ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

DEPARTMENT OF
INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Fall Semester 2015

LaTeX Report Template

Master Thesis

Ritvik Ranjan
rranjan@ethz.ch

Date of thesis hand-in

Advisors: Vincent Maillou, vmaillou@iis.ee.ethz.ch
Dr. Alexandros Nikolaos Ziogas, alziogas@iis.ee.ethz.ch

Professor: Prof. Dr. Mathieu Luisier, leader@iis.ee.ethz.ch

Abstract

The abstract summarizes what this report is about. It focusses on the big picture and does not go into details. You should write concisely about the following points:

- Describe the **background** of your project: what is the motivation for your project and why is it important?
- Describe the **objectives** of your project.
- Describe the **problems** that must be addressed to achieve the objectives—why are these problems difficult?
- Describe your **approach** and **methods**.
- Summarize the most important **results**.
- State the main **conclusion** and its significance.

The abstract typically takes half a page and should not be longer than one full page. Try to write a draft of the abstract early on to have a good idea of your project, but revise the abstract as the project progresses. Write the final version of the abstract once the report is otherwise complete.

The remainder of this document contains an example on structure and content of the report. This template is meant to guide you and not to force you into a certain structure—just make sure you and your advisors agree on content and structure of the report *before* you start writing it. Appendix A gives more specific guidelines for some major project areas (e.g., hardware designs). If you are new to LaTeX or want to learn some best practices, you should also check the short LaTeX guide in Appendix B.

Acknowledgments

You rarely work in complete isolation, and this is the place to acknowledge contributions by others.

Declaration of Originality

Download the official declaration of originality[1], print it, and sign it. When printing your thesis, replace this sheet with the physically signed original paper. For the digital version, scan the filled declaration of originality (either before signing or remove your signature[2] from the image) and replace the text inside this file with `\includepdf[scale=0.9]{declaration_of_originality}`.

[1] https://www.ethz.ch/content/dam/ethz/main/education/rechtliches-abschluesse/leistungskontrollen/declaration-originality.pdf

[2] By removing your signature from the digital version of your report, you enable us to share your report with collaborators without having to deal with privacy concerns.

Contents

Chapter 1

Introduction

- Why set of linear equations arise in scientific computing - Why some of the matrices are sparse - Time and memory complexity of solving linear systems - How does permutation help either reducing fill-in, memory or improving parallelism - In this work, we focus solely on the permutation problem. We explore existing methods for sparse symmetric matrix reordering. We explore any possible optimizations and parallelization strategies. We look for single thread, multi-thread and distributed memory implementations. - We thoroughly evaluate the performance of the different reordering methods and compare them in various qualities for a lot of different matrices.

Sparse linear systems of the form $Ax = b$, where $A \in \mathbb{R}^{n \times n}$ is sparse and symmetric positive definite, form the computational backbone of modern scientific computing. These systems arise naturally across virtually every domain of computational science and engineering, including quantum chemistry, computer graphics, computational fluid dynamics, power networks, machine learning, and optimization. The prevalence of sparse matrices stems from a fundamental mathematical property that many physical phenomena exhibit: local interactions—adjacent elements in a discretized domain interact strongly, while distant elements have minimal or no direct coupling.

Mathematically, a matrix A is considered sparse when the number of nonzero entries $\text{nnz}(A)$ satisfies $\text{nnz}(A) = O(n)$ rather than $O(n^2)$ as in dense matrices. This sparsity typically arises from the discretization of partial differential equations (PDEs) using finite difference, finite element, or finite volume methods.

The sparsity arises from the discretization process itself. When continuous problems described by partial differential equations are discretized using methods such as finite differences, finite elements, or finite volumes, the resulting algebraic equations connect only neighboring grid points or elements. Conceptually, sparsity corresponds to systems with few pairwise interactions. For instance, in a three-dimensional finite element mesh, each node typically connects to only a small neighborhood of adjacent nodes, regardless of the total problem size, resulting in matrices where the number of non-zero elements is roughly equal to the number of rows or columns rather than being proportional to n^2.

[example for quantum transport; vincent's ppt; add photos]

Matrix permutations serve as a preprocessing step that can dramatically improve both computational efficiency and memory requirements for sparse linear system solution. The basic principle involves reordering the rows and columns of the matrix A to obtain a permuted system $PAP^T\hat{x} = Pb$, where P is a permutation matrix, such that the reordered matrix exhibits superior properties for factorization.

The primary motivation for permutation is fill-in reduction during matrix factorization. Fill-in occurs when zeros in the original matrix become non-zero during factorization, effectively destroying the sparse structure.

==========

Chapter 2

Background

Before we look into the algorithms and theory behind reordering techniques, we first take a dive into the factorization process that produces further non-zeroes, which in this and the entire thesis's context, are called fill-in. Fill-in depends only on the structure of the matrix, i.e., the positions where the initial non-zero entries are placed. It may be trivial but important to note, that if some numerical arithmetic results in a zero in the factored matrix, it is usually still considered as a fill-in.

Suppose the set of set of n equations that are to be solved are

$$Ax = b \tag{2.1}$$

where A is a sparse matrix, x is the vector of unknowns. The sparsity, or the number of non-zero entries in A determines the fill-in of it's Cholesky factor when it is employed to solve the aforementioned set of equations. Suppose the Cholesky factorization of A is given by LL^T, where L is a lower triangular matrix (with a positive diagonal) and L^T is the transpose of L. The efficiency of solving this set of equations is dependent on the number of non-zero entries in L. It has been shown (George and Liu) that we can factor and solve the set of equations in space proportional to $\sum\_jd\_j$ and time complexity $\sum\_jd\_j^2$, where d_j is the number of non-zeroes in the jth column of L.

2.1. Elimination Trees and Symbolic Factorization

2.1.1. Mathematical Foundations and Problem Formulation

Sparse Matrix Factorization Context

Consider a sparse symmetric positive definite matrix $A \in \mathbb{R}^{n\times n}$. The Cholesky factorization decomposes A into the form $A = LL^T$, where L is a lower triangular matrix. While A may contain relatively few nonzero entries, the factor L typically exhibits significantly more nonzeros due to a phenomenon known as fill-in. This fill-in occurs because the

elimination process creates new nonzero entries at positions that were originally zero in A.

The sparsity pattern of A is defined as the set pattern$(A) = \{(i,j) : A_ij \neq 0\}$. Similarly, pattern$(L)$ denotes the sparsity pattern of the Cholesky factor. The fundamental observation is that pattern(L) can be completely determined from pattern(A) using purely structural analysis, without requiring any numerical computation. This structural predictability forms the theoretical foundation for elimination trees and symbolic factorization.

Graph-Theoretic Representation of Sparse Matrices

Every sparse symmetric matrix A can be naturally represented as an undirected graph $G(A) = (V,E)$, where the vertex set $V = \{1,2,\ldots,n\}$ corresponds to the rows and columns of A, and the edge set E contains an edge (i,j) if and only if $A\_ij \neq 0$ for $i \neq j$. This graph representation transforms matrix operations into graph-theoretic problems, enabling the application of powerful combinatorial methods to numerical linear algebra.

The elimination process on matrix A corresponds to a sequence of vertex eliminations on graph $G(A)$. When vertex k is eliminated, all pairs of neighbors of k become connected by edges if they were not already connected. This edge addition process continues throughout the elimination sequence, producing what is known as the filled graph $G^+(A)$. The filled graph contains all edges that exist at any point during the elimination process, and its structure completely determines the sparsity pattern of the Cholesky factor L.

Formal Definition of Elimination Trees

The elimination tree provides a compact representation of the structural relationships inherent in sparse matrix factorization. This tree structure captures the essential dependencies between different stages of the elimination process.

Definition 2.1 (Elimination Tree)**.** Let A be a symmetric positive definite matrix with Cholesky factorization $A = LL^T$. The elimination tree $T(A) = (V,E\_T)$ is a directed tree defined by the parent function parent$: V \to V \cup \{\emptyset\}$, where:

$$\text{parent}(j) = \min\{i > j : L\_ij \neq 0\} \tag{2.2}$$

If no such index i exists, then parent$(j) = \emptyset$ and j is a root of the tree. The directed edges of the tree are given by $E\_T = \{(j,\text{parent}(j)) : \text{parent}(j) \neq \emptyset\}$.

This definition establishes a precise correspondence between the numerical structure of the Cholesky factor and the combinatorial structure of a directed tree. Each node j in the tree corresponds to column j of the matrix, and the parent relationship encodes which column will first modify column j during the factorization process.

Definition 2.2 (Ancestor Relationship)**.** In the elimination tree $T(A)$, node i is an ancestor of node j (denoted $i \succ j$) if there exists a directed path from j to i in the tree. Node i is a proper ancestor if $i \succ j$ and $i \neq j$.

The ancestor relationship in the elimination tree has a profound connection to the structure of the Cholesky factor, as formalized in the following fundamental theorem.

Theorem 2.1 (Fundamental Characterization Theorem)**.** *Let $T(A)$ be the elimination tree of symmetric positive definite matrix A with Cholesky factor L. Then for any indices $i > j$: L_ij $\neq 0$ if and only if $i \succ j$ in $T(A)$.*

This theorem establishes that the elimination tree completely characterizes the sparsity pattern of the Cholesky factor. The nonzero entries in column j of L correspond exactly to the ancestors of node j in the elimination tree. This remarkable result shows that purely combinatorial information (tree ancestry) determines numerical structural properties (matrix sparsity patterns).

2.1.2. Structural Properties and Theoretical Characterizations

Fill-in Characterization Through Path Analysis

Understanding when fill-in occurs during elimination requires analyzing connectivity patterns in the original matrix graph. The concept of fill paths provides a precise characterization of this phenomenon.

Definition 2.3 (Fill Path)**.** Let $G(A) = (V, E)$ be the graph of matrix A. A fill path from vertex i to vertex j (where $i < j$) is a path $P = (i = v\_0, v\_1, v\_2, \ldots, v\_k = j)$ in $G(A)$ such that all intermediate vertices satisfy $v\_l < \min(i, j)$ for $1 \leq l \leq k - 1$.

The fill path condition ensures that all intermediate vertices on the path are eliminated before either endpoint, which is precisely the condition under which the elimination process will create a direct connection between i and j.

Theorem 2.2 (Fill Path Characterization)**.** *An edge (i, j) with $i < j$ belongs to the filled graph $G^+(A)$ if and only if there exists a fill path from i to j in the original graph $G(A)$.*

This theorem provides the theoretical foundation for predicting fill-in patterns. It shows that structural properties of the original graph completely determine which new edges will be created during elimination, independent of the specific numerical values in the matrix.

Postorder Properties and Structural Invariants

The elimination tree possesses several important structural properties that make it amenable to efficient algorithmic manipulation.

Theorem 2.3 (Postorder Property)**.** *In any elimination tree $T(A)$, if parent$(j) = i$, then all nodes in the subtree rooted at j have indices less than i. Furthermore, the elimination tree is uniquely determined by the sparsity pattern of A, independent of the specific elimination ordering used (provided the ordering respects the natural ordering of indices).*

This property implies that elimination trees can be processed using postorder traversal, where children are visited before their parents. This traversal order naturally reflects the dependencies in the elimination process—a column can only be processed after all columns in its subtree have been completed.

2.1.3. Algorithmic Construction of Elimination Trees

The most straightforward approach to constructing elimination trees directly implements the definition by examining the sparsity structure of the matrix.

Algorithm 1: Direct Construction Method

Input : Symmetric positive definite matrix A with sparsity pattern
Output: Parent array representing elimination tree $T(A)$

```
1  Initialize parent[1...n] ← 0 ;                    // 0 represents null parent
2  for j = 1 to n do
3  |   parent[j] ← 0;
4  |   foreach i ∈ {k : A[j,k] ≠ 0 and k > j} do
5  |   |   if parent[j] = 0 or i < parent[j] then
6  |   |   |   parent[j] ← i;
7  |   |   end
8  |   end
9  end
10 return parent array
```

This algorithm requires $O(|A|)$ time, where $|A|$ denotes the number of nonzero entries in matrix A. The space complexity is $O(n)$ for storing the parent array. While simple to understand and implement, this direct approach does not exploit the structural properties of elimination trees for potential efficiency improvements.

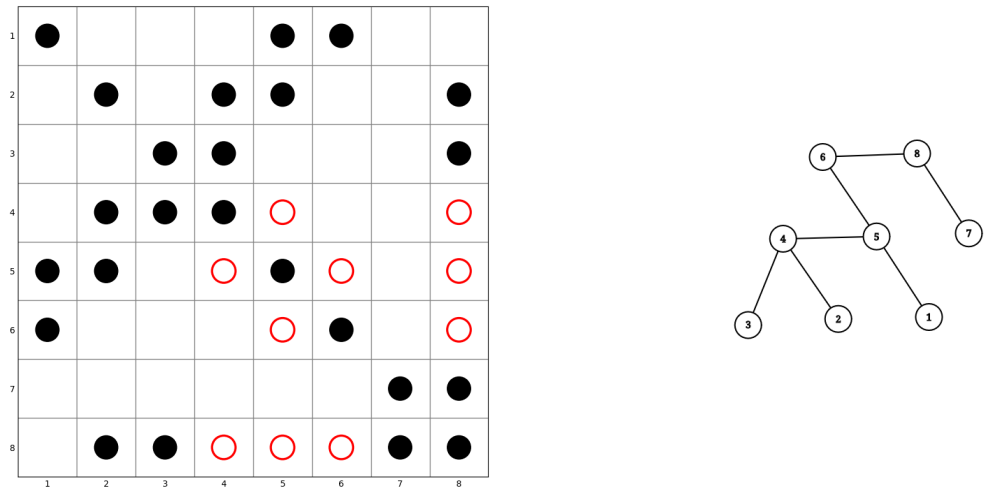
2.1.4. Symbolic Factorization

Symbolic factorization determines the sparsity pattern of the Cholesky factor L from the sparsity pattern of matrix A using the elimination tree. It does not involve any numerical computations and relies solely on the structure of the original matrix.

The key insight is that the sparsity structure of each column in L can be computed by analyzing how structural information propagates through the elimination tree. Each column inherits structure from two sources: the original matrix A and the previously computed columns corresponding to its children in the elimination tree.

Definition 2.4 (Column Structure Inheritance)**.** For column j of the Cholesky factor L, the nonzero pattern is determined by:

- Direct inheritance: All nonzero entries from column j of the original matrix A that lie on or below the diagonal



(a) Sparse matrix structure

(b) Corresponding elimination tree

Figure 2.1.: Illustration of elimination tree construction for the given sparse symmetric matrix. The sparsity is denoted by filled black circles and the fill-in induced is denoted by hollow red circles.

- Indirect inheritance: All entries from the union of nonzero patterns of columns corresponding to children of j in the elimination tree, restricted to rows numbered greater than j

This inheritance pattern reflects the computational dependencies during numerical factorization—column j can only be processed after all its children in the elimination tree have been completed.

The postorder traversal ensures that each column is processed only after all its descendants in the elimination tree have been completed, respecting the computational dependencies inherent in the factorization process.

Theorem 2.4 (Correctness of Symbolic Factorization)**.** *Algorithm 2 correctly computes the sparsity pattern of the Cholesky factor L in $O(|L|)$ time, where $|L|$ denotes the number of nonzero entries in L.*

Algorithm 2: Elimination Tree Symbolic Factorization

Input : Sparsity pattern of matrix A, elimination tree $T(A)$
Output: Sparsity pattern of Cholesky factor L

Initialize column_structure$[1 \dots n]$ as empty sets;
for $j = 1$ **to** n ; `// in postorder traversal of` $T(A)$
do
  `// Direct inheritance from original matrix`
  column_structure$[j] \leftarrow \{i \geq j : A[i,j] \neq 0\}$;
  `// Indirect inheritance from children in elimination tree`
  foreach *child c of j in* $T(A)$ **do**
    column_structure$[j] \leftarrow$ column_structure$[j] \cup \{i \in$ column_structure$[c] : i > j\}$;
  end
  `// Record structure for column` j `of` L
  pattern$(L[:,j]) \leftarrow$ column_structure$[j]$;
end
return *pattern*(L)

2.2. Minimum fill-in and NP-completeness

When performing Cholesky factorization on a sparse symmetric matrix, we want to minimize fill-in - the new nonzero entries that appear during elimination. This translates to a graph theory problem: given a graph representing the matrix's sparsity pattern, find an elimination order that creates the fewest new edges. Yannakakis [1] proved this optimization problem is NP-complete, meaning no polynomial-time algorithm can solve it optimally in general (unless P = NP).

The key insight connecting linear algebra to graph theory is that chordal graphs is essential in proving as such. A graph is chordal if every cycle of length ≥ 4 has a chord (an edge connecting non-consecutive vertices). The fundamental theorem states that a graph has perfect elimination (zero fill-in) if and only if it's chordal, and any elimination process on any graph produces a chordal result. Therefore, the minimum fill-in problem becomes: What's the smallest number of edges we must add to make our graph chordal?

The proof uses a three-step reduction to prove NP-completeness. First, we consider a bipartite graphs and their special case, chain graphs. A bipartite graph has vertices split into two independent sets P and Q, and it's a chain graph when neighborhoods form a nested sequence: $\Gamma(v\_1) \subseteq \Gamma(v\_2) \subseteq \ldots \subseteq \Gamma(v\_k)$. The key property is that a bipartite graph is a chain graph if and only if it contains no pair of independent edges.

Second, for any bipartite graph $G = (P, Q, E)$, we construct $C(G)$ by making P into a complete clique (all vertices in P connected), making Q into a complete clique (all vertices in Q connected), and keeping original edges between P and Q. The crucial lemma states that $C(G)$ is chordal if and only if G is a chain graph. This works because

if G has independent edges, they create a 4-cycle in $C(G)$ with no chord. Conversely, if G is a chain graph, $C(G)$ has a perfect elimination order.

Third, the reduction uses the Optimal Linear Arrangement Problem, which asks: given a graph, arrange vertices on a line to minimize the sum of distances between adjacent vertices. This problem is known to be NP-complete. Yannakakis constructs an transformation where, given graph G with n vertices and m edges, he creates bipartite graph G' where part P has one vertex for each vertex in G, and part Q has an elaborate gadget structure with $2m + n^2 - d(v)$ vertices, with connections that encode the linear arrangement constraints. The mathematical relationship is:

$$\text{Minimum fill-in of } G' = \text{Optimal arrangement cost of } G + \frac{n^2(n-1)}{2} - 2m \tag{2.3}$$

This result has practical implications that no perfect algorithm exists, as any algorithm guaranteeing optimal fill-in will require exponential time in the worst case. This explains why practical sparse matrix software uses heuristics like minimum degree ordering, nested dissection and such.

2.3. Heuristics classification

There are several heuristics that have been proposed to reduce the fill-in developed over the years. These heuristics can be broadly classified in the following categories: Bandwidth minimization, minimum degree (and it's variants), nested dissection, banded structure methods using hypergraphs, and recently, machine learning approaches.

2.3.1. Bandwidth Minimization

Bandwidth minimization refers to the problem of permuting the rows and columns of a matrix such that the non-zero entries are as close to the diagonal as possible. Gaussian elimination can be performed in $O(nb^2)$ time on matrices of dimension n and bandwidth b, which is faster than the forward $O(n^3)$ algorithm when b is smaller than n (Lim A. et al., 2006a). Additionally, this problem is NP-complete, but several heuristics have been developed to approximate the optimal solution.

The Cuthill-McKee algorithm employs breadth-first search to reduce matrix bandwidth by generating level structures. However, it has computational limitations and may not achieve optimal bandwidth reduction. The Reverse Cuthill-McKee algorithm addresses these issues by reversing the ordering, while the GPS algorithm by Gibbs et al. provides an alternative level structure approach.

We take a look at the reverse Cuthill-McKee algorithm, which is still widely used in practice.

[write the pseudocode here]

2.3.2. Minimum Degree

The minimum degree algorithm determines the order in which to eliminate variables (or pivot) during Gaussian elimination to minimize fill-in (creation of new non-zero entries) in sparse matrices. At each step, it chooses the node with the minimum degree (fewest connections) for elimination.

The minimum degree algorithm operates on the adjacency graph of the sparse matrix. It begins by initializing the graph, where each vertex represents a variable. At each iteration, the algorithm selects the vertex with the smallest degree (i.e., the fewest neighbors), which corresponds to the variable whose elimination is expected to introduce the least fill-in. Upon elimination, the chosen vertex is removed from the graph, and all its neighbors are connected to each other, forming a clique to preserve the matrix structure. The degrees of the affected vertices are then updated to reflect the new connections. This process is repeated until all vertices have been eliminated. The resulting elimination order directly determines the pivot sequence used during matrix factorization.

(placeholder for general algorithm)

Variants of the minimum degree algorithm include:

1. Multiple Minimum Degree (MMD)

   When multiple nodes have the same minimum degree, uses additional tie-breaking rules. Often selects based on secondary criteria like external degree or node numbering.

2. Approximate Minimum Degree (AMD)

   Uses approximations to avoid expensive exact degree calculations. Employs concepts like "supervariables" and "element absorption". Much faster than exact minimum degree while maintaining similar fill-in quality.

3. Constrained Minimum Degree

   Incorporates additional constraints (like maintaining bandwidth). Balances minimum degree objective with other structural requirements.

2.3.3. Nested Dissection

Nested dissection is a divide and conquer heuristic for the solution of sparse symmetric systems of linear equations based on graph partitioning. Nested dissection can be viewed as a recursive divide-and-conquer algorithm on an undirected graph; it uses separators in the graph, which are small sets of vertices whose removal divides the graph approximately in half [2].

Nested dissection consists of the following steps: Form an undirected graph in which the vertices represent rows and columns of the system of linear equations, and an edge represents a nonzero entry in the sparse matrix representing the system. Recursively

(a) Original graph before separator identification

(b) Graph after separator removal showing partitioned subgraphs

Figure 2.2.: Separator identification process. The separator vertices divide the graph into approximately equal subgraphs

partition the graph into subgraphs using separators, small subsets of vertices the removal of which allows the graph to be partitioned into subgraphs with at most a constant fraction of the number of vertices. Perform Cholesky decomposition (a variant of Gaussian elimination for symmetric matrices), ordering the elimination of the variables by the recursive structure of the partition: each of the two subgraphs formed by removing the separator is eliminated first, and then the separator vertices are eliminated [3].

All the sequential algorithms for determining the elimination ordering of a graph G can be described by the following general algorithm: Generate the tree of separators for G and perform a tree traversal on the separator tree to order the vertices, where this traversal must visit a node before any of its parents.

2.3.4. Hypergraph partitioning and other methods

In recent years, hypergraph partitioning methods [4] have been proposed to reduce fill-in during sparse matrix factorization. Hypergraphs generalize graphs by allowing edges (hyperedges) to connect more than two vertices, making them well-suited for modeling complex relationships in sparse matrices. In this paper we explore another Hypergraph based reordering method,

Other methods include machine learning approaches that learn optimal ordering strategies using reinforcement learning. One such approach is Alpha Elimination [5] which uses Convolutional Neural Networks (CNNs) and reinforcement learning to predict the next node to eliminate based on the current state of the graph. We have explored a more flexible architecture but fails with scalability, which we discuss in Chapter 5.

Chapter 3

Implementation and Optimizations

Discuss the state-of-the art and other related work. Depending on how much related work exists and how central the comparison to it is in your thesis (discuss this with your advisors), the introduction may contain sufficient related work and this chapter can be omitted.

3.1. SuiteSparse implementation of RCM and Minimum Degree

The Reverse Cuthill-McKee (RCM) algorithm in SuiteSparse provides bandwidth reduction for symmetric sparse matrices through a breadth-first search strategy combined with degree-based ordering heuristics. The implementation processes disconnected components separately and employs pseudo-peripheral vertex selection to minimize profile and bandwidth.

The SuiteSparse RCM implementation incorporates several refinements over the basic algorithm. The pseudo-peripheral vertex selection uses multiple BFS traversals to identify vertices that are approximately diametrically opposite, which typically results in better bandwidth reduction than arbitrary starting points. The degree-based sorting of neighbors during BFS traversal helps create a more systematic ordering that tends to group low-degree vertices together, further improving the resulting bandwidth.

The algorithm's effectiveness stems from its ability to produce orderings where vertices with similar connectivity patterns are placed close together in the permutation. This locality property translates directly into reduced bandwidth and improved cache performance during matrix operations, making RCM particularly valuable for iterative solvers and direct factorization methods that benefit from band structure preservation.

3.1.1. AMD Algorithm Overview

The Approximate Minimum Degree (AMD) algorithm implemented in SuiteSparse follows a refined elimination-based approach that balances computational efficiency with fill-in minimization. The algorithm operates on a quotient graph representation and

Algorithm 3: SuiteSparse RCM Algorithm

input : Symmetric matrix A ($n \times n$)
output: Permutation P for bandwidth reduction

Initialize:;
Compute degree$[i] = |\text{adj}(i)|$ for all vertices i;
Find connected components $R\_1, R\_2, \ldots, R\_k$ using DFS;
Initialize visited$[i]$ = false for all i;
Set perm_index = 0;
Process each component:;
foreach *connected component R_j* **do**
    Find pseudo-peripheral start vertex:;
    start = `FindPseudoPeripheral`(R_j);
    `// Select vertex with minimum degree among maximum-distance vertices`
    Cuthill-McKee BFS traversal:;
    Initialize queue $Q = \{\text{start}\}$;
    Set visited[start] = true;
    Initialize CM_order = [start];
    while $Q \neq \emptyset$ **do**
        $u = \text{dequeue}(Q)$;
        neighbors = sort(unvisited_adj(u),by_degree_ascending);
        foreach $v \in$ *neighbors* **do**
            if $\neg$*visited*$[v]$ **then**
                Set visited$[v]$ = true;
                enqueue(Q, v);
                Append v to CM_order;
            end
        end
    end
    Apply reverse ordering (RCM):;
    for $i = 0$ **to** $|CM\_order| - 1$ **do**
        $P[\text{perm\_index} + i] = \text{CM\_order}[|\text{CM\_order}| - 1 - i]$;
    end
    perm_index $\leftarrow$ perm_index + |CM_order|;
end
return P;

incorporates several key optimizations including aggressive absorption, approximate degree updates, and dense row detection.

The key innovation in SuiteSparse's AMD implementation lies in its aggressive absorption strategy and approximate degree computation. The aggressive absorption phase identifies elements that can be completely absorbed into the current pivot, reducing the size of the quotient graph and improving cache locality. The approximate degree updates provide a computationally efficient method to maintain ordering decisions without exact degree computation, which becomes prohibitively expensive as elimination progresses.

The dense row detection mechanism addresses a common pathology in minimum degree algorithms where elimination of high-degree vertices can lead to excessive fill-in. When the algorithm detects that a newly formed element exceeds a threshold based on the matrix size, it defers elimination of the associated variables, effectively implementing a hybrid strategy that combines minimum degree with nested dissection principles.

3.1.2. COLAMD Algorithm Overview

The Column Approximate Minimum Degree (COLAMD) algorithm in SuiteSparse extends the minimum degree concept to rectangular matrices by operating on a bipartite graph representation. Unlike AMD which works on the symmetric structure $A^T A$, COLAMD directly processes the rectangular matrix A to produce a column ordering that minimizes fill-in during factorization.

The COLAMD algorithm addresses the unique challenges of rectangular matrix ordering by maintaining both row and column degree information throughout the elimination process. The bipartite graph formulation allows the algorithm to track how column eliminations affect the sparsity structure without explicitly forming the potentially much denser $A^T A$ matrix. The dense row detection mechanism prevents pathological behavior when processing matrices with highly dense rows, which could otherwise lead to excessive fill-in during factorization.

The degree update strategy in COLAMD carefully accounts for the fill-in patterns specific to rectangular matrices, where eliminating a column affects all other columns that share nonzero entries in the same rows. The algorithm's ability to handle supernodes (groups of columns with identical sparsity patterns) further enhances its effectiveness for structured matrices commonly arising in finite element applications.

Algorithm 4: SuiteSparse AMD Algorithm

input : Symmetric matrix A ($n \times n$), Control parameters
output: Permutation P, Info statistics

Initialize:;
Convert A to $A + A^T$ pattern if unsymmetric;
Build quotient graph $G = (V, E)$ from A;
Initialize degree lists Head$[d]$ for $d = 0$ to n;
Set degree$[v] = |\text{adj}(v)|$ for all vertices v;
Place each vertex in appropriate degree list;
Main elimination loop:;
for $k = 1$ **to** n **do**
    Select pivot:;
    Find minimum degree d with non-empty Head$[d]$;
    Select pivot p from Head$[d]$;
    Remove p from degree list;
    Set $P[k] = p$ (add to elimination ordering);
    Element absorption:;
    foreach *element e adjacent to p* **do**
        if $|L\_e \cap L\_p| = |L\_e|$ **then**
            Absorb e into p (aggressive absorption);
        end
    end
    Form new element e_p:;
    $L\_e\_p = \text{adj}(p) \setminus \{\text{absorbed elements}\}$;
    Mark e_p as new element;
    Update degrees (approximate):;
    foreach *uneliminated vertex $v \in L\_e\_p$* **do**
        external_degree$[v] = |\text{adj}(v) \cap \text{uneliminated}|$;
        bound $= |L\_e\_p| - |L\_e\_p \cap \text{adj}(v)|$;
        degree$[v] \approx$ external_degree$[v]$ + bound;
        Move v to new degree list;
    end
    Dense row detection:;
    if $|L\_e\_p| > \max(\alpha\sqrt{n}, 16)$ **then**
        Mark e_p as dense element;
        Move dense variables to end of ordering;
    end
end
Post-processing:;
Apply elimination tree post-ordering;
Compute final permutation statistics;
return *P and Info*;

Algorithm 5: SuiteSparse COLAMD Algorithm

input : Matrix A ($m \times n$) in CSC format, Control parameters
output: Column permutation P, Info statistics

Initialize:;
Treat A as bipartite graph $G = (R \cup C, E)$;
Set col_degree$[j]$ = nonzeros in column j, place in degree list;
Detect dense rows: if row_degree$[i] >$ threshold, mark dense;
Main elimination loop:;
for $k = 1$ **to** n **do**
    Select minimum degree column c (tie-break by density);
    Set $P[k] = c$;
    Update degrees:;
    foreach *row i connected to c* **do**
        foreach *uneliminated column j in row i* **do**
            fill_count $= |R(c) \cap R(j)|$;
            new_degree = col_degree$[j] + |R(c)| -$ fill_count $- 1$;
            Apply dense penalty if row i is dense;
            Move j to Head[new_degree];
        end
    end
    Mark eliminated rows and handle supernode formation;
end
Place remaining dense columns at end of ordering;
return *P and Info*;

3.2. Nested Dissection using METIS and SCOTCH

Nested dissection represents a divide-and-conquer approach to matrix ordering that recursively partitions the graph using small vertex separators, ordering the separated components before the separator vertices. METIS and SCOTCH implement sophisticated multilevel nested dissection algorithms that combine graph coarsening, separator finding, and refinement techniques to produce high-quality orderings for large sparse matrices.

The multilevel nested dissection approach in METIS provides superior ordering quality compared to single-level methods by operating at multiple scales. The coarsening phase creates a hierarchy of increasingly smaller graphs while preserving essential structural properties through heavy edge matching. This matching strategy prioritizes edges with large weights, which in the context of matrix ordering typically correspond to strong structural connections that should be preserved during coarsening.

The separator computation on the coarsest level benefits from reduced problem size while maintaining global structural awareness. The refinement phase during projection ensures that separators remain high-quality as they are mapped back to finer graph levels. The recursive application of this process creates a natural hierarchy where large components are isolated first, followed by progressively smaller substructures, resulting in elimination orderings with excellent fill-in characteristics for sparse direct solvers.

Algorithm 6: Multilevel Graph Coarsening

input : Graph $G = (V, E)$
output: Hierarchy of progressively coarser graphs

```
Hierarchy = [G] // Start with original graph
CurrentGraph = G;
while |V(CurrentGraph)| > COARSENING_THRESHOLD do
    Find heavy edge matching to preserve graph structure;
    Matching = HeavyEdgeMatching(CurrentGraph);
    Contract matched edges to create coarser graph;
    CoarserGraph = ContractEdges(CurrentGraph, Matching);
    Add to hierarchy;
    Hierarchy.append(CoarserGraph);
    CurrentGraph = CoarserGraph;
end
return Hierarchy;
```

Algorithm 7: Heavy Edge Matching Algorithm

input : Graph G with edge weights
output: Maximal matching favoring heavy edges

```
1  Matching = {};
2  Matched = {} // Track matched vertices
3  Sort edges by weight (descending) for better matching quality;
4  SortedEdges = SortByWeight(E(G), descending = true);
5  foreach edge (u,v) in SortedEdges do
6      if u ∉ Matched and v ∉ Matched then
7          Matching.add((u,v));
8          Matched.add(u);
9          Matched.add(v);
10     end
11 end
12 return Matching;
```

3.3. Parallel-Nested Dissection

3.4. Parallelizing minimum degree

There haven't been many attempts to parallelize the minimum degree algorithm due to its inherently sequential nature. The only known approximate parallel implementation of the minimum degree algorithm is the ParAMD algorithm proposed by Chang et al. in [6].

The sequential AMD algorithm has inherent bottlenecks that make parallelization difficult. Each elimination step requires selecting a pivot with minimum approximate degree, after which the degrees of neighboring variables must be updated. These steps are inherently sequential since you cannot select the next pivot until all updates from the previous elimination are complete.

Instead of eliminating one pivot at a time, the algorithm selects multiple pivots simultaneously using "distance-2 independent sets" - pivots that are at least 2 steps apart in the graph. This ensures no overlap in pivot neighborhoods, eliminating contention between parallel threads.

The algorithm allows selection of pivots whose degrees are within a multiplicative factor (`mult`) of the minimum degree. This relaxation increases the pool of available pivots for parallel processing while balancing against ordering quality - too much relaxation degrades the solution.

The ParAMD algorithm employs concurrent connection updates by pre-allocating 1.5x the original graph storage to avoid dynamic memory allocation. Each thread claims space atomically after collecting all updates, eliminating garbage collection synchronization bottlenecks.

For degree management, each thread maintains its own degree lists instead of sharing a global structure. The algorithm uses an affinity array to track which thread has the most current information for each variable, with lazy cleanup of stale entries during traversal.

3.5. New Coarsening approaches in Nested Dissection

3.6. Hypergraph Based Ordering

3.7. GPU Implementation of RCM

GPU Implementation of RCM is basically a parallel implementation of the breadth-first search (BFS) algorithm. Much research is available on parallel BFS, and the implementation in this thesis is based on the NVIDIA work on GPU-accelerated BFS in [7].

This GPU breadth-first search implementation uses a level-synchronous algorithm that processes the graph one depth level at a time, maintaining two key data structures: a vertex frontier (vertices to be explored in the current iteration) and an edge frontier (all neighbors of vertices in the current frontier). The algorithm begins with a single source vertex and alternates between two fundamental operations across BFS levels.

Algorithm 8: Parallel AMD Algorithm

input : Matrix A ($n \times n$), parameters mult, lim
output: Elimination ordering

Preprocessing:;
$G \leftarrow$ `initialize_quotient_graph`$(A + A^T)$ `// Compute symmetric pattern`
Initialize degree lists for each thread:;
forall *tid* $= 0$ **to** *num_threads* $- 1$ ***in parallel*** **do** `initialize_degree_list`(tid) ;
Main elimination loop:;
while $|V| > 0$ **do**
    Find minimum approximate degree across all threads:;
    amd $\leftarrow$ `find_global_minimum_degree`();
    Select distance-2 independent set of pivots:;
    $D \leftarrow$ `distance_2_independent_set`(amd, mult, lim);
    if $|D| = 0$ **then**
        break `// No more valid pivots`
    end
    Eliminate pivots in parallel:;
    forall *pivot* $p \in D$ ***in parallel*** **do** tid $\leftarrow$ `get_thread_id`();
    `eliminate_pivot`(tid, p);
    ;
    Barrier synchronization;
    barrier();
end
return *elimination ordering*;

Each BFS iteration follows a two-phase process. In the expansion phase, the algorithm takes the current vertex frontier and performs parallel neighbor gathering to create the edge frontier. Multiple threads cooperatively read the adjacency lists of frontier vertices from the compressed sparse row (CSR) representation, collecting all outgoing edges. In the contraction phase, the algorithm filters this edge frontier by checking each neighbor's visitation status, removing already-visited vertices and duplicates to produce the vertex frontier for the next iteration. This process continues until the vertex frontier becomes empty, indicating the traversal is complete.

The expansion phase uses a multi-granularity approach to handle the irregular degree distributions common in real graphs. For vertices with small adjacency lists, the algorithm employs scan-based gathering where threads use prefix sum to compute scatter offsets, creating a perfectly packed array of neighbors that allows all threads to participate in memory reads without SIMD lane waste. For medium-sized adjacency lists, warp-based gathering assigns entire 32-thread warps to cooperatively process single vertices, with threads strip-mining through the adjacency list in parallel. For very large adjacency lists, CTA-based gathering enlists entire thread blocks (hundreds of threads) to process individual high-degree vertices. This hybrid strategy automatically adapts to the workload characteristics and ensures efficient GPU utilization regardless of degree distribution.

I took an already existing implementation of the parallel BFS algorithm from the NVIDIA CUDA SDK [8] and adapted it for reordering the graph using RCM.

Chapter 4

Results

In this chapter, we first describe our evaluation setup, which consists of the HPC cluster used for our experiments, the matrix datasets, and the algorithmic implementations evaluated. We then present a detailed analysis of various performance metrics, including execution time, memory usage, parallelization and matrix fill-in reduction.

4.1. Evaluation setup

Everything that is evaluated is ran on Fritz, a high-performance computing cluster at NHR@FAU.

4.1.1. Hardware Infrastructure

Fritz is a parallel CPU cluster operated by NHR@FAU, featuring Intel Ice Lake and Sapphire Rapids processors with an InfiniBand (IB) network and a Lustre-based parallel filesystem accessible under `$FASTTMP`. The cluster configuration consists of:

Nodes	**CPUs and Cores**	**Memory**	**Slurm Partition**
992	2 × Intel Xeon Platinum 8360Y (Ice Lake) 2 × 36 cores @2.4 GHz	256 GB	singlenode, multinode
48	2 × Intel Xeon Platinum 8470 (Sapphire Rapids) 2 × 52 cores @2.0 GHz	1 TB	spr1tb
16	2 × Intel Xeon Platinum 8470 (Sapphire Rapids) 2 × 52 cores @2.0 GHz	2 TB	spr2tb

Table 4.1.: Fritz cluster node configuration

The login nodes fritz[1-4] are equipped with 2 × Intel Xeon Platinum 8360Y (Ice Lake) processors and 512 GB main memory. Additionally, the remote visualization node fviz1 features 2 × Intel Xeon Platinum 8360Y processors with 1 TB main memory, one Nvidia A16 GPU, and 30 TB of local NVMe SSD storage.

4.2. Matrices dataset

We selected a diverse set of sparse matrices from the SuiteSparse Matrix Collection and our custom dataset of matrices encountered in Quantum Transport to evaluate our implementations. The SuiteSparse matrices chosen are usually which were commonly used in the literature for *METIS* and *SCOTCH* own evaluations. [TODO about Quantum transport matrices]

Table 4.2.: SuiteSparse matrix dataset used for evaluation

Graph name	No. of vertices	No. of edges	Description
144	144649	1074393	3D Finite element mesh
598A	110971	741934	3D Finite element mesh
ADD32	4960	9462	32-bit adder
AUTO	448695	3314611	3D Finite element mesh
BBMAT	38744	99341	2D Stiffness matrix
BCSSTK30	28294	1007284	3D Stiffness matrix
BCSSTK31	35588	572914	3D Stiffness matrix
BCSSTK32	44609	985046	3D Stiffness matrix
BRACK2	62631	366559	3D Finite element mesh
CANT	54195	1960797	3D Stiffness matrix
COPTER2	55476	352238	3D Finite element mesh
FINAN512	74752	261120	Linear programming
LHR10	10672	209093	Chemical engineering
LHR71	70304	1449248	Chemical engineering
M14B	214765	3358036	3D Finite element mesh
MEMPLUS	17758	54196	Memory circuit
PWT	36519	144793	3D Finite element mesh
SHYY161	76480	152002	CFD/Navier-Stokes
TORSO1	201142	1479989	3D Finite element mesh
TROLL	213453	5858829	3D Stiffness matrix
VENKAT01	62424	827684	2D Coefficient matrix
WAVE	156317	1059331	3D Finite element mesh

Table 4.3.: Quantum Transport matrix dataset used for evaluation

Graph name	No. of vertices	No. of edges	Description
airpoll_conditional_st3	8499	1218651	[TODO: Description]
airpoll_prior_st3	8499	1205931	[TODO: Description]
cnt_cp2k	9152	6154350	[TODO: Description]
cnt_w90	768	71680	[TODO: Description]
kmc_potential_0	403605	10007089	[TODO: Description]
kmc_potential_1	1632355	41208963	[TODO: Description]
qubit_fem_b4	116380	2517228	[TODO: Description]
sinw_w90	7488	5310160	[TODO: Description]

4.3. Methodology

For our evaluation, we systematically assessed the performance of eleven different reordering algorithms across all matrices listed in Tables 4.2 and 4.3. The evaluated algorithms encompass both sequential and parallel approaches, ranging from classical degree-based methods to graph partitioning and hypergraph-based techniques:

Classical Methods:

- **AMD** – Approximate Minimum Degree
- **COLAMD** – Column Approximate Minimum Degree
- **RCM** – Reverse Cuthill-McKee
- **Natural** – Original matrix ordering (baseline)

Graph Partitioning Methods:

- **METIS** – Sequential graph partitioning-based reordering
- **METIS+IC** – METIS with improved coarsening integration
- **SCOTCH** – Sequential graph partitioning
- **PT-SCOTCH** – Parallel graph partitioning
- **ParMETIS** – Parallel graph partitioning

Hypergraph-based Methods:

- **HG-2, HG-4, HG-8, HG-16** – Hypergraph-based reordering with block partition sizes of 2, 4, 8, and 16 vertices respectively

The hypergraph-based methods (HG-2, HG-4, HG-8, HG-16) employ different block partition sizes to explore the trade-off between computational complexity and reordering/parallelization quality. All these algorithms are described in detail in Chapter 2.

4.3.1. Performance Metrics and Measurement Protocol

Our comprehensive evaluation measures five key performance indicators to assess the effectiveness of each reordering algorithm:

Fill-in Reduction: We measure the number of non-zero entries introduced during symbolic Cholesky factorization using the `cmpfillin` binary provided by the METIS library. This tool performs symbolic factorization (as described in Chapter 2) to count the additional non-zero entries without executing the actual numerical computation.

Computational Complexity: We track the total floating-point operation count required for factorization, providing insight into the computational efficiency gains achieved by different reordering strategies.

Reordering Time: The wall-clock time required to compute the reordering itself is measured using Python's `time.perf_counter()` with microsecond precision. This overhead cost is crucial for understanding the practical applicability of each method.

Memory Usage: We implement a dedicated memory monitoring system using a separate thread that samples memory consumption every 50 milliseconds during reordering operations. The monitoring captures both peak memory usage (maximum RSS observed) and average memory consumption throughout the reordering process. Memory measurements are obtained via the `psutil` library, tracking the resident set size (RSS) of the process and computing the memory increase relative to a baseline measured before algorithm execution.

Parallelization Potential: We assess the elimination tree depth as an indicator of the inherent parallelism available in the factorization. As mentioned in Chapter 2, the elimination tree is constructed by identifying parent-child relationships in the symbolic factorization structure, where the tree depth represents the critical path length and thus the theoretical lower bound on parallel factorization time.

For memory profiling, we initially attempted to use the Massif profiler from Valgrind, but encountered issues with our Python-based implementations and the cluster environment. Consequently, we developed a custom memory measurement protocol that employs a thread-safe monitoring system to capture accurate memory usage patterns during reordering operations.

Our memory measurement implementation works by first establishing a baseline memory consumption before starting any reordering algorithm, recording the initial resident set size (RSS) using `psutil.Process().memory_info().rss`. During algorithm execution, a dedicated monitoring thread runs concurrently, sampling memory usage every 50 milliseconds throughout the entire operation. This thread continuously updates the maximum observed memory consumption and maintains a comprehensive list of all samples for subsequent average computation.

Upon completion of the reordering algorithm, we compute both the peak memory increase (calculated as the maximum observed RSS minus the baseline) and the average memory increase (computed as the mean of all samples minus the baseline), with results reported in kilobytes. For cases where the monitoring thread fails to capture meaningful data, such as very fast operations that complete before sufficient samples can be collected, we provide conservative estimates based on matrix characteristics, using approximately 10% of the matrix storage size as an upper bound for memory consumption.

All experiments are executed on the Fritz cluster to ensure consistent hardware conditions across all measurements. Each algorithm is tested on every matrix in our dataset, and results are averaged over five independent runs to mitigate variability.

4.4. Evaluation

For each matrix in our dataset with different reordering algorithms applied, we have collected all the aforementioned metrics, and these can be found in our appendix/supplementary material. Here, we present aggregated results and visualizations for our underlying metrics.

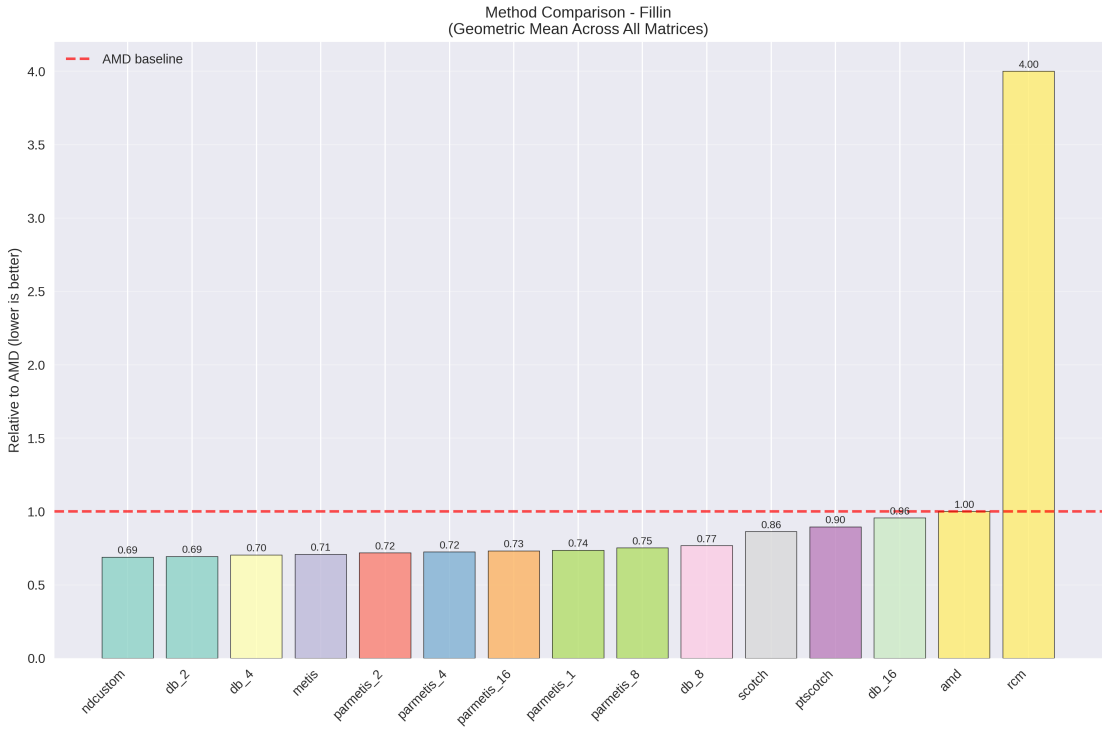


Figure 4.1.: Fill-in comparison across different reordering algorithms with AMD as baseline. The geometric mean of fill-in ratios shows the relative performance of each method, where values below 1.0 indicate better performance than AMD.

The sparsity pattern visualization in Figure 4.4 provides intuitive insight into how different reordering algorithms restructure the matrix 144. The natural ordering shows a scattered, unstructured pattern that leads to extensive fill-in during factorization. In contrast, the optimized reorderings demonstrate clear improvements in matrix structure, with methods like METIS and the hypergraph-based approaches creating more clustered patterns that minimize fill-in. Similarly, the results for matrix copter2 in Figure 4.5 reveal comparable trends.

We compare the fill-in results with different reordering algorithms by taking geometric means across all matrices, as shown in Figure 4.1.

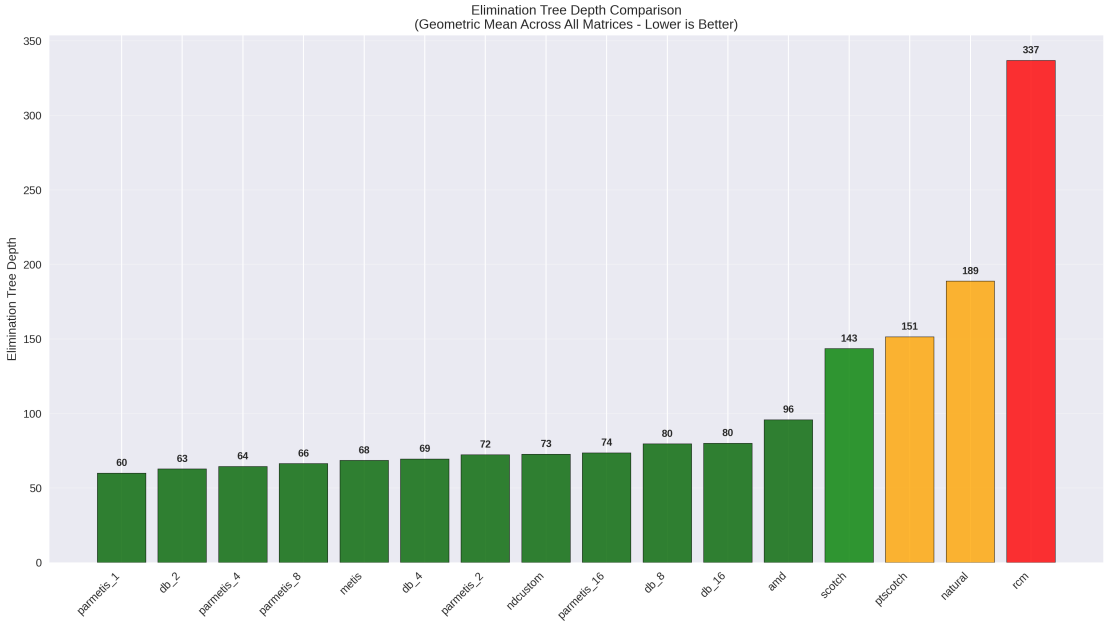


Figure 4.2.: Elimination tree depth comparison across different reordering algorithms. The geometric mean shows the relative parallelization potential, where lower depths indicate better parallel scalability for factorization.

The elimination tree depth analysis in Figure 4.2 reveals the parallelization potential of each reordering method. Lower elimination tree depths indicate greater opportunities for parallel factorization, as they represent shorter critical paths in the dependency graph.

Reordering time is another practical consideration regarding computational overhead, particularly how execution time scales with matrix size. Figure 4.3 illustrates the relationship between matrix size (number of vertices) and reordering time for different algorithms across our entire dataset.

The performance analysis reveals distinct computational overhead characteristics across different algorithm categories. AMD and RCM are the fastest methods, benefiting from their simple computational kernels. Among nested dissection approaches, METIS is the fastest, while ParMETIS and PT-SCOTCH are slower, likely due to parallelization

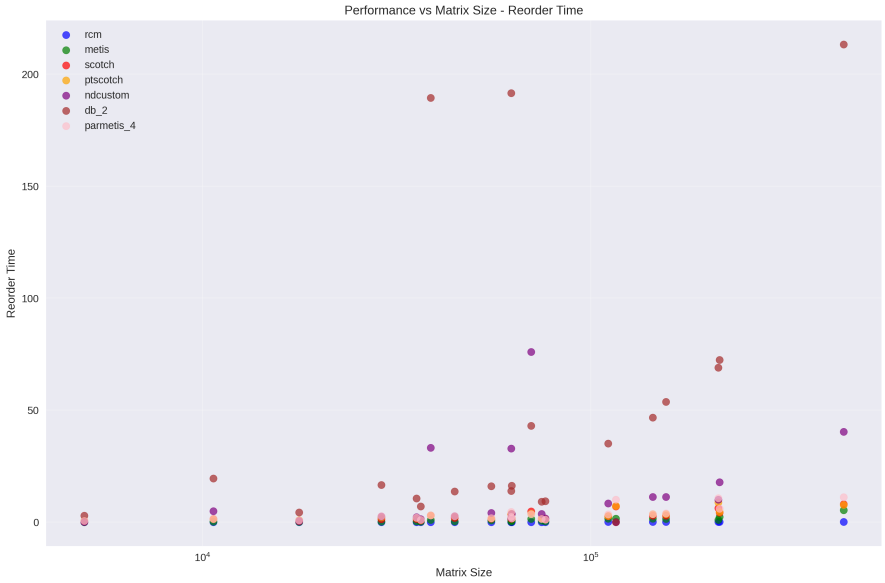


Figure 4.3.: Reordering time as a function of matrix size for different algorithms

overhead that outweighs benefits for our matrix sizes. The hypergraph methods (HG-2, HG-4, HG-8, HG-16) were considerably the slowest by far, due to the inherent complexity of hypergraph partitioning algorithms involving multiple coarsening, partitioning, and refinement phases.

Table 4.4.: Fill-in results for different reordering algorithms (number of non-zeros after symbolic factorization)

Matrix	AMD	COLAMD	HG-16	HG-2	HG-4	HG-8	METIS	Natural	METIS+IC	RCM	SCOTCH
144	93410000	93410000	68090000	46850000	47320000	51880000	47060000	1.845E+19	46230000	925100000	51300000
598a	45810000	45810000	40670000	25590000	25870000	29360000	25480000	-	25460000	-	28370000
add32	9417	9417	9830	10030	10120	10120	10050	2658000	9995	14610	20300
auto	569500000	569500000	-	220800000	226000000	-	220600000	-	217900000	-	243900000
bbmat	16860000	16860000	18340000	15780000	16100000	16370000	15910000	-	15750000	33750000	17800000
bcsstk30	3823000	3823000	4806000	4354000	4507000	4345000	4277000	15690000	4300000	21110000	4541000
bcsstk31	5533000	5533000	4698000	4248000	4209000	4313000	4186000	23140000	4164000	22400000	4755000
bcsstk32	4944000	4944000	5855000	5383000	5382000	5451000	5311000	60770000	5297000	44020000	6201000
brack2	7386000	7386000	8666000	5732000	5906000	6693000	5870000	759500000	5800000	45130000	7636000
cant	28880000	28880000	17850000	18210000	18200000	18090000	18200000	16780000	18220000	17150000	20440000
copter2	13880000	13880000	15040000	8914000	9284000	10970000	8906000	702700000	8880000	68550000	10830000
finan512	2763000	2763000	1562000	1650000	1577000	1626000	1647000	6190000	1623000	7511000	2197000
lhr10	8078000	8078000	7500000	3130000	3858000	5469000	3469000	-	3323000	14980000	3937000
lhr71	93700000	93700000	210300000	36470000	36060000	35450000	36040000	210300000	34550000	150100000	41460000
m14b	110300000	110300000	95120000	62320000	63220000	70130000	62610000	-	62130000	-	68790000
memplus	52430	52430	57320	55730	54660	55180	54780	138400000	55830	142700	127300
pwt	1519000	1519000	1320000	1307000	1313000	1319000	1306000	3396000	1309000	5362000	1601000
shyy161	1746000	1746000	1653000	1692000	1639000	1660000	1657000	8141000	1672000	11110000	2042000
torso1	12450000	12450000	-	-	-	-	13570000	-	-	113500000	16560000
troll	91680000	91680000	78570000	60850000	61560000	63540000	58910000	913600000	58210000	941600000	67080000
venkat01	5739000	5739000	5797000	5378000	5378000	5472000	5405000	64940000	5349000	45380000	5954000
wave	121700000	121700000	85080000	61300000	61410000	66910000	59710000	1008000000	62180000	835800000	66530000

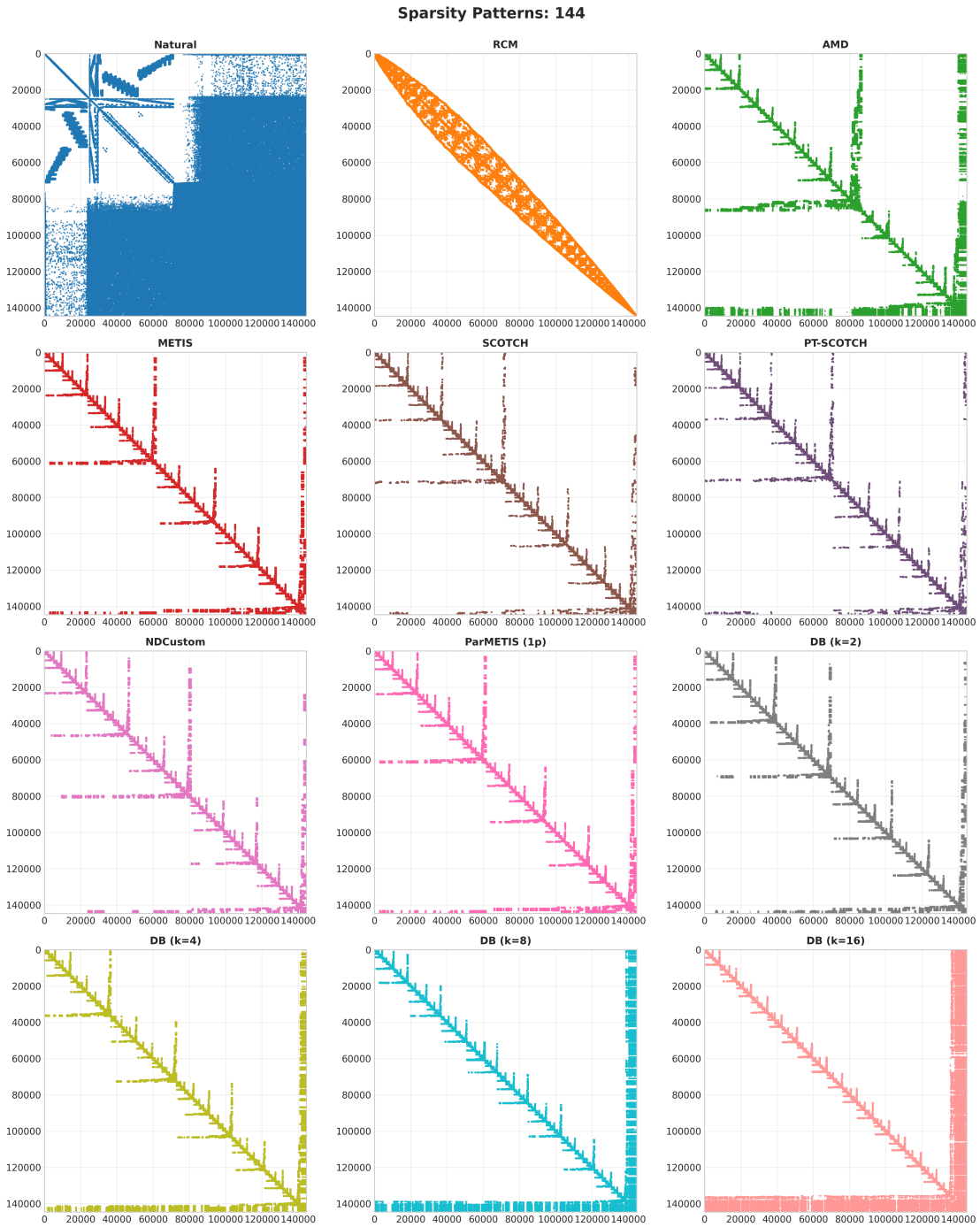


Figure 4.4.: Sparsity patterns of matrix 144 with different reordering algorithms applied. The natural ordering (top-left) shows a scattered pattern leading to high fill-in, while optimized reorderings (other panels) demonstrate more defined structures.

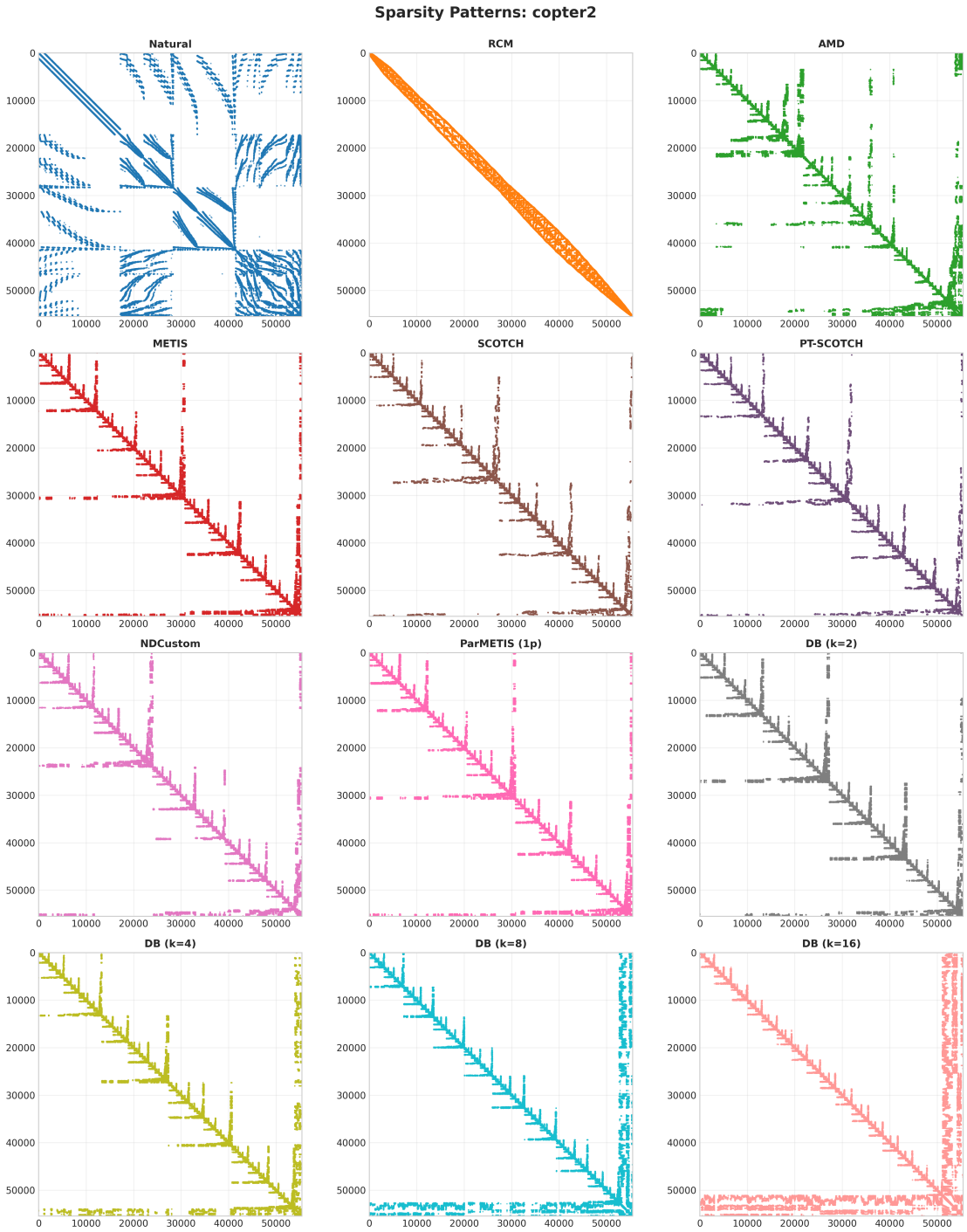


Figure 4.5.: Similarly, sparsity patterns of matrix copter2 with different reordering algorithms applied.

Chapter 5

Other Approaches

In this chapter, we explore alternative and promising approaches to the sparse matrix reordering problem beyond traditional heuristic methods. These methods were found by myself to be either computationally difficult to scale for large matrices or were found to have some bottlenecks, but nevertheless represent interesting directions for future work.

5.1. Graph Reinforcement Learning for Reordering

As we know that sparse symmetric matrices can be represented as undirected graphs, it seems promising that this representation allows GNNs to naturally capture the local neighborhood relationships that classical ordering heuristics rely on, such as node degree and clustering patterns.

Traditional heuristics like minimum degree ordering make greedy decisions based on limited local information. GNNs, however, can propagate information across multiple hops in the graph, enabling each node to consider not just its immediate neighbors but also the broader structural context. This multi-hop reasoning capability allows the network to anticipate how eliminating one node will affect distant parts of the matrix, potentially leading to more effective ordering decisions.

The message-passing architecture of GNNs naturally models the fill-in process during matrix factorization. When a node is eliminated, it creates new connections between its neighbors, which GNNs can represent through their aggregation and update mechanisms. The network can learn to predict these fill-in patterns and make elimination choices that minimize overall structural complexity.

The model shows.

5.2. GPU Accelerated Nested Dissection

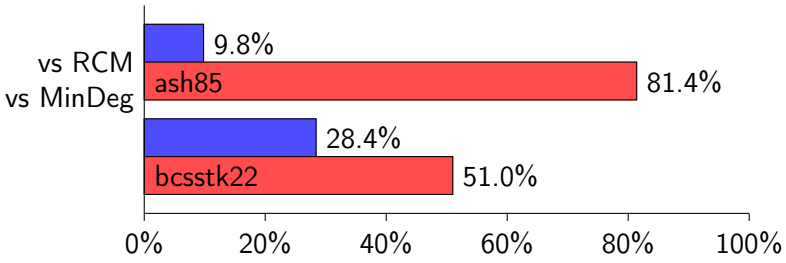


Figure 5.1.: GNN-based method with RCM and MinDegree

Chapter 6

Conclusion and Future Work

Draw your conclusions from the results and summarize your contributions. Point out aspects that need to be investigated further.

The conclusion can be structured inversely to the introduction: Summarize how the *evaluation* backs the *solution*, which solves the *problem*. Describe how your contributions improve the *situation* and what other (potentially newly discovered) problems have to be solved in the future.

Be concise: the conclusion normally fits on a single page and is rarely longer than two pages.

Appendix A

Topic-Specific Guidelines

A.1. IC design (ASIC or FPGA) projects

For IC design projects, the part of the report where you present your main contributions (the *Implementation* chapter in this template) usually consists of two chapters: *Hardware Architecture* and *Design Implementation*.

A.1.1. Hardware architecture

In the Hardware Architecture chapter, you should describe the architecture and decisions that led to it. Block diagrams and descriptions of control flow, data flow, and interfaces go there. The described architecture can be more general than what you actually implemented, e.g., through parameters.

A.1.2. Design implementation

The Design Implementation chapter is about the architecture variant you actually implemented. It can be meaningful to merge this chapter with the Results chapter to relate central figures of merit directly to implementation choices and tradeoffs; discuss this with your advisors.

Functional verification

Describe how you verified the design implementation functionally. For example, describe how your testbench interfaced the golden model and your circuit. Figure A.1 illustrates a sample setup.

Reference a ReadMe file that describes how the testbench can be launched.

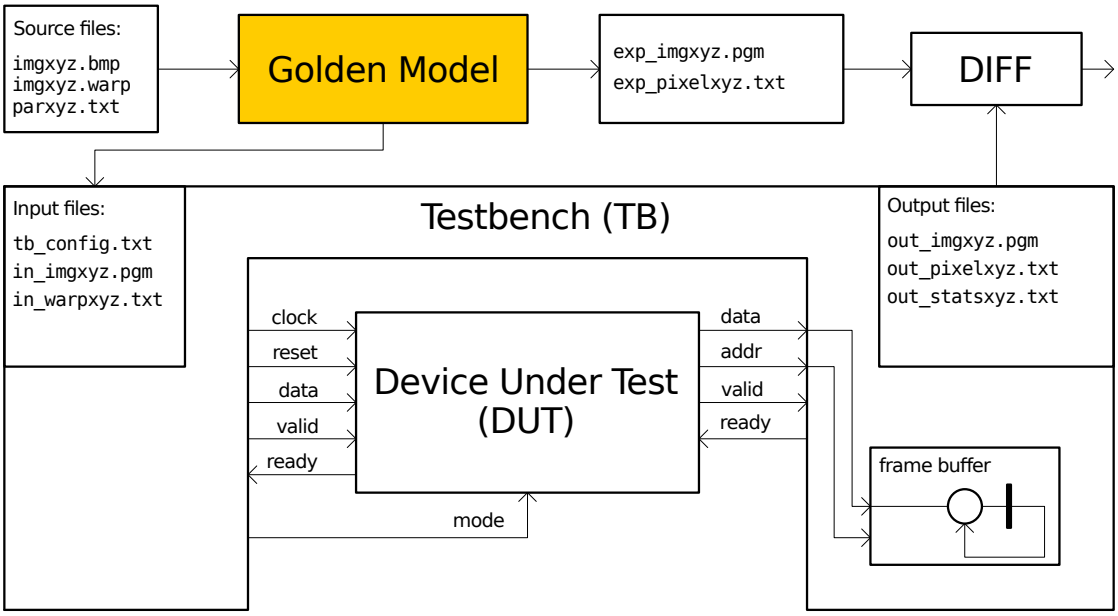


Figure A.1.: Testbench used for functional verification.

Back-end implementation

In ASIC projects, remember to discuss both front- and back-end implementation. That is, if you took special measures for floorplanning, DFT, or clock or power distribution, you will want to mention it here. In this case, make sure to add results that show the impact of your measures.

A.1.3. Results

Typical application-specific figures of merit of your hardware design are SNR, throughput, and memory/interface bandwidth. Moreover, you should also specify technology-specific figures such as area (for ASICs) or resource (for FPGAs) requirements, timing constraints, and power and energy consumption.

A.1.4. Data sheet

If your ASIC is getting fabricated, you need to write a data sheet for it. You should put this data sheet into the appendix of your report. You are free to write the data sheet in a standalone document and include a PDF file here or to write it in the source files of your report.

Sections of a typical IC data sheet are:

- Features
- Applications

- Packaging
- Bonding diagram like the one in Fig. A.2.
- Pinout diagram like the one in Fig. A.3.
- Interface description
- Register map
- Operation modes:
  - Functional modes
  - Test modes
- Electrical specifications
  - Recommended operating regions
  - Absolute maximum ratings

For more information, up-to-date bonding diagrams, and other technology-specific data ask the DZ.

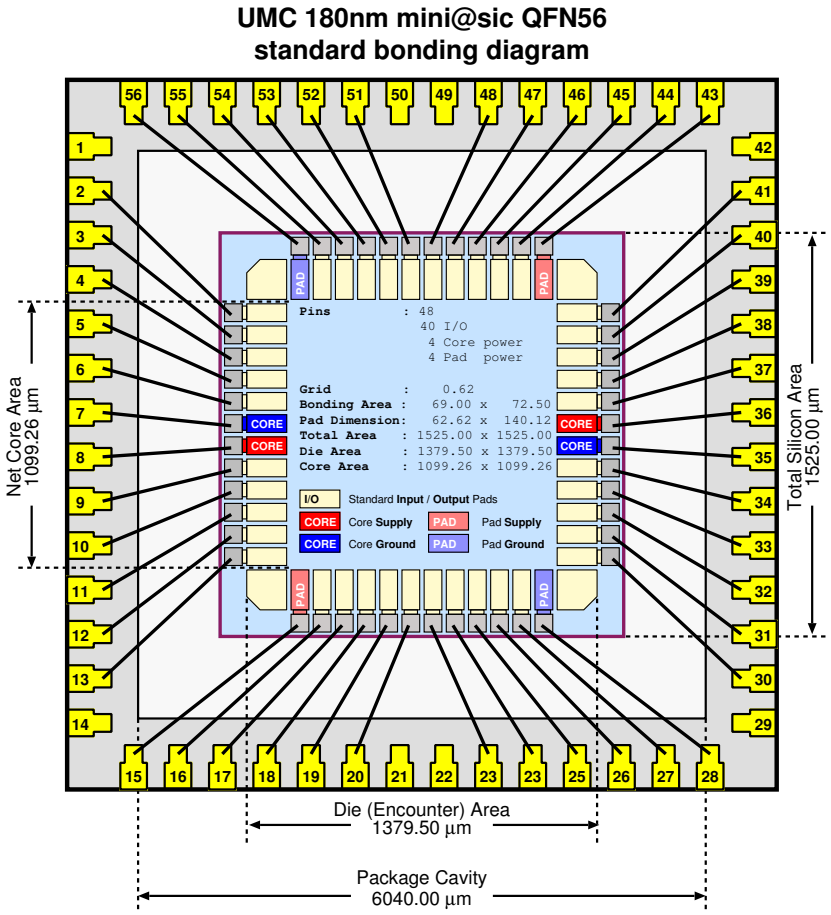


Figure A.2.: Standard bonding diagram for QFN56 UMC 180 nm mini@sics.

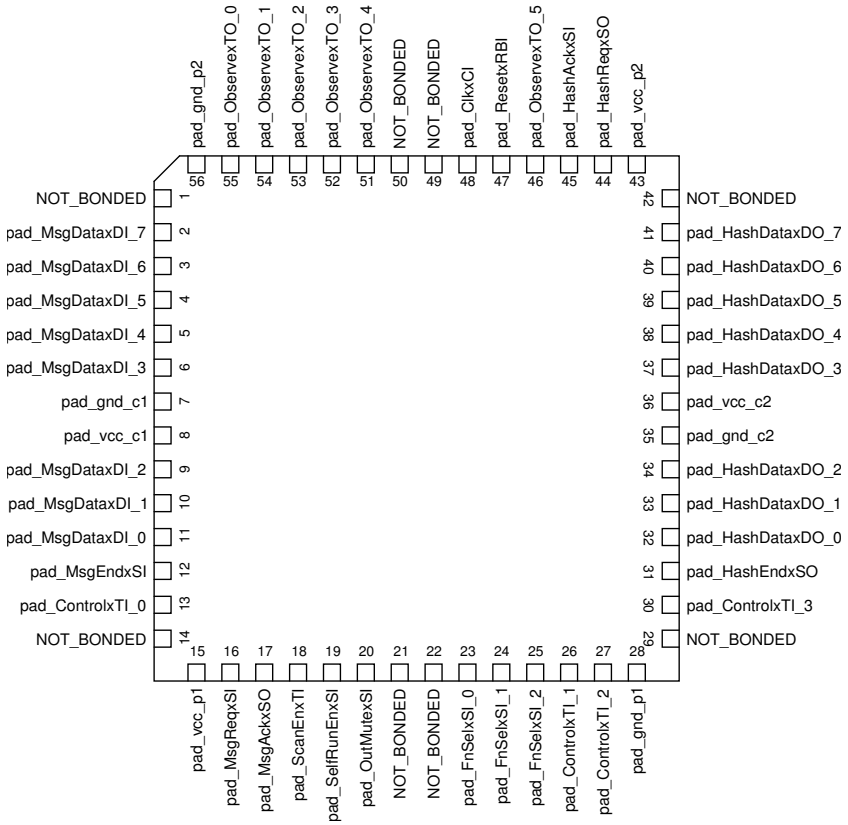


Figure A.3.: Pinout for MyFancyChip.

Appendix B

Compact Guide to LaTeX and the *iisreport* Class

Writing a report with LaTeX might at first not be as intuitive as with WYSIWYG editors. However, once you get used to the (rather simple) syntax, you will soon discover how powerful it is and how it helps you achieve tasks that are very difficult (if not impossible) to achieve with WYSIWYG editors.

This report template provides everything you need to get you started working with LaTeX. The rest of this chapter contains a short guide with examples for commonly used features.

B.1. Building the document

Generate a PDF file from this template by simply executing `make` in the directory where the top-level `.tex` file is in. Internally, this will invoke the `latexmk` program, which is the simplest and most consistent way to build a LaTeX document and is included in all recent LaTeX distributions. Additionally, `make` will check the `fig/` directory and generate PDF files for those figure raw files it knows how to compile (more on this in Appendix B.7).

B.2. Text editing and spacing

White spaces and line breaks are automatically inserted when the document is compiled. For this, the number of spaces between two words is irrelevant, but a different space length is automatically inserted after a period to make sentences better distinguishable. If you write a period that does not end a sentence, e.g., when mentioning Prof. Dr. S. Body, you need to escape the spaces between the abbreviated words with a backslash \. If you want to prevent LaTeX from breaking a line at a specific white space, you have to replace that space with a tilde ~. LaTeX also automatically hyphenates English words, so it can break a line within a word, although it does so only cautiously. Automatic

hyphenation can fail, e.g., for non-standard words, causing overly long lines. In such cases you have two options: First, if you have to break a standard word at an uncommon position, you can insert a `\-` at that position in the word. Second, you can define the hyphenation of a non-standard word by adding `\hyphenation{Jab-ber-woc-ky}` to the preamble[1] of your document.

A line of text is not broken at the same position as in the source code. For this reason, we suggest you put one sentence on one line of source code because this allows your VCS to track content changes much better than if you wrap lines within sentences. To start a new paragraph, insert an empty line. You can manually break a line by writing `\\`; however, this is rarely necessary and wide usage of it is a sign of fighting the typesetting system.

By default in this template, paragraphs start with a short indentation and are not separated by vertical white space, but this can be changed. If you prefer the latter, pass the `parskip` option to the `iisreport` document class, i.e., change the first line of your main document to `\documentclass[parskip]{iisreport}`.

B.2.1. Special characters

Many special characters are available, and they are all listed in *The Comprehensive LaTeX Symbol List* [9]. At the beginning you might not know what to search for, though, so it can be more helpful to use the *Detexify*[2] web app where you can draw the symbol you are looking for.

A frequent mistake is to mix up the three dashes (-, –, and —). The rules are simple [10]:

- The *hyphen*, `-`, is used between the elements of compound words; e.g., “run-time”.
- The *en-dash*, `--`, is used for ranges; e.g., “3–7”.
- The *em-dash*, `---`, is used for digressions within or at the end of a sentence—although you should use it sparingly.

Another frequent mistake are wrong quotation marks. Fortunately, this can also easily be avoided: Use `'text'` for ‘single quotation marks’ and `''text''` for “double quotation marks” (have a look at the source code to see the matching pairs). In American English, double quotes prevail and single quotes are typically only used inside double quotes.

B.2.2. Font faces and emphasis

The font face can be changed locally with the commands in Table B.1. The text to appear differently has to be put between the curly braces `{}`, i.e., the text is an *argument* to one of the commands. Some font faces can also be combined by nesting them. For example, `\textbf{\textit{some words}}` becomes ***some words***.

[1] The *preamble* of a document is formed by all code between the `\documentclass` command and the beginning of the document body after `\begin{document}`.

[2] http://detexify.kirelabs.org/classify.html

Command	Output
`\textrm{}`	Roman (the default in this document)
`\textsf{}`	Sans serif
`\texttt{}`	`Typewriter (i.e., all characters have the same width)`
`\textbf{}`	**Bold**
`\textit{}`	*Italic*
`\textsl{}`	*Slanted*
`\textsc{}`	SMALL CAPS

Table B.1.: Different font faces.

Command	Output sample
`\tiny`	quick brown foxes
`\scriptsize`	quick brown foxes
`\footnotesize`	quick brown foxes
`\small`	quick brown foxes
`\normalsize`	quick brown foxes
`\large`	quick brown foxes
`\Large`	quick brown foxes
`\LARGE`	quick brown foxes
`\huge`	quick brown foxes
`\Huge`	quick brown foxes

Table B.2.: Different font sizes.

When you want to *emphasize* text, use the `\emph{}` command instead of one of the commands in Table B.1. In this way, you separate a *property* of a piece of text (i.e., which text is emphasized) from its *formatting* (i.e., how emphasized text looks like). This is an important principle in typesetting with LaTeX. In this case, it allows you to define the formatting of all emphasized text independently of which text is meant to be emphasized.

B.2.3. Font sizes

The font size can be changed with the commands in Table B.2. These commands change the size within a given *scope*; for instance `{\Large some words}` only prints "some words" large.

Figure B.1.: Selected predefined colors, sorted by hue.

B.2.4. Coloring text

The color of text can be changed with the `\textcolor` and `\color` commands: The former takes two arguments, a declared color and the text to color. For example, `\textcolor{RoyalBlue}{I am royal}` becomes I am royal. The latter takes only a defined color and colors all text in its scope. For example, `{\color{RoyalPurple}So am I}` becomes So am I.

Figure B.1 shows a selection of predefined colors. The `xcolor` package manual [11] lists more colors and describes how to define custom colors.

B.3. Debugging

Occasionally, you will make syntax mistakes while writing a document, causing compilation to fail. In this case, the last lines of the console output will mention an error and point you to a `.log` file in the directory where you ran `make`. Open that log file. Even though that file can be very long and contains many technical details that are of no interest to you, finding errors is easy: simply search for lines starting with an exclamation mark!

If you use an undefined command, e.g., due to a typo, the error messages should be very helpful. If, however, you cause parentheses or environment delimiters to mismatch, the position of your mistake is hard to derive from the error messages.

A good technique to locate a mistake is to comment out recent changes until the document compiles neatly again. For recompilation, you should use `make clean all` to prevent errors that crept into temporary files from disturbing your bug hunt. To reduce compilation time for large documents, you can comment out chapters that are known to

be good. When you have a working version again, re-enable the code you commented out last piece-by-piece. This piecewise reduction should help you systematically find the mistake.

B.4. Math mode

LaTeX is probably the most powerful and elaborate tool to typeset mathematical content. Once you know a few core concepts, writing properly formatted mathematical content becomes quite simple.

To distinguish maths from regular text, maths is written in *math mode*. There are two categories that differ in their presentation: inline and displayed. Inline maths, e.g., $a^2 + b^2 = c^2$ is enclosed in `$` signs. It is meant for simple expressions. More complex content is displayed separately from the text. The most common way to display maths is inside the `equation` environment[3], for example:

$$\int_{-\infty}^{\infty} x \, \mathrm{d}x = 0. \tag{B.1}$$

Subscripts are written with `\sb{}`[4], superscripts with `\sp{}`, integrals with `\int`, and sums with `\sum`. Have a look at the source code of the last paragraph for usage examples.

Equations get a number by default, so you can label and refer to them (more on this in Appendix B.9). If you want to suppress an equation number, you can use the *starred* version of the equation environment, i.e., `equation*`.

Many mathematical symbols and functions are predefined, letting you express relations such as $\forall\, x \in \mathbb{R}\ \exists\, n \in \mathbb{N} : \ldots$ fluently. Wikipedia[5] has a list of all predefined mathematical symbols.

Variable names are by default one character long, causing `$ xy z $` to be typeset with identical spacing as `$ xyz $`. You should thus use single-letter variables whenever possible. If you have to use multi-letter variables, write them inside the `\var{}` command[6]. This causes x and y in the two-letter variable xy to be closer together. To make the variable clearly distinguishable from the next one, however, you may still have to insert a space[7] (as in $xy\,z$) or even an operator (as in $xy \cdot z$).

[3] *Environments* in LaTeX are similar to commands, but are usually used for larger chunks of code. For example, the entire document except the preamble is inside the `document` environment. Environments are formed with `\begin{environmentname} ... \end{environmentname}`.

[4] By default, LaTeX would allow to use the underscore _ for subscripts. In the `iisreport` document class, however, the underscore is a regular, printable character. The rationale is that the underscore is very common in technical designators and having to escape every single one is a common source of errors.

[5] https://en.wikibooks.org/wiki/LaTeX/Mathematics#List_of_Mathematical_Symbols

[6] The `var` command is not standard LaTeX but defined by the `iisreport` document class. If you want to use it elsewhere, it is very simple to implement: https://tex.stackexchange.com/a/129434/92384.

[7] The most common horizontal spacing macros in math mode are (in increasing order): `\,`, `\;`, `\enspace`, `\quad`, and `\qquad`. A complete list with examples is available here: https://tex.stackexchange.com/a/74354/92384. Keep in mind that frequent insertion of manual spacing may be a hack around a more fundamental problem.

When you want to use text in math mode (subscripts are a common use case for this), you must write that text inside the `\text{}` command to avoid the same problems as with multi-letter variables.

B.4.1. Delimiters: Parentheses, brackets, bars, and intervals

For simple parentheses and square brackets, you can readily use the `()` and `[]` characters, respectively. Curly braces are a bit more involved because `{}` are grouping characters. Thus, you would have to escape them and write `\{\}` instead. However, we recommend to use the `\cbr{}` command (short for "curly braces") instead. That command has the additional advantage of automatically sizing parentheses to the content. (The automatic sizing can be disabled by passing `[0]` as optional first argument to the command.) If you need automatically sized parentheses and square brackets, the commands are `\del{}` (for "delimiter") and `\sbr{}` (for "square bracket"), respectively. This allows you to effortlessly maintain readability even in deeply nested equations:

$$\left(E\left[\min\left\{X_1, X_2\right\}\right] - \left(\pi - \arccos\left(\frac{y}{r}\right)\right)\right)^n. \tag{B.2}$$

Absolute values are written with `\abs{}`, e.g.,

$$\left|\exp\left(x\pi i\right)\right| = 1 \quad \forall\, x \in \mathbb{R}, \tag{B.3}$$

while vector norms are written with `\norm{}`, e.g.,

$$\left\|\vec{x}\right\|_2 = \sqrt{\sum_{k=1}^{n} x_k^2} \quad \forall\, \vec{x} \in \mathbb{R}^n, \tag{B.4}$$

where the vector $\vec{x}$ was written with `\vec{x}` and the square root with `sqrt{..}`.

Intervals are written with the `\intxy{}` commands, where each of `x` and `y` are either `o` for open or `c` for closed.

B.4.2. Differential and derivative operators

Differential and derivative operators are written with the following commands: `\dif x` is the simple differential operator, e.g., $\mathrm{d}x$. `\Dif x` is a derivative operator, e.g., $\mathrm{D}x$. `\od[n]{f}{x}` is the ordinary n-th derivative operator, e.g., $\frac{\mathrm{d}^n f}{\mathrm{d}x^n}$. `n` is optional and should be omitted for the first derivative. `\pd[n]{f}{x}` is the partial n-th derivative operator, e.g., $\frac{\partial^n f}{\partial x^n}$. Finally, `\md{f}{n}{x}{q}{y}{r}` is the mixed partial derivative operator, e.g., $\frac{\partial^n f}{\partial x^q \partial y^r}$, where `n` is the total order of differentiation and `q` and `r` are the orders of differentiation for `x` and `y`, respectively.

B.4.3. Vectors, matrices, and distinction of cases

Both vectors and matrices are written with the `Xmatrix` environments, where `X` defines the delimiter of the matrix and can be `p` for parentheses, `b` for brackets, `B` for curly braces, `v` for vertical bars, `V` for double vertical bars, or omitted for no delimiters. The matrix is written row-wise with the elements of a row separated by `&` and each row is terminated by `\\`. A column vector is just a matrix with one column, a row vector one with one row. For example:

$$x = \begin{pmatrix} x_1 & x_2 \end{pmatrix} \quad y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} \quad A = \begin{pmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{2,2} \end{pmatrix} \tag{B.5}$$

The `Xmatrix` environments center the columns by default. If you want a different alignment, use the starred variant[8] of the environments, which accepts a single character as optional argument[9]: `r` for right, `c` for center, and `l` for left.

To write case distinctions, use the `dcases` environment. For example:

$$a(v) = \begin{cases} 0 & \text{if } v \geq \mathrm{c}, \\ \epsilon > 0 & \text{else.} \end{cases} \tag{B.6}$$

If you want the curly brace to be on the right of the cases, use the `rcases` environment.

B.4.4. Multi-line equations

If you want to write a single equation that is longer than one line, use the `multline` (without 'i'!) environment. That environment switches to math mode by itself, so you *must not* use it inside `equation`. Use the line break command, `\\`, to define the two lines of the equation. For example:

$$\begin{aligned} \alpha + \beta + \gamma + \delta + \epsilon + \zeta + \eta + \theta + \iota + \kappa + \lambda + \mu + \nu + \xi + + \pi + \rho + \sigma + \tau \\ = \upsilon + \phi + \chi + \psi + \omega. \end{aligned} \tag{B.7}$$

To write multiple equations in series or an especially complicated multi-line equation, use the `IEEEeqnarray` environment. That environment takes a series of characters specifying the columns as argument. The most common argument is `rCl`, meaning one right-aligned column followed by a center-aligned separator followed by a left-aligned column. As with matrices, use `&` to separate columns and `\\` to separate equation lines. For example:

$$a = b + c \tag{B.8}$$

$$\begin{aligned} &= d + e + f + g + h + i + j + k \\ &\quad + l + m + n + o \end{aligned} \tag{B.9}$$

$$= p + q + r + s, \tag{B.10}$$

[8] The *starred variant* of an environment or a command simply has a star $*$ at the end of the environment or command name, respectively. Not all environments and commands have a starred variant.

[9] *Optional arguments* are always enclosed in square brackets `[]`.

where the first line of the second equation was ended by `\nonumber` to suppress numbering that part of the equation.

Browse through *How to Typeset Equations in LaTeX* [12] for further informations and solutions to more complex examples.

B.4.5. Definitions, theorems, lemmas, and proofs

Here are some examples on writing definitions, theorems, lemmas, and proofs.

Definition B.1 (Singularity)**.** Let U be an open subset of the complex numbers $\mathbb{C}$, $a \in U$, and f be a complex differentiable function defined on $U \setminus \{a\}$.

The point a is a *removable singularity* of f if there exists a holomorphic function g defined on all of U such that $f(z) = g(z)\ \forall\, z \in U \setminus \{a\}$.

The point a is a *pole* or *non-essential singularity* of f if there exists a holomorphic function g defined on U with $g(a) \neq 0$ and $n \in \mathbb{N}$ such that

$$f(z) = \frac{g(z)}{(z-a)^n} \quad \forall\, z \in U \setminus \{a\}\,. \tag{B.11}$$

The lowest such number n is called the *order of the pole*.

The point a is an *essential singularity* of f if it is neither a removable singularity nor a pole. The point a is an essential singularity iff the Laurent series has infinitely many powers of negative degree.

Theorem B.1 (Residue Theorem)**.** *Let f be analytic in the region G except for the isolated singularities $a_1, a_2, \ldots, a_m$. If γ is a closed rectifiable curve in G that does not pass through any of the points a_k and if $\gamma \approx 0$ in G, then*

$$\frac{1}{2\pi i}\int_{\gamma} f = \sum_{k=1}^{m} n(\gamma; a_k)\, Res(f; a_k). \tag{B.12}$$

Proof. Left as an exercise for the reader. □

Lemma B.2 (Schwarz)**.** *Let $D := \{z \in \mathbb{C} : |z| < 1\}$ be the open unit disk in the complex plane centered at the origin, and let $f : D \to \mathbb{C}$ be a holomorphic map such that $f(0) = 0$ and $|f(z)| \leq 1$ on D. Then, $|f(z)| \leq |z|\ \forall\, z \in D$ and $|f'(0)| \leq 1$. Moreover, if $|f(z)| = |z|$ for some $z \neq 0$ or $|f'(0)| = 1$, then $f(z) = az$ for some $a \in \mathbb{C}$ with $|a| = 1$.*

Proof. Beyond the scope of this document. □

B.5. Quantities with SI units

- quantity with a unit: 300 MHz
- unit alone: GV

- ranges of quantities with units: 2 Mibit/s to 256 Mibit/s
- number (especially for engineering notation or very large numbers): 10 000, 3.14×10^6, 5×10^{-12}
- ranges of numbers 5×10^{-12} to 3.14×10^6

Math mode not required, but can be used with it.

B.6. Enumerations and itemizations

Itemizations are ...

- unnumbered and
- written inside the `itemize` environment, where every item starts with `\item`.

Enumerations, on the other hand, are ...

1. numbered and
2. written inside the `enumerate` environment.

Both itemizations and enumerations can be nested. The indentation level and itemization items are then automatically adjusted:

1. This demonstrates that
   a) enumerations and
   b) itemizations
      - can be nested.

B.7. Floats: figures and tables

Both figures and tables normally form *floating* environments. This means that LaTeX will automatically place them near to where they were in the source code, but not at the exact same position. The placement algorithm is fairly sophisticated [13] and usually works reasonably well.

The base environment for figures is `figure`, the one for tables is `table`. Floats usually get a caption with the `\caption{}` command. If you want to refer to them (more on this in Appendix B.9), you additionally have to put a `\label{}` after the `\caption{}` but on the same line (to have correct page numbers even near page breaks).

To center-align the content of a float, use the `\centerfloat` command at the beginning of that float.

Figure B.2.: Example figure.

Decimal	Hexadecimal	Octal	Binary
10	A_{16}	12_8	1010_2
13	D_{16}	15_8	1101_2

Table B.3.: Simple example table with some values in different number systems.

B.7.1. Figures

Images can be included with the `\includegraphics[properties]{file_name}` command, where `properties` allows to, e.g., define the `width`, `height`, or `scale` of an image in the `key=value` syntax. The `file_name` is relative to the `fig/` directory and the default suffix, `.pdf`, can be omitted. An example figure is given in Fig. B.2.

Whenever possible, you should use SVGs for two reasons: First, they can be scaled losslessly to the target size and resolution. Second, they allow you to keep a small, modifiable source file of the graphic under version control and have the PDF file to be included built automatically.

We recommend using *Inkscape*[10]. Simply draw a figure in Inkscape, set the canvas to where you want the image border, save the original `.svg` file in the `fig/` directory, and use `\includegraphics` on the file name without suffix. When you run `make`, the corresponding PDF file will get built automatically and included in your document. If you use a VCS (we highly recommend to do so!), track the original `.svg` file but add the auto-built `.pdf` file to the ignore list (e.g., `fig/.gitignore`). If you include PDF files of which you have no source files, track that `.pdf` file in your VCS.

As Inkscape does not support embedding fonts in its SVG files, you should either only use standard, widely-available fonts[11] or track the auto-built PDF images in `fig/` with your VCS. If you choose the latter, though, be aware that other collaborators who do not have the font installed must not commit changes to the built PDF images (because text with missing fonts will be rendered incorrectly by their Inkscape).

B.7.2. Tables

B.8. Algorithms and source code listings

Algorithm 9 shows an example algorithm. Have a look at the source code to discover how it works.

[10] Freely available at `https://inkscape.org`.
[11] Web safe fonts are good candidates for widely available fonts.

Algorithm 9: Disjoint decomposition.

input : A bitmap Im of size $w \times l$
output: A partition of the bitmap

```
1  special treatment of the first line;
2  for i ← 2 to l do
3      special treatment of the first element of line i;
4      for j ← 2 to w do
5          left ← FindCompress(Im[i, j − 1]);
6          up ← FindCompress(Im[i − 1, ]);
7          this ← FindCompress(Im[i, j]);
8          if left compatible with this then // O(left,this)==1
9              if left < this then Union(left,this);
10             else Union(this,left);
11         end
12         if up compatible with this then                    // O(up,this)==1
13             if up < this then Union(up,this);
               // this is put under up to keep tree as flat as possible
14             else Union(this,up);
               ;                                          // this linked to up
15         end
16     end
17     foreach element e of the line i do FindCompress(p);
18 end
```

Some	title	words
odd	odd	odd
even	even	even
odd	odd	odd

Table B.4.: Simple example table with different row and cell colors.

Day	Min. Temp.	Max. Temp.	Description
Monday	11 °C	22 °C	A clear day with lots of sunshine. However, a strong breeze will bring down the temperatures.
Tuesday	9 °C	19 °C	Cloudy with rain across many northern regions. Clear spells across most of Scotland and Northern Ireland, but rain reaching the far northwest.

Table B.5.: Example table with fixed column widths: 1.5 cm for columns two and three, 7 cm for column four. Content adopted from Wikibooks.

Source code listings are written inside the `lstlisting` environment, which takes optional arguments such as the `language` of the code, a `caption`, and a `label` in `key=value` syntax. Listing B.1 shows an example listing.

Listing B.1: Simple C code snippet.

```
int main()
{
  return 0;
}
```

External source code files can be included as listing with the `\lstinputlisting{}` command. Since you rarely want to include an entire file, you can specify the first and the last line with the `firstline` and the `lastline` key, respectively.

B.9. Citing and referencing

Use the `\cref{}` command to reference a document-internal label. Use the `\cite{}` command to cite an entry in the bibliography. You should always use a nonbreaking space, ~, before the `\cite{}` command to prevent the citation label from falling to the next line.

B.10. Printing and binding

When printing this document, pass the `print` option to the `iisreport` class. This will cause the layout to be optimized for printing.

B.11. Further reading

For a guide on writing research reports, we recommend the *Manual for Writers of Research Papers, Theses, and Dissertations* [14]. Furthermore, *The Elements of Style* [15] and *The Chicago Manual of Style* [16] are useful guides on concise writing in general. The latter is freely available online within the ETH network.[12]

If you are interested to learn more about LaTeX, you will find many resources online but the majority of them are written by casual amateurs and can teach you bad practices. Their approach is not strictly wrong but in the long term can cost you a lot of time compared to proper solutions. Thus, let us suggest to start all your TeX-related searches at the TeX StackExchange[13] community. Especially in the beginning, you will encounter common problems, for which there are usually several high-quality solutions on StackExchange, complete with an explanation of why the problem arose.

There are also some very good books on LaTeX. *The Not So Short Introduction to LaTeX2e* [17] is an excellent start, and *More Math into LaTeX* [18] teaches 99 % of what you need to know about typesetting maths. The first book[14] and the first section of the second book[15] are freely available online. For more advanced users, *The LaTeX Companion* [19] and the *The TeXbook* [10] are the definitive books.

B.12. *iisreport* options quick reference

The *iisreport* document class offers a few options that allow you to easily customize the layout of your report and configure certain features. Options can be specified inside the square brackets on the first line of the `report.tex` file, with individual options separated by commas. Here is an overview of all available options in alphabetic order:

- `oldfonts` brings back the fonts from the legacy IIS report template.
- `oldsubscript` disables making the underscore printable and makes it usable for subscripts instead.
- `parskip` causes paragraphs to start with a vertical space of half a line instead of indentation.
- `print` optimizes the page layout for printing and binding.

[12] http://www.chicagomanualofstyle.org
[13] https://tex.stackexchange.com/
[14] http://tobi.oetiker.ch/lshort/lshort.pdf
[15] http://www.ctan.org/tex-archive/info/Math_into_LaTeX-4/Short_Course.pdf

Appendix C

Task Description

Include the task description PDF file you got from your advisors with `\includepdf[pages=-, scale=0.9]{../path/to/task/description.pdf}`.

List of Acronyms

ASIC . . . . . application-specific integrated circuit

DFT . . . . . design for testability

DZ . . . . . Microelectronics Design Center at ETH Zurich

FPGA . . . . . field-programmable gate array

IC . . . . . integrated circuit

IIS . . . . . Integrated Systems Laboratory

PDF . . . . . Portable Document Format

SNR . . . . . signal-to-noise ratio

SPSE . . . . . Situation-Problem-Solution-Evaluation

SVG . . . . . scalable vector graphics

VCS . . . . . version control system

WYSIWYG . . . "what you see is what you get"

List of Figures

List of Tables

Bibliography

[1] M. Yannakakis, "Computing the Minimum Fill-In is NP-Complete," *SIAM Journal on Algebraic Discrete Methods*, vol. 2, no. 1, pp. 77–79, Mar. 1981, publisher: Society for Industrial and Applied Mathematics. [Online]. Available: https://epubs.siam.org/doi/10.1137/0602010

[2] R. J. Lipton, D. J. Rose, and R. E. Tarjan, "Generalized Nested Dissection," *SIAM Journal on Numerical Analysis*, vol. 16, no. 2, pp. 346–358, Apr. 1979, publisher: Society for Industrial and Applied Mathematics. [Online]. Available: https://epubs.siam.org/doi/10.1137/0716027

[3] A. George, "Nested Dissection of a Regular Finite Element Mesh," *SIAM Journal on Numerical Analysis*, vol. 10, no. 2, pp. 345–363, Apr. 1973, publisher: Society for Industrial and Applied Mathematics. [Online]. Available: https://epubs.siam.org/doi/10.1137/0710032

[4] U. V. Çatalyürek, C. Aykanat, and E. Kayaaslan, "Hypergraph Partitioning-Based Fill-Reducing Ordering for Symmetric Matrices," *SIAM Journal on Scientific Computing*, vol. 33, no. 4, pp. 1996–2023, Jan. 2011, publisher: Society for Industrial and Applied Mathematics. [Online]. Available: https://epubs.siam.org/doi/10.1137/090757575

[5] A. Dasgupta and P. Kumar, "Alpha Elimination: Using Deep Reinforcement Learning to Reduce Fill-In During Sparse Matrix Decomposition," in *Machine Learning and Knowledge Discovery in Databases: Research Track: European Conference, ECML PKDD 2023, Turin, Italy, September 18–22, 2023, Proceedings, Part IV*. Berlin, Heidelberg: Springer-Verlag, Sep. 2023, pp. 472–488. [Online]. Available: https://doi.org/10.1007/978-3-031-43421-1_28

[6] Y.-H. Chang, A. Buluç, and J. Demmel, "Parallelizing the approximate minimum degree ordering algorithm: Strategies and evaluation," 2025. [Online]. Available: https://arxiv.org/abs/2504.17097

[7] D. Merrill, "Scalable GPU Graph Traversal."

[8] P. Kaleta, "kaletap/bfs-cuda-gpu," May 2025, original-date: 2019-11-14T12:55:50Z. [Online]. Available: https://github.com/kaletap/bfs-cuda-gpu

[9] S. Pakin, *The Comprehensive LaTeX Symbol List*, Jan. 2017. [Online]. Available: http://mirrors.ctan.org/info/symbols/comprehensive/symbols-a4.pdf

[10] D. E. Knuth, *The TEXbook*. Addison-Wesley, 1984.

[11] U. Kern, *xcolor: Driver-independent color extensions for LATEX and pdfLATEX*, May 2016. [Online]. Available: http://mirrors.ctan.org/macros/latex/contrib/xcolor/xcolor.pdf

[12] S. M. Moser, *How to Typeset Equations in LATEX*, Sep. 2017. [Online]. Available: http://moser-isi.ethz.ch/docs/typeset_equations.pdf

[13] F. Mittelbach, "How to influence the position of float environments like figure and table in LATEX," *TUGboat*, vol. 35, no. 3, pp. 248–254, 2014. [Online]. Available: https://www.latex-project.org/publications/tb111mitt-float.pdf

[14] K. L. Turabian, *A Manual for Writers of Research Papers, Theses, and Dissertations*. University of Chicago Press, 2013.

[15] W. Strunk, *Elements of Style*. Start Publishing LLC, 2012.

[16] U. of Chicago Press and U. of Chicago Press Editorial Staff, *The Chicago Manual of Style*. University of Chicago Press, 2017.

[17] T. Oetiker, *The Not So Short Introduction to LATEX2e*, Jun. 2016. [Online]. Available: https://tobi.oetiker.ch/lshort/lshort.pdf

[18] G. Grätzer, *More Math Into LATEX*. Springer International Publishing, 2016.

[19] F. Mittelbach, M. Goossens, J. Braams, D. Carlisle, and C. Rowley, *The LATEX Companion*, ser. Tools and Techniques for Computer Typesetting. Pearson Education, 2004.

[20] H. Kaeslin, *Digital Integrated Circuit Design: From VLSI Architectures to CMOS Fabrication*. Cambridge University Press, Apr. 2008.

[21] D. Hankerson, S. Vanstone, and A. Menezes, *Guide to Elliptic Curve Cryptography*, ser. Springer Professional Computing. Springer, 2004.

[22] NIST, *Advanced Encryption Standard (AES) (FIPS PUB 197)*, National Institute of Standards and Technology, Nov. 2001.

[23] P. Rogaway, M. Bellare, J. Black, and T. Krovetz, "OCB: A block-cipher mode of operation for efficient authenticated encryption," in *ACM Conference on Computer and Communications Security*, 2001, pp. 196–205.

[24] Xilinx. (2011, Nov.) Virtex-6 FPGA Configuration User Guide. UG360 (v3.4). [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug360.pdf

[25] ——. (2011, Oct.) 7 Series FPGAs Configuration User Guide. UG470 (v1.2). [Online]. Available: http://www.xilinx.com/support/documentation/user_guides/ug470_7Series_Config.pdf

[26] Wikipedia, "Isaac Newton," accessed October 1, 2012. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Isaac_Newton&oldid=514997436

[27] J. Wright, *siunitx: A comprehensive (SI) units package*, Aug. 2017. [Online]. Available: http://mirrors.ctan.org/macros/latex/contrib/siunitx/siunitx.pdf

[28] T. Miller and D. Parker, "Writing for the reader: A problem-solution approach," *English Teacher Forum*, no. 3, pp. 21–27, 2012. [Online]. Available: http://files.eric.ed.gov/fulltext/EJ997525.pdf

[29] M. Hoey, *On the Surface of Discourse*. Allen and Unwin, 1983.