saad [27] for ILU\((p)\)). For \( p = 0 \) which allows zero extra fill, this is IC(0), a static pattern.

Saad [27, Chapter 10] combines both \( t \) and \( p \) for an incomplete LU factorization, ILU\((t, p)\) or ILUTP. These parameters attempt to reduce the number of nonzeros in the incomplete factor, making it easier to compute for large systems and also making solver iterations quicker by having faster matrix operations. We aim to implement an analogous method for incomplete Cholesky, allowing two dynamic drop parameters, IC\((t, p)\), or ICHOLTP.

The dynamic dropping parameters both aim to achieve the same goal: significantly reduce the number of nonzeros in the incomplete factor for faster factorization time and faster matrix operations during PCG, along with a smaller memory footprint. Fewer nonzeros while building \( \tilde{L} \) speeds up factorization time because there are fewer flops during matrix-vector multiplication. The same logic applies
once the factor has been built: during PCG, mainly on line 11, fewer nonzeros in $M$ will result in a faster multiplication. These operations are significant while solving very large systems. Aside from computation time, the required memory for a full Cholesky factor also grows infeasibly large quickly. Both dropping parameters aim to restrict memory consumption.

### 3.3.1 Drop Tolerance and Fill Factor

The 3D anisotropic matrix, with a dimension $n^3 \times n^3$, has a bandwidth of $n^2$, and the full Cholesky factor will essentially cause 100% fill in within the band (minus some zeros within the band in earlier columns). That means the number of nonzeros is roughly given by $|L| = \text{nnz}_\text{bandwidth} \times \text{matrix size} = n^2 \times n^3 = n^5$.

**Example 3.3.1** Consider a system with $n = 50$, a matrix dimension $n^3 \times n^3 = 125,000 \times 125,000$. The full Cholesky would require $50^5 = 312,500,000 = 313$ million entries. Holding 64-bit (8 bytes) floating points numbers, this becomes $3.13 \times 10^8 \times 8 = 25.3 \times 10^9 = 2.5GB$. It is typical for these systems to be on the order of magnitude of $n > 1,000,000$, so 2.5GB of memory for a system well under a million in size is clearly prohibitive.

The full Cholesky factorization for the anisotropic diffusion Equation (1.3) mostly has small values, shown in Figure 3.1. Figure 3.1 is a histogram of nonzero values for the full Cholesky factorization of an $n^3 = 15625$ sized system. The binning is done such that the smallest values exist in the $[-10^{-5}, 10^{-5}]$ section, which would contain an overwhelming number of zeros if included. There are typically values greater than one along the diagonal, however, their bar is so small compared to the others that it is virtually non-existent. The largest bins contain numbers small in magnitude. Consequently, a drop tolerance can be selected to eliminate a large number of entries. Beyond just dropping those small values, zeroing out elements will impact future steps of the factorization, creating a cascading effect that will result in even more zero entries and greatly reducing the memory footprint (more detail in Chapter 4). However, the drop tolerance does not guarantee a certain number of nonzeros in the incomplete factor. The required amount of memory allocation required for the incomplete factor is unknown. It may be possible to dynamically allocate memory, but that will come with a performance cost.
Figure 3.1: Histogram of nonzero values in $L$. Most values in the factorization tend to be small. Values greater than one only exists along the diagonal, and their bin is virtually indistinguishable in the histogram.

The fill factor, $p$, allows the user to define a maximum allowable number of fill values across the incomplete factor according to Equation (3.2). Without $p$, some applications allow the user to define their maximum allowed memory allocation and will fail if exceeded [13]. Specifying the fill-in will guarantee that only a certain amount of memory is allocated and that the factorization will succeed. Even in the smallest case where $p = 0$ allows zero additional fill, the incomplete factorization will run to completion, outputting the static pattern IC(0).

Choosing Drop Tolerance and Fill Factor

The drop tolerance and fill factor must be carefully chosen. Extreme values in either direction may result in an incomplete factor which is too difficult/large to compute, or too simple to be useful. For example, consider the case for IC($t$) with no $p$ parameter. Choosing $t = 0$ drops no elements, and would result in the full Cholesky factor being constructed, which will take too long. On the converse, choosing $t$ large will drop nearly every element, creating a preconditioning matrix which does a poor job approximating the inverse of $A$ and not helping with PCG.
convergence. Similar arguments can be struck for $p$, with values too large keeping too much fill in with respect to computational complexity and memory usage, and values too small creating a less complex and potentially poor approximation. That is why a reasonable middle ground for both $t$ and $p$ must be found, allowing for quick computation while being an effect preconditioner.

3.3.2 Factorization Strategies

We discuss three methods for creating a dynamically controlled incomplete Cholesky factor. The first is a modified version of the optimal algorithm, the second is given by [18], and the third is a modification of [18].

Modified Optimal

Starting from the fully optimal Cholesky factorization in Section 2 is a natural choice. Algorithm 7 modifies the optimal solve from Algorithm 3, including a drop tolerance on line 8, and a fill factor on line 10. This modified solve algorithm replaces the triangular solve in the full Cholesky factorization, Algorithm 4. The output vector with dropped entries becomes the $k$th row of the incomplete factor.

**Algorithm 7** Modified Optimal Solve

**Input** $A$, $T$, $L$, $b$, drop tolerance $t$, fill factor $p$

```latex
1: $x = b$
2: for $i \in \text{Reach}_{T_k}(A_k)$ do
3:     $\alpha = b_i$
4:     for $j = 0 \rightarrow i - 1$ where $l_{ij} \neq 0$ do
5:         $\alpha = \alpha - l_{ij}x_j$
6:     end for
7:     $v = \frac{\alpha}{l_{ii}}$
8:     $x_i = v$ if $|v| > t$ else 0
9: end for
10: Keep $p$ largest elements in $x$ ($p - n$ smallest $x_i = 0$)
```

The full Cholesky algorithm worked in an optimal manner by grabbing only relevant columns for the triangular solve but inserting the solved vector as a row. Inserting a row into a compressed-column format requires full knowledge of the matrix nonzero pattern. But as part of the incomplete factorization, entries are
dropped dynamically, and a static pattern cannot be set beforehand.

Compressed-column sparse matrices are difficult to build row-wise but very easy to build column-wise. Instead of solving for the lower triangular factor $L$, storing the Cholesky factor in its upper triangular counterpart $L^T$ also comes with downsides. Since each row in $L$ is a column of $L^T$, it may seem that reach row $\mathcal{L}_k$ can simply be inserted into $L^T$ without knowledge of the column counts ahead of time. This is true, however, it eliminates one of the largest optimizations from computing the nonzero pattern. When computing the triangular solve in Algorithm 3, the nonzero pattern of the row ensures that only necessary flops are used. Implementation-wise, this means it only accesses columns relevant to the computation, where nonzeros will exist in the $k$th row. The ability select columns from the upper sub-matrix is removed when storing the upper triangular $L^T$, as the columns in $L$ become rows in $L^T$. The triangular solve would have to iterate over every single column while searching for proper row indices, removing the optimization from the elimination tree’s reach entirely.

A compromise must be met in order to properly use the symbolic analysis. One idea is to solve the system in lower triangular $L$ form, then transpose the current factor $\mathcal{L}$ into upper triangular, store the row $\mathcal{L}_k$ as a column, and transpose it back to begin the next iteration of the triangular solve. For a general sparse matrix, $B$, the transpose has a run-time complexity of $\mathcal{O}(|B|)$ \cite{8}, linearly proportional to the number of nonzeros. Transposing twice clearly comes with two operations of this complexity, however, it is possible to achieve the same results with only a single $\mathcal{O}(|B|)$ transposition.

Instead of transposing to insert the column, it is possible to shift and insert values into the sparse data structure. First, the maximum number of entries to allocate to the sparse structure is determined by $p$. If an $n \times n$ matrix $B$ exists with at most $p$ elements per row/column, the maximum number of nonzeros is given by

$$\max(|B|) = |\text{tril}(B)| + p(n - p) + \sum_{i=1}^{p-n-1} i$$

(3.2)

where the first term accounts for the nonzeros in the original lower triangular matrix (including diagonal), the second term assumes maximum additional $p$ elements
in columns sized greater than or equal to $p$, and the final summation sums the ending $p - n - 1$ columns which cannot have $p$ elements of fill due to their size. Typically $p$ is fairly small, and the final two terms can be combined into $pn$ for simplicity. With the maximum number of nonzeros allocated, any intermediate steps during the solve will contain unused space in the structure. The method for shifting and inserting works by keeping track of the number of elements to insert, $s = |\mathcal{L}_k|$, and by starting at the back of the sparse data and index arrays, shifting the elements backwards (i.e. $d[i + s] = d[i]$) until we hit an entry to insert an element. Insert that element and decrement the shift index, $s = s - 1$. Using this technique, all new elements can be inserted into $\mathcal{L}$ with a single pass over the sparse matrix in $O(|\mathcal{L}|)$ time. This process repeats for every iteration of the factorization, resulting in a total additional cost of $O(n|\mathcal{L}|)$. A small example of these shifting operations is in Figure 3.2. The insertion points are determined by which column the element belongs in. For example, if an element requires insertion in column $j$, then there will be an insertion operation when the shifting index reaches column pointer $p[j + 1] - 1$, which will insert it in the last position of column $j$.

This additional shifting cost is significant, but another part of the optimal algorithm adds even more time to execute. Computing the reach of the elimination tree at every iteration is unchanged by $t$ and $p$. Even for values that heavily restrict the incomplete Cholesky factor, the reach still computes in full. This will result in the reach predicting a potentially incorrect pattern for $\mathcal{L}_k$ if certain entries from previous rows have been dropped. The extra time spent on the reach and useless floating point operations causes this method of modifying the optimal algorithm to break down quite easily for $n$ not much more than full Cholesky can handle. See more discussion in Chapter 4.

**Jones & Plassman [18]**

The algorithm proposed by Jones & Plassman is a modified version of left-looking Cholesky. As a brief introduction, left-looking defined in [8] computes the Cholesky factor by breaking down the system as
**Insert** $\mathcal{L}_k = \{x_4, x_5\}, s = 2$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th></th>
<th></th>
</tr>
</thead>
</table>

Shift last element $s$ amount backwards

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_3$</th>
<th></th>
</tr>
</thead>
</table>

Insert element, decrease shift $s = 1$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_4$</th>
<th>$x_3$</th>
<th></th>
</tr>
</thead>
</table>

Continue shifting

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_2$</th>
<th>$x_4$</th>
<th>$x_3$</th>
</tr>
</thead>
</table>

Insert element, decrease shift $s = 0$

<table>
<thead>
<tr>
<th>$x_1$</th>
<th>$x_5$</th>
<th>$x_2$</th>
<th>$x_4$</th>
<th>$x_3$</th>
</tr>
</thead>
</table>

**Figure 3.2:** Shift and insertion example for inserting row elements into a column form sparse matrix. Insertion points are determined by tracking the columns, which have been excluded here. Shifting stops once all elements are inserted, $s = 0$.

\[
\begin{bmatrix}
A_{k-1,k-1} & a_{k-1,k} & A^T_{k+1,k-1} \\
a^T_{k-1,k} & a_{k,k} & a^T_{k+1,k} \\
A_{k+1,k-1} & a_{k+1,k} & A_{k+1,k+1}
\end{bmatrix} = \\
\begin{bmatrix}
L^T_{k-1,k-1} & I_{k-1,k} & L_{k+1,k-1} \\
I^T_{k-1,k} & I_{k,k} & I^T_{k+1,k} \\
L_{k+1,k-1} & I_{k+1,k} & L_{k+1,k+1}
\end{bmatrix}
\]
where \( \mathbf{a}_{k,k} \) and \( \mathbf{l}_{k,k} \) are the \( k \)th row and column of \( L \) and \( A \). If the \( k-1 \) rows of \( L \) are known, then \( \mathbf{l}_{k,k} \) and \( \mathbf{l}_{k+1,k} \) can be computed from

\[
\mathbf{l}_{k,k} = \sqrt{\mathbf{a}_{k,k} - \mathbf{l}_{k-1,k}^T \mathbf{l}_{k-1,k}},
\]

\[
\mathbf{l}_{k+1,k} = (\mathbf{a}_{k+1,k} - L_{k+1,k-1} \mathbf{l}_{k-1,k}) / \mathbf{l}_{k,k}.
\]

This algorithm is dubbed “left-looking” because the computations above require information from to the left of \( k \). For example, and the most expensive operation in the algorithm, the matrix-vector product \( L_{k+1,k-1} \mathbf{l}_{k-1,k} \) is a multiplication of the strictly lower sub-matrix of \( L \) that has been computed so far, with the \((k-1)\)st row. This builds the Cholesky factor column-by-column.

Jones introduces the use of linked lists to track where nonzero elements are in the sub-matrix and sub-vectors. To illustrate why this is helpful, consider the case in Figure 3.3, which is solving for the \( k = 3 \)rd column, denoted by crosses (which may or may not be nonzero, depending on the matrix multiplication). Since only the entries on and below the \( k \)th diagonal are required as part of the computation, Jones keeps a new index array, pointing to the first element on/below the \( k \)th diagonal. This reduces the number of iterations when searching down the compressed-column structure to complete the matrix-vector product, as the \((k-1)\) row elements can be directly accessed. Another linked list is kept to track the nonzeros in \( \mathbf{l}_{k-1,k} \), the vector outlined in red. For example, if entry \( l_2 \) was zero, then the two multiplications \( L_{42} l_2 \) and \( L_{52} l_2 \) would result in zero and can be skipped. Using both new lists of information, left-looking can be completed much faster than the base case general Cholesky. However, this method is still slower than optimal Cholesky when computing complete factors.

This new left-looking technique becomes much quicker when the sub-matrix and sub-vectors are sparse. In the case of incomplete Cholesky, when lots of elements are dropped, this algorithm becomes much faster than optimal Cholesky. The optimal algorithm wastes time computing the reach when there are many elements being dropped, while Jones’ algorithm computes only the required matrix-vector products at each step with little overhead. The number of elements dropped depends on the incomplete parameters and will be discussed further in Chapter 4.
Left-looking Cholesky, $k = 3$

$$
\begin{pmatrix}
1 & 2 \\
L_{11} & L_{22} & 3 \\
L_{41} & L_{42} & \times 4 \\
L_{51} & L_{52} & \times 5 \\
\end{pmatrix}
\begin{pmatrix}
l_1 \\
l_2 \\
\end{pmatrix}
= 
\begin{pmatrix}
L_{41}l_1 + L_{42}l_2 \\
L_{51}l_1 + L_{52}l_2 \\
\end{pmatrix}
$$

Figure 3.3: Example left-looking Cholesky for a $5 \times 5$ matrix, solving for the $k = 3$ column, marked with crosses. The blue and red rectangles outline the sub-matrix and sub-vector used for the computation of the new column, denoted with \times's.

Modified Jones

We modify the Jones left-looking algorithm to also accept a fill factor parameter, $p$. To find the largest $p$ fill values, all values must be stored in an additional data structure during the solve. The values can then be sorted by their magnitude, and the top $p$ will be inserted into the incomplete factor. The simplest data structure is a dense array of size $n$ (sorting takes $O(n \log n)$), however it is possible to use a minimum-heap of size $p$ (potentially have to insert $n$ times into a size $p$ heap, a total run-time of $O(n \log p)$). Due to simplicity, we use a dense vector and quicksort for our implementation.

Implementing the drop tolerance typically entails a magnitude check before committing the value to the Cholesky factor, line 8 in Algorithm 7. The drop tolerance, $t$, may be an absolute tolerance (i.e. check if $|l_{ij}| < t$), or a relative tolerance. An example of relative tolerance is given in [23], where for the $k$th step, check if the element is smaller relative to the diagonal elements from its row and column:

$$
|l_{ij}^{(k)}| < t \sqrt{a_{ii}^{(k)} a_{kk}^{(k)}}.
$$
3.3.3 Experimental Decisions

For all incomplete factorizations, drop tolerances are absolute: $|l_{ij}| < t$. The fill factor includes $p$ additional fill values, always keeping values where $A$ is nonzero. The largest $p$ fill values are determined by storing fill-values in a dense array, performing a quicksort based off item magnitude once the row has been fully computed, then extracting the top $p$ elements.

The modified optimal algorithm uses the shifting approach to insert the $p$ elements into the compressed-column form matrix. The Jones & Plassmann algorithm uses only an absolute drop tolerance $t$, no fill factor $p$. The modified Jones & Plassmann algorithm introduces a fill factor parameter $p$, where values are stored in an array and sorted similar to the modified optimal algorithm. The modified Jones & Plassmann algorithm does not require the shift/insert technique since left-looking Cholesky builds the factor column-wise, therefore each column generated can easily be inserted into the compressed-column structure.
Numerical experiments are all run on an AMD Ryzen-5 3600 CPU with 16GB RAM. Code testing CPU times for incomplete Cholesky techniques are compiled C programs with GCC version 9.4.0. Code for all experiments is written using the CSparse library [8] for its sparse matrix data structures, along with sparse matrix operations such as transposition, multiplication, and triangular solves. The full Cholesky algorithm, including both symbolic and numeric factorization, is used directly from CSparse without modification.

All incomplete factorizations are implemented using the same patterns and style as CSparse. Three new major functions were written in C, one for each the modified optimal algorithm, a port of Jones & Plassmann from Python [13] to C, and the modified Jones & Plassmann algorithm. We created a command line application in C which allowed the input of a matrix file and user-defined drop tolerance and fill factor parameters. Statistics including run-time, CG iterations, CG residual, and the number of nonzeros in the incomplete factor are written to an output file. A Python interface was created to call the compiled C executable and read the output file, allowing for easy analysis of the generated statistics. All systems are 3D anisotropic diffusion problems from Equation (1.3), which increase problem complexity over 2D and show larger performance gains when introducing a preconditioner.
4.1 Evaluation Metrics

Deciding on a comprehensive evaluation metric is crucial for rating and comparing various techniques. The goal is to solve a differential within a reasonable amount of time given a consumer-grade computer. A common choice determining how quickly these systems solve is looking at the number of iterations to convergence for preconditioned conjugate gradient. However, for large problems that require preconditioning, this ignores the construction time of the preconditioner and the computational time per iteration. Optimizing only for the number of iterations would result in picking a preconditioner which is effectively the full Cholesky factorization - something vastly time and memory consuming. The number of PCG iterations is still significant though, as a good preconditioner should lower the number of iterations and potentially result in a lower overall run time.

Therefore, we need to consider both the PCG iterations and some information regarding the preconditioner. One idea to measure the “work” behind the preconditioner is the number of nonzero entries it contains. A potential evaluation metric can be found by multiplying the nonzeros in the preconditioner by the number of PCG iterations: $|\tilde{L}| \times |\text{PCG\_iters}|$. This metric is difficult to measure accurately in an environment which may not do things sequentially and tends to always choose a minimum $|\tilde{L}|$, even if the actual run-time is longer.

Since the end goal is to solve a system quickly, timing metrics are also possible. The first is wall-clock time, which measures the real-world time a program takes to execute. While this may be the most representative of real-world performance, wall-clock time will include non-related tasks to the program such as any other tasks the CPU schedules.

Another timing metric is the CPU time, which will directly measure the time spent by the CPU executing the program. CPU time is not always accurate, as timings of the same function can be slightly different. Execution time depends on a variety of different factors (CPU scheduling, frequency, etc.) which creates small variance between function calls. Inconsistent timings are exacerbated for small problems, where small differences are more impactful to the overall time. For larger problems, CPU timing discrepancies do not significantly impact the final measurement in our experiments. CPU time results in the most consistent metric.
for our environment. The CPU time implicitly measures the total work taken while forming the incomplete factor, and also while performing matrix operations during PCG.

### 4.2 Cholesky Factorization

Figure 4.1 shows the sparsity pattern of the anisotropic diffusion equation, and Figure 4.2 shows the Cholesky factorization sparsity pattern. Since this is a direct factorization, solving the system $LL^T x = b$ via two triangular solves still provides an exact answer via a triangular solve that is more efficient than Gaussian elimination. The factorization causes fill-in within the banded structure of the matrix, but zeros outside the band. The Cholesky factor is imaged in Figure 4.3, where there is still diagonal dominance, like in the full inversion. The fill-in values are typically smaller compared to the values in $L$ which are nonzero in $A$.

Figure 4.4 shows a histogram of the nonzero values present in each full Cholesky factor. Note that the histogram goes from negative to positive, with the smallest values being in the $[-10^{-5}, 10^{-5}]$ bin. It is evident that most values are small, with a greater proportion of small numbers in the larger systems. The smallest system has more large and positive values as the ratio of diagonal elements to off-diagonal elements is high, and the biggest values are along the diagonal. In the bigger systems where there are comparatively more off-diagonal values, the number of “big” (i.e. $x > 1$) entries is minuscule. This trend will become significant in Section 4.3.
Figure 4.2: Cholesky factor, $A = LL^T$, sparsity pattern for three different anisotropic diffusion systems.

Figure 4.3: Cholesky factor, $L$, image for anisotropic diffusion system. There is still diagonal dominance, however, all off-diagonals are negative (dark blue) while the majority of the matrix is zero.

when dropping small elements since most values are small and there is potential to save plenty of space.

4.3 ICHOLTP

The following section will focus on different incomplete Cholesky factorizations and their effect on solving the anisotropic diffusion equation. For testing on large
Figure 4.4: Cholesky factor histogram of nonzero values. As $n$ increases, the number of small values in the $[-10^{-5}, 10^{-5}]$ bin also increases.
proceedures, we pick $\varepsilon = 0.01$ as the benchmark because it creates a more challenging problem.

4.3.1 Full Cholesky, Modified Optimal

Table 4.1 provides the CPU time taken for a relatively small system, $n^3 = 27000$. The top row contains a full Cholesky factorization. Both the symbolic factorization and the CG solution time are very small for the full Cholesky, but the time taken while carrying out the series of triangular solves is two orders of magnitude larger than any other method in the table. This clearly demonstrates that obtaining a full Cholesky factorization becomes infeasible, and will be omitted in future experiments with much larger $n$. All incomplete factors contain the same number of nonzeros. This is due to the drop tolerance being large enough such that the fill factor does not have an effect. In the case $p = 50$, this means that every column has 50 or fewer values just from the drop tolerance, so the fill factor does not contribute additional dropping.

Similarly, the modified optimal algorithm from Section 3.3.2 begins to slow down. Although much better than full Cholesky, the modified optimal algorithm is an order of magnitude slower than Jones and modified Jones. The symbolic factorization remains unchanged as mentioned previously, and does not contribute significantly to the overall run time. Most of the time is spent during the triangular solve stage, where the algorithm fully iterates over the elimination tree, consuming additional time. Running the executable using a profiler, about 55% of the total execution time is spent traversing the elimination tree. Additional time would be spent on nonzero entries predicted by the elimination tree which remains zero in the incomplete factor. For this reason, the modified optimal algorithm will also be ignored in future sections with much larger systems.

4.3.2 Effect of Drop Tolerance

Figures 4.5-4.7 show the effect of only including a drop tolerance, $t$, when solving a large and hard problem. Figure 4.6 shows the overall CPU run time for computing the incomplete factor and it as the preconditioner in PCG to solve a system. This graph has a distinct “U” shape, which is expected as there are large computational
Table 4.1: Solution CPU time breakdown for $n^3 = 27000$. Column headers contain the type of factorization, drop tolerance, fill factor, number of CG iterations, symbolic factorization time, triangular solve time, CG solve time, total time, and the number of nonzeros in the preconditioner. From top to bottom, the types are complete Cholesky, unpreconditioned conjugate gradient, Jones [18], modified optimal, and modified Jones.

<table>
<thead>
<tr>
<th>type</th>
<th>t</th>
<th>p</th>
<th>iter</th>
<th>symb</th>
<th>flop</th>
<th>sol</th>
<th>total</th>
<th>nnz</th>
</tr>
</thead>
<tbody>
<tr>
<td>chol</td>
<td>1</td>
<td></td>
<td>0.002</td>
<td>8.398</td>
<td>0.084</td>
<td>8.48</td>
<td>23543129</td>
<td></td>
</tr>
<tr>
<td>cg</td>
<td></td>
<td>115</td>
<td></td>
<td></td>
<td>0.039</td>
<td>0.039</td>
<td></td>
<td></td>
</tr>
<tr>
<td>mod</td>
<td>0.01</td>
<td>50</td>
<td>13</td>
<td>0.002</td>
<td>0.095</td>
<td>0.010</td>
<td>0.107</td>
<td>211620</td>
</tr>
<tr>
<td>jones</td>
<td>0.01</td>
<td>13</td>
<td></td>
<td></td>
<td>0.006</td>
<td>0.010</td>
<td>0.01</td>
<td>211620</td>
</tr>
<tr>
<td>jonestp</td>
<td>0.01</td>
<td>50</td>
<td>13</td>
<td></td>
<td>0.008</td>
<td>0.010</td>
<td>0.018</td>
<td>211620</td>
</tr>
</tbody>
</table>

costs associated with both a small and large drop tolerance. A small drop tolerance keeps more nonzeros in the factor, taking longer to create. A large tolerance drops many nonzeros, creating a less effective preconditioner which causes CG to require more iterations for convergence.

As the drop tolerance approaches zero, the incomplete factor will drop fewer elements, therefore becoming more like the full factorization. This will result in fewer PCG iterations, but a significantly longer time to compute the factorization. As the drop tolerance becomes very large, in this case, $t \approx 1$, the incomplete factor will begin dropping many/all elements. With a high enough drop tolerance to completely drop all off-diagonal entries, the resultant factor will only contain diagonal elements, becoming the Jacobi preconditioner. The Jacobi preconditioner does not have a great impact on solving the system during PCG, as shown by the number of iterations equally the unpreconditioned solve in Figure 4.5.

The drop tolerance has a nonlinear relationship with the number of nonzeros in the incomplete factor, as shown in Figure 4.7. To better understand how/why this happens, Figure 4.8 is a histogram of all nonzeros in the incomplete factor for a small system, $n^3 = 15625$. A small system is chosen here to allow the full Cholesky factorization for comparison. It is not equivalent to computing the full Cholesky factor, then dropping elements based on the drop tolerance versus dropping elements dynamically during the factorization. This is because, during the factorization, one fill value may cause more fill-in during future iterations. Drop-
Dropping elements causes a cascading effect of dropping future nonzeros in the factor. Notice for the full Cholesky histogram, the bin $x \in [-10^{-5}, 10^{-5}]$ contains about 4 million elements, and the bin $x \in [-10^{-4}, -10^{-5}]$ contains just under 3 million elements. However, after computing an incomplete factorization with a drop tolerance of $t = 10^{-5}$, the bin $x \in [-10^{-4}, -10^{-5}]$ only contains about 2 million elements, although no elements in that range were explicitly dropped. The reduction in numbers occurs from dropping elements from $[-10^{-5}, 10^{-5}]$, which are then unable to contribute during future computation. Table 4.2 shows the total number of nonzeros for more drop tolerances.

![ICHOLT PCG iterations, $n^3=1000000, \varepsilon=0.01$](image)

**Figure 4.5**: Number of PCG iterations versus different drop tolerances for ICHOLT. A smaller drop tolerance results in less iterations.

### 4.3.3 Effect of Fill Factor

Performing similar operations as above, the effect of only dropping via the fill factor (holding $t = 0$) is shown in Figures 4.9-4.11. The fill factor has a slightly more nonlinear behaviour, particularly at moderately small values of $p$. Observed in Figures 4.9 and 4.10, there is a large spike in iterations and CPU time around $p \in [3, 7]$. This range of unstable $p$ appears in other large-sized matrices with hard $\varepsilon$ values, sometimes causing the incomplete factorization to fail on account of not
Figure 4.6: Total CPU time for the ICHOLT factorization plus the PCG solve versus different drop tolerances. Minimum total time occurs at $t = 10^{-2}$ for the tested drop tolerances.

<table>
<thead>
<tr>
<th>Drop Tolerance</th>
<th>nnz(L)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>9405649</td>
</tr>
<tr>
<td>1e-5</td>
<td>4598024</td>
</tr>
<tr>
<td>1e-4</td>
<td>2060667</td>
</tr>
<tr>
<td>1e-3</td>
<td>696486</td>
</tr>
<tr>
<td>1e-2</td>
<td>197556</td>
</tr>
<tr>
<td>1e-1</td>
<td>60625</td>
</tr>
</tbody>
</table>

Table 4.2: Drop tolerance and number of nonzeros in the incomplete factor for the matrix in Figure 4.8 with $n^3 = 15625$.

being positive definite. Only happening on very large systems, it is unclear exactly why these values of $p$ cause the factorization to fail.

In the region $p \geq 10$, the incomplete factorization stabilizes. The number of iterations drops quickly, eventually beginning to plateau. In Figure 4.10, the total CPU time begins to monotonically increase with additional $p$. A larger $p$ requires a larger sort during the factorization, causing the increase in time while the number of PCG iterations does not noticeably benefit.

Figure 4.11 shows the total number of nonzeros in the factor along with the as-
sociated memory footprint for allocating the entire structure in a compress-column format, using 64-bit values. Each additional $p$ allows an increase of at most 1 more value per column. For $n^3$ columns, an increase of $p$ increases the memory allocation by an additional $2n^3$ elements (one $n^3$ for each data entry and row index entry), or $2n^3 \times 8$ bytes (for 64-bit floats/integers). In the case of $n^3 = 1,000,000$, each additional $p$ increases memory by $2 \cdot 10^6 \times 8 = 16000000 = 16$MB. Figure 4.11 demonstrates that the memory allocation can grow quickly, and limited $p$ can restrict the memory footprint of the incomplete factor.

4.3.4 Combined Drop Tolerance and Fill Factor

We can now observe the effects of combining both $t$ and $p$ during the incomplete Cholesky factorization, $IC(t,p)$ or ICHOLTP. Combining both parameters should be able to combine the large dropping power of $t$ along with the memory control of $p$. We will generate matrices with two different sizes. The first system will contain a matrix with $n^3 = 1,728,000$, with varying $\varepsilon = \{0.01, 0.1\}$, with a simulated memory limit of 1GB. This will allow a maximum $p = 31$. The second system will
have size $n^3 = 9,938,375$ and $\varepsilon = 0.01$ with a simulated memory limit of 8GB. This memory restriction is much more realistic on modern consumer-grade systems, particularly for mobile devices. Parameters which would require too much memory, or cause a factorization failure due to indefiniteness, are defaulted to maximum values for iterations, CPU time, and the number of nonzeros.

Figures 4.12-4.16 with $\varepsilon = 0.01$ are generated from 10 values for $t$ and 10 values for $p$, resulting in 100 simulated parameter combinations. Figure 4.12 plots the PCG iterations taken with the preconditioner. As expected, the more nonzeros allowed in the preconditioner, the lower the iterations. More nonzeros occur in the preconditioner when $t \to 0$ and $p \to \infty$. On the contrary, Figure 4.13 plots the total number of nonzeros in the incomplete factor, which increases the number of iterations decreases. Figures 4.14 and 4.15 (logarithm scale to better emphasis differences) show the total CPU time taken for the factorization and PCG solve. From Figure 4.6, which is a comparably large problem, the optimal $t = 10^{-2}$ is consistent. Same with the optimal $p$ from Figure 4.10. $t = 10^0 = 1$ results in no convergence, as the diagonal preconditioner is insufficient for this problem.

The most important thing to notice from Figures 4.12-4.15 is that the feature space is not one-to-one: there are multiple different parameter choices which return the same results. In these Figures, the range $t = 10^{-2}$ with $p \in [12,31]$ appear to return nearly identical results. This phenomenon is briefly described in Section 4.3.2, where the drop tolerance will also affect the number of nonzeros in all ranges. At certain combinations of $t$ and $p$, if the drop tolerance drops enough values, then the effects from $p$ may not exist within certain ranges. For example, if a fully dense vector of length $m$ had a drop tolerance applied which dropped half the elements to have $\frac{m}{2}$ nonzeros in total, only $p \in [0, \frac{m}{2}]$ would have an effect on dropping more elements, as any $p > \frac{m}{2}$ would only include additional zeros.

It may appear that the fill factor has little effect in this regard: simply pick $t = 10^{-2}$ for optimal results. However, $p$ still plays an important factor in controlling the initial memory allocation. Jones’ algorithm [18] which only uses $t$ does not guarantee any level of memory consumption/allocation because it is impossible to know without executing the factorization. Figure 4.16 visually represents the initial memory allocation, and its independence from $t$. Selecting $p$ to restrict the memory allocation can be important when dealing with very large problems that
may use an unknowingly large amount of memory.

Figure 4.17 images the CPU time on a logarithmic scale for \( n^3 = 1,728,000 \) and \( \varepsilon = 0.1 \). In comparison to Figure 4.15 for \( \varepsilon = 0.01 \), the range of competitive \( t \) and \( p \) values is larger. The easier system does not require as much information in the preconditioner, allowing a higher drop tolerance and lower fill factor.

Figures 4.18-4.22 contains the numerical experiments for the second system, \( n^3 = 9,938,375 \) and \( \varepsilon = 0.01 \) with a memory restriction at 8GB. The \( n^3 = 9,938,375 \) experiments are very similar to their 1,728,000 and \( \varepsilon = 0.01 \) counterpart. The logarithmic CPU images both have the most narrow band at \( t = 10^{-2} \), with a fill factor range extending to approximately a quarter of the maximum allowed \( p \). The similarities between all the figures suggest that the optimal parameters do not change much between different scales of large systems.

Both \( t \) and \( p \) play an important role in the incomplete Cholesky factorization. The drop tolerance \( t \) eliminates small values which creates a preconditioner capable of significantly reducing the number of CG iterations, while fill factor \( p \) controls the amount of memory required by the preconditioner. For large systems with the hardest \( \varepsilon = 0.01 \), a drop tolerance around \( t = 10^{-2} \) gives the best results, dropping enough values to stay within a reasonable memory bound while also minimizing the amount of overall work required to solve the system. The CPU times at \( t = 10^{-2} \) for hard problems may allow for a range of \( p \) that is sufficient, the best may be considered the \( p \) which minimizes memory allocation. A fill factor roughly greater than 10 for the problems presented allows for the full factorization of difficult problems to complete successfully, creating a useful preconditioner within memory constraints. The minimum \( p \) between problems of size \( n^3 = 1,728,000 \) and \( n^3 = 9,938,375 \) did not change drastically, suggesting that even larger problems may benefit from similar parameters of the incomplete Cholesky factorization.
Figure 4.8: Number of nonzeros in the incomplete Cholesky factor versus various drop tolerances. Full Cholesky factor on top corresponds to $t = 0$. 

53
**Figure 4.9:** Number of PCG iterations.

**Figure 4.10:** Total CPU time for the ICHOLP factorization plus the PCG solve.
Figure 4.11: Maximum number of nonzeros and memory size allocated for the fill-factor incomplete Cholesky factor.

Figure 4.12: Number of PCG iterations when building preconditioner using different $t$, $p$ parameter combinations. $n^3 = 1,728,000$, $\varepsilon = 0.01$. 
Figure 4.13: Number of nonzeros in the preconditioner. \( n^3 = 1,728,000, \varepsilon = 0.01 \).

Figure 4.14: CPU time taken for constructing the preconditioner and solving the system with PCG. \( n^3 = 1,728,000, \varepsilon = 0.01 \).
Figure 4.15: $\log_{10}$ CPU time, better emphasizing the differences between parameters. $n^3 = 1,728,000$, $\varepsilon = 0.01$.

Figure 4.16: Allocated memory in GB, only dependent on $p$. $n^3 = 1,728,000$, $\varepsilon = 0.01$. 
Figure 4.17: log10 CPU time. $n^3 = 1,728,000$, $\epsilon = 0.1$.

Figure 4.18: PCG Iterations. $n^3 = 9,938,375$, $\epsilon = 0.01$
Figure 4.19: Number of nonzeros in the preconditioner. $n^3 = 9,938,375$, $\varepsilon = 0.01$

Figure 4.20: CPU time for incomplete factorization and PCG. $n^3 = 9,938,375$, $\varepsilon = 0.01$
**Figure 4.21:** Logarithmic scale for CPU time. $n^3 = 9,938,375, \varepsilon = 0.01$

**Figure 4.22:** Initial memory allocation for the incomplete factor. $n^3 = 9,938,375, \varepsilon = 0.01$
Chapter 5

Conclusions

We have described how to build a discretization scheme of the anisotropic diffusion differential equation, how to convert the scheme into a linear system of equations, how to solve the system directly using an asymptotically optimal Cholesky factorization, how to solve the system using the conjugate gradient method, and how to perform an incomplete Cholesky factorization to use as a preconditioner for preconditioned conjugate gradient. Our code implements modern techniques for full and incomplete Cholesky factorizations and can solve problems with dimensions in the tens of millions.

We have shown that combining two incomplete factorization parameters, drop tolerance and a fill factor, allows in most cases we have tested the construction of a sufficiently good preconditioner in terms of iteration reduction, computation time, and memory allocation. A common incomplete Cholesky factorization in MATLAB includes a drop tolerance parameter but does not include a user-controlled parameter for memory allocation.

Both the drop tolerance and fill factor can be used in tandem for an incomplete factorization, with each parameter contributing its own strength. The drop tolerance can have a significant effect, removing small and relatively unimportant values to decrease factorization time while still being an effective preconditioner. However, the drop tolerance does not guarantee any memory constraints. The fill factor will guarantee only a certain number of fill values will exist in the incomplete factor. The fill factor allows for specific control over memory allocation while
allowing the drop tolerance to handle the majority of the element removal. We have experimentally found a range of values for the drop tolerance and the fill factor, which accomplishes the goal of keeping memory allocation in check while yielding a reasonable iteration count. When plotting in 2D parameter space \((t \text{ versus } p)\), the optimal range of parameters will exist within a band, typically around \(t \in [10^{-1}, 10^{-2}]\) and \(p \approx 10\) for the problems tested in this report. Harder problems may have stricter ranges for the parameters. In the scenario where multiple parameter combinations generate the same preconditioner and very similar overall CPU times, the best choice would be the fill factor which minimizes memory allocation.

It is desirable to aim for the parameters which result in the lowest overall CPU time for the incomplete factorization and iterative solve while also minimizing the memory usage of the incomplete factor.

The iterations, number of nonzeros, and the CPU time change non-linearly with different \(t\) and \(p\) values. We have observed situations in which there is an abrupt reduction in the nonzero. It is also possible for two close drop tolerances to create identical incomplete factors. The fill factor increases the number of nonzeros in the factor linearly as \(p\) increases, however, the number of PCG iterations can abruptly change for modest fill factors. In some cases, the fill factor will cause the factorization to become non-positive definite and fail. The factorization breakdown occurs for small isolated sets of drop tolerances, which is generally not a concern because the optimal drop tolerance exists within a range of larger values. There are ways to avoid a breakdown by means of static pivoting; this is beyond the scope of our work.

The incomplete factorization methods could include more optimizations in terms of the code we have generated. Potential areas for optimization include using a minimum-heap data structure or introducing a parallel implementation. We studied the anisotropic equation using a scalar value, however, variable coefficients may generate even harder problems. Studying larger systems with an extended parameter space may also have a different range of optimal parameters. Making the incomplete Cholesky factorization more robust may help improve special conditions where the factorization fails. Equilibration and reordering strategies (such as approximate minimum degree) may further improve the performance and reliability of ICHOLTP.


