UC Berkeley

UC Berkeley Electronic Theses and Dissertations

Title

Partial Left-Looking Structured Multifrontal Factorization & Algorithms for Compressed Sensing

Permalink

https://escholarship.org/uc/item/7zf0h7x0

Author

Wu, Cinna Julie

Publication Date

2012

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

Partial Left-Looking Structured Multifrontal Factorization & Algorithms for Compressed Sensing

by

Cinna Julie Wu

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Applied Mathematics

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor Ming Gu, Chair
Professor Jack Silver
Professor Laurent El Ghaoui

Fall 2012

Partial Left-Looking Structured Multifrontal Factorization & Algorithms for Compressed Sensing

Copyright 2012
by
Cinna Julie Wu

Abstract

Partial Left-Looking Structured Multifrontal Factorization & Algorithms for Compressed Sensing

by

Cinna Julie Wu

Doctor of Philosophy in Applied Mathematics

University of California, Berkeley

Professor Ming Gu, Chair

In this dissertation, we explore two problems involving large matrix computations. In the first half, we study the fast factorization of a large, sparse symmetric positive definite matrix and explain the underlying structures of the intermediate steps in the computation. Using these underlying structures, we present a new algorithm which takes advantage of these structures to reduce computational and memory costs. Our new algorithm is an improvement upon the well known and commonly used multifrontal method. We incorporate the use of fast structured computations to improve the algorithm as in [95, 92, 99].

In the second part of the dissertation, we explore a new algorithm for solving minimization problems of the form

$$\min \|Ax - b\| \quad \text{s.t.} \|x\|_1 \leq \tau,$$

where A is a wide matrix. This problem is often referred to in some fields as the Lasso problem. We improve upon an existing method by making a simple modification. Then we prove that this method is locally linearly convergent under certain reasonable conditions.

To my Sister

Contents

Contents ii

List of Figures v

List of Tables vi

I Partial Left-Looking Structured Multifrontal Method 1

1 Introduction 2
- 1.1 Problem Statement . . . 2
- 1.2 Applications . . . 3
  - 1.2.1 Finite element mesh . . . 4
  - 1.2.2 Laplace's equation . . . 5
- 1.3 Cholesky Factorization . . . 6
- 1.4 Existing Algorithms and Methods . . . 7
- 1.5 Organization of Part I of Dissertation . . . 9

2 Multifrontal Method 10
- 2.1 Introduction to the Multifrontal Method . . . 10
- 2.2 Preprocessing Steps . . . 10
  - 2.2.1 Step 1: Nested dissection ordering . . . 10
    - 2.2.1.1 Outline . . . 11
    - 2.2.1.2 Separator ordering . . . 12
    - 2.2.1.3 Ordering of nodes within separators . . . 12
    - 2.2.1.4 Node elimination and fill-in . . . 13
  - 2.2.2 Step 2: Symbolic factorization . . . 16
- 2.3 Step 3: Multifrontal Method . . . 16
  - 2.3.1 Elimination tree . . . 19
  - 2.3.2 Frontal and update matrices . . . 21
    - 2.3.2.1 Storage of the update matrices . . . 23
  - 2.3.3 Multifrontal method with dense update block storage . . . 23

3 Partial Left-Looking Multifrontal Method **25**
3.0.4 Terminology 26
3.1 Frontal and Update matrix structures 27
3.2 Partial Left-Looking Multifrontal Method 33
3.2.1 Left-looking multifrontal method 33
3.2.2 Stack storage 34
3.2.3 Storage of the supernodal frontal matrix within the stack 35
3.2.4 Partial left-looking structured multifrontal method 37
3.2.5 An example 37
3.3 Comparison of Memory Costs 37
3.4 Numerical Experiments 42

4 Cholesky Factorization of the Frontal Matrices **48**
4.1 Overview 49
4.2 Preliminaries 50
4.2.1 Notation and terminology 50
4.2.2 Introduction to symmetric HSS matrices 50
4.2.3 Low-rank matrix approximation 53
4.3 Generalized HSS Cholesky Factorization for SPD matrices 55
4.3.1 Merging child blocks 59
4.3.2 HSS solver with generalized Cholesky factors 61
4.3.3 Complexity of construction 61
4.4 Incorporating into the Partial Left-Looking Structured Multifrontal Method 63
4.4.1 Complexity and memory requirements 65
4.4.2 Schur-monotonicity 65

II An Algorithm for Compressed Sensing **66**

5 Introduction **67**
5.1 Problem Statement and Background 68
5.1.1 Motivation 69
5.1.2 Various l_1-relaxed formulations 70
5.1.3 Notation, terminology, and assumptions 71
5.1.4 Organization of Part II of dissertation 72
5.2 NESTA-LASSO 72
5.2.1 Nesterov's algorithm 72
5.2.2 NESTA-LASSO-K: An accelerated proximal gradient algorithm for LASSO 73
5.2.3 l_1-projector 74

6 Convergence Proofs **77**

6.1 Local Linear Convergence and Optimality 77
6.1.1 Almost sure sparsity of Nesterov's method 78
6.1.2 Local linear convergence of NESTA-LASSO 81

7 Parnes: Solving the Basis Pursuit Denoise Problem 85
7.1 PARNES 85
7.1.1 Pareto curve 85
7.1.2 Root finding 86
7.1.3 Solving the LASSO problem 87

8 Numerical Experiments 88
8.1 Other Solution Techniques and Tools 88
8.1.1 NESTA [9] 88
8.1.2 GPSR: Gradient projection for sparse reconstruction [34] 88
8.1.3 SpaRSA: Sparse reconstruction by separable approximation [35] 89
8.1.4 SPGL1 [12] and SPARCO [11] 89
8.1.5 FISTA: Fast iterative soft-thresholding algorithm [8] 89
8.1.6 FPC: Fixed point continuation [54, 55] 89
8.1.7 FPC-AS: Fixed-point continuation and active set [91] 90
8.1.8 Bregman iteration [100] 90
8.1.9 C-SALSA [1, 2] 90
8.2 Numerical Results 91
8.2.1 Choice of parameters 93
8.3 Conclusions 96

Bibliography 97

List of Figures

1.1 Example of an FEM mesh. 4
1.2 Example of a sparse symmetric positive matrix A from discretizing a PDE. . . . 5
1.3 Rectangular domain with a regular mesh and five-point stencil. 6

2.1 Nested dissection. 11
2.2 One step of block Cholesky. 15
2.3 Connections during the factorization. 16
2.4 Matrix reordered with nested dissection. 17
2.5 Nonzero structure of the Schur complement components. 19
2.6 Elimination Tree. 20
2.7 Schur complement vs. frontal matrix. 22

3.1 Levels in a elimination tree with height l and switching level at level s. 27
3.2 Illustration of the representation of $\mathcal{F}_i$ in (3.3) 28
3.3 Element containing i 29
3.4 Illustration of the representation of $\mathcal{F}_i$ in (3.3) 32
3.5 2D discrete Laplacian on a 5-point stencil with $n = 500$ 43
3.6 Computation time comparisons for different mesh sizes. 44
3.7 Maximum stack sizes for different mesh sizes. 45
3.8 PMF comparisons for different switching levels; $l = 12, tol = 10^{-8}$. 47

4.1 Matrix partition and the corresponding index sets and binary tree $\mathcal{T}$. 51
4.2 The visited set $\mathcal{V}_5$. (The number under each node i in Figure 4.2 denotes the cardinality s_i of $\mathcal{V}_i$.) 52
4.3 The factorization process. 57
4.4 The second HSS block row 57

7.1 An example of a Pareto curve. The solid line is the Pareto curve; the dotted red lines give two iterations of Newton's method. 86

List of Tables

3.1 Comparison of two stack uses. Highlighted nodes are nodes in the stack at step i. 36
3.2 First 15 steps of Algorithm 4 when applied to the example in Figure 3.4. The elimination tree has 31 nodes, five levels, and switching level $s = 4$. 39
3.3 Computational time comparisons; $\tau = 10^{-4}$. 46
3.4 Maximum stack size comparisons; $\tau = 10^{-4}$. 46

4.1 Flops counts of some matrix operations. 61

6.1 Number of products with A and $A^\top$ for NESTA-LASSO without prox-center updates (cf. Algorithm 9) and NESTA-LASSO with prox-center updates (cf. Algorithm 11). These values are given by N_A and N_A^{update} respectively. 84

8.1 Comparison of accuracy using experiments from Table 8.2. Dynamic range 100 dB, $\sigma = 0.100$, $\mu = 0.020$, sparsity level $s = m/5$. Stopping rule is (8.1). 93
8.2 Number of function calls where the sparsity level is $s = m/5$ and the stopping rule is (8.1). 94
8.3 Number of function calls where the sparsity level is $s = m/5$ and the stopping rule is (8.2). 94
8.4 Recovery results of an approximately sparse signal (with Gaussian noise of variance 1 added) and with (8.2) as a stopping rule. 95
8.5 Recovery results of an approximately sparse signal (with Gaussian noise of variance 0.1 added) and with (8.2) as a stopping rule. 95

Acknowledgments

First and foremost, I would like to thank my advisor Ming Gu. Not only was he an excellent advisor in a mathematical sense, he also played an enormous role in my personal growth over the years. I am very grateful for his continuous patience, kindness, and fascinating stories. Without him, I would not be where I am today. I thank him for the tremendous amount of time and energy he put into coming up with projects and ideas, always making time for helpful discussions, and giving all around great advice.

Second of all, I am very thankful for my wonderful family. They have stood behind me throughout this degree, supporting whatever decision I made. I thank my parents, Jeff and Shelly Wu, for their love and generosity, my sister, Regina Wu, for her patience, and my brother, Andrew Wu, for being there....in front of a computer.

Third, I would like to thank the people I have worked with throughout the years for their helpful discussions - Lek-Heng Lim, Shengguo Li, and Joon-Ho Lee. I am grateful to have had the opportunity to work with such a wide variety of wonderful people. Thanks to Shengguo Li for the use of his code.

Many thanks to Jianlin Xia at Purdue University. The first part of this thesis was built upon his work in the field of direct methods for structured matrix computations. He also kindly provided the initial multifrontal code from which our code was built upon.

Special thanks to my good friends in the math department: Shuchao Bi, Trevor Potter, Kevin Lin.

Part I

Partial Left-Looking Structured Multifrontal Method

Chapter 1

Introduction

Rapidly performing computations with extremely large matrices has become a ubiquitous task in applications involving massive data sets such as solving large scale PDEs. Such large, structured linear systems of equations are critical in many computational and engineering problems. The special structures are often a result from using various techniques to linearize or discretize the problem at hand. For a general matrix of size $n \times n$, such computations cost $O(n^2)$ to $O(n^4)$ operations and require huge amounts of memory, scaling dramatically with matrix size. When matrices have special structures, one can exploit these properties to decrease the costs. Thus, designing efficient and reliable structured matrix computations has been an intensive focus of recent research. Numerically stable fast and superfast algorithms have been developed for structured matrices such as Toeplitz matrices, Vandermonde matrices, and various forms of semi-separable matrices. In this dissertation, we focus on fast structured methods for computing the Cholesky factorization of a large, sparse symmetric positive-definite matrix.

1.1 Problem Statement

Consider a symmetric positive definite (SPD) system of equations

$$\mathcal{A}x = b$$

where $\mathcal{A}$ is a large, sparse structured matrix of size $N \times N$. We wish to quickly and accurately solve such a system for x. Typical Cholesky factorization methods have a computational complexity of $O(N^3)$ and are not practical for large problems. Thus, we develop a direct, generalized Cholesky factorization method which reduces computational and memory costs by taking advantage of the structure of $\mathcal{A}$. Our algorithm factors $\mathcal{A}$ as

$$\mathcal{A} = \mathcal{L}\mathcal{L}^T$$

where $\mathcal{L}$ is product of lower triangular and orthogonal matrices.

In particular, we are interested in large matrices $\mathcal{A}$ such that, with proper ordering, the intermediate matrices arising in the factorization possess a low-rank property: all off-diagonal blocks have small numerical ranks. These intermediate matrices can be approximated using a structure called a *hierarchically semiseparable* (HSS) representation. Existing fast algorithms for HSS matrices may then be applied to reduce costs. Such structured matrices often arise in the discretization of partial differential equations (PDEs) on a *finite element mesh* (grid) $\mathbb{M}$. See Section 1.2.1 for more details.

In this dissertation, we present a general direct solver for solving such a system via a generalized Cholesky factorization method. The method combines and improves upon various existing methods and is made up of the following five components:

- First, reorder the matrix $\mathcal{A}$ using the *nested dissection technique* (George 1973) [39]. This determines an order to eliminate the variables of $\mathcal{A}$ which will reduce fill-in (nonzero entries created in the Cholesky decomposition not in the original matrix) during the factorization.
- Next, apply a *symbolic Cholesky decomposition* to pre-determine the nonzero pattern of the factors $\mathcal{L}$ during the actual computation. This has been found to be essential for designing efficient sparse factorization algorithms.
- Perform the decomposition using the *multifrontal method* by Duff and Reid [30] which essentially reorganizes the factorization of a large, sparse matrix into factorizations of smaller dense matrices. This method is integrated with structured matrices as seen in [99, 92]. We incorporate a new way of storing the intermediate dense matrices which reduces memory costs by maintaining their special structures.
- The smaller dense matrices are factorized via a new *generalized HSS Cholesky factorization* which reduces costs by avoiding the need of computing Schur complements. Improvements in speed are made by incorporating randomization techniques with the matrix structures.
- Apply forward and backward substitutions to solve for x.

We have developed a Matlab software package which implements this algorithm.

1.2 Applications

Many computational engineering problems require numerically solving linear second-order partial differential equations (PDEs), i.e.,

$$-\sum_{i,j=1}^{d} c_{ij}\frac{\partial}{\partial x_i}\left(\frac{\partial u}{\partial x_j}\right) + -\sum_{i=1}^{d} b_i\frac{\partial u}{\partial x_i} + au = f \quad \text{in} \quad \Omega,$$

where the function u is unknown. Here, we examine the $2D$ and $3D$ cases where $d = 2, 3$. Some common examples of such PDEs include the Helmholtz equation, Laplace equation, and Poisson's equation. The Laplace and Poisson equations are examples of elliptic PDEs, i.e. the coefficient matrix $C = (c_{ij})$ is symmetric positive definite.

Two common numerical methods to solve PDEs are the finite element method (FEM) and the finite difference methods (FDM). Both methods involve first subdividing the domain Ω using a grid called a *mesh* where the intersection points are called *mesh points*, see Figure 1.1. The solution u is then approximated by solving a system of linear equations

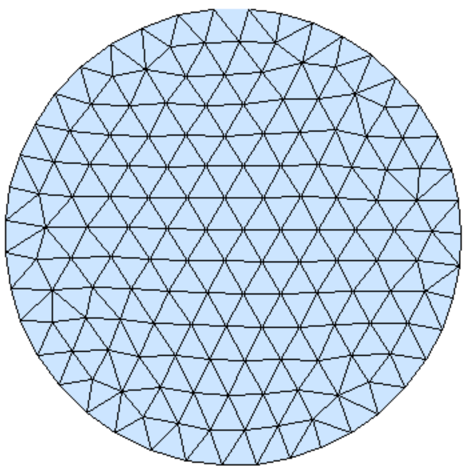


Figure 1.1: Example of an FEM mesh.

$$\mathcal{A}x = b \tag{1.1}$$

where $\mathcal{A} \in \mathbb{R}^{N \times N}$ and N is the number of mesh points. The matrix $\mathcal{A}$ is typically very large, sparse, and symmetric positive definite (SPD), see Figure 1.2. We focus on the problem of computing the Cholesky factorization of a large, sparse, SPD matrix arising from the discretization of a PDE. For the purpose of presentation, we focus on the $2D$ case with rectangular domains Ω.

1.2.1 Finite element mesh

In this manuscript, a *finite element mesh* $\mathbb{M}$ refers to a grid which is formed by partitioning a 2D or 3D region into subregions. The subregions are called *elements*, the boundaries of these subregions are called *edges*, and the connecting points of the edges are the *nodes* or *mesh points*. A *finite element system* associated to a mesh $\mathbb{M}$ is a symmetric positive definite (SPD) system

$$\mathcal{A}x = b$$

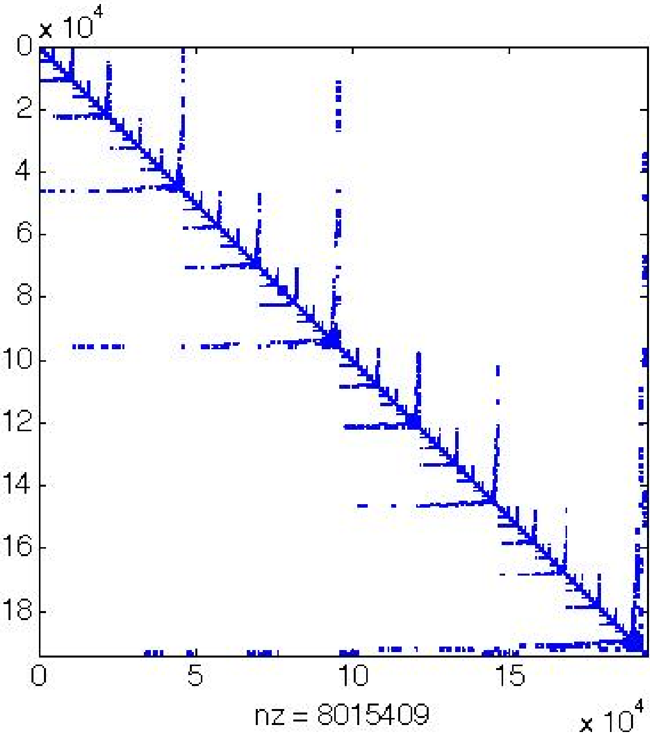


Figure 1.2: Example of a sparse symmetric positive matrix A from discretizing a PDE.

where each variable corresponds to a node of $\mathbb{M}$, and $\mathcal{A}_{ij} \neq 0$ if and only x_i and x_j belong to the same element in $\mathbb{M}$. Thus, an $n \times n$ regular mesh corresponds to an $n^2 \times n^2$ matrix $\mathcal{A}$. Such a matrix $\mathcal{A}$ arises when applying finite difference or finite element techniques to solve linear boundary value problems as seen in the next subsection.

1.2.2 Laplace's equation

As an example, Laplace's Equation, the simplest example of an elliptic PDE, is given by

$$\nabla^2 u := \frac{\partial^2 u}{\partial x_1^2} + \cdots + \frac{\partial^2 u}{\partial x_1^2} = 0. \tag{1.2}$$

As mentioned above, we assume that the domain Ω is rectangular. Given boundary conditions of u on Ω, the finite difference method (FDM) may be applied to solve the boundary-value problem for u in the following way.

First the domain, Ω, is given an $n \times n$ square grid (*regular mesh*); see Figure 1.3. The *five-point stencil* of a mesh point is defined to be the point itself together with its four neighbors, and for each mesh point (x_i, y_j), the five-point stencil may be used to write a finite difference approximation to $u(x_i, y_j)$. To solve for all the values of u on the mesh points, this leads to

solving the system of equations

$$\mathcal{A}x = b$$

where $\mathcal{A}$ is a five-diagonal $n^2 \times n^2$ sparse matrix of the form

$$\mathcal{A} = \begin{pmatrix} T & -I & & 0 \\ -I & \ddots & \ddots & \\ & \ddots & \ddots & -I \\ 0 & & -I & T \end{pmatrix}, \quad T = \begin{pmatrix} 4 & -1 & & 0 \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ 0 & & -1 & 4 \end{pmatrix}. \tag{1.3}$$

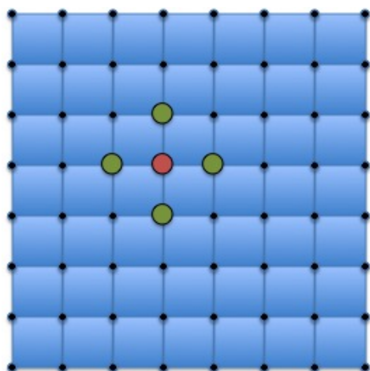


Figure 1.3: Rectangular domain with a regular mesh and five-point stencil.

When the right-hand side of Laplace's equation (1.2) is generalized to any function f, we have what is known as the Poisson's equation. Boundary-value problems involving Poisson's equation may also be solved using a discretization on a five-point stencil.

1.3 Cholesky Factorization

The Cholesky factorization of a matrix $\mathcal{A}$ is the decomposition of a symmetric positive-definite matrix into the product of a lower triangular matrix and its transpose, i.e.

$$\mathcal{A} = \mathcal{L}\mathcal{L}^T$$

where $\mathcal{L}$ is a lower triangular matrix. It is mainly used for finding the solution to a system of equations $\mathcal{A}x = b$: If $\mathcal{A} = \mathcal{L}\mathcal{L}^T$, one can easily solve for x by first solving $\mathcal{L}y = b$ for y using forward substitution and then $\mathcal{L}^T x = y$ for x using backward substitution.

We illustrate the Cholesky factorization by presenting one step of the computation. We say that a variable x_j has been *eliminated* when the j-th step of the Cholesky factorization has been computed. Write an $N \times N$ matrix $\mathcal{A}$ as

$$\mathcal{A} = \begin{pmatrix} A_{11} & A_{21}^T \\ A_{21} & A_{22} \end{pmatrix}.$$

This factorization is easily generalized to block form if we interpret A_{11} as a submatrix of $\mathcal{A}$. Eliminating A_{11} via one step of the Cholesky factorization gives

$$\begin{aligned}\mathcal{A} &= \begin{pmatrix} L_{11} & \\ L_{21} & I \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T \\ & \bar{A}_{22} \end{pmatrix} \\ &= \begin{pmatrix} L_{11} & \\ L_{21} & I \end{pmatrix} \begin{pmatrix} I & \\ & \bar{A}_{22} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T \\ & I \end{pmatrix}\end{aligned}$$

where $A_{11} = L_{11}L_{11}^T$, $L_{21} = A_{21}L_{11}^{-T}$, and $\bar{A}_{22} = A_{22} - A_{21}A_{11}^{-1}A_{21}^T$. The submatrix $\bar{A}_{22}$ is what is known as the *Schur complement* and is what remains to be factored. If $\bar{A}_{22}$ is then factored as $L_{22}L_{22}^T$, then

$$\mathcal{A} = \begin{pmatrix} L_{11} & \\ L_{21} & L_{22} \end{pmatrix} \begin{pmatrix} L_{11}^T & L_{21}^T \\ & L_{22}^T \end{pmatrix}.$$

Note that since $A_{21}A_{11}^{-1}A_{21}^T = (A_{21}L_{11}^{-T})(A_{21}L_{11}^{-T})^T$, we can interpret the formation of the Schur complement as subtracting an *outer-product update* from the original matrix.

1.4 Existing Algorithms and Methods

In general, there are two types of linear system solvers: *direct methods* and *iterative methods*. Iterative methods can be designed to take advantage of the sparsity of the matrix, and they also require less storage. However, theyf can be slow to converge or diverge without the use of effective preconditioners. On the other hand, direct methods are reliable and are efficient when multiple solves are required. However, they can be very expensive due to the generation of fill-in (loss of sparsity) but can be designed to take advantage of special structures to decrease costs.

Depending on the algorithm, there are very different complexities for computing the factorizations. Some iterative methods, such as multi-grid, may cost as little as $O(n)$ for certain PDEs. In the case of direct methods, for a 2D regular grid of size $n \times n$, a basic Cholesky factorization has a complexity of $O(N^3) = O(n^6)$. When column-wise/row-wise mesh orderings are first performed, the complexity may be reduced to $O(N^2) = O(n^4)$. In fact, there exist orderings, reducing fill-in, which have complexity of $O(N^{3/2}) = O(n^3)$ for 2D problems and $O(n^6)$ for 3D problems. Well known methods such as the multifrontal method can then be applied to these large, sparse matrices to further reduce costs.

Our algorithm is a direct method which combines nested dissection with a structured multifrontal method. The matrix elements are preordered using nested dissection and the elimination process is organized via the multifrontal method. The multifrontal method is structured in the sense that it takes advantage of the structures that can be found in the *frontal* and *update* matrices (intermediate dense matrices defined in Chapter 3). Nested dissection was first introduced by George in 1973 and is a divide and conquer heuristic based on graph partitioning for solving sparse SPD systems of equations. It involves the following steps:

1. Form an undirected graph where the vertices represent the variables in the matrix and each edge represents a nonzero matrix entry.

2. Recursively partition the graph into two subgraphs using separators (small subset of vertices such that upon removal, graph will be partitioned into disjoint subgraphs). The fill-in is then at most the size of separator squared.

3. Vertices of matrix are then ordered so that the vertices in the two subgraphs are ordered before their parent separator. This helps determine the order of elimination when later performing the Cholesky factorization.

The multifrontal method was first introduced by Duff and Reid in 1983. A detailed and exhaustive description of the multifrontal method by J. W. H. Liu may be found in [64]. The multifrontal method was a huge advancement in direct methods for solving sparse matrix equations and is widely used in many large-scale problems such as finite element problems [5, 10, 77], computational fluid dynamics [3], separable optimization problems [26], and semiconductor device simulations [66]. Here, we use a *supernodal* version of the multifrontal method where each node corresponds to the group of elements that make up a separator from the nested dissection method. The factorization of the matrix is then organized by creating an elimination tree where each tree node is one of these supernodes. Sparse matrix factorization now becomes a series of factorizations of small dense blocks.

As seen in the work by Xia and Gu, the multifronal method may be combined with HSS methods to reduce complexity costs. Although the nested dissection method helps to reduce fill-in, there is still a significant amount of fill-in that can occur. Typically, fill-in is handled by approximating the intermediate matrices by structured matrices such as $\mathcal{H}$-matrices [48, 50, 53], quasiseparable matrices [32], semiseparable matrices [20, 23, 24], etc. In particular, when the matrices arise from solving discretized PDEs as described in Section 1.2.1, the off-diagonal blocks of the fill-in have been observed to have small numerical ranks [99]. In [99], Xia combines the multifrontal method with algorithms for HSS matrices for a direct solver with total complexity $O(pn^2)$ where p is a constraint related to the PDE and accuracy in the matrix approximations and n is the mesh size. The dense submatrices in the multifrontal method are approximated by hierarchically semiseparable (HSS) matrices, first introduced by Chandrasekaran, Gu, et al. in [22, 21]. For 2D elliptic equations, the method computes structured approximate factorizations with nearly linear complexity and linear storage. Improvements to this method are made in a more recent paper by Xia [92]. Additionally, this structured multifrontal method may be implemented as a parallel solver. In [90], Wang, Hoop, et al. present a parallel structured multifrontal solver for time-harmonic elastic waves in 3D anisotropic media.

In this dissertation, we present an algorithm along the same lines as [99] with two major improvements.

1. Storage of the frontal/update matrices: The frontal/update matrices can be seen to be made up of three components - a sparse matrix, sum of low-rank updates, and small

dense blocks. We preserve the structure by storing the low-rank updates and small dense blocks as vectors in a stack. Pieces of the frontal matrix are then reconstructed as needed. This helps cut down on memory and operation costs.

- In Xia's Superfast Multifrontal Method [99], the dense intermediate blocks are stored as HSS matrices, and all subsequence operations (dense factorization, block permutation, splitting, merging) on these dense blocks involve HSS algorithms. That is, all operations take in an HSS matrix and output the matrix in HSS form. However, because the HSS form is quite involved, there arise many difficulties and complications in preserving this HSS format during implementation.
- In the more recent paper [92], Xia simplifies the previous method by only storing some (frontal matrices) of the dense submatrices in HSS form and the rest (update matrices) as dense matrices. This change helps simplify the implementation since some of the HSS methods may be replaced by their non-structured versions.

2. Usage of a new algorithm for HSS Cholesky factorization; see Chapter 4: In our new algorithm for computing the HSS Cholesky factorization, we organize the factorization so that there is no need to compute Schur complements. This helps to cuts down on operation costs. Moreover, we use a randomization technique [67, 62, 56] for compression to speed up computations. Unlike some previous methods, compression and factorization occur at the same time. Thus, the method does not require the input matrix to be of HSS form and integrates well with our first improvement.

    - In [99], Xia uses a generalized Cholesky factorization. Input and output are in HSS format and Schur complement computations are done via HSS computations.
    - In [92], partial ULV-type factorizations are used.

1.5 Organization of Part I of Dissertation

This part of the dissertation is organized as follows. In Chapter 2, we introduce the multifrontal method. We give brief descriptions of the nested dissection and symbolic factorization steps, and then give an outline of the method. The update and frontal matrices are defined in detail.

In Chapter 3, we discuss the structure of the frontal and update matrices. Different storage and computation methods for these dense submatrices are discussed, and our new storage framework is presented. Examples are used to illustrate the structure. We also present our main algorithm which we call the *partial left-looking multifrontal structured method*.

Finally, in the Chapter 4, we present our new algorithm for computing a generalized HSS factorization. The HSS matrix structure is defined, and we show how to incorporate this algorithm into our new structured multifrontal method framework.

Chapter 2

Multifrontal Method

2.1 Introduction to the Multifrontal Method

In this section we review the multifrontal method and present the framework of our algorithm. Here, we describe the three main components of our algorithm: (1) Nested Dissection (2) Symbolic factorization (3) Cholesky factorization. One of our main contributions involves the storage of the *frontal* and *update* matrices, defined in Section 2.3.2.

2.2 Preprocessing Steps

Before applying the multifrontal method to compute the factorization, we use two well known steps which greatly improve the algorithm. The first involves reordering the matrix, and the second involves symbolically performing the factorization.

2.2.1 Step 1: Nested dissection ordering

As previously mentioned, when solving a sparse symmetric positive-definite (SPD) system, $\mathcal{A}x = b$, direct methods are often expensive due to the amount of *fill-in* (nonzero matrix entries created in the Cholesky decomposition that are not part of $\mathcal{A}$) that is generated. This fill-in results in a loss of sparsity. In the context of discretizing PDEs, $\mathcal{A}$ is associated to a mesh on its domain. It turns out that reordering the mesh points or equivalently, reordering $\mathcal{A}$, can reduce fill-in and improve the performance of direct methods.

Nested dissection was first proposed by George to solve systems of equations defined on square meshes. For a mesh of size $n \times n$, the algorithm requires $O(n^2 log_2 n)$ memory and $O(n^3)$ time [39]. It has been shown [39], that for the problems we are concerned with, given any ordering of the mesh nodes, the Cholesky factorization will require at least such a complexity. Thus, nested dissection gives an optimal ordering. In [63], the method is generalized to work with any linear system on a planar/almost-planar graph with the same computational costs. The method is further improved in [82]. In our Matlab package, we use

the METIS package for the nested dissection step by the Karypis Lab at the University of Minnesota [60].

2.2.1.1 Outline

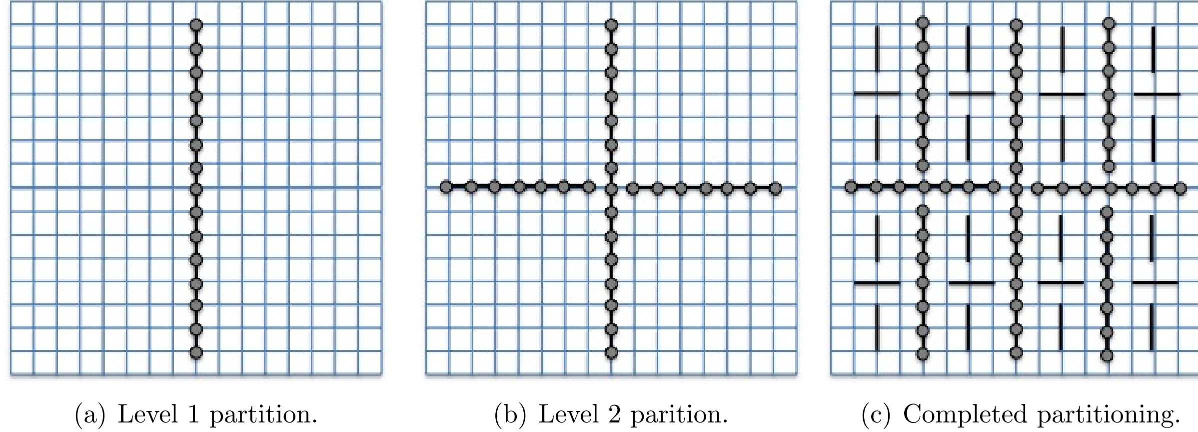
Although our algorithm is a general solver, we describe the nested dissection method for square meshes. In this dissertation, we use the following example:

Example 2.2.1. *Consider the square $(0, n) \times (0, n)$ of size $n \times n$. Let $\mathbb{M}$ be the $n \times n$ mesh formed by partitioning the square in n^2 pieces, where each element is a square of size 1×1. The finite element matrix $\mathcal{A}$ corresponding to this mesh is then of size $n^2 \times n^2$. Each variable of $\mathcal{A}$ corresponds to a mesh node of $\mathbb{M}$, and $\mathcal{A}_{ij} \neq 0$ if and only if the variables x_i and x_j belong to the same element. See Figure 1.3 for an example.*

The nested dissection method is outlined as follows:

1. Choose a set of nodes of $\mathbb{M}$ such that upon removal, the remaining nodes are divided into two disjoint sets; i.e., no element in the mesh contains nodes from both sets. This set of nodes is called the *separator*; see Figure 2.1(a) for an example. In the general graph partitioner METIS, the separators are found using a multilevel recursive bisection [60].

2. Repeat this process recursively on the two disjoint pieces until the separators are below a certain size or a certain level of subdivisions is reached; see Figure 2.1(c).



(a) Level 1 partition. (b) Level 2 parition. (c) Completed partitioning.

Figure 2.1: Nested dissection.

George proved the following result about nested dissection.

Theorem 2.2.2. *[39] The number of multiplicative operations required to factorize $\mathcal{A}$ is $O(n^3)$, and the number of nonzero off-diagonal components in $\mathcal{L}$ (storage required) is $O(n^2 log_2 n)$.*

2.2.1.2 Separator ordering

Once the separators have been found, they may be organized using a binary tree structure, $\mathcal{T}$, where each node corresponds to a separator. In this dissertation and our algorithm, the binary tree is given a postordering. Later, we show how this tree integrates with the supernodal multifrontal method. Essentially, each separator will correspond to a supernode, and the tree determines the elimination order. The *levels* of the tree are are ordered top-down, with the first separator to divide $\mathbb{M}$ (*top-level separator*) at level 1. See Figure 3.4.

Remark 2.2.3. *In this dissertation, the separators (nodes) in each level of the separator tree correspond to the same level of division in the mesh and are either all horizontal or vertical. In some literature [39, 63], the separators in a level involve cross-shaped cuts in the same level of division, splitting the domain into four disjoint pieces.*

It is easy to show the following results [93] assuming that $n = 2^k - 1$ for a positive integer k for an $n \times n$ mesh. We use this lemma in the proofs of Theorems 3.3.1 and 3.3.4.

Lemma 2.2.4. *For an $n \times n$ mesh with $n = 2^k - 1$ and positive integer $k > 0$,*

1. *All the separators in the same level have the name number of nodes.*
2. *The total number of levels is $l = 2log_2(n+1) - 1$, and for general n, there are $O(log_2 n)$ levels.*

The matrix can now be reordered based on the organization of the separators in the postordered binary tree, $\mathcal{T}$. We order the variables of the matrix so that, the set of variables corresponding to each separator are ordered before the variables in the separators with higher numbering $\mathcal{T}$. That is, the variables in a separator S_i (separator corresponding to node i in $\mathcal{T}$) come before the variables in separator S_j (separator corresponding to node j in $\mathcal{T}$) in the reordered matrix $\mathcal{A}$. Thus, the variables in S_i are eliminated before S_j. See Figure 2.4 for more details.

2.2.1.3 Ordering of nodes within separators

In Section 2.2.1.2, we described how the sets of variables in the separators are ordered but not how the variables within each separator are ordered. We can think of reordering the variables in $\mathcal{A}$ as two levels of ordering: a courser level (order of the sets of separator variables) and a finer level (ordering of the variables within the separator). It has been shown in [93], that because of the physical nature of the problems described in Section 1.2.1, the intermediate dense matrices in the multifrontal method will possess the qualities of the HSS structure (off-diagonal blocks have low numerical rank) depending on how the nodes within a separator are ordered. See [99] for more details.

2.2.1.4 Node elimination and fill-in

Once the matrix has been reordered, one can compute the Cholesky factorization, $\mathcal{A} = \mathcal{L}\mathcal{L}^T$, by eliminating variables in the order of the new matrix. In our algorithm, the factorization is organized via the multifrontal method, and the separator variables are eliminated according to the postorder binary tree we described above. Recall that eliminating variable x_i refers to computing the ith step of the Cholesky factorization.

We now illustrate the effects of eliminating the set of variables in a separator and how the matrix is "filled-in" as a result. First we outline the *block Cholesky factorization* algorithm. Assume the matrix $\mathcal{A}$ is split up into K^2 blocks of sizes $m_1, \ldots, m_K$ (the (i,j)-th block is of size $m_i \times m_j$). The block Cholesky factorization is the same as the Cholesky algorithm except one eliminates entire submatrices at once. See Algorithm 1.

Algorithm 1 Block Cholesky Factorization

Require: SPD matrix $\mathcal{A}$ partitioned into K^2 blocks; block sizes $m_1, \ldots, m_K$.
Ensure: Cholesky factor $\mathcal{L}$ where $\mathcal{A} = \mathcal{L}\mathcal{L}^T$ with lower-triangular block columns $L_i = \left[L_i^{(1)}; L_i^{(2)}\right]$.

1: Set $A_0 = \mathcal{A}$.
2: **for** $i = 1, \ldots, K-1$ **do**
3: Write A_{i-1} as

$$A_{i-1} = \begin{pmatrix} D_i & B_i^T \\ B_i & \bar{A}_i \end{pmatrix},$$

where D_i is of size $m_i \times m_i$.
4: Compute the following:

- $D_i = L_i^{(1)} L_i^{(1)T}$ where $L_i^{(1)}$ is the Cholesky factor of D_i,
- $L_i^{(2)} = B_i L_i^{(1)-T}$,
- $A_i = \bar{A}_i - L_i^{(2)} L_i^{(2)T}$.

5: **end for**
6: Compute $A_{K-1} = L_K^{(1)} L_K^{(1)T} = L_K L_K^T$.

The Cholesky factor $\mathcal{L}$ of $\mathcal{A}$ is made up of the submatrices, $L_i^{(1)}$ and $L_i^{(2)}$, defined in Algorithm 1 in the following way. Let

$$\mathcal{L} = \begin{pmatrix} L_{11} & & & \\ L_{21} & L_{22} & & \\ \vdots & \vdots & \ddots & \\ L_{K1} & L_{K2} & \ldots & L_{KK} \end{pmatrix}$$

where the i-th block column has m_i columns. Then

$$L_i^{(1)} \equiv L_{ii} \quad \text{and} \quad L_i^{(2)} \equiv \begin{pmatrix} L_{i+1,i} \\ \vdots \\ L_{Ki} \end{pmatrix}.$$

As a result, the nonzero pattern of the ith off-diagonal block column of the resulting Cholesky factor $\mathcal{L}$ corresponds to the nonzero pattern of the corresponding off-diagonal block column B_i in A_{i-1} since $L_i^{(2)} = B_i L_i^{(1)-T}$. Moreover, the nonzero pattern in A_i is "updated" by the nonzero pattern of B_i since $A_i = \bar{A}_i - L_i^{(2)} L_i^{(2)T}$. Although the original matrix may be sparse, subsequent Schur complements A_i may become dense, and the Cholesky factor $\mathcal{L}$ may no longer be sparse. The nonzeros created are commonly referred to as the *fill-in*.

One step of the block factorization process is shown in Figure 2.2 to illustrate the fill-in process of a sparse matrix. Figure 2.2(a) shows the nonzero pattern of the sparse matrix $\mathcal{A} = \begin{pmatrix} D_1 & B_1^T \\ B_1 & \bar{A}_1 \end{pmatrix}$ to be factorized. Figures 2.2 (b) and (c) show the nonzero patterns of B_1 and $L_1^{(2)}$, respectively. Notice how they have the same nonzero rows. Finally, Figures 2.2 (d) and (e) show the nonzero patterns of $L_1^{(2)} L_1^{(2)T}$ and A_1, respectively, showing their very similar sparsity patterns.

We use the following terminology in this dissertation to describe changes to the variables during elimination as in [93].

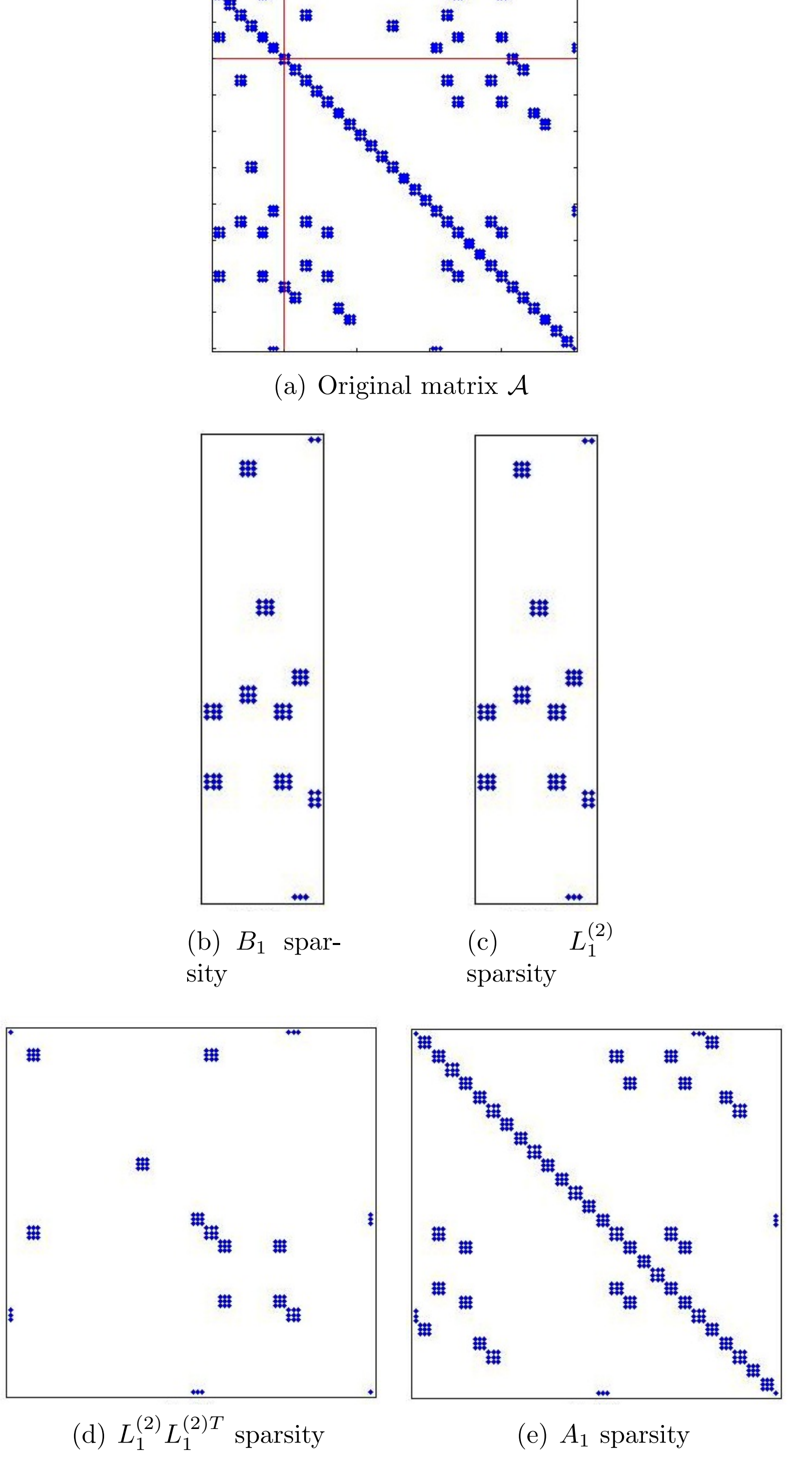
Definition 2.2.5. *For $i = 1, \ldots, K$, consider the matrix A_i defined in Algorithm 1 and variables x_j and x_k for $j, k > i$. Then, x_j and x_k are* ***connected*** *if the corresponding off-diagonal component in A_i is nonzero.*

As seen above, in the i-th step of the block Cholesky factorization x_j and x_k will be connected if they both are connected to the same variable x_i in D_i. This is because the (j, k)-th component in $L_i^{(2)} L_i^{(2)T}$ will be nonzero (assuming that it does not cancel with the corresponding component in $\bar{A}_i$. This observation is summarized in the following theorem.

Theorem 2.2.6. *[39] If the variable x_i is connected to variables x_k for $k > i$, then upon elimination of x_i, all such x_k will be connected pairwise.*

For a finite element system $\mathcal{A}x = b$, recall that $\mathcal{A}_{ij} \neq 0$ if and only x_i and x_j belong to the same element in the corresponding mesh. In the context of a mesh, if x_j and x_k belong to the same element as the nodes in separator i, then upon elimination of the i-th separator, they will become connected as shown in Figure 2.3.

In the case of nested dissection, each of the blocks in $\mathcal{A}$ correspond to a separator. Consider the example where we have seven nodes in the separator tree $\mathcal{T}$. The corresponding matrix structure is shown in Figure 2.4. Following the block Cholesky factorization with the blocks as the separators and eliminating them in the order of the nodes in $\mathcal{T}$, we see that eliminating separators 1 and 2 connects the variables in separators 3 and 7. Similarly,



(a) Original matrix $\mathcal{A}$

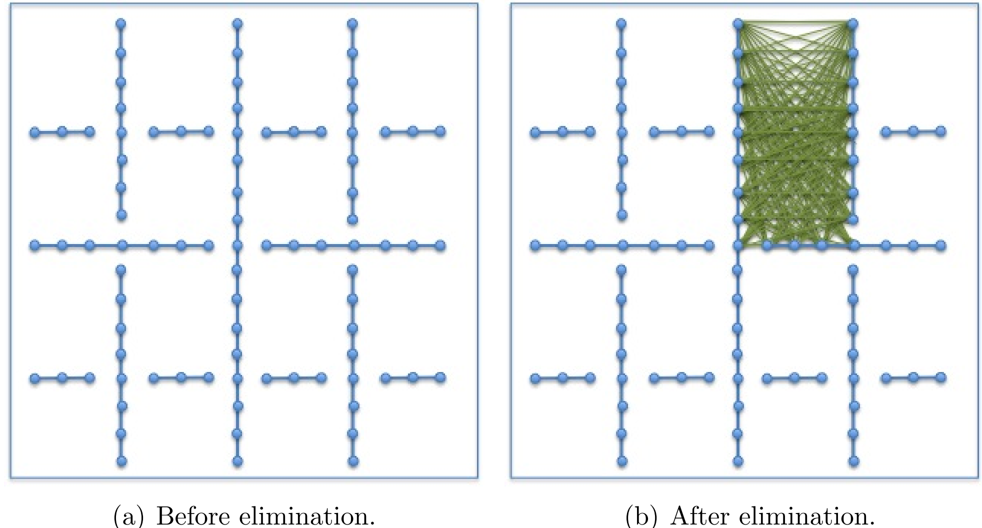
(b) B_1 sparsity

(c) $L_1^{(2)}$ sparsity

(d) $L_1^{(2)}L_1^{(2)T}$ sparsity

(e) A_1 sparsity

Figure 2.2: One step of block Cholesky.



(a) Before elimination. (b) After elimination.

Figure 2.3: Connections during the factorization.

eliminating separators 4 and 5 connects the variables in separators 6 and 7. Finally, eliminating the separators 3 and 6 will connect the variables in separator 7. Thus, for a matrix reordered using nested dissection, upon elimination of node i, only the ancestors of i in $\mathcal{T}$ are affected. In the next section, we show how to take advantage of this elimination pattern using the multifrontal method.

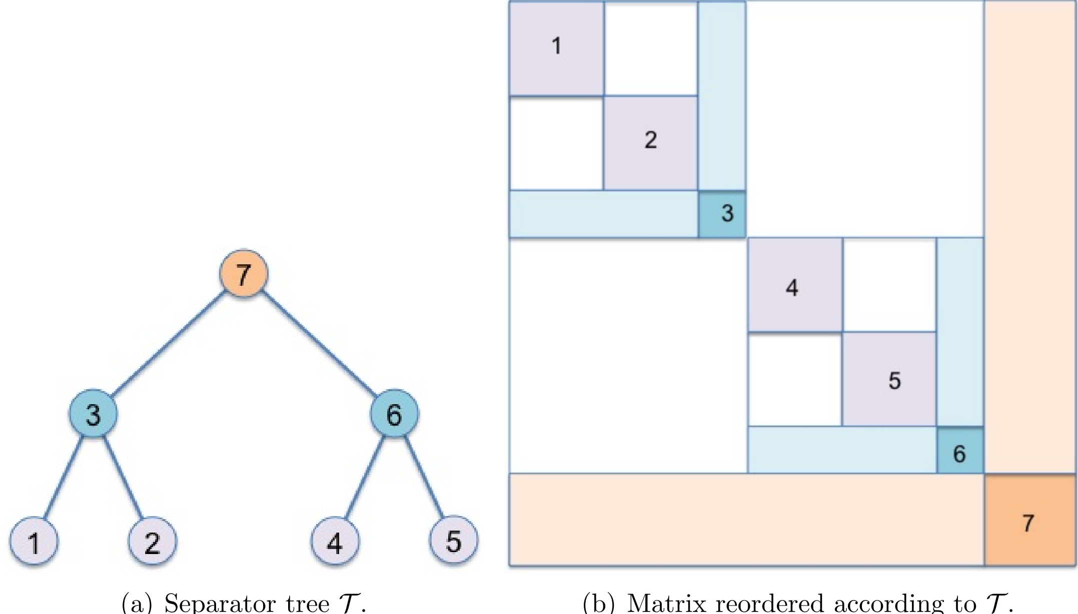
2.2.2 Step 2: Symbolic factorization

Before the factorization, we apply a symbolic factorization stage to predict the nonzero pattern in $\mathcal{L}$. This allows us to create suitable data structures for the factorization and efficiently implement the sparse factorization algorithm. No computations are done and for each separator, the indices of the nonzero rows in the corresponding block column of $\mathcal{L}$ are predicted and stored for later usage.

2.3 Step 3: Multifrontal Method

As seen in the previous section, eliminating the i-th separator only affects its ancestors in $\mathcal{T}$. The multifrontal method can be used to organize the entire factorization process to take advantage of this. The basic idea is to organize the factorization into factorizations of smaller dense matrices called *frontal matrices* which contain only the indices that are affected by the elimination of an appropriate separator.

The multifrontal method is a well known, significant direct method for solving sparse matrix problems. It was first developed for solving indefinite sparse symmetric linear equations and is a generalization of the *frontal* method by Irons [59]. In our algorithm, we use



(a) Separator tree $\mathcal{T}$. (b) Matrix reordered according to $\mathcal{T}$.

Figure 2.4: Matrix reordered with nested dissection.

the *supernodal* version of the multifrontal method. This is a generalization of multifrontal method which works with groups of nodes called *supernodes* rather than individual nodes. In our algorithm, the supernodes correspond to the separators from the nested dissection step. In this section we describe the *supernodal multifrontal method*. A detailed description of both methods can be found in the paper by Liu [64].

Remark 2.3.1. *From this point on, we refer to the nodes in a tree as supernodes, and the variables within a supernode as nodes.*

We first revisit the block Cholesky factorization in Algorithm 1. Given an $N \times N$ matrix $\mathcal{A}$ with Cholesky factor $\mathcal{L}$, partition them into K^2 blocks in the following way:

$$\mathcal{A} = \begin{pmatrix} A_{11} & \dots & A_{K1}^T \\ \vdots & \ddots & \vdots \\ A_{K1} & \dots & A_{KK} \end{pmatrix} \quad \text{and} \quad \mathcal{L} = \begin{pmatrix} L_{11} & & \\ \vdots & \ddots & \\ L_{K1} & \dots & L_{KK} \end{pmatrix}. \tag{2.1}$$

In the i-th step of the block Cholesky algorithm, we wish to factor the i-th Schur complement A_i. Thus, one of the main components of the block Cholesky algorithm is to form and factor A_i efficiently. To do this, we explore the underlying structure of A_i. For ease of notation, we define the *block outer product update* as the block form of the outer product update described in Section 1.3.

Definition 2.3.2. *In the i-th step of the block Cholesky algorithm, the* ***block outer product update*** *from the j-th block column of $\mathcal{L}$ is*

$$\tilde{U}_j^{(i)} = \begin{pmatrix} L_{ij} \\ \vdots \\ L_{nj} \end{pmatrix} \begin{pmatrix} L_{ij}^T & \dots & L_{nj}^T \end{pmatrix},$$

and the j-th block outer product update is defined to be

$$\tilde{U}_j := \tilde{U}_j^{(j)}.$$

Recall that the Schur complement A_i is defined inductively where $A_i = \bar{A}_i - \tilde{U}_i^{(i)}$ and $\bar{A}_i$ is a subblock of A_{i-1}. Using induction, it is easy to show that

$$A_i = \begin{pmatrix} A_{ii} & \dots & A_{Ki}^T \\ \vdots & \ddots & \vdots \\ A_{Ki} & \dots & A_{KK} \end{pmatrix} - \sum_{j=1}^{i} \tilde{U}_j^{(i)}. \tag{2.2}$$

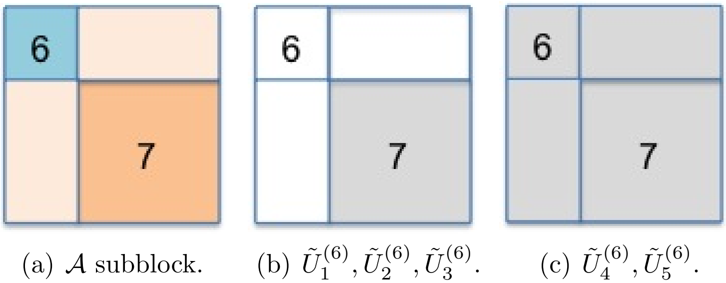
In other words, the i-th Schur complement is made up of the appropriate subblock of the original matrix $\mathcal{A}$ minus the first i block outer product updates.

It turns out that for sparse matrices or matrices with a sparse block pattern, it is not necessary to include all i block outer product updates in the formation of the Schur complement since the computation of $L_i^{(1)}$ and $L_i^{(2)}$ only involves the first block row and column of A_i. The multifrontal method is a systematic way to take advantage of this and remove these unnecessary steps. In the case of where nested dissection is used, each block corresponds to a separator (see Figure 2.4), and the matrix has a block sparse pattern as seen in Figure 2.4. As an example, consider applying the sixth step of the block Cholesky factorization. Then

$$A_6 = \begin{pmatrix} A_{66} & A_{76}^T \\ A_{76} & A_{77} \end{pmatrix} - \tilde{U}_1^{(6)} - \tilde{U}_2^{(6)} - \tilde{U}_3^{(6)} - \tilde{U}_4^{(6)} - \tilde{U}_5^{(6)},$$

where the sparsity pattern can be seen in Figure 2.5. Computing $L_6^{(1)}$ and $L_6^{(2)}$ only involves the first block row/column of A_6 which $\tilde{U}_1^{(6)}$, $\tilde{U}_2^{(6)}$, and $\tilde{U}_3^{(6)}$ do not contribute to. Thus, it is only necessary to include $\tilde{U}_4^{(6)}$ and $\tilde{U}_5^{(6)}$ in the computation of the Schur complement to correctly calculate L_i.

The block Cholesky factorization may be reorganized to work only with partial Schur complements involving the block outer products that contribute to the first block row/column of the Schur complement. This reorganization is the main idea of the multifrontal method. In what follows, we assume that we are working with a matrix that has been previously reordered by nested dissection.



(a) $\mathcal{A}$ subblock. (b) $\tilde{U}_1^{(6)}, \tilde{U}_2^{(6)}, \tilde{U}_3^{(6)}$. (c) $\tilde{U}_4^{(6)}, \tilde{U}_5^{(6)}$.

Figure 2.5: Nonzero structure of the Schur complement components.

2.3.1 Elimination tree

To understand the multifrontal method, we first define something called the *elimination tree* which organizes the update contributions to the Schur complement. The elimination tree was defined for the sparse symmetric case by Schreiber [81]. Here, we generalize the definition for the block matrix $\mathcal{A}$ and its Cholesky factor $\mathcal{L}$ in (2.1). A *supernode* [4, 83] is defined to be a set of consecutive variables in $\mathcal{A}$.

Definition 2.3.3. *[65, 64, 81] The* ***elimination tree****, $\mathcal{T}(\mathcal{A})$, of a $K \times K$ block matrix $\mathcal{A}$ is the tree with K supernodes $\{1, \ldots, K\}$. Each supernode of the tree corresponds to the set of variables in a block, and supernode p is the parent of j if and only if*

$$p = \min\{i > j : L_{ij} \neq 0\}.$$

As an example, consider the matrices in Figure 2.6 which show the block sparsity patterns for a matrix $\mathcal{A}$, its Cholesky factor $\mathcal{L}$, and the corresponding elimination tree $\mathcal{T}(\mathcal{A})$. Assume that no zeros are introduced through cancellation during the factorization.

When the block matrix is irreducible (can not be permuted to a block diagonal matrix), the elimination tree is a tree, but if the matrix is reducible, then the structure is a forest [64]. In this dissertation, unless specified otherwise, we assume that the matrix $\mathcal{A}$ is reducible. It turns out that the nested dissection ordering tree $\mathcal{T}$ is related to the elimination tree of the corresponding matrix $\mathcal{T}(\mathcal{A})$.

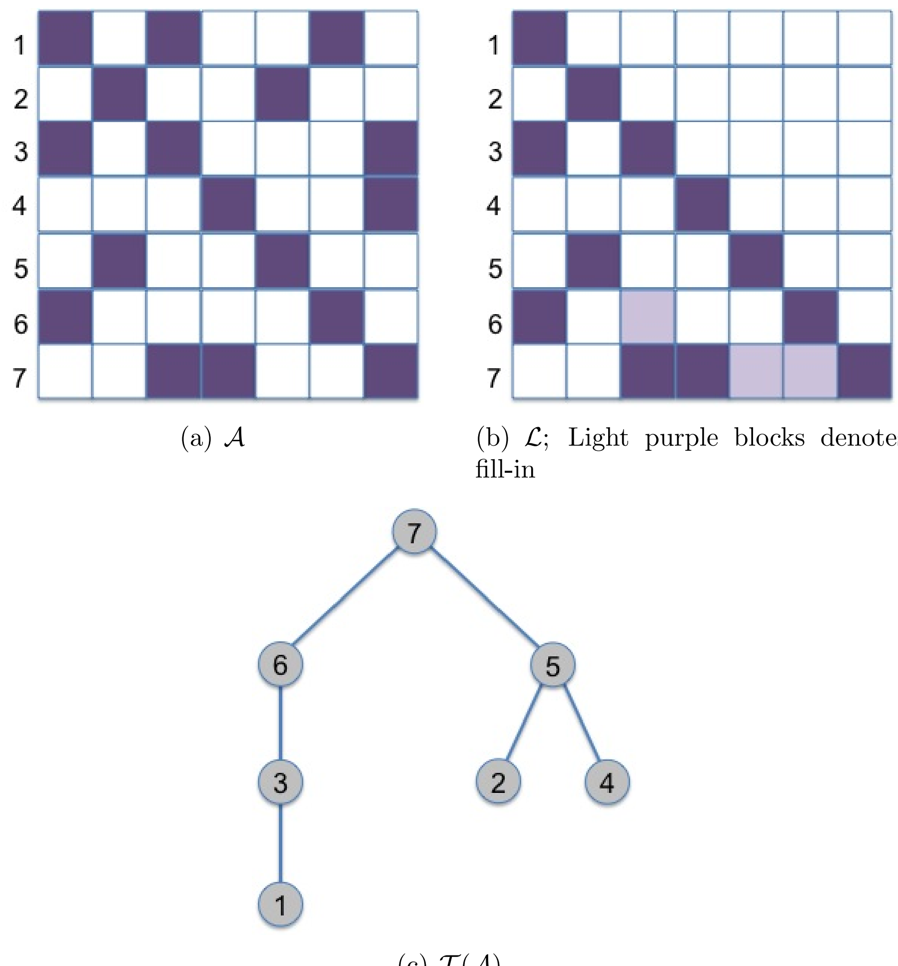
Theorem 2.3.4. *For a matrix $\mathcal{A}$ ordered with nested dissection, its postordered nested dissection ordering tree $\mathcal{T}$ and elimination tree $\mathcal{T}(\mathcal{A})$ are the same.*

Proof. In $\mathcal{T}(\mathcal{A})$, supernode p is the parent of j if and only if

$$p = \min\{i > j : L_{ij} \neq 0\}.$$

By definition of the postordered nested dissection tree, p is also the parent of j in $\mathcal{T}$. □

In the following discussions, we assume that the elimination tree is a balanced binary tree. Additionally, the following two theorems have been proven regarding the elimination tree for single nodes. We state the equivalent results for the case of supernodes.



(a) $\mathcal{A}$

(b) $\mathcal{L}$; Light purple blocks denotes fill-in

(c) $\mathcal{T}(\mathcal{A})$

Figure 2.6: Elimination Tree.

Theorem 2.3.5. *[64] If a supernode k is a descendant of j in the elimination tree, then the nonzero block structure of the block matrix $\left(L_{jk}^T, \ldots, L_{nk}^T\right)^T$ is contained in the nonzero block structure of $\left(L_{jj}^T, \ldots, L_{nj}^T\right)^T$*

Theorem 2.3.6. *[81] If $L_{jk} \neq 0$ and $k < j$, then the supernode k is a descendent of j in the elimination tree.*

These theorems imply that for the computation of the j-th block column of the Cholesky factor $\mathcal{L}$, since the block sparsity pattern is the same as the corresponding block column in the Schur complement matrix A_i, only the block outer product updates from the descendants of j in $\mathcal{T}(\mathcal{A})$ are necessary. This was observed in the previous subsection in Figure 2.5.

2.3.2 Frontal and update matrices

We are now ready to define the *frontal* and *update* matrices. The idea of the j-th frontal matrix is to isolate only what is needed in the Schur complement A_{j-1} to compute the j-th block column of $\mathcal{L}$. This involves three key changes from the Schur complement:

- Recall that only the block outer product contributions from the descendants of j are necessary in the Schur complement. The frontal matrix contains only these updates. In other words, we form a partial Schur complement.
- Only the j-th block column and row from the original matrix $\mathcal{A}$ are included.
- In the case of the nested dissection and other sparse matrix structures, the first block column of A_{j-1} contains many zero blocks. Thus, the frontal matrix reduces storage by only considering a particular dense submatrix of the partial Schur complement containing only certain nonzero blocks.

The definition of the frontal matrix summarizes these concepts. Let $i_0 := i, i_1, \ldots, i_p$ be the block row subscripts of the nonzero blocks in the i-th block column of $\mathcal{L}$.

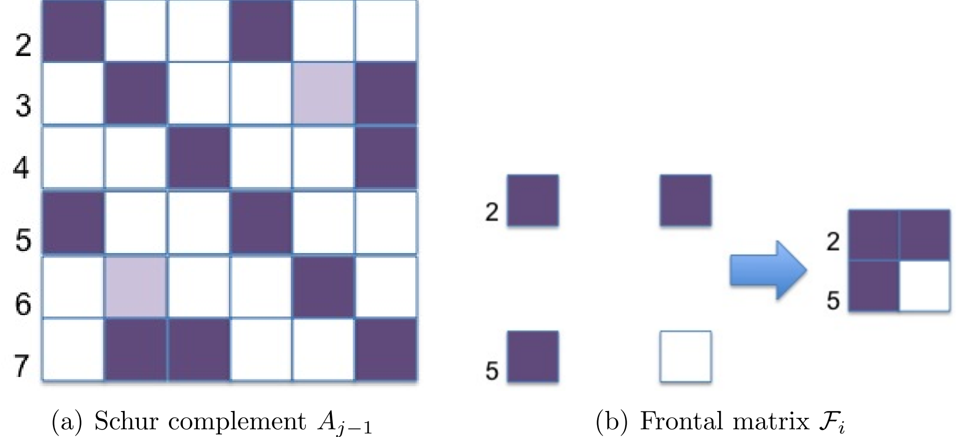
Definition 2.3.7. *The i-th **frontal matrix** $\mathcal{F}_i$ for $\mathcal{A}$ is*

$$\begin{aligned} \mathcal{F}_i &= \begin{pmatrix} A_{ii} & A_{i_1 i}^T & \ldots & A_{i_p i}^T \\ A_{i_1 i} & & & \\ \vdots & & 0 & \\ A_{i_p i} & & & \end{pmatrix} - \sum_{j:\textit{proper descendent of } i} \begin{pmatrix} L_{ij} \\ L_{i_1 j} \\ \vdots \\ L_{i_p j} \end{pmatrix} \begin{pmatrix} L_{ij} & L_{i_1 j}^T & \ldots & L_{i_p j}^T \end{pmatrix} && (2.3) \\ &= F_i^0 - \sum_{j:\textit{proper descendent of } i} \bar{U}_j^{(i)}. && (2.4) \end{aligned}$$

*The **initial frontal matrix** F_i^0 is the first matrix on the right-hand side, and the **update matrix** $\mathcal{U}_i$ is the Schur complement from one step of elimination of $\mathcal{F}_i$ to get the i-th block column of $\mathcal{L}$:*

$$\mathcal{F}_i = \begin{pmatrix} L_{ii} & & & \\ L_{i_1 i} & 1 & & \\ \vdots & & \ddots & \\ L_{i_p i} & & & 1 \end{pmatrix} \begin{pmatrix} L_{ii}^T & L_{i_1 i}^T & \ldots & L_{i_p i}^T \\ & & & \\ & & \mathcal{U}_i & \\ & & & \end{pmatrix}. \qquad (2.5)$$

Figure 2.7 compares the block sparsity pattern between the $(i-1)$-st Schur complement A_{i-1} and the i-th frontal matrix $\mathcal{F}_i$ of the matrix in Figure 2.6(a). Thus, the i-th frontal matrix contains the appropriate contributions from $\mathcal{A}$ and the block outer-product contributions from the proper descendants of i. The i-th update matrix contains the block outer-products from the subtree of $\mathcal{T}(\mathcal{A})$ with root i. It turns out that the i-th frontal matrix can be written as the i-th initial frontal matrix combined with the update matrices from its two children under an operation called *extend-add*. One can think of extend-adding



(a) Schur complement A_{j-1} (b) Frontal matrix $\mathcal{F}_i$

Figure 2.7: Schur complement vs. frontal matrix.

as a generalized matrix addition for matrices of different sizes. The idea of applying the extend-add operator, $\leftrightarrow\hspace{-0.9em}\updownarrow$, to two matrices is to add the two matrices after padding them with columns and rows of zeros to align indices.

To see an example of the extend-add operation, consider the two matrices

$$R = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad S = \begin{pmatrix} e & f \\ g & h \end{pmatrix}$$

which correspond to the indices $\{2, 4\}$ and $\{3, 4\}$, respectively. Then $R \mathbin{\leftrightarrow\hspace{-0.9em}\updownarrow} S$ is the following matrix corresponding to the indices $\{2, 3, 4\}$:

$$R \mathbin{\leftrightarrow\hspace{-0.9em}\updownarrow} S = \begin{pmatrix} a & 0 & b \\ 0 & 0 & 0 \\ c & 0 & d \end{pmatrix} + \begin{pmatrix} 0 & 0 & 0 \\ 0 & e & f \\ 0 & g & h \end{pmatrix} = \begin{pmatrix} a & 0 & b \\ 0 & e & f \\ c & g & d+h \end{pmatrix}.$$

We may now present the following theorem relating the frontal matrix with the update matrices of its children. In this formulation, frontal matrices are formed according to the postordering of the elimination tree.

Theorem 2.3.8. *[64] Let the supernodes c_1 and c_2 be the children of i in $\mathcal{T}(\mathcal{A})$. Then*

$$\mathcal{F}_i = \begin{pmatrix} A_{ii} & A_{i_1 i}^T & \dots & A_{i_p i}^T \\ A_{i_1 i} & & & \\ \vdots & & 0 & \\ A_{i_p i} & & & \end{pmatrix} \mathbin{\leftrightarrow\hspace{-0.9em}\updownarrow} \mathcal{U}_{c_1} \mathbin{\leftrightarrow\hspace{-0.9em}\updownarrow} \mathcal{U}_{c_2}. \tag{2.6}$$

In our algorithm, we use an equivalent definition of the frontal matrix using a combination of the last two representations of the frontal matrix. For an elimination tree of height l, fix a level $s \in \{0, \ldots, l-1\}$. We call this level the *switching level*. See Figure 3.1. This new definition represents the frontal matrix as the sum of the initial frontal matrix, block outer products updates from the proper descendants up to the switching level, and the update matrices of the supernodes at the switching level. In our algorithm, it turns out that the block outer product updates are of low-rank, and the update matrices have a sparse block structure. The definition is given in (3.3).

2.3.2.1 Storage of the update matrices

The update matrices are traditionally stored as dense matrices in a stack structure. That is, for the i-th separator where $i < K$ and K is the number of separators, one can update the stack with the update matrix $\mathcal{U}_i$ as follows. Assume that r is the parent of i.

1.
   - If i is a leaf node: Form the frontal matrix from the appropriate subblock of $\mathcal{A}$.
   - If i is a leaf node: Pop the frontal matrix from the stack.
2. Eliminate i to get $\mathcal{U}_i$.
3.
   - If i is the left child of r: Partially form the frontal matrix of the parent r with $\mathcal{F}_r = \mathcal{F}_r^0 \updownarrow\!\!\!\!\leftrightarrow \mathcal{U}_i$.
   - If i is the right child of r: Pop the partially formed $\mathcal{F}_r$ from the stack and extend-add $\mathcal{U}_i$ to it to complete the formation.
4. Push $\mathcal{F}_r$ back onto the stack.

This method outlined above is what is done in the case of [93]. In our algorithm, we continue to store the update matrix information in a stack. However, we do not explicitly form the update matrices. The details are explained in Chapter 3.

2.3.3 Multifrontal method with dense update block storage

The version of the multifrontal method which stores the update matrices as dense matrices in a stack is given in Algorithm 2. The following theorem regarding the complexity and memory requirements of this algorithm was proven in [93]. Notice that Algorithm 2 is the complexity of the nested dissection method.

Theorem 2.3.9. *Algorithm 2 requires $O(n^3)$ multiplicative operations and has $O(n^2 log_2 n)$ nonzeros in the Cholesky factor. If the elimination tree is full and balanced, then the update matrix stack requires $O(n^2)$ storage.*

Algorithm 2 Multifrontal method with dense update block storage.

Require: SPD matrix $\mathcal{A}$ partitioned into blocks; elimination tree $\mathcal{T}(\mathcal{A})$ with K nodes.
Ensure: Cholesky factor $\mathcal{L}$ where $\mathcal{A} = \mathcal{L}\mathcal{L}^T$ with block columns L_i.

1: **for** $i = 1, \ldots, K-1$ **do**
2:     **if** i is a leaf node **then**
3:         Form the frontal matrix $\mathcal{F}_i$ with the appropriate subblock of $\mathcal{A}$.
4:     **else**
5:         Pop the frontal matrix from the stack.
6:     **end if**
7:     Perform one step of a block Cholesky algorithm to $\mathcal{F}_i$ to get $\mathcal{U}_i$ and L_i.
8:     **if** i is a left child **then**
9:         Form $F_i = \mathcal{F}_r^0 \mathbin{\updownarrow\mkern-14mu\leftrightarrow} \mathcal{U}_i$.
10:     **else**
11:         Pop F_i off the stack.
12:         Form $F_i = F_i \mathbin{\updownarrow\mkern-14mu\leftrightarrow} \mathcal{U}_i$.
13:     **end if**
14:     Push F_i onto the stack.
15: **end for**
16: Pop $\mathcal{F}_K$ off the stack and factorize to get L_K.

Step 7 of Algorithm 2 requires performing one step of a block Cholesky factorization. Many different structured algorithms may be used. In particular, we use a new algorithm for factoring HSS matrices which we describe in Chapter 4.

Chapter 3

Partial Left-Looking Multifrontal Method

In this chapter, we describe and motivate our main algorithm which we call the *Partial Left-Looking Multifrontal Method*. In [95], Xia proposes a randomized direct solver for general large, sparse matrix equations based on integrating randomization into a structured multifrontal method such as the one proposed in [92]. As we will show, the frontal and update matrices have a certain low-rank structure which can be taken advantage of when they are passed to their parent in the elimination tree. One can save memory and computation time by passing these structures to the parents rather than losing them by passing the update/frontal matrices as dense matrices. Xia accomplishes this by using randomized HSS construction and factorization techniques. Moreover, Xia only partially factorizes the frontal matrix in each step. In our algorithm, we adopt the same overarching principles of passing the structures of the update/dense matrices up to the parents and only partially factorizing the frontal matrix. However, we carry out these concepts in a different manner and tailor our method to symmetric positive definite matrices while Xia's algorithm applies to general sparse matrices. Moreover, our factorization scheme (see Chapter 4) is *Schur monotonic* and ensures a structured SPD factorization, leading to a guaranteed factorization of the entire matrix. Lastly, in our dense matrix factorizations, our compression (low-rank approximation) scheme is more efficient than Xia's. In our method, we use a randomized trucated SVD algorithm for compression (see Algorithm 5) rather than the randomized interpolative decomposition used in [95]. Our compression scheme gives a better (smaller) rank estimate, making the algorithm more efficient.

An outline of this chapter is as follows. We first describe the underlying structure of the frontal and update matrices which we take advantage of in our algorithm. Next we present our algorithm which is a version of the multifrontal method with a left-looking approach. Finally, we describe the memory and computational costs of our method along with some numerical results. First we review the terminology used in this chapter.

3.0.4 Terminology

Let $\mathcal{A}$ be the $N \times N$ matrix we wish to factorize. Assume that we have already applied the nested dissection method to $\mathcal{A}$, and we have the elimination tree $\mathcal{T}(\mathcal{A})$ with K supernodes. Although it will not always be the case, for simplicity of discussion, we assume that $\mathcal{A}$ is reducible and thus a tree rather than a forest. For each supernode i, define the *frontal matrix index set* to be the supernode indices of the nonzero blocks in the i-th block column of the Cholesky factor $\mathcal{L}$. We use the following notation:

$$\mathcal{I}_i = \{i := i_0, i_1, \ldots, i_p\} \quad \mathcal{J}_i = \{i_1, \ldots, i_p\}\,. \tag{3.1}$$

Suppose $\mathcal{F}_i$ is the i-th frontal matrix corresponding to the i-th separator. Write

$$\mathcal{F}_i = \begin{pmatrix} F_{11} & F_{21}^T \\ F_{21} & U_i^0 \end{pmatrix} \tag{3.2}$$

where the first block column F_{11} and F_{21} contain the nodes belonging to the i-th supernode. We refer to this block column as the separator block $\mathcal{S}_i$. Let S_i^0 be the initial separator block; the first block column of F_i^0. The remainder of the frontal matrix U_i^0 corresponds to the nodes belonging to the i-th update matrix $\mathcal{U}_i$. Indeed, $\mathcal{U}_i$ is U_i^0 updated by the i-th block outer-product update. We call this matrix the *initial update matrix.*

In the discussion that follows, we refer to the subblock of the i-th block outer product update $L_i^{(2)} L_i^{(2)T}$, consisting only of the blocks in $\mathcal{J}_i$, as $\bar{U}_i$. For each j, let $\bar{U}_i^{(j)}$ be the submatrix of $\bar{U}_i$ containing only the supernodes with index greater than or equal to j. In the case where $\bar{U}_i^{(j)} = \bar{U}_i$, we use the notation $\bar{U}_i$. The notation $\mathcal{U}_i^{(j)}$ is used in the same way.

Remark 3.0.10. *Given the notation in this section:*

- *Only $\mathcal{S}_i$ is required to calculate the i-th block column of $\mathcal{L}$, $L_i = \begin{pmatrix} L_i^{(1)} \\ L_i^{(2)} \end{pmatrix}$, since $F_{11} = L_i^{(1)} L_i^{(1)T}$ and $L_i^{(2)} = F_{21} L_i^{(1)-T}$. Thus, the (initial) update matrices need to be stored but not necessarily explicitly formed.*
- *$\bar{U}_i$, U_i^0, $\mathcal{U}_i$ have supernode indices $\mathcal{J}_i$.*
- *$\mathcal{S}_i$ has row supernode indices $\mathcal{I}_i$ and column supernode index i.*
- *$\mathcal{F}_i$, F_i^0, have supernode indices $\mathcal{I}_i$*

Finally, for a supernode i, let $lvl(i)$ equal the level of i in the elimination/nested dissection tree.

3.1 Frontal and Update matrix structures

We now explore the structure of the frontal and update matrices. In our algorithm, we use the following, equivalent definition of the frontal matrix using a combination of the two representations of the frontal matrix found in Chapter 2.

For an elimination tree of height l, fix a level $s \in \{0, \ldots, l-1\}$. We call this level the *switching level*. See Figure 3.1. Let $lvl(i)$ equal the level of supernode i. This new definition represents the frontal matrix as the sum of the initial frontal matrix, block outer products updates from the proper descendants up to the switching level, and the subblocks of the update matrices of the supernodes at the switching level. In our algorithm, it turns out that the block outer product updates are of low rank, and the update matrices have a sparse block structure. Moreover, these matrices have a hierarchical block structure that we will describe. See Figure 3.2 for an illustration of this representation.

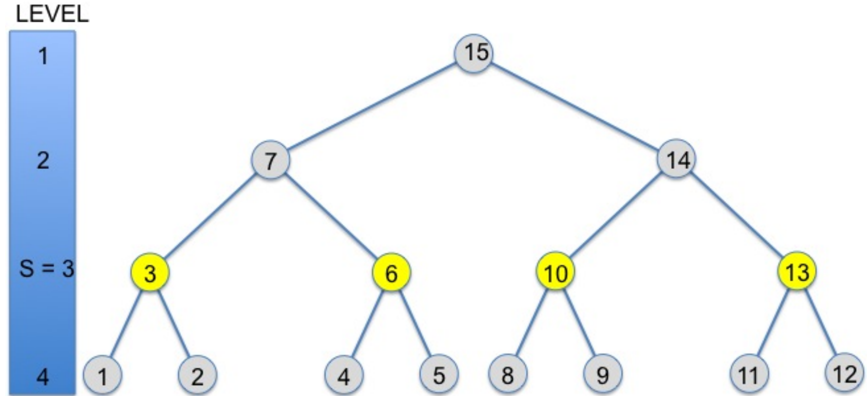


Figure 3.1: Levels in a elimination tree with height l and switching level at level s.

Theorem 3.1.1. *Fix a switching level s in the tree $\mathcal{T}(\mathcal{A})$. Then*

$$\mathcal{F}_i = \begin{pmatrix} A_{ii} & A_{i_1 i}^T & \ldots & A_{i_p i}^T \\ A_{i_1 i} & & & \\ \vdots & & 0 & \\ A_{i_p i} & & & \end{pmatrix} \updownarrow\!\!\leftrightarrow \sum_{k:lvl(k)=s} \mathcal{U}_k^{(i)} \updownarrow\!\!\leftrightarrow \sum_{k:lvl(j)<lvl(k)<s} -\bar{U}_k^{(i)}. \tag{3.3}$$

Proof. By definition of the frontal matrix, for a non-leaf separator i with children c_1 and c_2, the initial update matrix is given by

$$U_i^0 = \mathcal{U}_{c_1}^{(i+1)} \updownarrow\!\!\leftrightarrow \mathcal{U}_{c_2}^{(i+1)}.$$

Thus, the update matrix can be written as

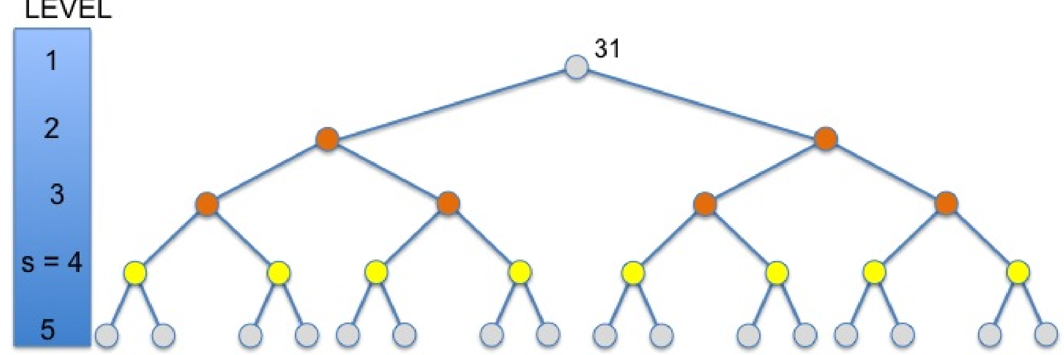
$$\mathcal{U}_i = \mathcal{U}_{c_1}^{(i+1)} \updownarrow\!\!\leftrightarrow \mathcal{U}_{c_2}^{(i+1)} \updownarrow\!\!\leftrightarrow -\bar{U}_i. \tag{3.4}$$

In general, since $\mathcal{U}_i^{(j)}$ is just a submatrix of $\mathcal{U}_i$, by Theorem 2.3.5, it is easy to see that that

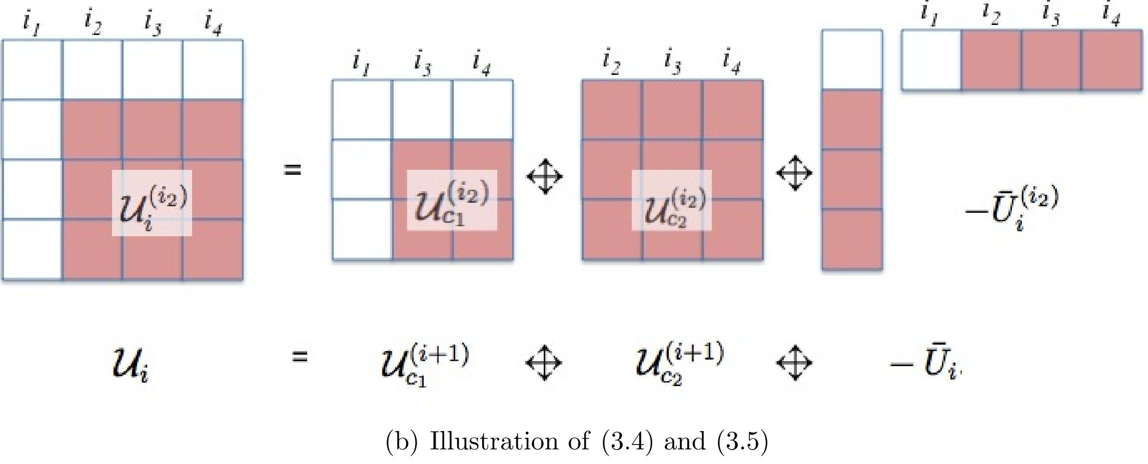
$$\mathcal{U}_i^{(j)} = \mathcal{U}_{c1}^{(j)} \oplus \mathcal{U}_{c2}^{(j)} \oplus -\bar{U}_i^{(j)}. \tag{3.5}$$

Repeatedly applying this equation to (3.4) until the update matrices in the right-hand side are at level s finishes the proof. □

Remark 3.1.2. *The last two summations on the right-hand side of* (3.3) *grow in the number of terms as* j *increases, but each term shrinks in size. This implies that eventually, each of these terms consist of small dense/low-rank blocks. In fact, we later show that in some cases, many of these pieces are mutually almost disjoint in a hierarchical way.*



(a) How the supernodes contribute to $\mathcal{F}_{31}$. Yellow supernodes contribute to the second sum. Orange supernodes contribute to the third sum.



(b) Illustration of (3.4) and (3.5)

Figure 3.2: Illustration of the representation of $\mathcal{F}_i$ in (3.3)

We now show that in some applications, the matrices given by the two summations in (3.3) have a special hierarchical sparse block structure. In fact, the terms in the summations will correspond to almost disjoint matrices. To see this, we examine the case of a finite element system with a regular mesh ordered by nested dissection. Consider the separator i. The element containing i consists of the (parts of) separators bordering i. That is, the separators $\{p_1, p_2, p_3, p_4\}$ form an *element* as shown in Figure 3.3.

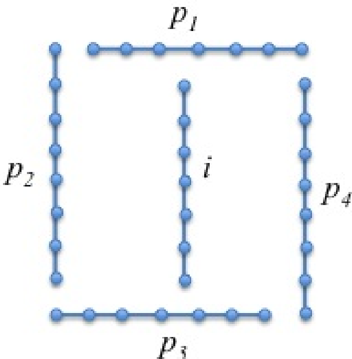


Figure 3.3: Element containing i

We have the following observation.

Lemma 3.1.3. *For a separator i:*

- *The frontal matrix $\mathcal{F}_i$ will contain exactly the nodes from the separators $\{i, p_1, p_2, p_3, p_4\}$.*
- *The nodes in the update matrix $\mathcal{U}_i$ are the nodes from the separators $\{p_1, p_2, p_3, p_4\}$.*

Proof. Suppose i is a leaf separator. Then by definition, $\mathcal{F}_i = \mathcal{F}^0$ which is a submatrix of $\mathcal{A}$. Since $\mathcal{A}_{ij}$ is nonzero if and only if x_j and x_i belong to the same element, $\mathcal{F}_i$ consists only of the nodes belonging to the same element as i. Moreover, in this case, by definition of the update matrix, $\mathcal{U}_i$ consists of the nodes belonging to $\{p_1, p_2, p_3, p_4\}$.

If i is a non-leaf separator, then $\mathcal{F}_i = \mathcal{F}^0 \oplus \mathcal{U}_{c_1} \oplus \mathcal{U}_{c_2}$ where c_1 and c_2 are the children of i. Thus, $\mathcal{F}_i$ consists of the union of nodes belonging to $\mathcal{F}^0$ (nodes in $\{i, p_1, p_2, p_3, p_4\}$), $\mathcal{U}_{c_1}$, and $\mathcal{U}_{c_2}$. Using induction, we can show that the nodes of $\mathcal{U}_{c_1}$, and $\mathcal{U}_{c_2}$ are a subset of the nodes in $\{i, p_1, p_2, p_3, p_4\}$. □

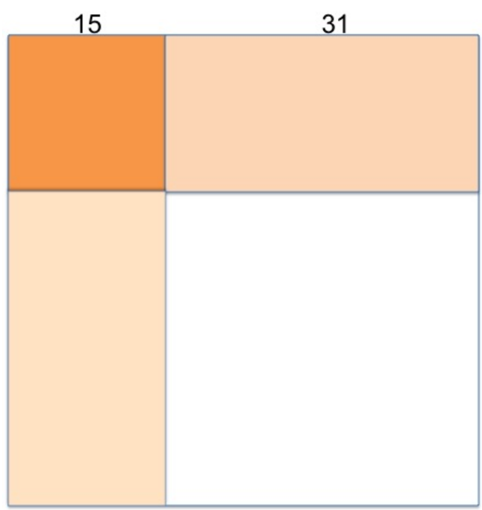
Remark 3.1.4. *By definition, the nodes contained in $\bar{U}_i$ are exactly the same as the nodes contained in $\mathcal{U}_i$.*

Thus, in the representation of the frontal matrix in (3.3), we can determine the structure of the contribution of each term in the right-hand side sums by looking at the separators in the element that contains them and ignoring the ones that have already been eliminated.

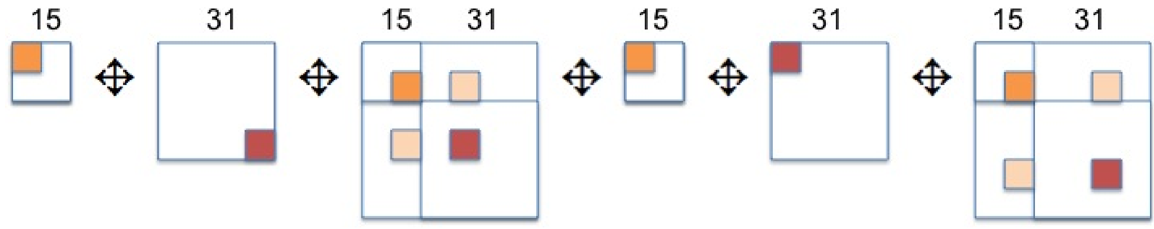
Consider the case in in Figure 3.4 where there are 31 separators/supernodes, five levels of dissection, and switching level $s = 5$.

Suppose $i = 15$. Then the $\mathcal{F}_{15}$ has the nodes belonging to $\{15, 31\}$, and the terms in (3.3) have the following structures:

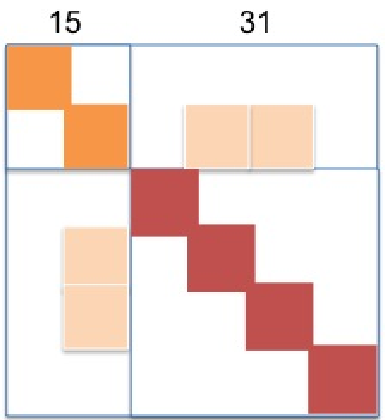
- $\mathcal{F}_{15}^0$:



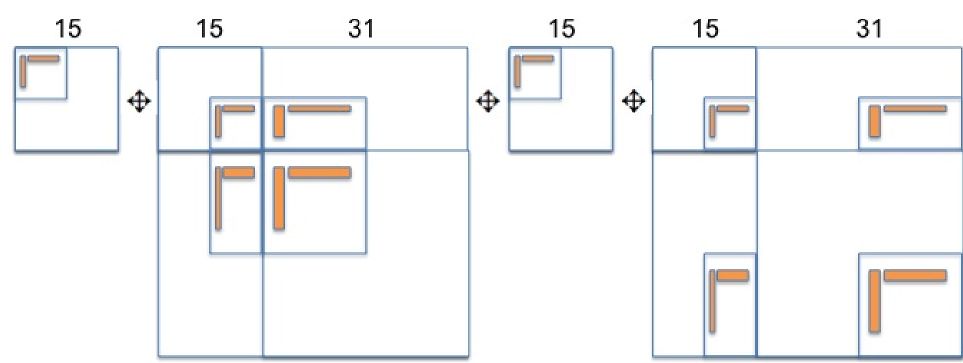
- $\sum_{k:lvl(k)=5} \mathcal{U}_k^{(15)}$ before the extend-add:



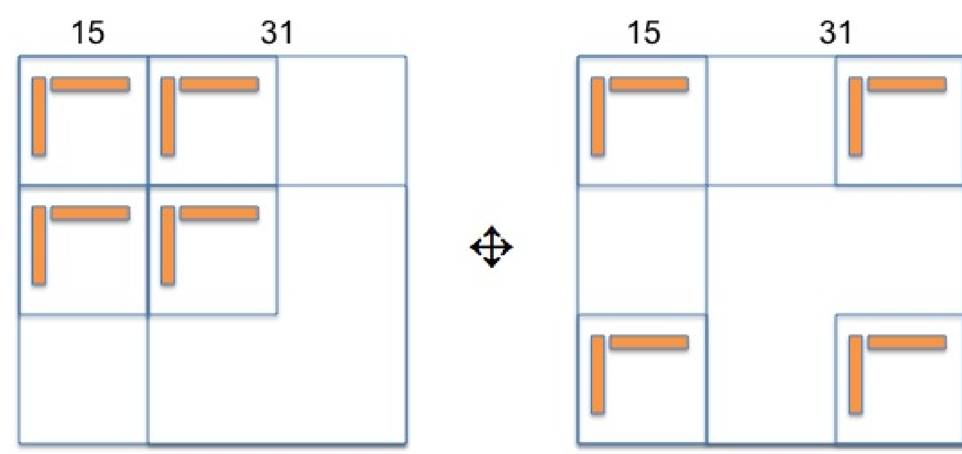
- $\sum_{k:lvl(k)=5} \mathcal{U}_k^{(15)}$ after the extend-add (note the overlap of the blocks):

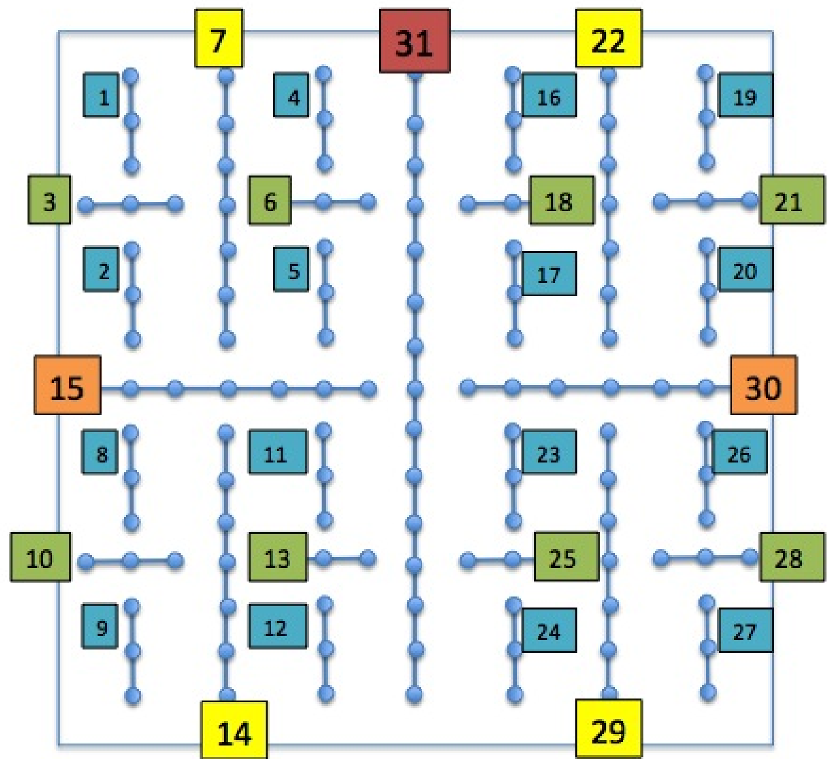


- $\sum_{k:2<lvl(k)<5} \bar{U}_k^{(15)}$:
  - $\sum_{k:lvl(k)=3} \bar{U}_k^{(15)}$:

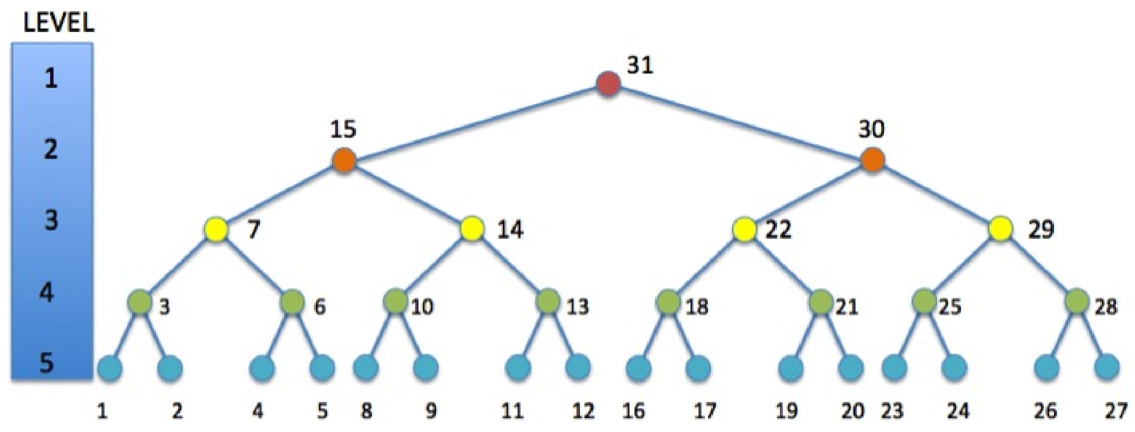


  - $\sum_{k:lvl(k)=3} \bar{U}_k^{(15)}$:





(a) Regular mesh with 31 separtors.



(b) Separator tree $\mathcal{T}$ with five levels.

Figure 3.4: Illustration of the representation of $\mathcal{F}_i$ in (3.3)

These examples illustrate the underlying structure of the frontal matrices. If we write

$$\mathcal{F}_i = \begin{pmatrix} F_{11} & F_{21}^T \\ F_{21} & U_i^0 \end{pmatrix} \tag{3.6}$$

then one can see that not only do the pieces F_{11}, F_{21} and U_i^0 have a hierarchical block structure, each of these blocks are made up of low rank pieces. Thus, each of these matrices contain a hierarchical low-rank structure which can be characterized by the HSS format. In [99], Xia verifies the low-rank structure of these matrices through numerical experiments where he uses the standard multifrontal method to solve a 5-point discretized Laplacian. Exploiting this structure can result in better storage of update matrices, fast multiplication to random matrices, and storage reduction. Xia incorporated HSS operations into the multifrontal method to improve operation costs [99]. In our algorithm, we also exploit this structure to reduce memory and computational costs.

3.2 Partial Left-Looking Multifrontal Method

There three high level approaches to Cholesky factorization. *Right-looking Cholesky* (eager method) traverses the columns from left to right, and the entire Schur complement matrix is formed at every step. *Left-looking Cholesky* (lazy method) traverses the columns from left to right, and at each step, only updates to the current column from previous columns. Computations are then done on the current column. In this case, the updates to the current column are done at the very last minute. Both these methods are easily generalized to the block Cholesky factorization case. In the *multifrontal method*, columns are traversed in the order of the elimination tree. If the matrix has been already been permuted, the columns are traversed from left to right. At each step, update matrices (subblock of a partially formed Schur complement) are fully formed. Thus, the multifrontal method is considered a type of right-looking method.

3.2.1 Left-looking multifrontal method

In our algorithm, we use a combination of a *left-looking multifrontal method* (Algorithm 3) with the *traditional multifrontal method* (Algorithm 2). Recall the representation of the frontal matrix in Theorem 3.3. In the left-looking multifrontal method, the update/frontal matrices are not stored. Instead, we compute the appropriate block column of the frontal matrix (separator block $\mathcal{S}_i$) on the fly by updating the appropriate piece in the initial frontal matrix from the previously computed blocks of $\mathcal{L}$. Although we could get these previously computed pieces from L, we store them in a stack which we explain in Section 3.2.3. See Algorithm 3 for the left-looking multifrontal method. Recall that $\mathcal{S}_i$ is the separator block (first block column of $\mathcal{F}_i$).

Algorithm 3 Left-looking multifrontal method.

Require: SPD matrix $\mathcal{A}$ partitioned into blocks; elimination tree $\mathcal{T}(\mathcal{A})$ with K nodes.
Ensure: Cholesky factor $\mathcal{L}$ where $\mathcal{A} = \mathcal{L}\mathcal{L}^T$ with block columns L_i.

1: **for** $i = 1, \ldots, K$ **do**
2:    **if** i is a leaf node **then**
3:       Form the $\mathcal{F}_i$ with the appropriate subblock of $\mathcal{A}$.
4:    **else**
5:       Form the $\mathcal{F}_i$ using using the formula

$$\mathcal{F}_i = F_i^0 \mathbin{\updownarrow\!\!\!\!\leftrightarrow} \sum_{k:lvl(j)<lvl(k)<s} -\bar{U}_k^{(i)},$$

      where the matrix $\bar{U}_k^{(i)}$ is a product of subblocks in L_i.
6:    **end if**
7:    Apply a Cholesky algorithm to $\mathcal{F}_i$ to get L_i.
8: **end for**

Remark 3.2.1. *In Algorithm 3, it is unnecessary to form the entire matrix $\mathcal{F}_i$ at each step. It suffices to only form the separator block $\mathcal{S}_i$ (first block column of $\mathcal{F}_i$). The algorithm does not change with this modification.*

In this dissertation, we use a *partial left-looking multifrontal method* in the sense that given a switching level s in the elimination tree, we use the traditional multifrontal method for supernodes with level greater than or equal to s and the left-looking multifrontal method for levels less than s. Thus, the frontal matrices $\mathcal{F}_i$ with $lvl(i) < s$ are formed according to (3.3). At the switching level, the update matrices are stored to be looked up later when forming the frontal matrix.

3.2.2 Stack storage

In Algorithm 2, the update/frontal matrices are stored as dense blocks in a stack. This stack structure is related to the postorder traversal of the elimination tree $\mathcal{T}(\mathcal{A})$. We first examine this stack structure. For simplicity, we assume that the elimination tree is full and balanced with l levels and $2^l - 1$ nodes.

One can either store the update matrices in the stack or the frontal matrices in the stack. We discuss both methods. In the first method, the update matrices are stored in the stack as follows. In the i-th step, the i-th update matrix is pushed onto the stack.

1. - If i is a leaf node: Form the frontal matrix with the appropriate subblock of $\mathcal{A}$.
   - If i is a leaf node: Pop the update matrices $\mathcal{U}_{c_1}$ and $\mathcal{U}_{c_2}$ from the top of the stack and form $\mathcal{F}_i$.

2. Eliminate i in $\mathcal{F}_i$ to get $\mathcal{U}_i$.

3. Push $\mathcal{U}_i$ onto the stack.

One thing to notice from this basic algorithm is that at Step i, both $\mathcal{U}_{c_1}$ and $\mathcal{U}_{c_2}$ are stored in the stack, and both need to be popped from the stack to be combined into $\mathcal{F}_i$. However, one can save space by forming $\mathcal{F}_i$ in two steps: first partially form $\mathcal{F}_i$ in Step c_1 and finish forming $\mathcal{F}_i$ in Step c_2. Once we reach Step i, $\mathcal{F}_i$ will be on the top of the stack. At Step i, the (partial) frontal matrix of the parent of i is stored in the stack as follows. Let r be the parent of i.

1. - If i is a leaf node: Form the frontal matrix from the appropriate subblock of $\mathcal{A}$.
   - If i is a non-leaf node: Pop the frontal matrix from the stack.

2. Eliminate i in $\mathcal{F}_i$ to get $\mathcal{U}_i$.

3. - If i is the left child of r: Partially form the frontal matrix of the parent r with $\mathcal{F}_r = \mathcal{F}_r^0 \updownarrow\!\!\!\!\leftrightarrow \mathcal{U}_i$.
   - If i is the right child of r: Pop the partially formed $\mathcal{F}_r$ from the stack and extend-add $\mathcal{U}_i$ to it to complete the formation.

4. Push $\mathcal{F}_r$ back onto the stack.

Both methods are compared in Table 3.1. Notice that storing the frontal matrices in the stack require less memory. We have the following theorem regarding the maximum length of the stack compared to the height of the tree.

Theorem 3.2.2. *Suppose the elimination tree is balanced and full with l levels, then the maximum height of the stack is l when used to store the update matrices and $l-1$ when used to store the frontal matrices.*

Proof. It is easy to see that adding one more level increases the maximum height of the stack by one. Thus, the theorem follows immediately from an induction argument. □

In the algorithms that we consider, we will use the stack for storing the frontal matrix.

3.2.3 Storage of the supernodal frontal matrix within the stack

There are many different ways to store the frontal matrices themselves within the stack structure. In the traditional multifrontal method, the frontal matrices are stored as dense blocks. In Liu's multifrontal paper [64], he discusses the storage required in this case: Working storage is needed for the entire frontal matrix during its formation and partial elimination. However, since the first supernodal column (separator block) of $\mathcal{F}_j$ can overlap with the storage of the corresponding block column L_j in $\mathcal{L}$, we can store the frontal matrix in two pieces:

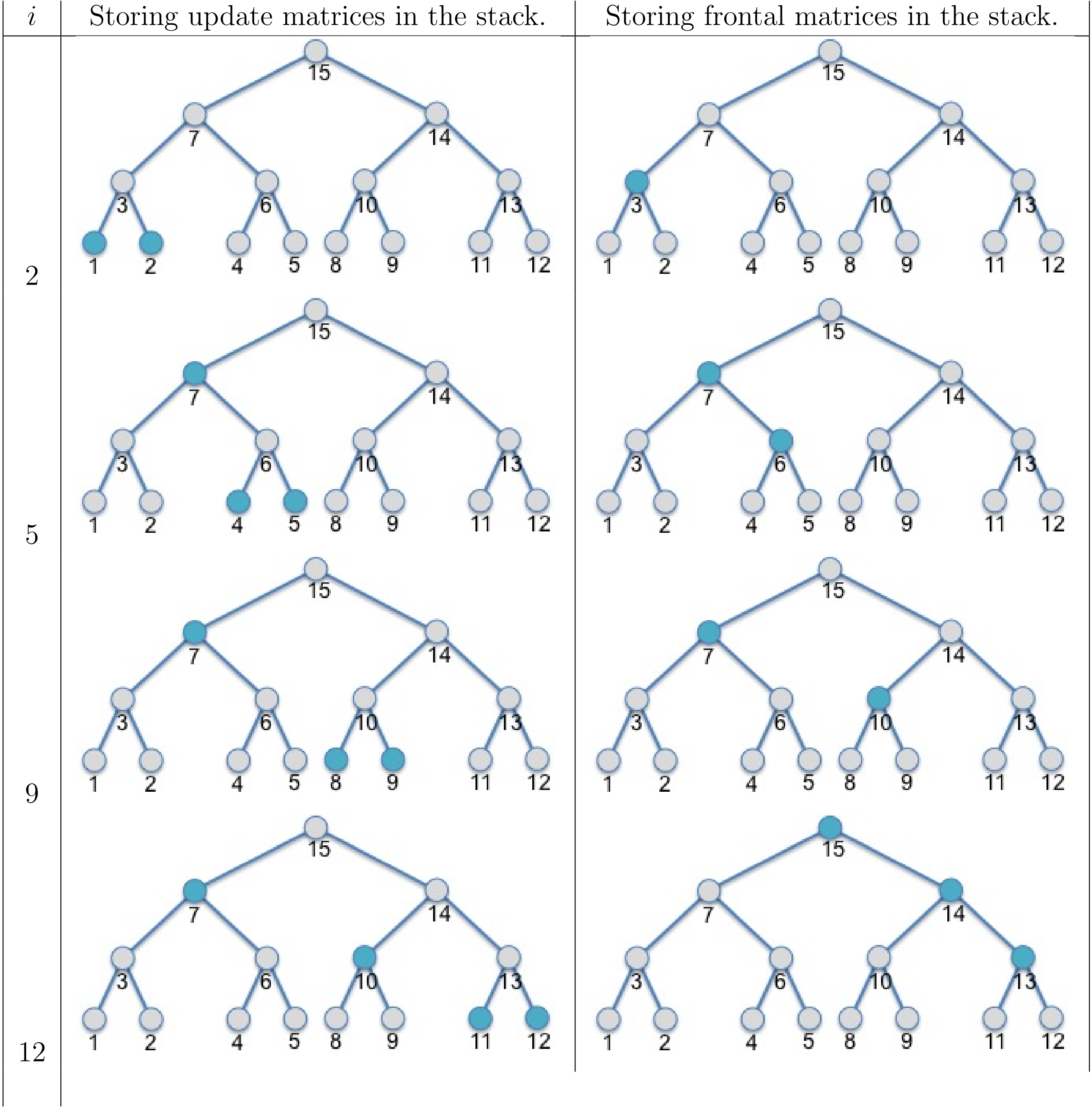


Table 3.1: Comparison of two stack uses. Highlighted nodes are nodes in the stack at step i.

1. Store the first supernode column (separator block) in the respective block column of $\mathcal{L}$. This reduces the amount of data movement after partial elimination, since the nodes are already in place.

2. Working array for the update matrix portion (initial update matrix). Thus, only working storage for the update matrices is needed.

In the Section 3.1, we described the underlying structure of the frontal and update matrices and its representation as HSS matrices (see Chapter 4 for a definition). In [99], Xia discusses storing the frontal and update matrices as HSS matrices within the stack. In [92], the frontal matrices are stored as HSS matrix, and the update matrices are stored as dense blocks. Algorithms for HSS structures are then applied to these matrices when computing the partial factorizations.

3.2.4 Partial left-looking structured multifrontal method

In our partial left-looking *structured* multifrontal method, Algorithm 4, we use a stack to store the frontal matrices in their structured form. That is, the frontal matrices are not formed until needed, but the components that make up the frontal matrices are organized in a stack. The components consist of the pieces from the update matrices at the switching level and the outer product updates from levels less than the switching level as in (3.3). The update and outer product update matrices are each stored as vectors. Thus, each element of the stack contains two vectors. Although the outer-product pieces can be retrieved from $\mathcal{L}$, we use this schema to improve data locality. Moreover, this schema may be used in a parallel implementation. In Chapter 4, we show that the block outer-product updates may be approximated by low-rank matrices. This algorithm has the advantage in that the amount of storage needed is reduced. Also, because the structure is preserved, the frontal matrices can be quickly multiplied to random matrices in the essential compression step in our factorization method.

3.2.5 An example

Here we give an example of Algorithm 4 when applied to the example in Figure 3.4. Our elimination tree has 31 nodes, five levels, and switching level $s = 4$. The first 15 steps of the algorithm are shown below in Table 3.2.

3.3 Comparison of Memory Costs

The main goal of our new multifrontal algorithm is to reduce the working memory costs for the update matrices. These matrices can become very large for large mesh sizes and no longer fit into the available working memory. Here, we compare the complexity and memory costs of the traditional multifrontal method with our partial left-looking structured multifrontal

Algorithm 4 Partial left-looking structured multifrontal method.

Require: SPD matrix $\mathcal{A}$; $\mathcal{T}(\mathcal{A})$ with K nodes, height l; switching level $s < l$
Ensure: Cholesky factor $\mathcal{L}$ where $\mathcal{A} = \mathcal{L}\mathcal{L}^T$ with block columns L_i.

1: **for** $i = 1, \ldots, K-1$ **do**
2:   **if** i is a leaf node **then**
3:     Form the frontal matrix $\mathcal{F}_i$ with the appropriate subblock of $\mathcal{A}$.
4:     Apply a Cholesky algorithm to $\mathcal{F}_i$ to get L_i and $\mathcal{U}_i$.
5:   **else**
6:     **if** $lvl(i) \geq s$ **then**
7:       Pop the frontal matrix $\mathcal{F}_i$ from the stack S.
8:       Apply a Cholesky algorithm to $\mathcal{F}_i$ to get L_i and $\mathcal{U}_i$.
9:     **else**
10:       Pop the top element of S, $\{S_1, S_2\}$, which contain the pieces of $\mathcal{F}_i$.
11:       Apply a structured Cholesky algorithm to $\{S_1, S_2\}$ to get $L_i = \left[L_i^{(1)}; L_i^{(2)}\right]$.
12:       Update S_1 and S_2 by replacing each $U_k^{(i)}, \mathcal{U}_k^{(i)}$ with $U_k^{(i+1)}, \mathcal{U}_k^{(i+1)}$, respectively.
13:     **end if**
14:   **end if**
15:   **if** $lvl(i) > s$ **then**
16:     **if** i is a left child **then**
17:       Form $\mathcal{F}_i = F_r^0 \updownarrow\!\!\leftrightarrow \mathcal{U}_i$. Push $\mathcal{F}_i$ onto the stack.
18:     **else**
19:       Pop $\mathcal{F}_i$ off the stack. Form $\mathcal{F}_i = \mathcal{F}_i \updownarrow\!\!\leftrightarrow \mathcal{U}_i$. Push $\mathcal{F}_i$ onto the stack.
20:     **end if**
21:   **else if** $lvl(i) = s$ **then**
22:     **if** i is a left child **then**
23:       Initialize $S_1 = [\ ]$. Set $S_2 = [U_i]$. Push $\{S_1, S_2\}$ onto the stack.
24:     **else**
25:       Pop $\{S_1, S_2\}$ off the stack. Update $S_2 = [S_2\ U_i]$. Push $\{S_1, S_2\}$ onto the stack.
26:     **end if**
27:   **else**
28:     **if** i is a left child **then**
29:       Update $S_1 = [S_1\ L_i^{(2)}]$. Push $\{S_1, S_2\}$ onto the stack.
30:     **else**
31:       Pop $\{S_1, S_2\}$ off stack. Set $S_1 = [S_1\ L_i^{(2)}]$, $S_2 = [S_2\ U_i]$. Push $\{S_1, S_2\}$ on stack.
32:     **end if**
33:   **end if**
34: **end for**
35: Pop the top element of S, $\{S_1, S_2\}$, which contain the pieces of $\mathcal{F}_K$.
36: Apply a structured Cholesky algorithm to $\{S_1, S_2\}$ to get L_K.

$$\mathcal{F}_1 = \begin{bmatrix}1\\3\\7\end{bmatrix}, \mathcal{U}_1 = \begin{bmatrix}3\\7\end{bmatrix}, S = [\mathcal{F}_3]$$

$$\mathcal{F}_2 = \begin{bmatrix}2\\7\\15\end{bmatrix}, \mathcal{U}_2 = \begin{bmatrix}7\\15\end{bmatrix}, S = [\mathcal{F}_3]$$

$$\mathcal{F}_3 = \begin{bmatrix}3\\7\\15\end{bmatrix}, \mathcal{U}_3 = \begin{bmatrix}7\\15\end{bmatrix}, S = \left[\begin{array}{c|c} & \mathcal{U}_3\end{array}\right]$$

$$\mathcal{F}_4 = \begin{bmatrix}4\\6\\7\\31\end{bmatrix}, \mathcal{U}_4 = \begin{bmatrix}6\\7\\31\end{bmatrix}, S = \left[\begin{array}{c|c} & \mathcal{F}_6 \\ \hline & \mathcal{U}_3\end{array}\right]$$

$$\mathcal{F}_5 = \begin{bmatrix}5\\6\\15\\31\end{bmatrix}, \mathcal{U}_5 = \begin{bmatrix}6\\15\\31\end{bmatrix}, S = \left[\begin{array}{c|c} & \mathcal{F}_6 \\ \hline & \mathcal{U}_3\end{array}\right]$$

$$\mathcal{F}_6 = \begin{bmatrix}6\\7\\15\\31\end{bmatrix}, \mathcal{U}_6 = \begin{bmatrix}7\\15\\31\end{bmatrix}, S = \left[\begin{array}{c|c} & \mathcal{U}_3 \\ & \mathcal{U}_6\end{array}\right]$$

$$\mathcal{F}_7 = \begin{bmatrix}7\\15\\31\end{bmatrix} \mathbin{\updownarrow\!\!\!\leftrightarrow} \mathcal{U}_3 \mathbin{\updownarrow\!\!\!\leftrightarrow} \mathcal{U}_6, \bar{\mathcal{U}}_7 = \begin{bmatrix}15\\31\end{bmatrix},$$

$$S = \left[\begin{array}{c|c} \bar{\mathcal{U}}_7 & \mathcal{U}_3^{(8)} \\ & \mathcal{U}_6^{(8)}\end{array}\right]$$

$$\mathcal{F}_8 = \begin{bmatrix}8\\10\\14\\15\end{bmatrix}, \mathcal{U}_8 = \begin{bmatrix}10\\14\\15\end{bmatrix}, S = \left[\begin{array}{c|c} & \mathcal{F}_{10} \\ \hline \bar{\mathcal{U}}_7 & \mathcal{U}_3^{(8)} \\ & \mathcal{U}_6^{(8)}\end{array}\right]$$

$$\mathcal{F}_9 = \begin{bmatrix}9\\10\\14\end{bmatrix}, \mathcal{U}_9 = \begin{bmatrix}10\\14\end{bmatrix}, S = \left[\begin{array}{c|c} & \mathcal{F}_{10} \\ \hline \bar{\mathcal{U}}_7 & \mathcal{U}_3^{(8)} \\ & \mathcal{U}_6^{(8)}\end{array}\right]$$

$$\mathcal{F}_{10} = \begin{bmatrix}10\\14\\15\end{bmatrix}, \mathcal{U}_{10} = \begin{bmatrix}14\\15\end{bmatrix}, S = \left[\begin{array}{c|c} & \mathcal{U}_{10} \\ \hline \bar{\mathcal{U}}_7 & \mathcal{U}_3^{(8)} \\ & \mathcal{U}_6^{(8)}\end{array}\right]$$

$$\mathcal{F}_{11} = \begin{bmatrix}11\\13\\14\\15\\31\end{bmatrix}, \mathcal{U}_{11} = \begin{bmatrix}13\\14\\15\\31\end{bmatrix}, S = \left[\begin{array}{c|c} & \mathcal{F}_{13} \\ \hline & \mathcal{U}_{10} \\ \hline \bar{\mathcal{U}}_7 & \mathcal{U}_3^{(8)} \\ & \mathcal{U}_6^{(8)}\end{array}\right]$$

$$\mathcal{F}_{12} = \begin{bmatrix}12\\13\\14\\31\end{bmatrix}, \mathcal{U}_{12} = \begin{bmatrix}13\\14\\31\end{bmatrix}, S = \left[\begin{array}{c|c} & \mathcal{F}_{13} \\ \hline & \mathcal{U}_{10} \\ \hline \bar{\mathcal{U}}_7 & \mathcal{U}_3^{(8)} \\ & \mathcal{U}_6^{(8)}\end{array}\right]$$

$$\mathcal{F}_{13} = \begin{bmatrix}13\\14\\15\\31\end{bmatrix}, \mathcal{U}_{13} = \begin{bmatrix}14\\15\\31\end{bmatrix}, S = \left[\begin{array}{c|c} & \mathcal{U}_{13} \\ & \mathcal{U}_{10} \\ \hline \bar{\mathcal{U}}_7 & \mathcal{U}_3^{(8)} \\ & \mathcal{U}_6^{(8)}\end{array}\right]$$

$$\mathcal{F}_{14} = \begin{bmatrix}14\\15\\31\end{bmatrix} \mathbin{\updownarrow\!\!\!\leftrightarrow} \mathcal{U}_{10} \mathbin{\updownarrow\!\!\!\leftrightarrow} \mathcal{U}_{13}, \bar{\mathcal{U}}_{14} = \begin{bmatrix}15\\31\end{bmatrix}, S = \left[\begin{array}{c|c} \bar{\mathcal{U}}_7 & \mathcal{U}_3^{(8)} \\ & \mathcal{U}_6^{(8)} \\ \bar{\mathcal{U}}_{14} & \mathcal{U}_{10}^{(15)} \\ & \mathcal{U}_{13}^{(15)}\end{array}\right]$$

$$\mathcal{F}_{15} = \begin{bmatrix}15\\31\end{bmatrix} \mathbin{\updownarrow\!\!\!\leftrightarrow} \mathcal{U}_3^{(8)} \mathbin{\updownarrow\!\!\!\leftrightarrow} \mathcal{U}_6^{(8)} \mathbin{\updownarrow\!\!\!\leftrightarrow} \mathcal{U}_{10}^{(15)} \mathbin{\updownarrow\!\!\!\leftrightarrow} \mathcal{U}_{13}^{(15)} \mathbin{\updownarrow\!\!\!\leftrightarrow} \bar{\mathcal{U}}_7 \mathbin{\updownarrow\!\!\!\leftrightarrow} \bar{\mathcal{U}}_{14},$$

$$\bar{\mathcal{U}}_{15} = [31], S = \left[\begin{array}{c|c} \bar{\mathcal{U}}_7^{(16)} & \mathcal{U}_3^{(16)} \\ & \mathcal{U}_6^{(16)} \\ \bar{\mathcal{U}}_{14}^{(16)} & \mathcal{U}_{10}^{(16)} \\ \bar{\mathcal{U}}_{15} & \mathcal{U}_{13}^{(16)}\end{array}\right]$$

Table 3.2: First 15 steps of Algorithm 4 when applied to the example in Figure 3.4. The elimination tree has 31 nodes, five levels, and switching level $s = 4$.

method for a 2D discretized problem with mesh size $n \times n$. In our new multifrontal method, we factorize the frontal matrices using the algorithm in Chapter 4 which is shown to have complexity $O(pN^2)$ where N is the size of the matrix, and p is an upper bound for numerical ranks of the off-diagonal blocks. This algorithm requires $O(pN)$ storage. Lastly, we give some numerical examples comparing the costs of computation.

Theorem 3.16 of [93] gives the complexity and storage requirements for the traditional

multifrontal method. We outline the proof since the ideas are used again in Theorem 3.3.4. Note that the complexity is the complexity of nested dissection.

Theorem 3.3.1. *The traditional multifrontal method requires $O(n^3)$ multiplicative operations to factorize $\mathcal{A}$ and has $O(n^2 \log_2 n)$ nonzeros in the Cholesky factors. If the elimination tree is full and balanced, the storage requirement for the update matrix stack is $O(n^2)$.*

Proof. Since $\mathcal{A}$ is of size $n^2 \times n^2$, there exist a total of $l = O(\log_2 n)$ levels in the elimination tree and $O(2^{l/2}) = O(n)$. At each level k, there are 2^{k-1} supernodes (subproblems). Each problem (frontal matrix) has dimension $O(2^{\frac{l-k}{2}})$ since it involves the nodes in the element containing the corresponding separator. Each traditional Cholesky factorization takes $O(2^{\frac{l-k}{2}})^3$. Thus, the total cost is

$$\sum_{k=1}^{l} 2^{k-1} O(2^{\frac{l-k}{2}})^3 = O(2^{3l/1}) = O(n^3). \tag{3.7}$$

Each update matrix in the k-th level requires $O(2^{l-k})$ space to store the corresponding factor in $\mathcal{L}$. Thus, the total storage space for $\mathcal{L}$ is

$$\sum_{k=1}^{l} 2^{k-1} O(2^{\frac{l-k}{2}})^2 = O(l2^l) = O(n^2 \log_2 n). \tag{3.8}$$

For the update matrix stack, as mentioned before, there exist $l - k + 1$ update matrices in the stack at the k-th level. Thus, the total storage for the update matrices in the stack is

$$O(2^{l-k}) + O(2^{l-k-1}) + \cdots + O(1^2) = O(2^{l-k}). \tag{3.9}$$

The maximum storage needed for the stack occurs at $k = 1$, and we have $O(2^{l-1}) = O(n^2)$. Thus, the total space needed is $O(n^2 \log_2 n)$. □

Before we state the complexity and storage requirements for the partial left-looking structured multifrontal method, we first prove a few facts about its stack elements; that is, the frontal matrices stored in structured form. The following lemma shows that a frontal matrix in structured form of order N requires $O(N^2)$ storage, $O(N^2 q)$ operations to multiply it to a matrix of size $N \times q$, and $O(pN^2)$ to reconstruct it, where p is an upper bound on the rank of the off-diagonal blocks.

Lemma 3.3.2. *A frontal matrix of size $n \times n$ stored in the format in* (3.3) *requires*

1. *$O(N^2)$ storage,*
2. *$O(N^2 q)$ operations to multiply it to a matrix of size $n \times q$,*
3. *$O(N^2 p)$ to reconstruct the matrix.*

Proof. Suppose s is the switching level of the elimination tree. We first compute the memory needed to store the i-th frontal matrix $\mathcal{F}_i$ as the terms $\left\{\mathcal{U}_k^{(i)} : lvl(k) = s\right\}$ and $\left\{U_k^{(i)} : s_0 < lvl(k) < s, lvl(i) = s_0\right\}$. At level k, there are 2^{k-1} supernodes, and the frontal matrices are of size $O(2^{\frac{l-k}{2}})$ Thus, $\mathcal{F}_i$ is of size $O(2^{\frac{l-s_0}{2}})$, and at level s, the terms in $\left\{\mathcal{U}_k^{(i)} : lvl(k) = s\right\}$ require

$$2^{s-s_0} O\left(2^{\frac{l-s}{2}}\right)^2 = O\left(N^2\right) \tag{3.10}$$

storage.

Similarly, the terms in $\left\{U_k^{(i)} : s_0 < lvl(k) < s\right\}$ require

$$\begin{aligned}\sum_{j=s_0+1}^{s-1} 2^{j-s_0} O(2^{\frac{l-j}{2}})p &= O\left(p \sum_{j=s_0+1}^{s-1} 2^{l-s_0} \frac{1}{2^{(l-j)/2}}\right) \\ &= O\left(p2^{\frac{l-s_0}{2}}(1-2^{\frac{s-s_0+1}{2}})\right) = O\left(p2^{\frac{l-s_0}{2}}\right) = O\left(pN\right)\end{aligned}$$

storage. Thus, we need a total of $O(N^2)$ memory.

In the case where the frontal matrix is multiplied to a matrix of size $n \times q$, each dense block operation will require $O(2^{\frac{l-s}{2}})^2 q$ operations at level s and $O(2^{\frac{l-k}{2}})^2 pq$ operations at level k between s and s_0. The calculation is carried out in exactly the same way to show that we need $O(N^2 q)$ operations.

Finally, in order to reconstruct $\mathcal{F}_i$, we need to perform 2^{s-s_0} extend-adds of matrices of size $O(2^{\frac{l-s}{2}})^2$, and at each level j between s and s_0, we need to perform 2^{j-s_0} extend-adds of matrices of size $O(2^{\frac{l-j}{2}})^2$ and 2^{j-s_0} multiplications requiring $O(2^{\frac{l-j}{2}})^2 p$ operations. This results in a total of

$$O(N^2) + O(N^2) + O(N^2 p) = O(N^2 p) \tag{3.11}$$

operations. □

Remark 3.3.3. *In the case of a 2D finite element problem on a regular grid, the number of nonempty terms in* (3.3.3) *is reduced by a factor of* $2^{\lfloor s-s_0-1 \rfloor}$. *Thus, the number of terms becomes* $2^{\frac{s-s_0}{2}}$ *and becomes* $O(N^2/2^{\frac{s}{2}})$. *If we choose* $2^{s/2} \approx N/p$ *then we have* $O(N^2/2^{\frac{s}{2}}) = O(Np)$, *and the total storage needed for a frontal matrix in structured form is* $O(Np)$.

We can now state and prove the following theorem.

Theorem 3.3.4. *The partial left-looking structured multifrontal method requires* $O(n^2 p \log_2 n)$ *multiplicative operations to factorize* $\mathcal{A}$ *if* $p \approx 2^s$ *and has* $O(n^2 \log_2 p)$ *nonzeros in the Cholesky factors. If the elimination tree is full and balanced, then the storage requirement for the update matrix stack is* $O(n^2)$.

Proof. The proof is similar to the proof in Theorem 3.3.1. Before the switching level s, we use direct Cholesky and have the following computation costs:

- s bottom levels,
- for each level k, 2^{k-1} factorizations, each of dimension $O(2^{\frac{l-k}{2}})$,
- subcost $\sum_{k=l-s+1}^{l} 2^{k-1}O(2^{\frac{l-k}{2}})^3 = O(2^l 2^s) = O(n^2 2^s)$.

After the switching level, we have the structured Cholesky factorization which has a computational cost of $O(pN^2)$. There are:

- $l-s$ upper levels,
- for each level k, 2^{k-1} factorizations, each of dimension $O(2^{\frac{l-k}{2}})$,
- subcost $\sum_{k=1}^{l-s} 2^{k-1}O(p2^{2^{\frac{l-k}{2}}}) = O(p2^l(l-s)) = O(pn^2(l-s))$.

If we choose $p \approx 2^s$, then we have a total cost of $O(pn^2 \log_2 n)$.

Similarly, since the structured Cholesky factors require $O(pN)$ storage, the total storage needed for the Cholesky factor $\mathcal{L}$ is:

$$\sum_{k=l-s+1}^{l} 2^{k-1}O(2^{\frac{l-k}{2}})^2 + \sum_{k=1}^{l-s} 2^{k-1}O(p2^{\frac{l-k}{2}}) = O(2^l s) + O(p2^l/2^s) = O(n^2 log_2 p). \quad (3.12)$$

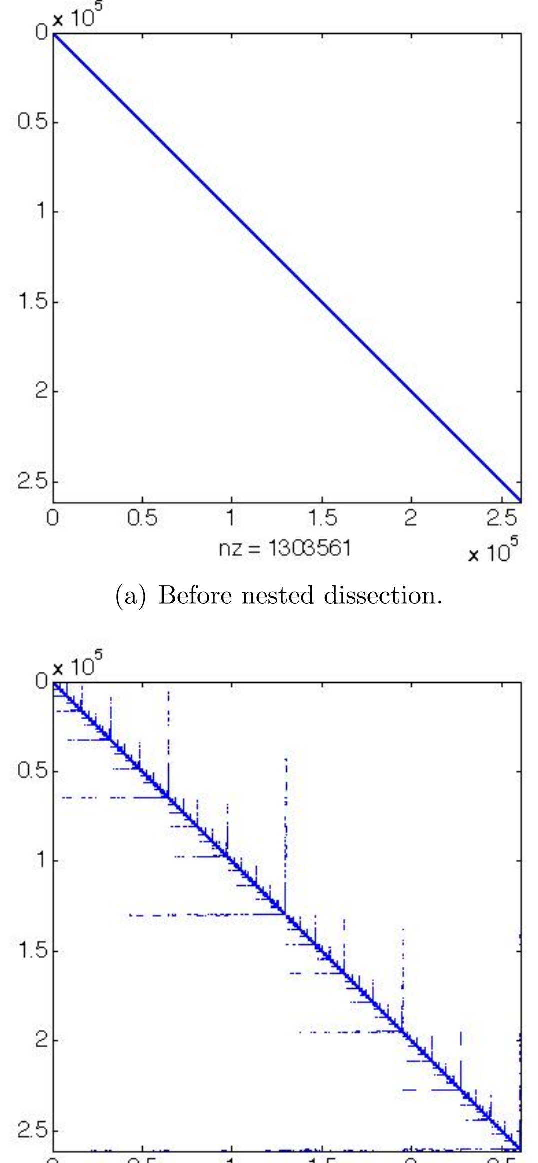
The storage needed for the update matrix stack is exactly the same as in (3.9). Thus, the total space needed is $O(n^2 \log_2 p)$. □

Remark 3.3.5. *In the case of a 2D finite element problem on a regular grid as in Remark 3.3.3, the storage requirement for the update matrix stack is $O(np)$ since each frontal matrix only needs a storage space of $O(pN)$.*

3.4 Numerical Experiments

Here we give some numerical examples comparing our method with the traditional multifrontal method. We test our method on a 2D discrete Laplacian on a 5-point stencil where $\mathcal{A}$ is a 5-diagonal $n^2 \times n^2$ sparse matrix. An example of such a matrix before and after nested dissection is given in Figure 3.5. The two methods are implemented in Matlab and are tested on a 2.2 GHz Intel Core i7 processor with 8GB memory. We use the following notation:

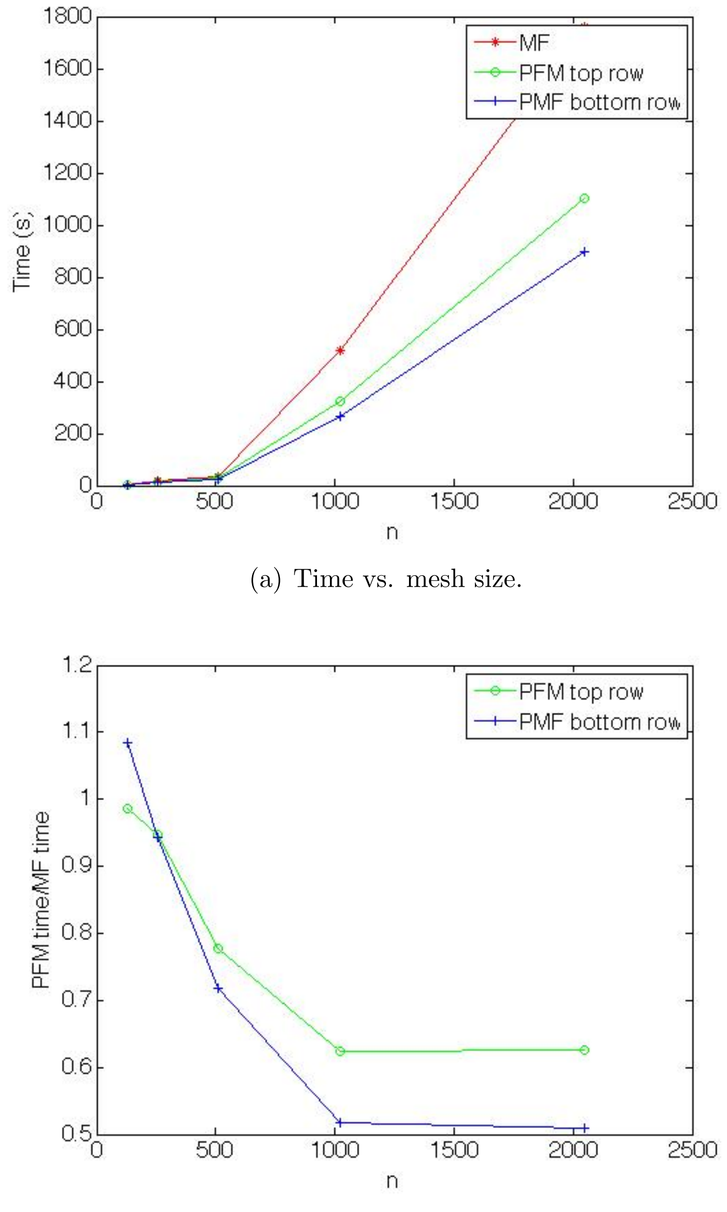
We first compare the number of flops needed on our model problem for different values of n, l, and s. See Table 3.3. In Table 3.4, we run the same experiment and compare the maximum sizes of the memory used in the frontal matrix stacks. In Figures 3.6 and 3.7, we plot the ratios of the time and memory for PMF over MF, respectively. We see that PMF does better than MF and does better when more structured levels are used. Finally in Figure 3.8, we show how the computation time, maximum stack size, and relative error change with switching level for $n = 511$ and $n = 1023$.



(a) Before nested dissection.

(b) After nested dissection.

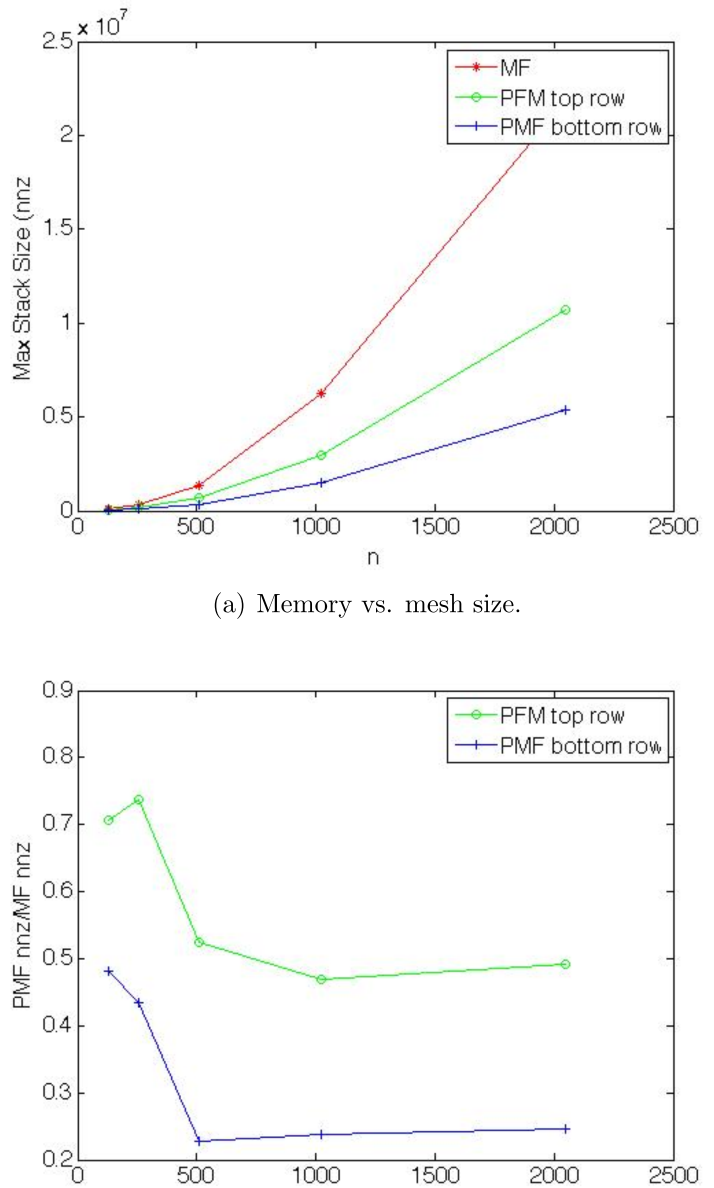
Figure 3.5: 2D discrete Laplacian on a 5-point stencil with $n = 500$



(a) Time vs. mesh size.

(b) Ratio of computation times.

Figure 3.6: Computation time comparisons for different mesh sizes.



(a) Memory vs. mesh size.

(b) Ratio of memory costs.

Figure 3.7: Maximum stack sizes for different mesh sizes.

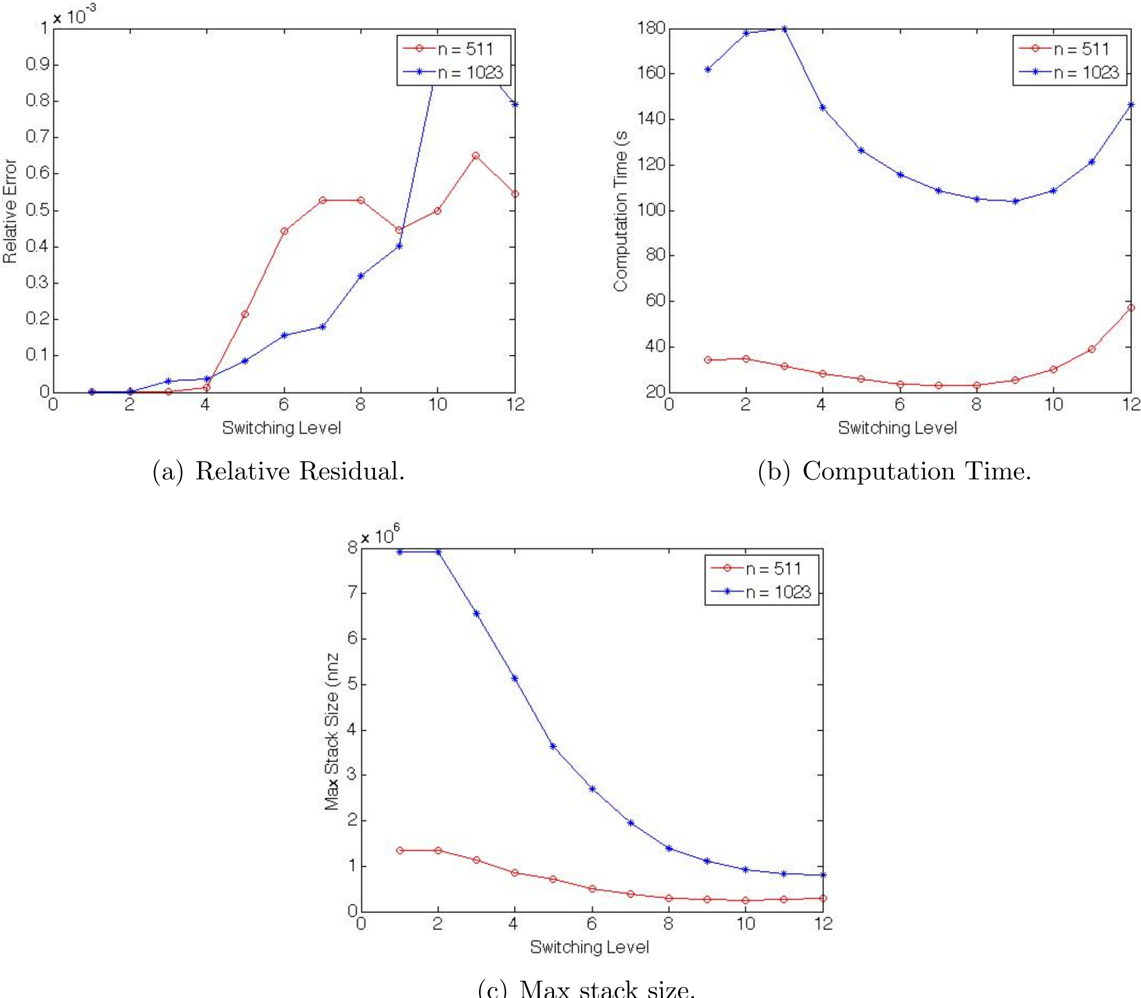
Notation	Meaning
TMF	Traditional Multifrontal Method
PMF	Partial Left-Looking Structured Multifrontal Method
n	mesh size
τ	tolerance
l	total number of levels
s	switching level
time	time in seconds
ms	max stack size (nonzeros)

Mesh	$n = 127$			$n = 255$			$n = 511$			$n = 1023$			$n = 2047$		
	l	s	flops	l	s	time	l	s	time	l	s	time	l	s	time
MF	10		3.83s	12		16.19s	12		36.25s	14		517.47s	14		1758.49s
PMF	10	4	3.78s	12	5	15.33s	12	5	28.18s	14	5	323.28s	14	5	1101.77s
	10	6	4.15s	12	7	15.28s	12	7	26.02s	14	7	267.80s	14	7	896.74s

Table 3.3: Computational time comparisons; $\tau = 10^{-4}$.

Mesh	$n = 127$			$n = 255$			$n = 511$			$n = 1023$			$n = 2047$		
	l	s	ms	l	s	ms	l	s	ms	l	s	ms	l	s	ms
MF	10		7.91E4	12		2.94E5	12		1.35E6	14		6.28E6	14		2.18E7
PMF	10	4	5.59E4	12	5	2.17E5	12	5	7.09E5	14	5	2.95E6	14	5	1.07E7
	10	6	3.81E4	12	7	1.28E5	12	7	3.07E5	14	7	1.49E6	14	7	5.35E6

Table 3.4: Maximum stack size comparisons; $\tau = 10^{-4}$.



(a) Relative Residual.

(b) Computation Time.

(c) Max stack size.

Figure 3.8: PMF comparisons for different switching levels; $l = 12, tol = 10^{-8}$.

Chapter 4

Cholesky Factorization of the Frontal Matrices

It has been observed [99] that the frontal/update matrices in the generalized multifrontal method, when applied to problems such as the ones in Section 1.2.1, possess certain low-rank properties with appropriate choices of tolerance and block sizes. In particular, if we write the i-th frontal matrix as

$$\mathcal{F}_i = \begin{pmatrix} A_{ii} & A_{iB} \\ A_{Bi} & A_{BB} \end{pmatrix} = \begin{pmatrix} L_{ii} & \\ L_{Bi} & I \end{pmatrix} \begin{pmatrix} L_{ii}^T & L_{Bi}^T \\ & \mathcal{U}_i \end{pmatrix},$$

where $\mathcal{U}_i$ is the ith update matrix, then

- A_{iB} is numerically of low-rank,
- A_{ii} and $\mathcal{U}_i$ have off-diagonal blocks with numerically low-ranks.

Matrices with this low-rank property can be approximately described using a *hierarchically semiseparable* (HSS) representation. Such matrices are called HSS matrices. As a result, fast algorithms which exploit this structure may be used to compute the Cholesky factorization of the frontal matrices. In this chapter, we propose a robust Cholesky factorization method for symmetric positive definite (SPD), hierarchically semiseparable (HSS) matrices. Classical Cholesky factorizations and some semiseparable methods need to sequentially compute Schur complements. In contrast, we develop a strategy involving orthogonal transformations and approximations which avoids the explicit computation of the Schur complement in each factorization step. The overall factorization requires fewer floating point operations and has better data locality when compared to the recent HSS method in [96]. Our strategy utilizes a robustness technique so that an approximate generalized Cholesky factorization is guaranteed to exist.

In [61], three different methods are tested for compressing the off-diagonal blocks in each iteration, i.e., rank-revealing QR, SVD, and SVD with random sampling. In the comparisons,

using SVD with random sampling is fast and stable with high probability. Thus, in this dissertation, we use the random sampling technique for compression. Later, we show how to combine this randomization with our method of storing the frontal matrices to improve speed. The complexity of this algorithm is $O(N^2p)$, where N is the dimension of matrix and p is the maximum off-diagonal (numerical) rank.

4.1 Overview

The HSS matrix structure was first discussed in [23, 24] and arose from an algebraic abstraction of the fast integral equation solver developed in [47]. More broadly, the HSS matrix is closely related to other rank-structured matrices such as the $\mathscr{H}$ [48, 51], $\mathscr{H}^2$ [52, 49], quasiseparable [33, 89, 88], and sequentially semiseparable (SSS) [23, 24] matrices. Among other things, some of these matrix structures, such as the HSS, $\mathscr{H}$, and $\mathscr{H}^2$ matrices, have proven to be invaluable tools in the fast numerical solutions of integral equations. Recently, they have been shown to play central roles in the superfast direct factorization and preconditioning of certain classes of large, sparse matrices [43, 42, 48, 49, 99].

The semi-separable matrix structures share the common feature that all their off-diagonal blocks have rapidly decaying singular values. Thus, the numerical ranks of off-diagonal blocks are significantly smaller than the matrix dimensions [22, 21, 40, 68, 79, 7, 14, 23]. Recently, Xia and Gu have proposed an efficient algorithm that computes the approximation

$$A = R^{\top} R + O\left(\tau\right), \tag{4.1}$$

where $R^{\top}R$ is an HSS matrix and R is upper triangular [96]. This algorithm costs $O(N^2k)$ floating operations (*flops*), where N is the order of A and k is the HSS rank. The main attractiveness over the algorithms in [23, 24, 21, 67] is that all Schur complements of A are kept SPD throughout the computation, thereby ensuring the existence of R for any given positive τ value. This robustness characteristic, often referred to as *Schur-monotonic*, is achieved by an approximation process, whereby the difference between the approximated and true Schur complement is a small, non-negative definite matrix. This technique is referred to as *Schur compensation* in [96].

Given an SPD matrix A, we are interested in the rapid computation of an SPD, *hierarchical semi-separable* (HSS) matrix S such that

$$A = S + O\left(\tau\right), \tag{4.2}$$

where τ is a user-prescribed tolerance. We propose a new algorithm for computing the approximation S in (4.2), where $S = PP^{\top}$ is a generalized Cholesky factorization with P an HSS matrix, computed through a sequence of Householder transformations and Cholesky factorizations. Our algorithm is designed to be Schur-monotonic and free of any *direct* Schur complement computations, resulting in faster computation and better data locality.

Recent work has suggested the efficiency and effectiveness of utilizing randomized algorithms [56, 97, 44, 45] for low-rank matrix compression during the HSS matrix construction.

As pointed out in [44, 45], some of these randomized algorithms are equivalent to subspace iteration methods with an excellent start matrix, and the feature that the off-diagonal blocks of A have rapidly decaying singular values allows such algorithms to compute low-rank approximations quickly. Our algorithm adopts a reliable version of the randomized algorithm that maintains Schur-monotonicity and yet allows fast low-rank compression. When the matrix is large, our algorithm is much faster than that of [96] (see [61]). Note that Martinsson [67] has developed an efficient algorithm to approximately construct HSS matrices using random sampling techniques. However, this algorithm does not appear to maintain Schur-monotonicity during the factorization process, and can produce an indefinite HSS approximation even when the original matrix is SPD.

Depending on the tolerance level, the matrix S in (4.2) can either be used as a matrix factorization for a rapid linear system solver or as a preconditioner in the context of preconditioned conjugate gradient iterations.

4.2 Preliminaries

In this section, we introduce some notation and give a brief introduction to the key concepts of HSS structures. We also describe some standard low-rank matrix approximation methods, including the random sampling method, which will be used to compress off-diagonal blocks.

4.2.1 Notation and terminology

As in the previous notation, let $\mathcal{T}$ be a full binary tree. The root of $\mathcal{T}$ is denoted by $\text{root}(\mathcal{T})$, and for each node i, $\text{sib}(i)$ and $\text{par}(i)$ denote the sibling and parent of i. If i is a non-leaf node, we represent the left and right child of i with i_1 and i_2, respectively. For our purposes, $\mathcal{T}$ is assumed to be *postordered*. That is, the nodes are ordered so that non-leaf nodes i satisfy the ordering $i_1 < i_2 < i$. We assume the levels of $\mathcal{T}$ are ordered *top-down*. In other words, $\text{root}(\mathcal{T})$ is at level 0 and the leaves of $\mathcal{T}$ are at the largest level; see Figure 4.1(c).

Let $A \in \mathbb{R}^{N\times N}$ be a symmetric matrix with indexing set $\mathcal{I} := \{1, \ldots, N\}$. For a subset t_i of $\mathcal{I}$, let t_i^c be the set of all indices less than those of t_i and t_i^r be the set of all indices greater than those of t_i; then, $\mathcal{I} = t_i^c \cup t_i \cup t_i^r$. Allow $A_{t_i t_j}$ to represent the submatrix of A with row index set t_i and column index set t_j.

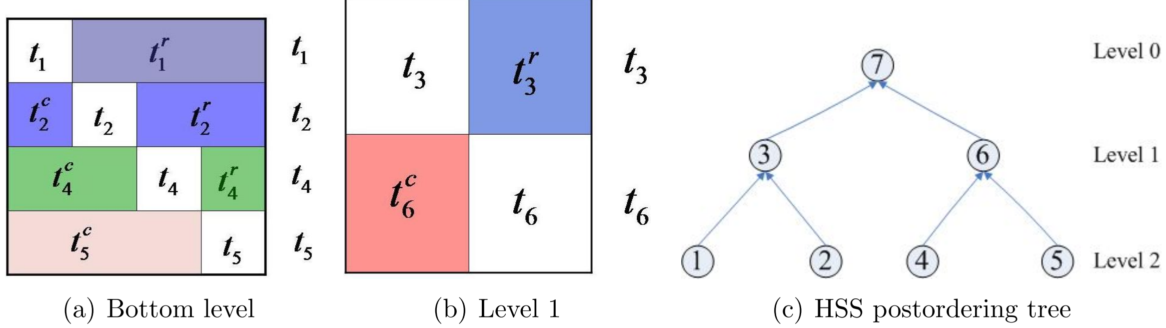
4.2.2 Introduction to symmetric HSS matrices

We introduce the postordered HSS form of a *symmetric* matrix A; see [96] for the general case. The HSS representation of A depends on a recursive partitioning of the rows and columns. Since A is symmetric, we assume the rows and columns have the same partitioning, and it is understood that the ith partition of A refers to both the ith row and column partition. As in [96], the partitioning is organized via a full, postordered binary tree $\mathcal{T}$; i.e., the ith

node of T corresponds to the ith partition of A. The indices of the ith partition of A are contiguous and satisfy the following:

- $t_i \cup t_{\mathrm{sib}(i)} = t_{\mathrm{par}(i)}$,
- $t_{\mathrm{root}(\mathcal{T})} = \mathcal{I}$.

Figure 4.1 illustrates these sets.



(a) Bottom level (b) Level 1 (c) HSS postordering tree

Figure 4.1: Matrix partition and the corresponding index sets and binary tree $\mathcal{T}$.

Each node i of $\mathcal{T}$ is associated with a set of matrices D_i, U_i, R_i, B_i called *generators*, where B_i is empty if i is a right child. The generators satisfy the recursive relationships

$$D_i = \begin{pmatrix} D_{i_1} & U_{i_1}B_{i_1}U_{i_2}^\top \\ U_{i_2}B_{i_1}^\top U_{i_1}^\top & D_{i_2} \end{pmatrix}, \; U_i = \begin{pmatrix} U_{i_1}R_{i_1} \\ U_{i_2}R_{i_2} \end{pmatrix}, \tag{4.3}$$

where $D_i \equiv A_{t_i t_i}$. For example, a 4×4 block HSS form looks like

$$\begin{pmatrix} D_1 & U_1B_1U_2^\top & U_1R_1B_3R_4^\top U_4^\top & U_1R_1B_3R_5^\top U_5^\top \\ U_2B_1^\top U_1^\top & D_2 & U_2R_2B_3R_4^\top U_4^\top & U_2R_2B_3R_5^\top U_5^\top \\ U_4R_4B_3^\top R_1^\top U_1^\top & U_4R_4B_3^\top R_2^\top U_2^\top & D_4 & U_4B_4U_5^\top \\ U_5R_5B_3^\top R_1^\top U_1^\top & U_5R_5B_3^\top R_2^\top U_2^\top & U_5B_4^\top U_4^\top & D_5 \end{pmatrix}, \tag{4.4}$$

and the corresponding HSS tree is shown in Figure 4.1(c). Following the notation of [96], a block row (column) of A excluding the diagonal block is called an *HSS block row (column)*, or simply *HSS block*. For instance, the ith HSS block row and block column are

$$H_i^{row} = \begin{pmatrix} A_{t_i t_i^c} & A_{t_i t_i^r} \end{pmatrix} \text{ and } H_i^{col} = \begin{pmatrix} A_{t_i^c t_i} \\ A_{t_i^r t_i} \end{pmatrix},$$

respectively. Our algorithm is introduced using HSS block rows; the discussions for HSS block columns are similar. We call the maximum (numerical) rank of all HSS blocks the *HSS rank* of the matrix.

Many efficient algorithms have been developed for working with matrices represented or approximated by HSS structures. As shown in [21], there exist $O(N)$ algorithms for solving an HSS linear system. To clearly describe such HSS algorithms, we review the definition of a *visited set* [96].

Definition 4.2.1. *The visited set associated with a node i of a postordered binary tree $\mathcal{T}$ is*

$$\mathcal{V}_i := \{j | j \text{ is a left node and } \operatorname{sib}(j) \in \operatorname{pred}(i)\}, \tag{4.5}$$

where $\operatorname{pred}(i)$ *is the set of predecessors associated with node i, i.e.,*

$$\operatorname{pred}(i) = \begin{cases} \{i\}, & \text{if } i = \operatorname{root}(\mathcal{T}), \\ \{i\} \cup \operatorname{pred}(\operatorname{par}(i)), & \text{otherwise.} \end{cases}$$

The set $\mathcal{V}_i$ can be interpreted as the stack before the visit of i in the postordering traversal of $\mathcal{T}$ [96]. For example, we have

$$\mathcal{V}_4 = \mathcal{V}_6 = \{3\}, \ \mathcal{V}_5 = \{3, 4\}, \ \mathcal{V}_{11} = \mathcal{V}_{13} = \{7, 10\}, \ \mathcal{V}_{12} = \{7, 10, 11\};$$

see Figure 4.2.

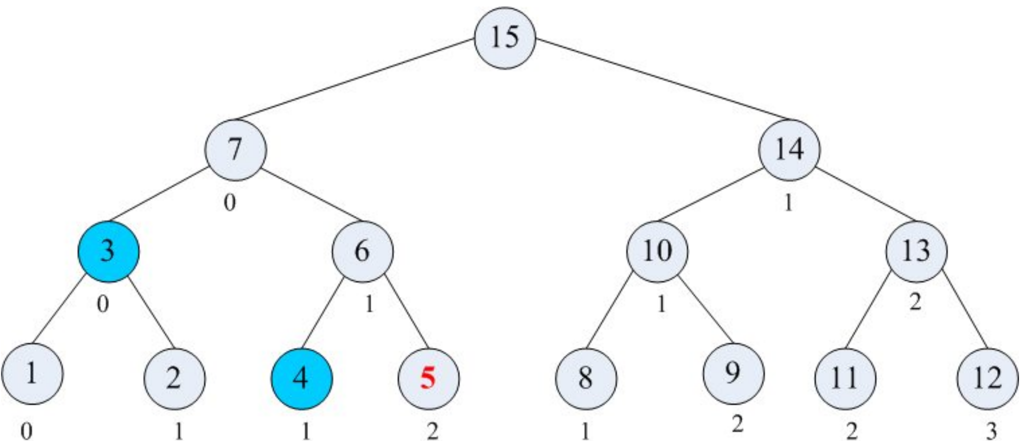


Figure 4.2: The visited set $\mathcal{V}_5$. (The number under each node i in Figure 4.2 denotes the cardinality s_i of $\mathcal{V}_i$.)

We later use the following theorem involving $\mathcal{V}_i$ to analyze the complexity of the HSS construction algorithm. In particular, the theorem shows that for a perfect binary tree, the maximum number of nodes in $\mathcal{V}_i$ is proportional to the height of the tree.

Theorem 4.2.2. *[94] Let $\mathcal{T}$ be a perfect binary tree with levels ordered top-down and s_i be the cardinality of $\mathcal{V}_i$. Then for any node i at level l, $1 \leq s_i \leq l$. Moreover, there are exactly $f_j^l := \binom{l}{j}$ nodes i at level l with $s_i \equiv j$ for some $1 \leq j \leq l$.*

4.2.3 Low-rank matrix approximation

In [61], three different methods are used for computing low-rank approximations to a matrix $B \in \mathbb{R}^{m\times n}$. Each method can be computed either by setting a tolerance τ or an explicit rank k. That is, there are two types of approximations:

- *Fixed-precision approximation:* Seek matrices $U \in \mathbb{R}^{m\times r_\tau}$ and $T \in \mathbb{R}^{r_\tau \times n}$ such that

$$\|B - UT\|_2 \leq \tau, \tag{4.6}$$

  where r_τ is determined by τ.

- *Fixed-rank approximation:* Seek matrices $U \in \mathbb{R}^{m\times r}$ and $T \in \mathbb{R}^{r\times n}$ such that

$$\|B - UT\|_2 = \min_{\text{rank}(X)\leq k} \|B - X\|_2. \tag{4.7}$$

The first method is a rank-revealing QR (RRQR) factorization. It is well known that RRQR can be used to compute low-rank approximations [19, 46] since RRQR allows one to (approximately) factor B as

$$BP \approx QR,$$

where $Q \in \mathbb{R}^{m\times k}$ has orthonormal columns, $R \in \mathbb{R}^{k\times n}$ is upper triangular, and $P \in \mathbb{R}^{n\times n}$ is a permutation matrix. The second method is the commonly used truncated singular value decomposition (SVD) [41] where

$$B \approx U\Sigma V^\top$$

with $U \in \mathbb{R}^{m\times k}, V \in \mathbb{R}^{n\times k}$, and $\Sigma \in \mathbb{R}^{k\times k}$.

The third method is a randomized algorithm to compute the low-rank approximations. Since this algorithm has been shown to typically be very fast and accurate [61], we use this method of compression in our algorithm. In general, such randomized algorithms are divided into two stages [62, 56]. First, a low-dimensional subspace approximately spanning the range of B is constructed. Then, the desired matrix decomposition is computed on a reduced matrix.

Stage A: Compute an approximate low-rank basis $Q \in \mathbb{R}^{m\times k}$ of the range of B such that Q has orthonormal columns and

$$B \approx QQ^*B.$$

Stage B: Compute the desired matrix decomposition on the smaller matrix $C := Q^*B \in \mathbb{R}^{k\times n}$.

Algorithm 5 Random Low-Rank Approximation

Require: l such that $k \leq l < \min\{m, n\}$ where k is an approximation for the rank of B
Ensure: U, T where UT is a low-rank approximation of B

1: Draw an $n \times l$ random matrix Ω whose entries are Gaussian random variables with zero mean and unit variance.
2: Compute the sample matrix
$$Y = B\Omega.$$
3: Let $Q \in \mathbb{R}^{m\times k}$ consist of the left singular vectors correspond to the k largest singular values of Y. This can be computed using an SVD where
$$Y = B\Omega = U\Sigma V^\top = [Q \mid P]\Sigma V^\top.$$
Here, $U \in \mathbb{R}^{m\times l}$ and $V \in \mathbb{R}^{l\times l}$ have orthonormal columns, Σ is an $l \times l$ nonnegative diagonal matrix, and $P \in \mathbb{R}^{m\times(l-k)}$.
4: **return** $U = Q$ and $T = Q^*B$

The HSS construction requires computing low-rank approximations of H_i^{row} (or H_i^{col}) satisfying (4.6) or (4.7). This can be achieved by approximating the orthonormal row (or column) bases. Thus, it is enough to compute the basis Q in **Stage A**. The following algorithm, equivalent to the random SVD algorithm proposed in Section 5.2 of [69], is used to quickly find Q.

The following theorem, summarized from [69], says that QQ^*B closely approximates B with very high probability for small values of p as long as the $(k+1)$st singular value of B is small. For instance, we can choose $p = 8, 10$.

Theorem 4.2.3. *Let $B \in \mathbb{R}^{m\times n}$, k and p be positive integers such that $1 \leq k \leq k + p \leq \min\{m, n\}$, and $\Omega \in \mathbb{R}^{n\times(k+p)}$ be a Gaussian random matrix with zero mean and unit variance. Let Q be the $m \times k$ matrix computed from Algorithm 5 and σ_{k+1} be the $(k+1)$st largest singular value of B. Then*

$$\|B - QQ^*B\|_2 \leq 10\sigma_{k+1}\sqrt{(k+p)n},$$

with probability at least $1 - \phi(p)$ for a decreasing function ϕ.

Remark 4.2.4. *1. The function ϕ decreases rapidly. For example, $\phi(8) < 10^{-5}$ and $\phi(20) < 10^{-17}$.*

2. In [61], Algorithm 5 is typically observed to be faster than the deterministic algorithms RRQR and SVD.

4.3 Generalized HSS Cholesky Factorization for SPD matrices

In this section, we discuss our new algorithm for computing a generalized HSS Cholesky factorization. We begin with a simple 2×2 block partitioning of an $N \times N$ SPD matrix A where

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{12}^{\top} & A_{22} \end{pmatrix}, \tag{4.8}$$

with $A_{12} \in R^{m\times(N-m)}$ and $m \leq \frac{N}{2}$. We will assume the off-diagonal submatrix A_{12} has rapidly decaying singular values. Thus, A_{12} is a *low-rank matrix* up to a given tolerance $\tau > 0$. Our approach exploits this low-rank property.

To motivate this approach, we introduce the scheme developed in [96]. First compute the Cholesky factorization of A_{11} as $L_{11}L_{11}^{\top}$, and let $L_{21} = A_{12}^{\top}L_{11}^{-\top}$. Then A can be factored as

$$A = \begin{pmatrix} L_{11} & \\ L_{21} & I \end{pmatrix} \begin{pmatrix} L_{11}^{\top} & L_{21}^{\top} \\ & S \end{pmatrix},$$

where $S = A_{22} - L_{21}L_{21}^{\top}$ is the Schur complement. The computation of this factorization can be sped up by taking the truncated SVD of $L_{21}^{\top}$. We have

$$L_{21}^{\top} = (U \quad \hat{U}) \begin{pmatrix} \Sigma & \\ & \hat{\Sigma} \end{pmatrix} \begin{pmatrix} V^{\top} \\ \hat{V}^{\top} \end{pmatrix} = U\Sigma V^{\top} + \hat{U}\hat{\Sigma}\hat{V}^{\top} = U\Sigma V^{\top} + O(\tau), \tag{4.9}$$

where $\Sigma = \operatorname{diag}(\sigma_1, \ldots, \sigma_k)$, $\hat{\Sigma} = \operatorname{diag}(\sigma_{k+1}, \ldots, \sigma_m)$, and $\sigma_k \geq \tau \geq \sigma_{k+1}$. Then $U\Sigma V^{\top}$ is the τ-truncated SVD of $L_{21}^{\top}$ and can be used to approximate the Schur complement S by

$$\tilde{S} = A_{22} - V\Sigma^2 V^{\top}. \tag{4.10}$$

Since $\tilde{S} = A_{22} - L_{21}L_{21}^{\top} + \hat{V}\hat{\Sigma}^2\hat{V}^{\top} = S + \hat{V}\hat{\Sigma}^2\hat{V}^{\top}$, $\tilde{S}$ is always SPD for any tolerance τ. Thus, one can continue the Cholesky factorization on $\tilde{S}$ in the same fashion, and an approximate HSS factorization of A is guaranteed for any $\tau > 0$. See [96] for more details.

In this algorithm, we take a different approach. Instead of keeping track of the approximate Schur complement $\tilde{S}$ throughout the computation, we completely avoid any explicit computation of the Schur complements throughout the generalized Cholesky factorization.

We again use the matrix (4.8) to illustrate the main idea of our new algorithm. To this end, we only factorize part of the first block row. There are two phases in this algorithm: *compression* and *merging*. The main idea is to find an orthonormal matrix $\mathscr{U}$ such that the Cholesky factorization of $\mathscr{U}^{\top}A\mathscr{U}$ be approximately computed without calculating the Schur complement.

Compute the Cholesky factorization $A_{11} = L_1L_1^{\top}$ and an orthogonal decomposition $L_1^{-1}A_{12} = Q_1W_1 + Q_2W_2$, where $Q = [Q_1 \quad Q_2]$ is an orthonormal matrix with $Q_2 \in R^{m\times k}$ and $\|W_1\|_2 = O(\tau)$. Now further compute a QL factorization $U^{(1)}\hat{L} = L_1Q$ which leads to

$$U^{(1)}\hat{L}\left(U^{(1)}\hat{L}\right)^{\top} = L_1Q\left(L_1Q\right)^{\top} = L_1L_1^{\top} = A_{11},$$

and

$$A_{12} = L_1 Q \begin{pmatrix} W_1 \\ W_2 \end{pmatrix} = U^{(1)} \hat{L} \begin{pmatrix} W_1 \\ W_2 \end{pmatrix}.$$

Defining

$$\mathscr{U}_1 := \begin{pmatrix} U^{(1)} & \\ & I \end{pmatrix} \tag{4.11}$$

leads to

$$\mathscr{U}_1^\top \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \mathscr{U}_1 = \begin{pmatrix} \hat{L}\hat{L}^\top & \hat{L} \begin{pmatrix} W_1 \\ W_2 \end{pmatrix} \\ \begin{pmatrix} W_1^\top & W_2^\top \end{pmatrix} \hat{L}^\top & A_{22} \end{pmatrix},$$

and the partitioning

$$\hat{L} = \begin{pmatrix} \hat{L}_{11} & \\ \hat{L}_{21} & \hat{L}_{22} \end{pmatrix}$$

yields

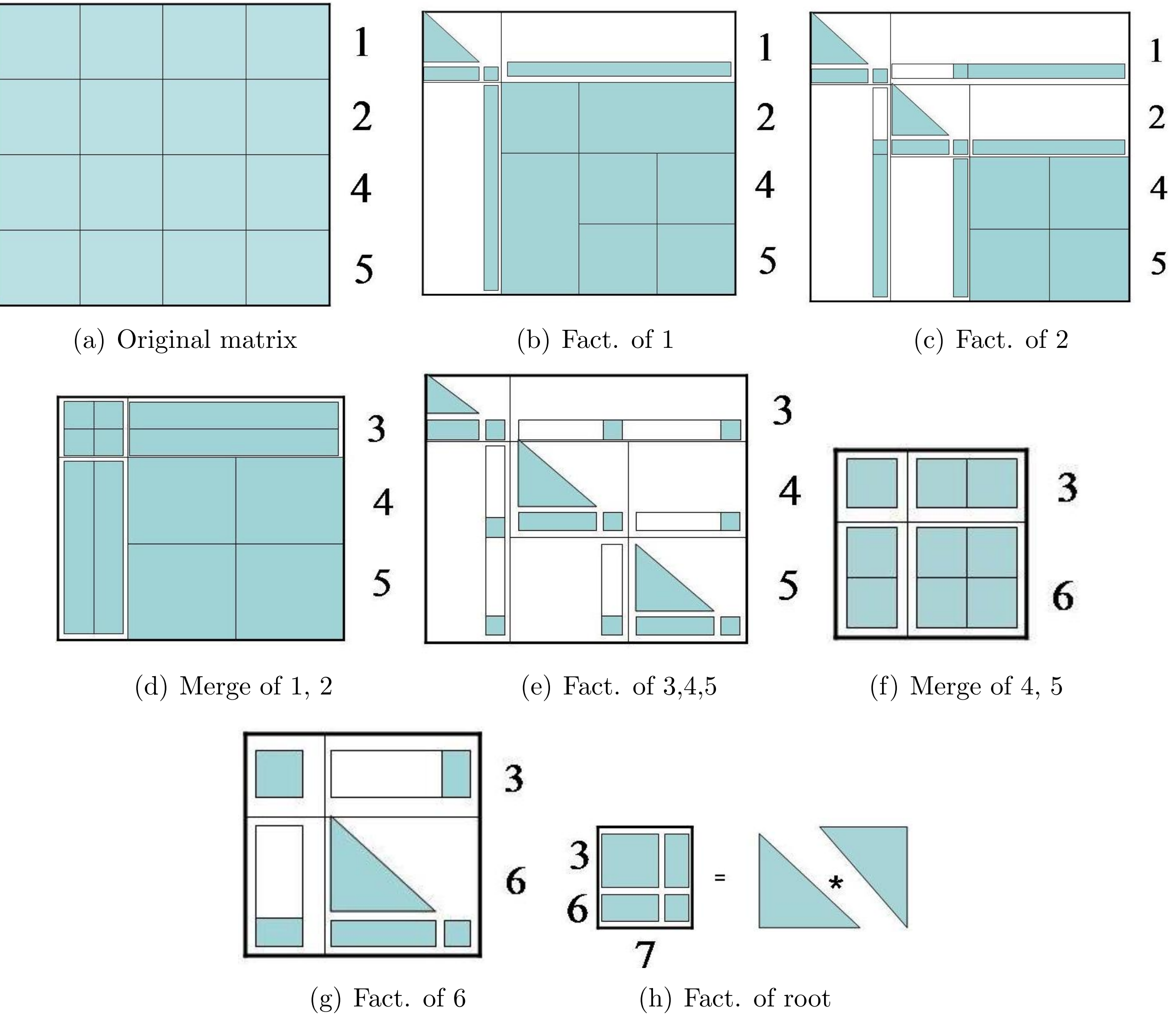
$$\begin{aligned} &\mathscr{U}_1^\top \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \mathscr{U}_1 \\ &= \begin{pmatrix} \hat{L}_{11} & & \\ \hat{L}_{21} & I & \\ W_1^\top & & I \end{pmatrix} \begin{pmatrix} I & & \\ & \hat{L}_{22}\hat{L}_{22}^\top & \hat{L}_{22} W_2 \\ & W_2^\top \hat{L}_{22}^\top & A_{22} - W_1^\top W_1 \end{pmatrix} \begin{pmatrix} \hat{L}_{11} & & \\ \hat{L}_{21} & I & \\ W_1^\top & & I \end{pmatrix}^\top \\ &\approx \begin{pmatrix} \hat{L}_{11} & & \\ \hat{L}_{21} & I & \\ & & I \end{pmatrix} \begin{pmatrix} I & & \\ & \hat{L}_{22}\hat{L}_{22}^\top & \hat{L}_{22} W_2 \\ & W_2^\top \hat{L}_{22}^\top & A_{22} \end{pmatrix} \begin{pmatrix} \hat{L}_{11} & & \\ \hat{L}_{21} & I & \\ & & I \end{pmatrix}^\top. \end{aligned}$$

In the last equation, we have set W_1 to zero in each of the matrices, resulting in an error of $O(\tau)$ in the first and last matrices and an error of $O(\tau^2)$ in the center matrix. Since A is SPD, the center matrix in the last equation is also SPD for any $\tau > 0$. In the following context, we denote the compressed off-diagonal block of node i as $A^{(i)}_{\hat{t}_i t_i^r}$. For example, after the compression of node 1, we may use $A^{(1)}_{\hat{t}_1 t_2}$ to denote $\hat{L}_{22} W_2$. $A^{(1)}_{\hat{t}_1 t_2}$ is a matrix with fewer rows than $A_{t_1 t_2} (= A_{12})$.

This summarizes how to compress the $(1, 2)$ block in a 2×2 block partition setting. In general, the compression process is similar to that of [96]. The main difference of our algorithm is in the need to determine the HSS block row

$$H_i^{row} := \left[A^{(j_1)T}_{\hat{t}_{j_1} t_i}, \ldots, A^{(j_{s_i})T}_{\hat{t}_{j_{s_i}} t_i} \mid A_{t_i t_i^r} \right],$$

where $j_1, j_2, \ldots, j_{s_i}$ are the elements of the visited set $\mathcal{V}_i$ for a node i. Figure 4.4 illustrates how to obtain the HSS block row for $i = 2$. To compress H_i^{row}, we apply the above compression procedure to node i. Below, we summarize the general procedure for leaf nodes i, following the ordering of the postordered tree. Note that since A is symmetric, it is enough to work on the block rows in the upper triangular section of A.



(a) Original matrix (b) Fact. of 1 (c) Fact. of 2

(d) Merge of 1, 2 (e) Fact. of 3,4,5 (f) Merge of 4, 5

(g) Fact. of 6 (h) Fact. of root

Figure 4.3: The factorization process.

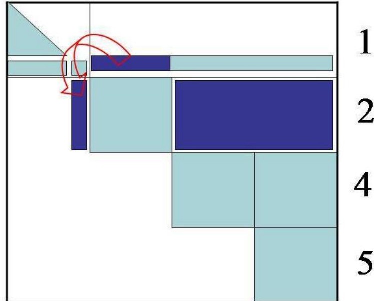


Figure 4.4: The second HSS block row

Algorithm 6 Compressing off-diagonal block rows

Require: SPD matrix A partitioned into n^2 blocks (n leaf nodes in the HSS tree $\mathcal{T}$); off-diagonal block corresponding to node i has m_i rows and rank k_i.

Ensure: Generators $U^{(i)}$, $\hat{L}_{11}^{(i)}$, $\hat{L}_{21}^{(i)}$

1: **for** $i = 0, 1, 2, \ldots,$root$(\mathcal{T})$ **do**

2: **if** i is a leaf node **then**

3: Identify the ith diagonal block $A_{t_i t_i}$ and ith HSS block row

$$H_i^{row} = \left[A_{\hat{t}_{j_1} t_i}^{(j_1)T}, \ldots, A_{\hat{t}_{j_{s_i}} t_i}^{(j_{s_i})T} \mid A_{t_i t_i^r} \right],$$

where $\mathcal{V}_i = \{j_1, j_2, \ldots, j_{s_i}\}$ is the visited set of node i.

4: Compute the Cholesky factorization of $A_{t_i t_i} = L_i L_i^\top$.

5: Compute the orthogonal decomposition $L_i^{-1} H_i^{row} = Q_1 W_1 + Q_2 W_2$, where $Q = [Q_1 \quad Q_2]$ is an orthonormal matrix with $Q_2 \in R^{m_i \times k_i}$, to obtain the *compression* $L_i^{-1} H_i^{row} \approx Q_2 W_2$.

6: Compute $U^{(i)}$ from the QL decomposition $L_i Q = U^{(i)} \hat{L}^{(i)}$, and write

$$\hat{L}^{(i)} = \begin{pmatrix} \hat{L}_{11}^{(i)} & \\ \hat{L}_{21}^{(i)} & \hat{L}_{22}^{(i)} \end{pmatrix},$$

where $\hat{L}_{22}^{(i)} \in R^{k_i \times k_i}$.

7: Compute the HSS block $\widehat{H}_i = \hat{L}_{22}^{(i)} W_2$ and $D_i = \hat{L}_{22}^{(i)} \hat{L}_{22}^{(i)T}$.

8: **end if**

9: **end for**

Remark 4.3.1. *1. The matrices $U^{(i)}$, $\hat{L}_{11}^{(i)}$, $\hat{L}_{21}^{(i)}$, and the scalar k_i are the* generators *of node i and are stored to later reconstruct the preconditioner or be used in the HSS solver. The matrix D_i and remaining HSS block $\widehat{H}_i$ are passed to par(i).*

2. In Step 3 of Algorithm 6, computing Q is a classical low-rank matrix approximation problem with many possible algorithms (RRQR, τ-truncated SVD, SVDR).

3. The HSS block H_i^{row} can be formed with the aid of the visited set $\mathcal{V}_i$: If node i is a left node, push i onto a stack $\mathcal{S}_v$; otherwise, pop an element from the stack. The elements of the stack $\mathcal{S}_v$ are exactly the nodes in $\mathcal{V}_i$ right before i is visited.

4. In Algorithm 6, we compress the off-diagonal block row H_i^{row}. We can similarly compress the off-diagonal block column H_i^{col}.

After compression, the first and second block rows, $A_{\hat{t}_1 t_4}^{(1)}$ and $A_{\hat{t}_2 t_4}^{(2)}$, are of full rank.

However, when the two block rows are merged to form the matrix

$$H_3 = \begin{bmatrix} A^{(1)}_{\hat{t}_1 t_4} \\ A^{(2)}_{\hat{t}_2 t_4} \end{bmatrix},$$

H_3 may be again be of low-rank. Thus, our algorithm hierarchically compresses the off-diagonal blocks. The process is outlined in the next section.

4.3.1 Merging child blocks

For each parent node, we merge the appropriate blocks of its children together and compress it again. Take for example node 3, where the resulting blocks from nodes 1 and 2 are merged to form

$$\mathcal{A}_3 = \begin{bmatrix} D_1 & B_1 & A^{(1)}_{\hat{t}_1 t_4} \\ B_1^\top & D_2 & A^{(2)}_{\hat{t}_2 t_4} \\ A^{(1)\top}_{\hat{t}_1 t_4} & A^{(2)\top}_{\hat{t}_2 t_4} & A_{t_4 t_4} \end{bmatrix}, \tag{4.12}$$

where B_1 is obtained when compressing the second HSS block, see Figure 4.3(c). The size of the original matrix is then reduced. In general, for parent nodes we need to first determine the ith diagonal block D_i and ith HSS block H_i. In the case of node 3,

$$D_3 = \begin{bmatrix} D_1 & B_1 \\ B_1^\top & D_2 \end{bmatrix}, \; H_3 = \begin{bmatrix} A^{(1)}_{\hat{t}_1 t_4} \\ A^{(2)}_{\hat{t}_2 t_4} \end{bmatrix} \tag{4.13}$$

are formed by merging the appropriate blocks of the children, node 1 and node 2. For a general parent node i, the diagonal block D_i and its off-diagonal block H_i are of the form

$$D_i = \begin{bmatrix} D_{i_1} & B_{i_1} \\ B_{i_1}^\top & D_{i_2} \end{bmatrix}, \; H_i = \left[\begin{array}{c|c} [A^{(j_1)\top}_{\hat{t}_{j_1}\hat{t}_{i_1}} \;\cdots\; A^{(j_{s_i})\top}_{\hat{t}_{j_{s_i}}\hat{t}_{i_1}} & A^{(i_1)}_{\hat{t}_{i_1} t_i^r}] \\ \hline [A^{(j_1)\top}_{\hat{t}_{j_1}\hat{t}_{i_2}} \;\cdots\; A^{(j_{s_i})\top}_{\hat{t}_{j_{s_i}}\hat{t}_{i_2}} & A^{(i_2)}_{\hat{t}_{i_2} t_i^r}] \end{array}\right], \tag{4.14}$$

where i_1 and i_2 are the children of node i. The blocks $\left[A^{(j_1)\top}_{\hat{t}_{j_1}\hat{t}_{i_1}}, \ldots, A^{(j_{s_i})\top}_{\hat{t}_{j_{s_i}}\hat{t}_{i_1}}\right]$ and $\left[A^{(j_1)\top}_{\hat{t}_{j_1}\hat{t}_{i_2}}, \ldots, A^{(j_{s_i})\top}_{\hat{t}_{j_{s_i}}\hat{t}_{i_2}}\right]$ make up the leftmost block of HSS block row H_i which makes up the portion in front of D_i in $\mathcal{A}$.

The computation is continued in the manner of the leaf nodes; that is, we Cholesky factorize $D_i = L_i L_i^\top$ and compute the compression $L_i^{-1} H_i \approx Q_2 W_2 + O(\tau)$, where $Q = [Q_1 \; Q_2]$ is orthonormal and $Q_2 \in R^{m_i \times k_i}$. The generators, $U^{(i)}$, $\tilde{L}^{(i)}_{11}$, $\tilde{L}^{(i)}_{21}$ and k_i are computed from $L_i Q = U^{(i)} \tilde{L}^{(i)}$. Traversing the HSS tree $\mathcal{T}$ in postorder, our algorithm alternates between *compressions* and *merges* until arriving at root($\mathcal{T}$). The complete *Generalized HSS Cholesky factorization* algorithm is summarized in Algorithm 7.

See Figure 4.3 for an illustration of the entire process. As seen in Figure 4.3(a), the original matrix A is partitioned into 16 blocks; i.e., there are four leaf nodes in the HSS tree.

Algorithm 7 Generalized HSS Cholesky factorization

Require: SPD matrix A; HSS tree $\mathcal{T}$ with $N_{\mathcal{T}}$ nodes.
Ensure: Generators $U^{(i)}$, $\hat{L}_{11}^{(i)}$, $\hat{L}_{21}^{(i)}$, k_i

1: **for** $i = 1, \ldots, N_{\mathcal{T}} - 1$ **do**
2:   **if** i is a leaf node **then**
3:     Cholesky factorize $A_{t_i t_i} = L_i L_i^\top$, and form the ith HSS block row,

$$H_i^{row} = \left[A_{\hat{t}_{j_1} t_i}^{(j_1)\top}, \ldots, A_{\hat{t}_{j_{s_i}} t_i}^{(j_{s_i})\top} \mid A_{t_i t_i^r} \right],$$

where $\mathcal{V}_i = \{j_1, j_2, \ldots, j_{s_i}\}$ is the visited set of node i.
4:     *Compress* $L_i^{-1} H_i^{row} \approx Q_2 W_2$, where $Q = [Q_1 \quad Q_2]$ is an orthonormal matrix.
5:     Compute $U^{(i)}$ from $L_i Q = U^{(i)} \hat{L}^{(i)}$.
6:     Factorize node i, and compute $\widehat{H}_i = \hat{L}_{22}^{(i)} W_2$, $D_i = \hat{L}_{22}^{(i)} \hat{L}_{22}^{(i)\top}$ (see Algorithm 6).
7:     **if** i is a right node **then**
8:       Construct $B_{\text{sib}\{i\}}$ from $\widehat{H}_i$.
9:     **end if**
10:   **else**
11:     *Merge* D_{i_1}, D_{i_2}, B_{i_1}, $\widehat{H}_{i_1}$, and $\widehat{H}_{i_2}$ to form

$$D_i = \begin{bmatrix} D_{i_1} & B_{i_1} \\ B_{i_1}^\top & D_{i_2} \end{bmatrix}, \; H_i = \left[\begin{array}{ccc|c} [A_{\hat{t}_{j_1} \hat{t}_{i_1}}^{(j_1)\top} & \cdots & A_{\hat{t}_{j_{s_i}} \hat{t}_{i_1}}^{(j_{s_i})\top} & A_{\hat{t}_{i_1} t_i^r}^{(i_1)}] \\ \hline [A_{\hat{t}_{j_1} \hat{t}_{i_2}}^{(j_1)\top} & \cdots & A_{\hat{t}_{j_{s_i}} \hat{t}_{i_2}}^{(j_{s_i})\top} & A_{\hat{t}_{i_2} t_i^r}^{(i_2)}] \end{array} \right].$$

12:     Compute $D_i = L_i L_i^\top$ and compress $L_i^{-1} H_i$ using Algorithm 1, to obtain $U^{(i)}$, $\hat{L}^{(i)}$, k_i, and $\widehat{H}_i$.
13:     **if** i is a right node **then**
14:       Construct $B_{\text{sib}\{i\}}$ from $\widehat{H}_i$.
15:     **end if**
16:   **end if**
17: **end for**
18: Merge $D_{N_{\mathcal{T}1}}$, $D_{N_{\mathcal{T}2}}$, $B_{N_{\mathcal{T}1}}$ to form $D_{N_{\mathcal{T}}}$.
19: Compute the Cholesky factorization $D_{N_{\mathcal{T}}} = L_{N_{\mathcal{T}}} L_{N_{\mathcal{T}}}^\top$.

Figure 4.3(b) represents the factorization of node 1 after compression; note that the first off-diagonal block row has been approximated by a low-rank matrix. The factorization of node 2 is represented in Figure 4.3(c), and the appropriate blocks of node 1 and node 2 are merged to form a smaller matrix (Figure 4.3(d)). Continuing the process in Figure 4.3(e), nodes 3, 4 and 5 are factorized. Nodes 4 and 5 are then merged to form node 6 as seen in Figure 4.3(f). Finally, node 6 is factorized and merged with node 3, which is then in turn factorized (Figure 4.3(h)).

4.3.2 HSS solver with generalized Cholesky factors

We briefly describe the HSS solver proposed in [98] for solving $Ax = b$ where A has a generalized Cholesky factorization organized by an HSS tree $\mathcal{T}$. As in a classical LU decomposition, the HSS solver involves a forward substitution and a backward substitution. Each node i of $\mathcal{T}$ has generators $U^{(i)}$, $\hat{L}_{11}^{(i)}$, $\hat{L}_{21}^{(i)}$ and k_i, where k_i is the approximate rank of node i. To solve the linear system, we traverse the HSS tree $\mathcal{T}$ in postorder to implement forward and backward substitution. We first partition b according to the bottom level (leaf) nodes, and denote the partition corresponding to leaf node i with b_i. Assume there are $N_{\mathcal{T}}$ nodes.

After the forward substitution, each node has updated an b_i. Then, the solution x can be computed from b_i using backward substitution. The procedure is very similar to forward substitution with similar operation counts. We omit the details.

4.3.3 Complexity of construction

Table 4.1: Flops counts of some matrix operations.

Operation	Flops
Cholesky factorization of an $n \times n$ matrix	$\frac{n^3}{3}$
Inverse of an $n \times n$ lower triangular matrix times an $n \times k$ matrix	n^2k
Product of a general $m \times n$ matrix and an $n \times k$ matrix	$2mnk$
QR factorization of an $m \times k$ tall matrix ($m > k$)	$2k^2(m - \frac{k}{3})$
QL factorization for an $n \times n$ matrix	$\frac{4}{3}n^3$
SVD of a general $m \times n$ matrix $A = U\Sigma V^*, m > n$, computing U, Σ	$4m^2n$
Product of an $n \times n$ lower triangular matrix and an $n \times n$ upper triangular matrix	$\frac{2n^3}{3}$

Algorithm 8 Forward Substitution

Require: HSS tree $\mathcal{T}$ with $N_{\mathcal{T}}$ nodes of an SPD matrix A; Generators $U^{(i)}$, $\hat{L}_{11}^{(i)}$, $\hat{L}_{21}^{(i)}$, k_i of the generalized Cholesky factor P of A where $A = PP^T$; b

Ensure: $P^{-1}b$

1: **for** $i = 1, \ldots, N_{\mathcal{T}} - 1$ **do**

2:    **if** i is a leaf node **then**

3:       Compute

$$\hat{b}_i = U^{(i)\top} \cdot b_i = \begin{bmatrix} \hat{b}_{i;1} \\ \hat{b}_{i;2} \end{bmatrix} \begin{matrix} m_i - k_i \\ k_i \end{matrix}, \; b_i = \begin{bmatrix} \hat{L}_{11}^{(i)} & \\ \hat{L}_{21}^{(i)} & I \end{bmatrix}^{-1} \cdot \hat{b}_i = \begin{bmatrix} b_{i;1} \\ b_{i;2} \end{bmatrix} \begin{matrix} m_i - k_i \\ k_i \end{matrix}.$$

4:    **else**

5:       Form b_i from the lower sections of its children, i.e., $b_i = \begin{bmatrix} b_{i_1;2} \\ b_{i_2;2} \end{bmatrix}$, where i_1, i_2 are the left and right child of node i, respectively. Then compute

$$b_i = \begin{bmatrix} \hat{L}_{11}^{(i)} & \\ \hat{L}_{11}^{(i)} & I \end{bmatrix}^{-1} \cdot U^{(i)\top} \cdot b_i = \begin{bmatrix} b_{i;1} \\ b_{i;2} \end{bmatrix} \begin{matrix} m_i - k_i \\ k_i \end{matrix}.$$

6:    **end if**

7: **end for**

8: Compute

$$b_{N_{\mathcal{T}}} = L_{N_{\mathcal{T}}}^{-1} \cdot b_{N_{\mathcal{T}}} \equiv \begin{bmatrix} b_{N_{\mathcal{T}1};2} \\ b_{N_{\mathcal{T}2};2} \end{bmatrix},$$

where $N_{\mathcal{T}1}, N_{\mathcal{T}2}$ are the left and right child of node $N_{\mathcal{T}}$, respectively.

Assume A is an $N \times N$ SPD matrix and has been assigned a full HSS tree $\mathcal{T}$. Furthermore, assume the HSS rank of A is k, and at the bottom level, each leaf node has m rows, where m is of $O(k)$. Then the *generalized HSS Cholesky factorization* method has complexity of $O(N^2 k)$.

We outline the complexity computation here. A detailed computation can be found in [61]. In the following discussion, assume $\mathcal{T}$ is ordered top-down with L levels so that the bottom level is at $L - 1$ and the root is at level 0. Since $\mathcal{T}$ is a full binary tree, $\mathcal{T}$ has $n := 2^{L-1}$ leaf nodes and a total of $2n - 1$ nodes. Moreover, assume all off-diagonal blocks of A have rank k, and each leaf node contains m rows (that is, $N = mn$). Denote the set of leaf nodes by $LN := \{i |$ node i is a leaf node$\}$. We compute the cost level by level. Let N_i be the number of columns in $A_{t_i t_i^r}$. According to Theorem 4.2.2 in [94],

$$\sum_{i \text{ at level } l} s_i = \sum_{j=1}^{l} j \binom{l}{j} = \frac{1}{2} l 2^l, \tag{4.15}$$

$$\sum_{i \text{ at level } l} N_i = \sum_{j=1}^{2^l} \left(n - j\frac{n}{2^l}\right) m = \frac{1}{2} mn \left(2^l - 1\right). \tag{4.16}$$

The following illustrates the computation when using SVD to compress the HSS block rows. The major operations of our generalized HSS Cholesky factorization are as follows.

For each leaf node i (bottom level nodes):

- Cholesky factorization of $A_{ii} = L_i L_i^\top$ requires $\frac{m^3}{3}$ flops.
- Compressing $H_i^\top L_i^{-\top} = Q_i R_i L_i^{-\top}$ requires $2m^2(N_i - \frac{m}{3}) + m^3 + 2m^2 k s_i$ flops, where $H_i \in \mathbb{R}^{m\times(k\times s_i + N_i)}$ and s_i is the cardinality of $\mathcal{V}_i$.
- Computing $W_2^\top = Q_i U_1 \Sigma_1$, where $R_i L_i^{-\top} = [U_1 \quad U_2] \begin{bmatrix} \Sigma_1 & \\ & \Sigma_2 \end{bmatrix} V^\top$ and $V = [V_1 \quad V_2]$ with $V_1 \in R^{m\times k}$, requires $2N_i mk + m^3 + mk + 2mk^2 s_i$ flops.
- Computing $U^{(i)}$ from $L_i Q = U^{(i)} \hat{L}^{(i)}$ where $Q = [V_2 \quad V_1]$ requires $m^3 + \frac{4m^3}{3}$ flops.
- Computing $D_i = \hat{L}_{22}^{(i)} \hat{L}_{22}^{(i)\top}$ and $\widehat{H}_i = \hat{L}_{22}(i) W_2$ requires $\frac{2k^3}{3} + 2N_i k^2 + 2k^3 s_i$ flops.

Therefore, the total cost of all the leaf nodes is approximately $C_f \approx O(N^2 k)$ where $N = mn$ and $m = O(k)$. At level l, there are 2^l parent nodes, and there are a total of $n-1$ non-leaf nodes. The analysis for non-leaf nodes is the same as for leaf nodes. The difference is that the main diagonal block of each parent node is a $2k \times 2k$ matrix; thus, $m = 2k$ in the above flop counts.

The complexity of each non-leaf node (except the root) is $14k^2 N_i + \frac{98}{3}k^3 + 14k^3 s_i$. Summing over the levels between 0 and $L-1$, $C_p \approx O(N^2 k)$ where $n = 2^{L-1}$, $N = mn$, and $m = O(k)$. The complexity of the root node is $C_r = \frac{(2k)^3}{3} < N^2 k$. Thus, the total complexity is $C = C_f + C_p + C_r = 8N^2 k + O(Nk^2)$.

Remark 4.3.2. 1. *The complexity of the algorithm in [96] is also $O(N^2 k)$. However, in our numerical results, our algorithm requires fewer flops when using the same low-rank matrix approximation method for compression.*

2. *With modern computer architectures, floating-point operations are no longer the dominant factor in execution speed. Although the randomized algorithm SVDR requires more flops than RRQR and SVD, in our experience, SVDR is much faster.*

4.4 Incorporating into the Partial Left-Looking Structured Multifrontal Method

In order to incorporate our new algorithm for Cholesky factorization into the partial left-looking structured multifrontal matrix, we use the randomized compression technique ver-

sion. Thus, we make the following modifications. Note that the changes will only occur in the leaf nodes.

1. Since we never explicitly form the update matrices, we only work with the separator block column of the frontal matrix.

2. For each leaf node, we form the i-th diagonal block $A_{t_i t_i}$ and HSS block row $A_{t_i t_i^r}$ on the fly from the pieces in the structured storage of $\mathcal{F}_i$.

3. The off-diagonal piece $A_{t_i t_i^r}$ needs to be multiplied on the right by a random matrix and on the left by the matrix $U_i(:, 1 : p)^T L_i^{-1}$ where p is the numerical rank of the compressed off-diagonal. These operations can be done on the fly without first forming the matrices. In particular, to multiply $A_{t_i t_i^r}$ to a random matrix, we apply the method of Martinsson in [67] for multiplying matrices to a random matrix. An outline of the method is as follows: Given an $N \times N$ matrix A with m^2 blocks, we wish to multiply each of its off-diagonal blocks to a random matrix. We can do the following.

   - Choose a random matrix Ω of size $N \times q$ and compute $B = A\Omega$.
   - For the i-th off-diagonal, obtain the i-th diagonal block of A, A_{ii}, and multiply it to the corresponding rows of Ω. Call this matrix B_{ii}.
   - The i-th off-diagonal matrix of A multiplied to a random matrix is then $B_i - B_{ii}$, where B_i is the i-th block row of B.

4. In our original implementation of the new Cholesky factorization, the intermediate compressed off-diagonal blocks are stored in the original matrix A itself. Since we no longer have an explicit A to store these blocks, we must store them in an alternate fashion that takes up less memory than the size of A. Thus, if the matrix A is of size N^2, we hope to store it in less $O(N^2)$ space. To do this, we store the compressed off-diagonal pieces in a stack structure as we traverse the postordered HSS tree. The method is outlined as follows: In Step i of the new Cholesky factorization,

   - If i is a left-leaf node, create a new stack element with the i-th compressed diagonal and off-diagonal blocks.
   - If i is a right-leaf node, merge the i-th compressed diagonal and off-diagonal blocks to the top stack element to form the merged diagonal/off-diagonal blocks for the parent.
   - If i is a left-parent node, replace the top stack element with the new compressed diagonal/off-diagonal blocks.
   - If i is a right-parent node, merge the i-th compressed diagonal and off-diagonal blocks to the top stack element to form the merged diagonal/off-diagonal blocks for the parent.

4.4.1 Complexity and memory requirements

For a matrix A of size $N \times N$, the Cholesky factorization algorithm we have presented has $O(pN^2)$ complexity where p is an upper bound on the rank of the off-diagonal blocks and $O(pN)$ storage space for the factors. We now show that the complexity and memory requirements are not affected by our modifications. For each of the changes, we have the following added computational and memory costs:

1. No added memory or computational costs.

2. By Lemma 3.3.2, reconstruction of the entire matrix requires $O(N^2p)$ operations. There are no significant extra storage requirements since we only temporarily form subblocks in each iteration.

3. By Lemma 3.3.2, the computation of $A\Omega$ requires $O(N^2q) = O(N^2p)$ operations. The workspace required is $O(qN) = O(pN)$ for the random matrix. The other multiplications do not add extra operations since they already need to be computed.

4. Assuming that HSS tree has height $l = O(\log_2 N)$, as seen in Chapter 3, the stack will have a maximum length of $O(l)$. Since each of the compressed blocks have size $O(pN)$, the maximum size of the stack is of $O(pN \log_2 N)$.

Thus, the total additional computation costs is $O(N^2p) + O(N^2p) = O(N^2p)$, and the total additional workspace required is $O(pN) + O(pN \log_2 N)$. Thus, the total computational complexity does not change, the storage for the factors does not change, but we do require an additional workspace of $O(pN \log_2 N)$.

4.4.2 Schur-monotonicity

As shown in Section 4.3, our generalized Cholesky factorization algorithm is Schur-monotonic (all Schur complements are positive-definite), thus ensuring an algorithm that is always successful. Because of this, our partial left-looking structured multifrontal method is also Schur-monotonic, and a solution is always guaranteed: It is easy to show by induction that after Step i of the multifrontal method, the ith Schur complement of the block Cholesky algorithm can be reassembled by extend-adding all matrices in the current stack to the corresponding sparse submatrix of $\mathcal{A}$. Additionally, the dense frontal matrix factorization in Step i of our new multifrontal method is equivalent to the partial factorization of the ith Schur complement (factorization of the first subblock) in the block Cholesky algorithm. Thus, the multifrontal method is just a reorganization of the block Cholesky algorithm, and since our dense matrix factorization algorithm has been shown to be Schur-monotonic, the partial factorization of the ith Schur complement will be Schur-monotonic as well. The partial left-looking multifrontal algorithm is then guaranteed to be Schur-monotonic.

Part II

An Algorithm for Compressed Sensing

Chapter 5

Introduction

Signal processing typically involves massive data sets. Most data sets are compressible in the sense that they have a sparse representation under a particular basis. The process of acquiring large amounts of data only to discard most of it (through compression) is wasteful in terms of space and time. Compressed sensing is the process of simultaneously acquiring and compressing data. In particular, $k << n$ measurements of an unknown signal of size n are taken in such a way that the unknown signal may be recovered. In this part of the dissertation, we propose a new algorithm which may be used to recover the unknown signal in compressed sensing problems.

Here, we present an algorithm, NESTA-LASSO, for the LASSO problem, i.e., an underdetermined linear least-squares problem with a 1-norm constraint on the solution:

$$\text{LS}(\tau) \quad \min \|Ax - b\|_2 \quad \text{s.t.} \quad \|x\|_1 \le \tau. \tag{5.1}$$

We prove under the assumption of the *restricted isometry property* (RIP) and a sparsity condition on the solution, that NESTA-LASSO is guaranteed to be almost always locally linearly convergent. As in the case of the algorithm NESTA, proposed by Becker, Bobin, and Candès, we rely on Nesterov's accelerated proximal gradient method, which takes $O(\sqrt{1/\varepsilon})$ iterations to come within $\varepsilon > 0$ of the optimal value. We introduce a modification to Nesterov's method that regularly updates the prox-center in a provably optimal manner. The aforementioned linear convergence is in part due to this modification.

Next, we attempt to solve the basis pursuit denoising (BPDN) problem (i.e., approximating the minimum 1-norm solution to an underdetermined least squares problem) by using NESTA-LASSO in conjunction with the Pareto root-finding method employed by van den Berg and Friedlander in their SPGL1 solver. The resulting algorithm is called PARNES. We provide numerical evidence to show that it is comparable to currently available solvers.

5.1 Problem Statement and Background

We would like to find a solution to the sparsest recovery problem with noise

$$\min \|x\|_0 \quad \text{s.t.} \quad \|Ax - b\|_2 \leq \sigma. \tag{5.2}$$

Here, σ specifies the noise level, A is an m-by-n matrix with $m \ll n$, and $\|x\|_0$ is the number of nonzero entries of x. This problem comes up in fields such as image processing [80], seismics [58, 57], astronomy [16], and model selection in regression [31]. Since (5.7) is known to be ill-posed and NP-hard [38, 70], various convex, l_1-relaxed formulations are often used.

Relaxing the 0-norm in (5.7) gives the basis pursuit denoising (BPDN) problem

$$\text{BP}(\sigma) \quad \min \|x\|_1 \quad \text{s.t.} \quad \|Ax - b\|_2 \leq \sigma. \tag{5.3}$$

The special case of $\sigma = 0$ is the basis pursuit problem [25]. Two other commonly used l_1-relaxations are the LASSO problem [85]

$$\text{LS}(\tau) \quad \min \|Ax - b\|_2 \quad \text{s.t.} \quad \|x\|_1 \leq \tau \tag{5.4}$$

and the penalized least-squares problem

$$\text{QP}(\lambda) \quad \min \|Ax - b\|_2^2 + \lambda \|x\|_1 \tag{5.5}$$

proposed by Chen, Donoho, and Saunders [25]. A large amount of work has been done to show that these formulations give an effective approximation of the solution to (5.7); see [28, 86, 17]. In fact, under certain conditions on the sparsity of the solution to (5.7), these formulations can exactly recover the solution whenever A satisfies the *restricted isometry property* (RIP).

There is a wide variety of algorithms which solve the BP(σ), QP(λ), and LS(τ) problems. Refer to Section 8.1 for descriptions of some of the current algorithms. Our work has been motivated by the accuracy and speed of the recent solvers NESTA and SPGL1. In [71], Nesterov presents an algorithm to minimize a smooth convex function over a convex set with an optimal convergence rate. An extension to the nonsmooth case is presented in [73]. NESTA solves the BP(σ) problem using the nonsmooth version of Nesterov's work.

For appropriate parameter choices of σ, λ, and τ, the solutions of BP(σ), QP(λ), and LS(τ) coincide [12]. Although the exact dependence is usually hard to compute [12], there are solution methods which exploit these relationships. The MATLAB solver SPGL1 is based on the Pareto root-finding method [12] which solves BP(σ) by approximately solving a sequence of LS(τ) problems. In SPGL1, the LS(τ) problems are solved using a spectral projected-gradient (SPG) method.

While we are ultimately interested in solving the BPDN problem in (5.9), our main result is an algorithm for solving the LASSO problem (5.11). Our algorithm, NESTA-LASSO (cf. Algorithm 11), essentially uses Nesterov's work to solve the LASSO problem. We introduce one improvement to Nesterov's original method, namely, we update the prox-center every

K steps instead of fixing it throughout the algorithm. With this modification, we prove in Theorem 6.1.8 that NESTA-LASSO is guaranteed to be almost always locally linearly convergent for sufficiently large K, as long as the solution is s-sparse and A satisfies the restricted isometry property of order $2s$. In fact, Theorem 6.1.8 also provides the choice for the optimal K.

Finally, we show that replacing the SPG method in the Pareto root-finding procedure, used in SPGL1, with our NESTA-LASSO method leads to an effective method for solving $\text{BP}(\sigma)$. We call this modification PARNES and compare its efficacy with the state-of-the-art solvers presented in Section 8.1.

5.1.1 Motivation

In the field of digital technology one hopes to efficiently find *digital* representations of *analog* signals, such as images and audio. The classical two stage approach consists first of collecting measurements of the analog signal. The number of measurements can be quite large depending on the desired resolution. In the case of an image, the number of measurements can be in the millions. In the second step, called *source coding*, the measurements are quantized, and the resulting signal is sparsified and compressed using an appropriate basis - e.g., Fourier, discrete cosine transform. In some cases, even sparser representations may be found by using a *redundant dictionary.* These dictionaries consist of unions of different bases. Mathematically, the aim is to have

$$f \approx Bx \tag{5.6}$$

where $f \in \mathbb{R}^k$ is the sampled signal and $B \in \mathbb{R}^{k\times n}$, $k \leq n$, represents a *redundant dictionary* under which there is a *compressed signal* x which is sparse. Since such dictionaries do not have a unique representation of f, they would only be useful if computationally tractable methods were available to recover a sparse x.

The method described above can be somewhat inefficient since most of the collected data is disregarded at the compression stage. In fact, for settings such as magnetic resonance imaging (MRI), it can be inconvenient to take large amounts of measurements. Thus, in such cases where measurements are difficult, one would like a method which only samples what is necessary without affecting the quality of the resulting signal. The ideas of *compressed sensing* aim to sample and compress in one stage.

Compressed sensing techniques may be useful in applications where signals are sparse in a known basis and/or measurements are expensive but computations are cheap. Some useful setting include image processing [80], seismics [58, 57], astronomy [16], and model selection in regression [31].

To understand the basics of compressed sensing, suppose $f \in \mathbb{R}^k$ is a signal that admits a sparse representation under the redundant dictionary $B \in \mathbb{R}^{k\times n}$. The idea of compressed sensing is to choose a measurement matrix $M \in \mathbb{R}^{m\times k}$ with $m \leq k$ which will allow us to recover the sparse signal x. In turn, this will let us approximately recover $f = Bx$. We have

$$b \approx Mf \quad \text{with} \quad f = Bx$$

where $b \in \mathbb{R}^m$ is the set of measurements. In practice one would like to choose M so that $m \ll k$.

Setting $A = MB$, compressed sensing hopes to recover x by solving the optimization problem

$$\min \|x\|_0 \quad \text{s.t.} \quad \|Ax - b\|_2 \leq \sigma, \tag{5.7}$$

where σ is a bound on the noise level of b. Since this problem is known to be NP-hard, the various l_1-relaxed formulations described below are often used.

5.1.2 Various l_1-relaxed formulations

An l_1-relaxation is naturally motivated by the fact that $\|\cdot\|_1$ is the largest convex underestimator of $\|\cdot\|_0$ on unit l_∞-ball. There has been a large amount of work done to show that this gives an effective approximation of the solution to (5.7); see [17, 28, 86]. In particular, under certain conditions on the sparsity of the solution x, x can be recovered exactly provided that the matrix A satisfies the *restricted isometry property*; see Appendix 3. We will make this and some other reasonable assumptions on A to prove the linear convergence of PARNES.

Relaxing the zero-norm in (5.7) gives the basis pursuit problem [25]

$$\text{BP} \quad \min \|x\|_1 \quad \text{s.t.} \quad Ax = b. \tag{5.8}$$

Similarly, relaxing the zero-norm in (5.7) gives the basis pursuit denoise (BPDN) problem

$$\text{BP}(\sigma) \quad \min \|x\|_1 \quad \text{s.t.} \quad \|Ax - b\|_2 \leq \sigma. \tag{5.9}$$

The special case of $\sigma = 0$ gives the basis pursuit problem.

Alternatively, there are two other commonly used l_1-relaxations. The penalized least-squares problem

$$\text{QP}(\lambda) \quad \min \|Ax - b\|_2^2 + \lambda\|x\|_1 \tag{5.10}$$

is one possible relaxation proposed by Chen, Donoho, and Saunders [25]. Another formulation is the LASSO problem [85] with the one-norm constrained by a parameter τ

$$\text{LS}(\tau) \quad \min \|Ax - b\|_2 \quad \text{s.t.} \quad \|x\|_1 \leq \tau. \tag{5.11}$$

Note that among all the formulations, BP(σ) is often the most natural one in applications. This is because one often has an idea of what the noise level σ is, while it is not so clear what λ and τ should be. Also note that the name, basis pursuit denoise, is often applied to what we call the penalized least-squares problem, originally proposed by Chen, Donoho, and Saunders in [25]. We will follow the naming convention in the SPGL1 and NESTA papers [12, 9].

Such formulations are useful since they are all convex optimization problems. In particular, BP is a linear program while QP(λ), LS(τ), and BP(σ) are quadratic programming problems. In fact, for appropriate parameter choices of σ, λ, and τ, the solutions of BP(σ),

QP(λ), and LS(τ) coincide [78]. The exact dependence is hard to compute unless A is orthogonal [12]. However, there are solution methods which exploit these relationships. In fact, PARNES is based on the algorithm SPGL1 [12] which solves BP(σ) by solving a sequence of LS(τ) problems. In SPGL1, the LS(τ) problems are solved using a spectral gradient-projection method whereas we use a method based on the ideas in NESTA [9].

There is now a wide variety of available algorithms which solve the BP(σ), QP(λ), and LS(τ) problems. The NESTA algorithm is part of a larger class of algorithms often described as promixal gradient methods. Algorithms such HOMOTOPY [74, 75] and LARS [31] belong to an early class of methods using a homotopy method; the solution to BP(σ) is found by solving QP(λ) for various values of λ. Another class consists of using Bregman iterative procedures. An example of this type of algorithm is the fixed-point continuation method, FPC [54, 55], which is tested in our experiments.

We focus on solving the relaxation BP(σ) where σ is an estimate of the noise level in the data. We present an algorithm that solves BP(σ) for any value of $\sigma \geq 0$ by combining certain features of the available solvers SPGL1 and NESTA.

As mentioned above, SPGL1 approaches the solution of BP(σ) by solving LS(τ) for a sequence of τ's. We incorporate it into PARNES since we have observed that it is suitable for large-scale problems and faster than many other solvers. In SPGL1, the BP(σ) problem is interpreted as finding the root of a single-variable nonlinear equation. An approximate Newton's method is used, and each iteration involves solving the LASSO problem and its dual.

The algorithm NESTA solves the BP(σ) problem based on a method by Nesterov which has been shown to achieve an optimal convergence rate [73, 71]. Combined with continuation techniques [54, 55], the algorithm is experimentally shown to be accurate, robust and comparable in speed to SPGL1. In PARNES, we use the methods of Nesterov to approximately solve the LASSO problem in each iteration of our Newton's iteration.

5.1.3 Notation, terminology, and assumptions

In this paper, a vector is s-sparse if it has exactly s nonzero elements. We say that a vector is *at least* s-sparse if it has at most s nonzero elements. For a nonzero, s-sparse vector $x \in \mathbb{R}^n$, let I_x be the set of indices of the nonzero coefficients of x, i.e. the support of x; $\overline{x}$ is the vector containing the nonzero elements of x. For an $I \subseteq \{1, \ldots, n\}$, I^c is the complement of I. Given a matrix $A \in \mathbb{R}^{m \times n}$ and $I \subseteq \{1, \ldots, n\}$, A_I is the submatrix of A containing the j-th columns of A where $j \in I$. Throughout the paper, we use MATLAB terminology to describe vectors and matrices. Thus, $x[s : r]$ represents the subvector of x containing elements s to r. For a set S, let $\mathrm{int}(S)$ be the interior of S and ∂S be the boundary of S.

Throughout the paper, we make the blanket assumption that $b \in \mathrm{range}(A)$. That is, $Ax - b = 0$ is always possible. In many applications, A has full rank and therefore automatically satisfies this assumption; see [12].

5.1.4 Organization of Part II of dissertation

In Section 5.2, we present and describe the background of NESTA-LASSO. We show in Section 6.1 that, under some reasonable assumptions, NESTA-LASSO is almost always locally linearly convergent. In Section 7.1, we describe the Pareto root-finding procedure behind the BPDN solver SPGL1 and show how NESTA-LASSO can be used to solve a subproblem. Section 5 describes some of the available algorithms for solving BPDN and the equivalent QP(λ) problem. Lastly, in Section 6, we show in a series of numerical experiments that using NESTA-LASSO in SPGL1 to solve BPDN is comparable with current competitive solvers.

5.2 NESTA-LASSO

We present the main parts of our method to solve the LASSO problem. Our algorithm, NESTA-LASSO (cf. Algorithm 11), is an application of the accelerated proximal gradient algorithm of Nesterov [71] outlined in Section 5.2.1. Additionally, we have a prox-center update improving convergence which we describe in Section 6.1. In each iteration, we use the fast l_1-projector of Duchi et al. [29] given in Section 5.2.3.

5.2.1 Nesterov's algorithm

Let $Q \subseteq \mathbb{R}^n$ be a convex closed set. Let $f : Q \to \mathbb{R}$ be smooth, convex and, Lipschitz differentiable with L as the Lipschitz constant of its gradient, i.e.

$$\|\nabla f(x) - \nabla f(y)\|_2 \le L\|x - y\|_2, \quad \text{for all } x, y \in Q.$$

Nesterov's accelerated proximal gradient algorithm iteratively defines a sequence x_k as a judiciously chosen convex combination of two other sequences y_k and z_k, which are in turn solutions to two quadratic optimization problems on Q. The sequence z_k involves a strongly convex *prox-function*, $d(x)$, which satisfies

$$d(x) \ge \frac{\alpha}{2}\|x - c\|_2^2. \tag{5.12}$$

For simplicity, we have chosen the right-hand side of (5.12) with $\alpha = 1$ as our prox-function throughout this paper. The c in the prox-function is called the *prox-center*. With this prox-function, we have:

$$\begin{aligned}
y_k &= \operatorname*{argmin}_{y \in Q} \nabla f(x_k)^\top (y - x_k) + \frac{L}{2}\|y - x_k\|_2^2, \\
z_k &= \operatorname*{argmin}_{z \in Q} \sum_{i=0}^{k} \frac{i+1}{2}[f(x_i) + \nabla f(x_i)^\top (z - x_i)] + \frac{L}{2}\|z - c\|_2^2, \\
x_k &= \frac{2}{k+3} z_k + \frac{k+1}{k+3} y_k.
\end{aligned}$$

Nesterov showed that if x^* is the optimal solution to

$$\min_{x \in Q} f(x),$$

then the iterates defined above satisfy

$$f(y_k) - f(x^*) \leq \frac{L}{k(k+1)} \|x^* - c\|_2^2 = O\left(\frac{L}{k^2}\right).$$

An implication is that the algorithm requires $O(\sqrt{L/\varepsilon})$ iterations to bring $f(y_k)$ to within $\varepsilon > 0$ of the optimal value.

Algorithm 9 Accelerated proximal gradient method for convex minimization

Require: function f, gradient ∇f, Lipschitz constant L, prox-center c.
Ensure: $x^* = \operatorname{argmin}_{x \in Q} f(x)$

1: initialize x_0;
2: **for** $k = 0, 1, 2, \ldots,$ **do**
3: compute $f(x_k)$ and $\nabla f(x_k)$;
4: $y_k = \operatorname{argmin}_{y \in Q} \nabla f(x_k)^\top (y - x_k) + \frac{L}{2}\|y - x_k\|_2^2$;
5: $z_k = \operatorname{argmin}_{z \in Q} \sum_{i=0}^{k} \frac{i+1}{2}[f(x_i) + \nabla f(x_i)^\top (z - x_i)] + \frac{L}{2}\|z - c\|_2^2$;
6: $x_k = \frac{2}{k+3} z_k + \frac{k+1}{k+3} y_k$;
7: **end for**

In [73], Nesterov extends his work to minimize nonsmooth convex functions f. Nesterov shows that one can obtain the minimum by applying his algorithm for smooth minimization to a smooth approximation f_μ of f. Since ∇f_μ is shown to have Lipschitz constant $L_\mu = 1/\mu$, if μ is chosen to be proportional to ε, it takes $O\left(\frac{1}{\varepsilon}\right)$ iterations to bring $f(x_k)$ within ε of the optimal value.

The recent algorithm NESTA solves BP(σ) using Nesterov's algorithm for nonsmooth minimization. Our algorithm, NESTA-LASSO, solves LS(τ) using Nesterov's smooth minimization algorithm. In [72], Nesterov suggests an algorithm for minimizing composite functions which has a complexity of $O(\frac{1}{\epsilon^{1/2}})$. We are motivated by the accuracy and speed of NESTA, and the fact that the smooth version of Nesterov's algorithm has a faster convergence rate than the nonsmooth version.

5.2.2 NESTA-LASSO-K: An accelerated proximal gradient algorithm for LASSO

We apply Nesterov's accelerated proximal gradient method, Algorithm 9, to the LASSO problem LS(τ). We make one slight improvement to Algorithm 9. Namely, we update our prox-centers every K steps (cf. Algorithm 10); that is, Algorithm 9 is restarted every K iterations

with a new prox-center. We will see that this leads to local linear convergence under a suitable application of RIP (see Corollary 6.1.9 for details). In fact, we show in Section 6.1 that the prox-centers may be updated in an optimal fashion (cf. Algorithm 11).

In our case, $f = \frac{1}{2}\|b - Ax\|_2^2$, $\nabla f = A^\top(Ax - b)$, and Q is the 1-norm ball $\|x\|_1 \leq \tau$. The initial point x_0 is used as the prox-center c. To compute the iterate y_k, we have

$$\begin{aligned} y_k &= \underset{\|y\|_1 \leq \tau}{\operatorname{argmin}} \nabla f(x_k)^\top (y - x_k) + \frac{L}{2}\|y - x_k\|_2^2 \\ &= \underset{\|y\|_1 \leq \tau}{\operatorname{argmin}}\, y^\top y - 2(x_k - \nabla f(x_k)/L)^\top y \\ &= \underset{\|y\|_1 \leq \tau}{\operatorname{argmin}} \|y - (x_k - \nabla f(x_k)/L)\|_2 \\ &= \operatorname{proj}_1(x_k - \nabla f(x_k)/L, \tau) \end{aligned}$$

where $\operatorname{proj}_1(v, \tau)$ returns the projection of the vector v onto the 1-norm ball of radius τ. By similar reasoning, computing z_k can be shown to be equivalent to computing

$$z_k = \operatorname{proj}_1\left(c - \frac{1}{L}\sum_{i=0}^{k} \frac{i+1}{2}\nabla f(x_i), \tau\right).$$

In each iteration, we use the fast l_1-projector proj_1 described in the next section.

In NESTA-LASSO-K, Nesterov's method is restarted every K steps with the new prox-center $\operatorname{proj}_1(y_{iK} - \nabla f(y_{iK})/L, \tau)$. Here, y_{iK-1} is the K-th iterate of Nesterov's method after the i-th prox-center change; see Algorithm 10. In NESTA-LASSO, Nesterov's method is restarted in the same manner, except K is chosen in an optimal way. Algorithms 2 and 3 are stopped when the duality gap η_k is sufficiently small.

5.2.3 l_1-projector

The projection of an n-vector, d, onto the 1-norm ball, $\|x\|_1 \leq \tau$, is the solution to the minimization problem

$$\operatorname{proj}_1(d, \tau) := \underset{x}{\operatorname{argmin}} \|d - x\|_2 \quad \text{s.t.} \quad \|x\|_1 \leq \tau.$$

Let $\hat{d}$ be a reordering of d with $|\hat{d}_1| \geq ... \geq |\hat{d}_n|$. Then $a = \operatorname{proj}_1(d, \tau)$, is given by

$$a_i = \operatorname{sgn}(d_i) \cdot \max\{0, |d_i| - \eta\} \quad \text{with} \quad \eta = \frac{(|\hat{d}_1| + \cdots + |\hat{d}_k|) - \tau}{k}, \tag{5.13}$$

where k is the largest index such that $\eta \leq |\hat{d}_k|$.

See [29], by Duchi et al., and [12] for fast algorithms to compute a. Such algorithms cost $O(n \log n)$ in the worst case but have been shown experimentally to cost much less [12]. The results in [87] imply the two calls to proj_1 in the inner loop of NESTA-LASSO can be reduced to one call, but due to the low cost of proj_1, we do not make this modification.

Algorithm 10 NESTA-LASSO-K algorithm with prox-center updates every K steps

Require: initial point x_0, LASSO parameter τ, tolerance η, steps to update K
Ensure: $x_\tau = \operatorname{argmin}\{\|b - Ax\|_2 : \|x\|_1 \le \tau\}$.

1: **for** $j = 0, \ldots, j_{\max}$, **do**
2: $c_j = x_0$, $h_0 = 0$, $r_0 = b - Ax_0$, $g_0 = -A^\top r_0$, $\eta_0 = \|r_0\|_2 - (b^\top r_0 - \tau\|g_0\|_\infty)/\|r_0\|_2$;
3: **for** $k = 0, \ldots, K - 1$ **do**
4: $y_k = \operatorname{proj}_1(x_k - g_k/L, \tau)$;
5: $h_k = h_k + \frac{k+1}{2} g_k$;
6: $z_k = \operatorname{proj}_1(c_j - h_k/L, \tau)$;
7: $x_k = \frac{2}{k+3} z_k + \frac{k+1}{k+3} y_k$;
8: $r_k = b - Ax_k$;
9: $g_k = -A^\top r_k$;
10: $\eta_k = \|r_k\|_2 - (b^\top r_k - \tau\|g_k\|_\infty)/\|r_k\|_2$;
11: **end for**
12: $x_0 = \operatorname{proj}_1(y_k + A^\top(b - Ay_k)/L, \tau)$;
13: **if** $\eta_k \le \eta$ **then**
14: **return** $x_\tau = y_k$;
15: **end if**
16: **end for**

Algorithm 11 NESTA-LASSO algorithm with optimal prox-center updates

Require: initial point x_0, LASSO parameter τ, tolerance η.

Ensure: $x_\tau = \text{argmin}\{\|b - Ax\|_2 : \|x\|_1 \leq \tau\}$.

1: **for** $j = 0, \ldots, j_{\max}$, **do**
2: $c_j = x_0$, $h_0 = 0$, $r_0 = b - Ax_0$, $g_0 = -A^\top r_0$, $\eta_0 = \|r_0\|_2 - (b^\top r_0 - \tau\|g_0\|_\infty)/\|r_0\|_2$;
3: **for** $k = 0, \ldots, k_{\text{opt}} - 1$, **do**
4: **if** $\eta_k \leq e^{-2}\eta_0$ **then**
5: **return** y_k, η_k
6: **end if**
7: $y_k = \text{proj}_1(x_k - g_k/L, \tau)$;
8: $h_k = h_k + \frac{k+1}{2} g_k$;
9: $z_k = \text{proj}_1(c_j - h_k/L, \tau)$;
10: $x_k = \frac{2}{k+3} z_k + \frac{k+1}{k+3} y_k$;
11: $r_k = b - Ax_k$;
12: $g_k = -A^\top r_k$;
13: $\eta_k = \|r_k\|_2 - (b^\top r_k - \tau\|g_k\|_\infty)/\|r_k\|_2$;
14: **end for**
15: $x_0 = \text{proj}_1(y_k + A^\top(b - Ay_k)/L, \tau)$;
16: **if** $\eta_k \leq \eta$ **then**
17: **return** $x_\tau = y_k$;
18: **end if**
19: **end for**

Chapter 6

Convergence Proofs

6.1 Local Linear Convergence and Optimality

Under reasonable assumptions on the matrix A and the solution x^* of the LASSO problem, we prove that NESTA-LASSO-K almost always has a local linear convergence rate for large enough K. We also show that we can update the prox-centers c in a provably optimal way (NESTA-LASSO). Let y_k be the k-th iterate of Nesterov's accelerated proximal gradient method when minimizing a function f. Recall,

$$f(y_k) - f(x^*) \leq \frac{L}{k(k+1)} \|x^* - c\|_2^2 \tag{6.1}$$

where L is the Lipschitz constant for ∇f and c is the prox-center [71, 73].

In our case, $f(x) = \frac{1}{2}\|Ax - b\|_2^2$, where $A \in \mathbb{R}^{m \times n}$ with $m < n$. We will assume that A satisfies the *restricted isometry property* (RIP) of order $2s$ as described in [18, 84]. Namely, there exists a constant $\delta_{2s} \in (0, 1)$ such that

$$(1 - \delta_{2s})\|x\|_2^2 \leq \|Ax\|_2^2 \leq (1 + \delta_{2s})\|x\|_2^2 \tag{6.2}$$

whenever $\|x\|_0 \leq 2s$. Since the RIP helps ensure that the solution to (5.7) is closely approximated by the solution to (5.9) [84], and we are ultimately interested in solving (5.9), this is a reasonable assumption. Moreover, since we hope to recover the sparse solution to the solution to (5.7), we assume that the solution x^* to the LASSO problem is s-sparse. We plan to analyze the approximately sparse case for future work.

Under these assumptions, the LASSO problem has a unique solution (see Theorem 5 in [75]). Since the 1-norm ball is compact, the sequence of y_k's converges to the solution x^*.

Lemma 6.1.1. *If A satisfies the restricted isometry property* (RIP) *of order $2s$, and the optimal solution x^* is s-sparse, then the sequence of y_k's converges to x^*.*

6.1.1 Almost sure sparsity of Nesterov's method

We first state and prove the following results before proving our main results, i.e. the local linear convergence of NESTA-LASSO-K and the optimality of NESTA-LASSO. In particular, we show that under certain assumptions on the LASSO problem, the solution is almost always non-degenerate (see Proposition 6.5), and the iterates of Algorithm 9 are almost always eventually s-sparse. Our first lemma describes when the image of proj_1 is s-sparse.

For $d \in \mathbb{R}^n$, recall from Section 5.2.3 that if $\hat{d}$ is a reordering of d with $|\hat{d}_1| \geq ... \geq |\hat{d}_n|$, then $a = \text{proj}_1(d, \tau)$ is given by

$$a_i = \text{sgn}(d_i) \cdot \max\{0, |d_i| - \eta\} \quad \text{with} \quad \eta = \frac{(|\hat{d}_1| + \cdots + |\hat{d}_k|) - \tau}{k}, \tag{6.3}$$

where k is the largest index such that $\eta \leq |\hat{d}_k|$. For each $i \in \{1, \ldots, n\}$, define

$$\eta_i := \frac{|\hat{d}_1| + \cdots + |\hat{d}_i| - \tau}{i}.$$

The η_i's satisfy the following property which is used in the proof of our next lemma.

Claim 6.1.2. $\eta = \max\{\eta_i : i = 1, \ldots, n\}$.

Proof. Assume, without loss of generality, that $d \geq 0$ and $d = \hat{d}$ so that $d_1 \geq ... \geq d_n \geq 0$. A simple algebraic manipulation shows that $\eta_i - \eta_{i-1} = \frac{1}{i-1}(d_i - \eta_i)$ for $i \in \{2, \ldots, n\}$. Thus, $\text{sgn}(\eta_i - \eta_{i-1}) = \text{sgn}(d_i - \eta_i)$. Suppose $\eta = \eta_k$ for some k. Then $\eta_k \leq d_k$. Since $\text{sgn}(\eta_i - \eta_{i-1}) = \text{sgn}(d_i - \eta_i)$, it follows that $\eta_{k-1} \leq \eta_k$ and so $\eta_{k-1} \leq d_{k-1}$; thus, we can repeatedly apply this argument to show that $\eta_i \leq \eta_k$ for any $i < k$. A similar argument shows that $\eta_i \leq \eta_k$ for any $i > k$. $\square$

Given a nonempty $I \subseteq \{1, \ldots, n\}$ with $|I| = s$ and $\tau > 0$, if $s < n$, define the set

$$C_{I,\tau} := \left\{x \in \mathbb{R}^n : \sum\nolimits_{i \in I} |x_i| - \tau \geq s \cdot |x_j| \text{ for } j \notin I\right\}.$$

If $I = \{1, \ldots, n\}$, let $C_{I,\tau} := \{x \in \mathbb{R}^n : \|x\|_1 \geq \tau\}$. The following lemma shows that proj_1 sends vectors in $C_{I,\tau}$ to vectors that are *at least* s-sparse.

Lemma 6.1.3. *Suppose* $I \subseteq \{1, \ldots, n\}$ *with* $|I| = s$. *If* $d \in C_{I,\tau}$ *then* $I_{\text{proj}_1(d,\tau)} \subseteq I$. *Namely,* $\text{proj}_1(d, \tau)$ *is at least* s*-sparse.*

Proof. Suppose $d \in C_{I,\tau}$. Assume, without loss of generality, that $d \geq 0$ and $d = \hat{d}$ so that $d_1 \geq ... \geq d_n \geq 0$. For simplicity, let $[1], \ldots, [n]$ be a labeling of the indices of d so that $I = \{[1], \ldots, [s]\}$, $d_{[1]} \geq \ldots \geq d_{[s]}$, and $d_{[s+1]} \geq \ldots \geq d_{[n]}$.

By (6.3), $a_{[s+1]} \geq \ldots \geq a_{[n]}$, so it is enough to show that $a_{[s+1]} = 0$. Since $d \in C_{I,\tau}$,

$$s \cdot d_{[s+1]} \leq d_{[1]} + \ldots + d_{[s]} - \tau. \tag{6.4}$$

Let $r \leq s$ be the largest index such that $d_{[r]} \geq d_{[s+1]}$. Such an r exists since $s \cdot d_{[1]} \geq d_{[1]} + \cdots + d_{[s]} - \tau \geq s \cdot d_{[s+1]}$. By (6.4),

$$\begin{aligned} r \cdot d_{[s+1]} &\leq d_{[1]} + \cdots + d_{[r]} + (d_{[r+1]} - d_{[s+1]}) + \cdots + (d_{[s]} - d_{[s+1]}) - \tau \\ &\leq d_{[1]} + \cdots + d_{[r]} - \tau, \end{aligned}$$

which implies,

$$d_{[s+1]} \leq \frac{d_{[1]} + \cdots + d_{[r]} + d_{[s+1]} - \tau}{r+1} = \eta_{r+1}.$$

The last equality holds since we assumed that r is the largest index such that $r \leq s$ and $d_{[r]} \geq d_{[s+1]}$. Thus, $d_{[1]} + \cdots + d_{[r]} + d_{[s+1]} = d_1 + \cdots + d_r + d_{r+1}$. By the above claim, $d_{[s+1]} \leq \eta$, and so $a_{[s+1]} = 0$ by definition of $a_{[s+1]}$.

□

The next few lemmas involve the LASSO problem. First note the following LASSO optimality conditions (see e.g. [36] and [37]).

Proposition 6.1.4 (LASSO optimality conditions)**.** *For an $x^* \in \mathbb{R}^n$, let $I = I_{x^*}$. Then x^* is the optimal solution to the* LASSO *problem if and only if the gradient, $-\nabla f(x^*) = A^\top(b - Ax^*)$, at x^* satisfies*

$$A_I^\top(b - A_I\overline{x}^*) = \gamma \cdot \operatorname{sgn}(\overline{x}^*), \tag{6.5}$$

$$\|A_{I^c}^\top(b - A_I\overline{x}^*)\|_\infty \leq \gamma. \tag{6.6}$$

for some $\gamma \geq 0$. Moreover, there is a one-to-one correspondence between the γ and τ. Following the typical convention, if (6.6) *is a strict inequality, we say that x^* is a non-degenerate solution. Otherwise, we say that x^* is a degenerate solution.*

The following lemma relates non-denerate LASSO solutions x^* to the previously defined set $\operatorname{int}(C_{I_{x^*},\tau})$.

Lemma 6.1.5. *If x^* is a non-degenerate solution with $I_{x^*} = I$, then $x^* - \nabla f(x^*)/L \in \operatorname{int}(C_{I,\tau})$.*

Proof. By (6.5) and (6.6), for any $j \notin I$, we have

$$\begin{aligned} \sum_{i \in I} \left| x_i^* + \frac{a_i^\top(b - A_I\overline{x}^*)}{L} \right| \quad - \quad \tau &= \sum_{i \in I} \left| x_i^* + \frac{\gamma \cdot \operatorname{sgn}(x_i^*)}{L} \right| \quad - \quad \tau \\ &= \sum_{i \in I} |x_i^*| \quad + \quad |I| \cdot \frac{\gamma}{L} \quad - \quad \tau \\ &\geq |I| \cdot |a_j^\top(b - A_I\overline{x}^*)|/L \\ &= |I| \cdot |x_j^* + a_j^\top(b - A_I\overline{x}^*)|/L. \end{aligned}$$

The third equation on the right holds since we must have $\|x^*\|_1 = \tau$. If not, then we must have $Ax^* - b = 0$ which is only possible when x^* is a degenerate solution. □

We now prove that under our assumptions on the LASSO problem, the gradient at the optimal solution will almost always lie in a desirable direction. In other words, we have the following result.

Theorem 6.1.6. *Suppose $A \in \mathbb{R}^{m\times n}$ satisfies the restricted isometry property* (RIP) *of order $2s$, and the optimal solution x^* is s-sparse. The solution x^* will almost always be non-degenerate.*

Proof. Fix positive integers m, n, and $I \subseteq \{1, \ldots, n\}$ with $|I| = s \leq m$. Define $\textsc{ls}(m, n, I)$ to be the set of LASSO problems

$$\min \|Ax - b\|_2 \quad \text{s.t.} \quad \|x\|_1 \leq \tau$$

with s-sparse solutions x^* such that $I_{x^*} = I$ and $A \in \mathbb{R}^{m\times n}$ satisfying the RIP of order $2s$. As seen in the proof of Lemma 6.1.1, x^* is unique.

The LASSO optimality conditions above say that x^* is the solution to a LASSO problem if and only if $A_I^\top(b - A_I\overline{x}^*) = \gamma \cdot \operatorname{sgn}(\overline{x}^*)$ and $\|A_{I^c}^\top(b - A_I\overline{x}^*)\|_\infty \leq \gamma$ for some $\gamma \geq 0$. Since there is a one-to-one correspondence between τ and γ, we represent each LASSO problem in $\textsc{ls}(m, n, I)$ with the quadruple $(A_I, A_{I^c}, b, \gamma)$. Following this notation,

$$\textsc{ls}(m, n, I) = T_1 \cup T_2$$

where

$$T_1 := \left\{(A_I, A_{I^c}, b, \gamma) \in \textsc{ls}(m, n, I) : \|A_{I^c}^\top(b - A_I\overline{x}^*)\|_\infty = \gamma\right\},$$
$$T_2 := \left\{(A_I, A_{I^c}, b, \gamma) \in \textsc{ls}(m, n, I) : \|A_{I^c}^\top(b - A_I\overline{x}^*)\|_\infty < \gamma\right\}.$$

We show that T_1 has Lebesgue measure zero and T_2 has nonzero Lebesgue measure.

By the RIP, A_I has full rank since

$$0 < (1 - \delta_{2s})\|x\|_2^2 \leq \|A_Ix\|_2^2 \leq (1 + \delta_{2s})\|x\|_2^2$$

for all nonzero $x \in \mathbb{R}^s$. Thus, $A_I^\top A_I$ is invertible, and if x^* is the solution to $(A_I, A_{I^c}, b, \gamma) \in \textsc{ls}(m, n, I)$ then

$$\overline{x}^* = (A_I^\top A_I)^{-1}(A_I^\top b - \gamma \cdot \operatorname{sgn}(\overline{x}^*)).$$

Let $U := \left\{(A_I, A_{I^c}, b, \gamma) \in \mathbb{R}^{m\times s} \times \mathbb{R}^{m\times(n-s)} \times \mathbb{R}^m \times \mathbb{R}^+ : A_I \text{ nonsingular}\right\}$. For each $w \in \{-1, 1\}^s$, define the function $g_w : U \to \mathbb{R}^{n-s}$ by

$$g_w(A_I, A_{I^c}, b, \gamma) = \frac{A_{I^c}^\top\left(b - A_I(A_I^\top A_I)^{-1}(A_I^\top b - \gamma \cdot w)\right)}{\gamma},$$

If $S := \left\{x \in \mathbb{R}^{(n-s)} : |x| \leq 1\right\}$ with boundary ∂S and interior $\operatorname{int}(S)$, then

$$T_1 \subseteq \bigcup_w g_w^{-1}(\partial S) \quad \bigcup \quad \mathbb{R}^{m\times s} \times \mathbb{R}^{m\times(n-s)} \times \mathbb{R}^m \times \{0\}.$$

Each component function of g_w involves exactly one row of the variables in $A_{I^c}^\top$, and g_w is the composition of matrix inversion and basic matrix operations. Thus, g_w is a smooth map of constant rank $(n-s)$ on the open set $U \setminus g_w^{-1}(0)$. An application of Theorem 1 of [76] shows that $g_w^{-1}(\partial S)$ has measure zero. Hence, T_1 has Lebesgue measure zero.

To see that T_2 has nonzero measure, note that T_2 is the set of $(A_I, A_{I^c}, b, \gamma) \in U$ such that A satisfies the RIP of order $2s$ intersected with

$$\bigcup_w g_w^{-1}(\text{int}(S)) \cap \left\{(A_I, A_{I^c}, b, \gamma) \in U : \text{sgn}\left((A_I^\top A_I)^{-1}(A_I^\top b - \gamma \cdot w)\right) = w\right\}.$$

Using the triangle inequality, it is easy to see that the former set is open since

$$(1-\delta_{2s})\|x\|_2^2 \leq \|Ax\|_2^2 \leq (1+\delta_{2s})\|x\|_2^2$$

holds under small perturbations of A. The latter set is open since g_w and $(A_I, A_{I^c}, b, \gamma) \mapsto (A_I^\top A_I)^{-1}(A_I^\top b - \gamma \cdot w)$ are continuous functions for each w. Thus, T_2 is open. Moreover, it is easy to see that if $(A_I, A_{I^c}, b, \gamma) \in T_1$ then there exists a small perturbation E such that $(A_I, A_{I^c} + E, b, \gamma) \in T_2$. If $\text{LS}(m, n, I)$ is nonempty, it must be that T_2 is nonempty and therefore, has nonzero measure.

This argument is easily extended for any $I \subseteq \{1, \ldots, n\}$. Since there are a finite number of I's and a finite union of measure zero sets has measure zero, our lemma holds. □

Let y_k be the k-th iterate of Nesterov's accelerated proximal gradient method applied to the LASSO problem. The previous results allow us to make the following conclusion regarding the sparsity of y_k.

Theorem 6.1.7. *Suppose A satisfies the restricted isometry property* (RIP) *of order $2s$, and the optimal solution x^* is s-sparse. The iterates y_k are almost always eventually s-sparse.*

Proof. By Lemma 6.1.1, the sequence $\{y_k\}$ converges to the optimal solution x^*. Since $x_k = \frac{2}{k+3} z_k + \frac{k+1}{k+3} y_k$ and $\nabla f(x) = A^\top(Ax - b)$ is continuous, the sequence $\{x_k - \nabla f(x_k)/L\}$ converges to $x^* - \nabla f(x^*)/L$.

Theorem 6.1.6 says that x^* is almost always nondegenerate, in which case, by Lemma 6.1.5, $x^* - \nabla f(x^*)/L \in \text{int}(C_{Ix^*,\tau})$, where $\text{int}(C_{Ix^*,\tau})$ is the interior of $C_{Ix^*,\tau}$. Thus, if x^* is non-degenerate, there exists an N such that for $k \geq N$, $x_k - \nabla f(x_k)/L \in \text{int}(C_{I_{x^*}})$. By Lemma 6.1.3, for such k, $y_k = \text{proj}_1(x_k - \nabla f(x_k)/L, \tau)$ is s-sparse. □

6.1.2 Local linear convergence of NESTA-LASSO

We now show that NESTA-LASSO-K, Algorithm 10, is almost always locally linearly convergent under certain assumptions. First we give some motivation for why we update the prox-centers in NESTA-LASSO-K.

Consider applying Nesterov's accelerated proximal gradient method, Algorithm 9, to the LASSO problem. Suppose A satisfies the *restricted isometry property* (RIP) of order $2s$ and

the optimal solution x^* is s-sparse. As seen in Theorem 6.1.7, the iterates y_k are almost always eventually s-sparse. Thus, it is reasonable to assume that y_k is s-sparse.

Let $\delta = 1 - \delta_{2s}$ where δ_{2s} is the RIP constant of A. We have

$$\|A(x^* - y_k)\|_2^2 + 2(y_k - x^*)^\top A^\top (Ax^* - b) = f(y_k) - f(x^*) \geq \|A(y_k - x^*)\|_2^2 \geq \delta \|y_k - x^*\|_2^2. \quad (6.7)$$

To see the first inequality, let $y = x^* + \tau(y_k - x^*)$ for $\tau \in [0, 1]$. Due to the convexity of the 1-norm ball, y is feasible. Since x^* is the minimum, for any $\tau \in [0, 1]$,

$$f(y) - f(x^*) = \tau^2 \|A(x^* - y_k)\|_2^2 + 2\tau(y_k - x^*)^\top A^\top (Ax^* - b) \geq 0.$$

Thus, $(y_k - x^*)^\top A^\top (Ax^* - b) \geq 0$. The second inequality follows from (6.2) since the vector $y_k - x^*$ has at most $2s$ nonzeros. Then from (6.1), we have

$$\delta \|y_k - x^*\|_2^2 \leq \frac{L}{k(k+1)} \|x^* - c\|_2^2.$$

Putting everything together gives

$$\|y_k - x^*\|_2 \leq \sqrt{\frac{L}{k(k+1)\delta}} \|x^* - c\|_2 \leq \frac{1}{k} \sqrt{\frac{L}{\delta}} \|c - x^*\|_2. \quad (6.8)$$

The above relation and (6.1) suggest that when solving the LASSO problem, we can speed up Algorithm 9 by updating the prox-center, c, every K steps. With our assumptions, we prove in the first part of following theorem that for every $K > \sqrt{\frac{L}{\delta}}$, restarting Algorithm 9 every K steps with the new prox-center, $\text{proj}_1(y_k - \nabla f(y_k)/L, \tau)$, is locally linearly convergent. In the second part of Theorem 6.1.8, we prove that there is an optimal number of such steps.

In the following, allow the iterates to be represented by y_{jk} where j is the number of times the prox-center has been changed (the outer iteration) and k is number of iterations after the last prox-center change (the inner iteration).

Theorem 6.1.8. *Suppose A satisfies the restricted isometry property of order $2s$ and the solution x^* is s-sparse. The following holds if x^* is non-degenerate and the initial point $p_0 := x_0$ in Algorithm 2 is chosen to be sufficiently close to x^*.*

(i) *Algorithm 10 is locally linearly convergent for any $K > \sqrt{\frac{L}{\delta}}$.*

(ii) *In Algorithm 2, let j_{tot} be the total number of prox-center changes. The total number of iterations, $j_{\text{tot}} \cdot K$, to get $\|p_j - x^*\|_2 \leq \varepsilon$ is minimized if K is equal to*

$$k_{\text{opt}} := e\sqrt{\frac{L}{\delta}} \quad (6.9)$$

where e is the base of the natural logarithm. Moreover, for each j,

$$\|p_j - x^*\|_2 \leq \frac{1}{e^j} \|p_0 - x^*\|_2.$$

Proof. (i) By Lemma 6.1.5, $x^* - \nabla f(x^*)/L \in \text{int}(C_{I_{x^*},\tau})$, where $\text{int}(C_{I_{x^*},\tau})$ is the interior of $C_{I_{x^*},\tau}$. Let U_α be a ball of radius $\alpha > 0$, centered at $x^* - \nabla f(x^*)/L$, such that $U_\alpha \subseteq \text{int}(C_{I_{x^*},\tau})$. By continuity, we may choose an $\epsilon > 0$ such that $\|x - x^*\|_2 < \epsilon$ implies $x - \nabla f(x)/L \in U_\alpha$.

Now choose $\beta > 0$ such that for all $\|x\|_1 \leq \tau$, $f(x) - f(x^*) < \beta$ implies $\|x - x^*\|_2 < \epsilon$. To see that $\beta > 0$ exists, suppose for a contradiction that $\forall\ n$, $\exists\ x_n$ with $\|x_n\|_1 \leq \tau$ where $f(x_n) - f(x^*) < 1/n$ but $\|x_n - x^*\|_2 \geq \epsilon$. Since the 1-norm ball is compact, there is a subsequence $\{x_{n_k}\}$ of $\{x_n\}$ converging to some x'. By continuity, $f(x_{n_k})$ converges to $f(x')$. As mentioned right before the statement of Lemma 6.1.1, x^* is a unique minimum. Thus, $f(x') \neq f(x^*)$ contradicting the assumption that $f(x_n)$ converges to $f(x^*)$.

We now show that Algorithm 10 is linearly convergent if the initial prox-center p_0 is close enough to x^*. Suppose $\|p_0 - x^*\|_2 < \beta/L$. Then (6.1) implies

$$f(y_{1K}) - f(x^*) \leq \frac{L}{K(K+1)}\|p_0 - x^*\|_2^2 < \beta,$$

and so $\|y_{1K} - x^*\| < \epsilon$. By Lemma 6.1.3, $p_1 = \text{proj}_1(y_{1K} - \nabla f(y_{1K})/L, \tau)$ is s-sparse, and by (6.7),

$$\delta\|p_1 - x^*\|_2^2 \leq f(p_1) - f(x^*). \tag{6.10}$$

Note that p_1 is the result of a step of the *projected gradient method*, i.e. $x_{k+1} = \text{proj}_1(x_k - \nabla f(x_k)/L, \tau)$. Since this method is monotonically decreasing (see [101] for a proof),

$$f(p_1) - f(x^*) \leq f(y_{1K}) - f(x^*). \tag{6.11}$$

Combining (6.10) and (6.11) with (6.1), gives

$$\|p_1 - x^*\|_2 \leq \frac{1}{K}\sqrt{\frac{L}{\delta}}\|p_0 - x^*\|_2.$$

Since we assume that $K > \sqrt{\frac{L}{\delta}}$, we have $\|p_1 - x^*\|_2 < \beta/L$. Thus, the above arguments can be repeatedly applied to show that for any j,

$$\|p_j - x^*\|_2 \leq \left(\frac{1}{K}\sqrt{\frac{L}{\delta}}\right)^j \|p_0 - x^*\|_2. \tag{6.12}$$

(ii) First observe that (6.12) implies

$$\|p_j - x^*\|_2 \leq \left(\frac{1}{K}\sqrt{\frac{L}{\delta}}\right)^j \|p_0 - x^*\|_2 \leq \varepsilon\|p_0 - x^*\|_2$$

Table 6.1: Number of products with A and $A^\top$ for NESTA-LASSO without prox-center updates (cf. Algorithm 9) and NESTA-LASSO with prox-center updates (cf. Algorithm 11). These values are given by N_A and N_A^{update} respectively.

Number of Rows of A	Number of Columns of A	τ	N_A	N_A^{update}
100	256	6.28	69	37
200	512	12.6	77	47
400	1024	25.1	157	45

when

$$j \log\left(\frac{1}{K}\sqrt{\frac{L}{\delta}}\right) = \log \varepsilon.$$

This relation allows us to choose K to minimize the product $j \cdot K$. Since

$$j \cdot K = \frac{K \log \varepsilon}{\log \sqrt{L/\delta} - \log K},$$

taking derivative of the expression on the right shows that $j \cdot K$ is minimized when

$$K = e\sqrt{\frac{L}{\delta}},$$

where e is the base of the natural logarithm. The total number of iterations will then be

$$j_{\text{tot}} \cdot K = -e\sqrt{\frac{L}{\delta}} \log \varepsilon.$$

□

Theorem 6.1.6 implies that we almost always have local linear convergence:

Corollary 6.1.9. *If A satisfies the restricted isometry property of order $2s$ and the solution x^* is s-sparse, Algorithm 10 is almost always locally linearly convergent for any $K > \sqrt{\frac{L}{\delta}}$.*

In our experiments, there are some cases where updating the prox-center will eventually cause the duality gap to jump to a higher value than the previous iteration. This can cause the algorithm to run for more iterations than necessary. A check is added to prevent the prox-center from being updated if it no longer helps.

In Table 6.1, we give some results showing that updating the prox-center is effective when using NESTA-LASSO to solve the LASSO problem.

Chapter 7

Parnes: Solving the Basis Pursuit Denoise Problem

7.1 PARNES

In applications where the noise level of the problem is approximately known, it is preferable to solve BP(σ). The Pareto root-finding method used by van den Berg and Friedlander [12] interprets BP(σ) as finding the root of a single-variable nonlinear equation whose graph is called the Pareto curve. Their implementation of this approach is called SPGL1. In SPGL1, an inexact version of Newton's method is used to find the root, and at each iteration, an approximate solution to the LASSO problem, LS(τ), is found using an SPG approach. Refer to [27] for more information on the inexact Newton method. In Section 8.2, we show experimentally that using NESTA-LASSO in place of the SPG approach for solving the LS(τ) subproblems can lead to improved results. We call this version of the Pareto root-finding method, PARNES. The pseudocode of PARNES is given in Algorithm 12.

7.1.1 Pareto curve

Suppose A and b are given, with $0 \neq b \in \text{range}(A)$. The points on the Pareto curve are given by $(\tau, \varphi(\tau))$ where $\varphi(\tau) = \|Ax_\tau - b\|_2$, $\tau = \|x_\tau\|_1$, and x_τ solves LS(τ). The Pareto curve gives the optimal trade-off between the 2-norm of the residual and 1-norm of the solution to LS(τ). It can also be shown that the Pareto curve also characterizes the optimal trade-off between the 2-norm of the residual and 1-norm of the solution to BP(σ). Refer to [12] for a more detailed explanation of these properties of the Pareto curve. An example of a Pareto curve is shown in Figure 7.1.

Let τ_{BP} be the optimal objective value of BP(0). The Pareto curve is restricted to the interval $\tau \in [0, \tau_{\text{BP}}]$ with $\varphi(0) = \|b\|_2 > 0$ and $\varphi(\tau_{\text{BP}}) = 0$. The following theorem, proven by van den Berg and Friedlander, shows that the Pareto curve is convex, strictly decreasing over the interval $\tau \in [0, \tau_{\text{BP}}]$, and continuously differentiable for $\tau \in (0, \tau_{\text{BP}})$.

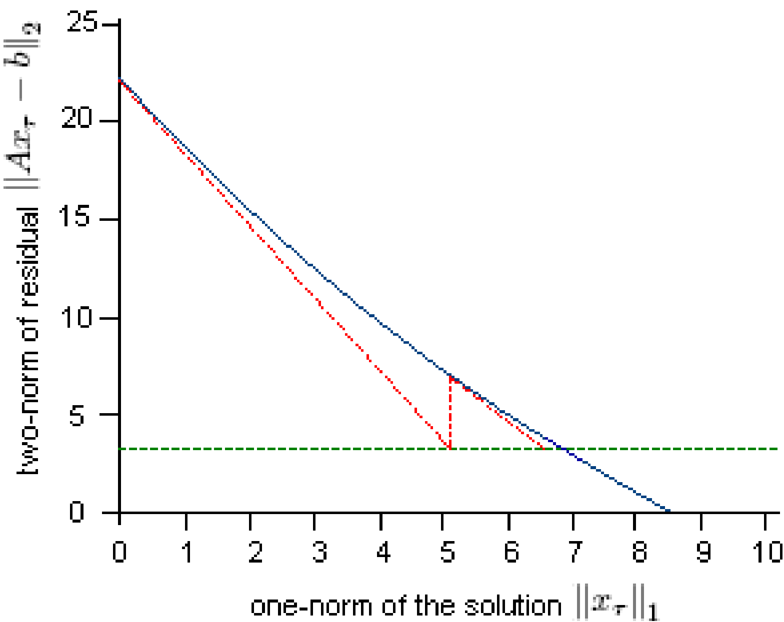


Figure 7.1: An example of a Pareto curve. The solid line is the Pareto curve; the dotted red lines give two iterations of Newton's method.

Proposition 7.1.1. *[12] The function φ is*

(i) *convex and nonincreasing;*

(ii) *continuously differentiable for $\tau \in (0, \tau_{\text{BP}})$ with $\varphi'(\tau) = -\lambda_\tau$ where $\lambda_\tau = \|A^T y_\tau\|_\infty$ is the optimal dual variable to $\text{LS}(\tau)$ and $y_\tau = r_\tau / \|r_\tau\|_2$ with $r_\tau = Ax_\tau - b$;*

(iii) *strictly decreasing and $\|x_\tau\|_1 = \tau$ for $\tau \in [0, \tau_{\text{BP}}]$.*

7.1.2 Root finding

Since the Pareto curve characterizes the optimal trade-off for both $\text{BP}(\sigma)$ and $\text{LS}(\tau)$, solving $\text{BP}(\sigma)$ for a fixed σ can be interpreted as finding a root of the non-linear equation $\varphi(\tau) = \sigma$. The iterations consist of finding the solution to $\text{LS}(\tau)$ for a sequence of parameters $\tau_k \to \tau_\sigma$ where τ_σ is the optimal objective value of $\text{BP}(\sigma)$.

Applying Newton's method to φ gives

$$\tau_{k+1} = \tau_k + (\sigma - \varphi(\tau_k))/\varphi'(\tau_k).$$

Since φ is convex, strictly decreasing and continuously differentiable, $\tau_k \to \tau_\sigma$ superlinearly for all initial values $\tau_0 \in (0, \tau_{\text{BP}})$ (see Proposition 1.4.1 in [13]). By Proposition 7.1.1, $\varphi(\tau_k)$

is the optimal value to LS(τ_k) and $\varphi'(\tau_k)$ is the dual solution to LS(τ_k). Since evaluating $\varphi(\tau_k)$ involves solving a potentially large optimization problem, an inexact Newton method is carried out with approximations of $\varphi(\tau_k)$ and $\varphi'(\tau_k)$.

Let $\overline{y}_\tau$ and $\overline{\lambda}_\tau$ be the approximations of the y_τ and λ_τ defined in Proposition 7.1.1. The duality gap at each iteration is given by

$$\eta_\tau = \|\overline{r}_\tau\|_2 - (b^T\overline{y}_\tau - \tau\overline{\lambda}_\tau).$$

The following convergence result has been proven by van den Berg and Friedlander.

Theorem 7.1.2. *[12] Suppose A has full rank, $\sigma \in (0, \|b\|_2)$, and the inexact Newton method generates a sequence $\tau_k \to \tau_\sigma$. If $\eta_k := \eta_{\tau_k} \to 0$ and τ_0 is close enough to τ_σ, we have*

$$|\tau_{k+1} - \tau_\sigma| = \gamma_1\eta_k + \zeta_k|\tau_k - \tau_\sigma|,$$

where $\zeta_k \to 0$ and γ_1 is a positive constant.

7.1.3 Solving the LASSO problem

Approximating $\varphi(\tau_k)$ and $\varphi'(\tau_k)$ require approximately minimizing LS(τ). The solver SPGL1 uses a spectral projected-gradient (SPG) algorithm. T he method follows the algorithm by Birgin, Martínez, and Raydan [15] and is shown to be globally convergent. The costs include evaluating Ax, $A^\top r$, and a projection onto the 1-norm ball $\|x\|_1 \le \tau$. In PARNES, we replace this SPG algorithm with our algorithm, NESTA-LASSO (cf. Algorithm 11).

Algorithm 12 PARNES: Pareto curve method with NESTA-LASSO

Require: initial point x_0, BPDN parameter σ, tolerance η.
Ensure: $x_\sigma = \text{argmin}\{\|x\|_1 : \|Ax - b\|_2 \le \sigma\}$

1: $\tau_0 = 0$, $\varphi_0 = \|b\|_2$, $\varphi'_0 = \|A^\top b\|_\infty$;
2: **for** $k = 0, \ldots, k_{\max}$, **do**
3: $\quad\tau_{k+1} = \tau_k + (\sigma - \varphi_k)/\varphi'_k$;
4: $\quad x_{k+1} = \text{NESTA-LASSO}(x_k, \tau_{k+1}, \eta)$;
5: $\quad r_{k+1} = b - Ax_{k+1}$;
6: $\quad\varphi_{k+1} = \|r_{k+1}\|_2$;
7: $\quad\varphi'_{k+1} = -\|A^\top r_{k+1}\|_\infty / \|r_{k+1}\|_2$;
8: $\quad$**if** $\|r_{k+1}\|_2 - \sigma \le \eta \cdot \max\{1, \|r_{k+1}\|_2\}$ **then**
9: $\quad\quad$**return** $x_\sigma = x_{k+1}$;
10: $\quad$**end if**
11: **end for**

Chapter 8

Numerical Experiments

8.1 Other Solution Techniques and Tools

In the our numerical experiments, we compare PARNES with other state-of-the-art methods. The algorithms we test and their experimental details are described below. Note that the algorithms either solve BP(σ) or QP(λ).

8.1.1 NESTA [9]

NESTA is used to solve BP(σ). Its code is available at `http://www.acm.caltech.edu/~nesta`. The parameters for NESTA are set to be

$$x_0 = A^\top b, \quad \mu = 0.02,$$

where x_0 is the initial guess and μ is the smoothing parameter for the 1-norm function in BP(σ).

Continuation techniques are used to speed up NESTA in [9]. Such techniques are useful when it is observed that a problem involving some parameter λ is faster for large λ, [74, 54]. Thus, the idea of continuation is to solve a sequence of problems for decreasing values of λ. In the case of NESTA, it is observed that convergence is faster for larger values of μ. When continuation is used in the experiments, there are four continuation steps with $\mu_0 = \|x_0\|_\infty$ and $\mu_t = (\mu/\mu_0)^{t/4}\mu_0$ for $t = 1, 2, 3, 4$.

8.1.2 GPSR: Gradient projection for sparse reconstruction [34]

GPSR is used to solve the penalized least-squares problem QP(λ). The code is available at `http://www.lx.it.pt/~mtf/GPSR`. The problem is first recast as a bound-constrained quadratic program (BCQP) by using a standard change of variables on x. Here, $x = u_1 - u_2$, and the variables are now given by $[u_1, u_2]$ where the entries are positive. The new problem is then solved using a gradient projection (GP) algorithm. The parameters are set to the default values in the following experiments.

A version of GPSR with continuation is also tested. The number of continuation steps is 40, the variable TOLERANCEA is set to 10^{-3}, and the variable MINITERA is set to 1. All other parameters are set to their default values.

8.1.3 SpaRSA: Sparse reconstruction by separable approximation [35]

SPARSA is used to minimize functions of the form $\phi(x) = f(x) + \lambda c(x)$ where f is smooth and c is non-smooth and non-convex. The QP(λ) problem is a special case of functions of this form. The code for SPARSA is available at `http://www.lx.it.pt/~mtf/SpaRSA`.

In a sense, SPARSA is an iterative shrinkage/thresholding algorithm. Utilizing continuation and a Brazilai-Borwein heuristic [6] to find step sizes, the speed of the algorithm can be increased. The number of continuation steps is set to 40 and the variable MINITERA is set to 1. All remaining variables are set to their default values.

8.1.4 SPGL1 [12] and SPARCO [11]

SPGL1 is available at `http://www.cs.ubc.ca/labs/scl/spgl1`. The parameters for our numerical experiments are set to their default values.

Due to the vast number of available and upcoming algorithms for sparse reconstruction, the authors of SPGL1 and others have created SPARCO [11]. In SPARCO, they provide a much needed testing framework for benchmarking algorithms. It consists of a large collection of imaging, compressed sensing, and geophysics problems. Moreover, it includes a library of standard operators which can be used to create new test problems. SPARCO is implemented in MATLAB and was originally created to test SPGL1. The toolbox is available at `http://www.cs.ubc.ca/labs/scl/sparco`.

8.1.5 FISTA: Fast iterative soft-thresholding algorithm [8]

FISTA solves QP(λ). It can be thought of as a simplified version of the Nesterov algorithm in Section 5.2.1 since it involves two sequences of iterates instead of three. In Section 4.2 of [9], FISTA is shown to give very accurate solutions provided enough iterations are taken. Due to its ease of use and accuracy, FISTA is used to compute reference solutions in [9] and in this paper. The code for FISTA can be found in the NESTA experiments code at `http://www.acm.caltech.edu/~nesta`.

8.1.6 FPC: Fixed point continuation [54, 55]

FPC solves the general problem $\min_x \|x\|_1 + \lambda f(x)$ where $f(x)$ is differentiable and convex. The special case with $f(x) = \frac{1}{2}\|Ax - b\|_2^2$ is the QP(λ) problem. The algorithm is available at `http://www.caam.rice.edu/~optimization/L1/fpc`.

FPC is equivalent to iterative soft-thresholding. The approach is based on the observation that the solution solves a fixed-point equation $x = F(x)$ where the operator F is a composition of a gradient descent-like operator and a shrinkage operator. It can be shown that the algorithm has q-linear convergence and also, finite-convergence for some components of the solution. Since the parameter λ affects the speed of convergence, continuation techniques are used to slowly decrease λ for faster convergence. A more recent version of FPC, FPC-BB, uses Brazilai-Borwein steps to speed up convergence. Both versions of FPC are tested with their default parameters.

8.1.7 FPC-AS: Fixed-point continuation and active set [91]

FPC-AS is an extension of FPC into a two-stage algorithm which solves QP(λ). The code can be found at `http://www.caam.rice.edu/~optimization/L1/fpc`. It has been shown in [54] that applying the shrinkage operator a finite number of times yields the support and signs of the optimal solution. Thus, the first stage of FPC-AS involves applying the shrinkage operator until an active set is determined. In the second stage, the objective function is restricted to the active set and $\|x\|_1$ is replaced by $c^T x$ where c is the vector of signs of the active set. The constraint $c_i \cdot x_i > 0$ is also added. Since the objective function is now smooth, many available methods can now be used to solve the problem. In the following tests, the solvers L-BFGS and conjugate gradients, CG (referred to as FPC-AS (CG)), are used. Continuation methods are used to decrease λ to increase speed. For experiments involving approximately sparse signals, the parameter controlling the estimated number of nonzeros is set to n, and the maximum number of subspace iterations is set to 10. The other parameters are set to their default values. All other experiments were tested with the default parameters.

8.1.8 Bregman iteration [100]

The Bregman Iterative algorithm consists of solving a sequence of QP(λ) problems for a fixed λ and updated observation vectors b. Each QP(λ) is solved using the Brazilai-Borwein version of FPC. Typically, very few (around four) outer iterations are needed. Code for the Bregman algorithm can be found at `http://www.caam.rice.edu/~optimization/L1/2006/10/bregman-iterative-algorithms-for.html`. All parameters are set to their default values.

8.1.9 C-SALSA [1, 2]

This state-of-the-art method solves BP(σ) and has been shown to be competitive with SPGL1 and NESTA. The method solves the general constrained optimization problem

$$\min_x \phi(x) \text{ s.t. } \|Ax - b\|_2 \leq \epsilon.$$

First, the method transforms the problem into an unconstrained problem which is then transformed into a different constrained problem and then solved with an augmented Lagrangian scheme.

The algorithm requires a method to compute the inverse of $(A^\top A + \alpha I)$ with $\alpha > 0$ and an efficient method for computing the denoising operator associated with ϕ. We have hand-tuned the parameters μ_1 and μ_2 for optimal performance. The code for C-SALSA can be found at `http://cascais.lx.it.pt/~mafonso/salsa.html`.

8.2 Numerical Results

In the NESTA paper [9] extensive experiments are carried out, comparing the effectiveness of the state-of-the-art sparse reconstruction algorithms described in Section 8.1. The code used to run these experiments is available at `http://www.acm.caltech.edu/~nesta`. We have modified this NESTA experiment infrastructure to include PARNES and C-SALSA, and we repeat some of the tests in [9] using the same experimental standards and parameters. Refer to the [9] for a detailed description of the experiments.

One difficulty that arises in carrying out such broad experiments is that some of the algorithms solve QP(λ) whereas others solve BP(σ). Comparing the algorithms thus requires a way of finding a (σ, λ) pair for which the solutions of QP(λ) and BP(σ) coincide. The NESTA experiments utilize a two-step procedure. Given the noise level ϵ, the authors choose $\sigma_0 := \sqrt{m + 2\sqrt{2m}\epsilon}$, and then use SPGL1 to solve the corresponding BP(σ_0) problem. The SPGL1 dual solution then provides an estimate of the corresponding λ. In practice, the computation of λ is not very stable, and so a second step is performed in which FISTA is used to compute a σ corresponding to λ using a very high accuracy of around 10^{-14}.

The highly accurate solution computed by FISTA is used to determine the accuracy of the solutions computed by the other solvers. Section 4.2 of [9] shows that this is reasonable since FISTA gives very accurate solutions provided that enough iterations are taken. For each test, FISTA is ran twice. In the first run, FISTA is ran with no limit on the number of iterations until the relative change in the function value is less than 10^{-14}. This solution is used to determine the accuracy of the computed solutions. The results recorded for FISTA are from running FISTA a second time with either stopping criterion (8.1) or (8.2).

Since the different algorithms utilize different stopping criteria, to maintain fairness, the codes have been modified to allow for two new stopping criteria. Intuitively, the algorithms are run until they achieve a solution at least as accurate as the one obtained by NESTA. In [9], NESTA (with continuation) is used to compute a solution x_{NES}. Let $\hat{x}_k$ be the k-th iteration in the algorithm being tested. The stopping criteria used are:

$$\|\hat{x}_k\|_{\ell_1} \le \|x_{\text{NES}}\|_{\ell_1} \quad \text{and} \quad \|b - A\hat{x}_k\|_{\ell_2} \le 1.05\,\|b - Ax_{\text{NES}}\|_{\ell_2}, \tag{8.1}$$

and

$$\lambda\|\hat{x}_k\|_{\ell_1} + \frac{1}{2}\|A\hat{x}_k - b\|_{\ell_2}^2 \le \lambda\|x_{\text{NES}}\|_{\ell_1} + \frac{1}{2}\|Ax_{\text{NES}} - b\|_{\ell_2}^2. \tag{8.2}$$

The rationale for having two stopping criteria is to reduce any potential bias arising from the fact that some algorithms solve QP(λ), for which (8.2) is the most natural, while others solve BP(σ), for which (8.1) is the most natural. It is evident from the tables below that

there is not a significant difference between using (8.1) and (8.2). For each test, the number of calls to A and $A^\top$ (N_A) is recorded, and the algorithms are said to have not converged (DNC) if the number of calls exceeds 20,000.

In Tables 8.2 and 8.3, we repeat the experiments done in Tables 5.1 and 5.2 of [9]. These experiments involve recovering an unknown, exactly s-sparse signal with $n = 262{,}144$, $m = n/8$, and $s = m/5$. For each run, the measurement operator A is a randomly subsampled discrete cosine transform, and the noise level is set to 0.1. The experiments are performed with increasing values of the dynamic range d where $d = 20, 40, 60, 80, 100$ dB.

The dynamic range d is a measure of the ratio between the largest and smallest magnitudes of the non-zero coefficients of the unknown signal. Problems with a high dynamic range occur often in applications. In these cases, high accuracy becomes important since one must be able to detect and recover low-power signals with small amplitudes which may be obscured by high-power signals with large amplitudes.

Table 8.1 compares the accuracy of the different solvers when used to calculate the results in the last column of Table 8.2. As this corresponds to a very high dynamic range (100 dB), one hopes to obtain very accurate results. Although FISTA produces the most accurate results ($\|x - x^*\|_1/\|x^*\|_1 = 3.63 \cdot 10^{-4}$), with at least twice the accuracy of the other solvers, it requires the over 10,000 calls to A and $A^\top$. In contrast, PARNES only requires 632 function calls to reach a relative accuracy of $\|x - x^*\|_1/\|x^*\|_1 = 6.93 \cdot 10^{-4}$. The solvers FPC-AS and FPC-AS (CG) do well and only require around 300 iterations to reach a relative accuracy of around $6.93 \cdot 10^{-4}$. The remaining algorithms reach relative accuracies of around $8 \cdot 10^{-4}$ or more, and GSPR does not converge. Without continuation, NESTA only achieves a relative accuracy of $4.12 \cdot 10^{-3}$ after 15,227 function calls. However, NESTA with continuation does much better and reaches a relative accuracy of $8.12 \cdot 10^{-4}$ after 787 function calls.

In Tables 8.2 and 8.3, the same experiment is ran for the two stopping criteria. Since there is no notable difference between the two sets of results, we only analyze Table 8.2. Here, FPC-AS and FPC-AS (CG) perform the best for large values of d, and the number of function calls ranges from 200 to 375 for all values of the dynamic range. In these cases, we see a relatively small increase in N_A as d increases from 20 dB to 100 dB. Our method, PARNES, and SPGL1 generally perform well and do particularly well for small values of d. However, both exhibit a larger increase in N_A with d, with PARNES increasing from 122 to 632 function calls and SPGL1 ranging between 58 and 504. The solvers NESTA + CT and SPARSA perform relatively well for large values of d with N_A ranging between 500 and 800.

In applications, the signal to be recovered is often approximately sparse rather than exactly sparse. Again, high accuracy is important when solving these problems. The last two tables, Tables 8.4 and 8.5, replicate Tables 5.3 and 5.4 of [9]. Each run involves an approximately sparse signal obtained from a permutation of the Haar wavelet coefficients of a 512×512 image. The measurement vector b consists of $m = n/8 = 512^2/8 = 32{,}768$ random discrete cosine measurements, and the noise level is set to have a variance of 1 in Table 8.4 and 0.1 in Table 8.5. For more specific details, refer to [9].

We have seen that NESTA + CT, SPARSA, SPGL1, PARNES, and both versions of FPC-AS perform well in the case of exactly sparse signals for all values of the dynamic range.

Table 8.1: Comparison of accuracy using experiments from Table 8.2. Dynamic range 100 dB, $\sigma = 0.100$, $\mu = 0.020$, sparsity level $s = m/5$. Stopping rule is (8.1).

Methods	N_A	$\|x\|_1$	$\|Ax-b\|_2$	$\frac{\|x-x^*\|_1}{\|x^*\|_1}$	$\|x-x^*\|_\infty$	$\|x-x^*\|_2$
PARNES	632	942197.606	2.692	0.000693	8.312	46.623
NESTA	15227	942402.960	2.661	0.004124	45.753	255.778
NESTA + CT	787	942211.581	2.661	0.000812	9.317	52.729
GPSR	DNC	DNC	DNC	DNC	DNC	DNC
GPSR + CT	11737	942211.377	2.725	0.001420	15.646	90.493
SPARSA	693	942197.785	2.728	0.000783	9.094	51.839
SPGL1	504	942211.520	2.628	0.001326	14.806	84.560
FISTA	12462	942211.540	2.654	0.000363	4.358	26.014
FPC-AS	287	942210.925	2.498	0.000672	9.374	45.071
FPC-AS (CG)	361	942210.512	2.508	0.000671	9.361	45.010
FPC	9614	942211.540	2.719	0.001422	15.752	90.665
FPC-BB	1082	942209.854	2.726	0.001378	15.271	87.963
BREGMAN-BB	1408	942286.656	1.326	0.000891	9.303	52.449
C-SALSA	1338	942219.455	2.317	0.000851	9.541	55.14

However, in the case of approximately sparse signals, SPARSA and all versions of FPC no longer converge in under 20,000 function calls. In Table 8.4, PARNES, SPGL1, and C-SALSA perform well, with PARNES and C-SALSA taking around 650 function calls for some runs (compare to NESTA + CT which takes at least 3,000 iterations). These algorithms also perform the best in Table 8.5, and most other algorithms no longer converge in under 10,000 function calls.

8.2.1 Choice of parameters

As Tseng observed, accelerated proximal gradient algorithms will converge so long as the condition given as equation (45) in [87] is satisfied. In our case this translates into

$$\min_{x\in\mathbb{R}^n} \left\{ \nabla f(y_k)^\top x + \frac{L}{2}\|x-x_k\|_2^2 + P(x) \right\} \geq \nabla f(y_k)^\top y_k + P(y_k), \tag{8.3}$$

upon setting $\gamma_k = 1$ and

$$P(x) = \begin{cases} 0 & \text{if } \|x\|_1 \leq \tau, \\ \infty & \text{otherwise,} \end{cases}$$

in (45) in [87]. In other words, the value of L need not necessarily be fixed at the Lipschitz constant of ∇f but may be decreased, and decreasing L has the same effect as increasing the stepsize. Tseng suggests to decrease L adaptively by a constant factor until (45) is violated,

Table 8.2: Number of function calls where the sparsity level is $s = m/5$ and the stopping rule is (8.1).

Method	20 dB	40 dB	60 dB	80 dB	100 dB
PARNES	122	172	214	470	632
NESTA	383	809	1639	4341	15227
NESTA + CT	483	513	583	685	787
GPSR	64	622	5030	DNC	DNC
GPSR + CT	271	219	357	1219	11737
SPARSA	323	387	465	541	693
SPGL1	58	102	191	374	504
FISTA	69	267	1020	3465	12462
FPC-AS	209	231	299	371	287
FPC-AS (CG)	253	289	375	481	361
FPC	474	386	478	1068	9614
FPC-BB	164	168	206	278	1082
BREGMAN-BB	211	223	309	455	1408
C-SALSA	242	602	702	970	1338

Table 8.3: Number of function calls where the sparsity level is $s = m/5$ and the stopping rule is (8.2).

Method	20 dB	40 dB	60 dB	80 dB	100 dB
PARNES	74	116	166	364	562
NESTA	383	809	1639	4341	15227
NESTA + CT	483	513	583	685	787
GPSR	62	618	5026	DNC	DNC
GPSR + CT	271	219	369	1237	11775
SPARSA	323	387	463	541	689
SPGL1	43	99	185	365	488
FISTA	72	261	1002	3477	12462
FPC-AS	115	167	159	371	281
FPC-AS (CG)	142	210	198	481	355
FPC	472	386	466	1144	9734
FPC-BB	164	164	202	276	1092
BREGMAN-BB	211	223	309	455	1408
C-SALSA	202	550	650	898	1230

Table 8.4: Recovery results of an approximately sparse signal (with Gaussian noise of variance 1 added) and with (8.2) as a stopping rule.

Method	Run 1	Run 2	Run 3	Run 4	Run 5
PARNES	838	810	1038	1098	654
NESTA	8817	10867	9887	9093	11211
NESTA + CT	3807	3045	3047	3225	2735
GPSR	DNC	DNC	DNC	DNC	DNC
GPSR + CT	DNC	DNC	DNC	DNC	DNC
SPARSA	2143	2353	1977	1613	DNC
SPGL1	916	892	1115	1437	938
FISTA	3375	2940	2748	2538	3855
FPC-AS	DNC	DNC	DNC	DNC	DNC
FPC-AS (CG)	DNC	DNC	DNC	DNC	DNC
FPC	DNC	DNC	DNC	DNC	DNC
FPC-BB	5614	7906	5986	4652	6906
BREGMAN-BB	3288	1281	1507	2892	3104
C-SALSA	742	626	630	1226	826

Table 8.5: Recovery results of an approximately sparse signal (with Gaussian noise of variance 0.1 added) and with (8.2) as a stopping rule.

Method	Run 1	Run 2	Run 3	Run 4	Run 5
PARNES	1420	1772	1246	1008	978
NESTA	11573	10457	10705	8807	13795
NESTA + CT	7543	13655	11515	3123	2777
GPSR	DNC	DNC	DNC	DNC	DNC
GPSR + CT	DNC	DNC	DNC	DNC	DNC
SPARSA	12509	DNC	DNC	3117	DNC
SPGL1	1652	1955	2151	1311	2365
FISTA	10845	12165	10050	7647	11997
FPC-AS	DNC	DNC	DNC	DNC	DNC
FPC-AS (CG)	DNC	DNC	DNC	DNC	DNC
FPC	DNC	DNC	DNC	DNC	DNC
FPC-BB	DNC	DNC	DNC	DNC	DNC
BREGMAN-BB	3900	3684	2045	3292	3486
C-SALSA	1886	1926	1770	1754	1854

then backtrack and repeat the iteration (cf. Note 6 in [87]). For simplicity, and very likely at the expense of speed, we do not change our L adaptively in PARNES and NESTA-LASSO. Instead, we choose a small fixed L by trying a few different values so that (8.3) is satisfied for all k and likewise for the tolerance η in Algorithm 11. However, even with this crude way of selecting L and η, the results obtained are still rather encouraging.

8.3 Conclusions

As seen in the numerical results, SPGL1 and NESTA are among some of the top performing solvers available for basis pursuit denoising problems. We have therefore made use of Nesterov's accelerated proximal gradient method in our algorithm NESTA-LASSO and shown that updating the prox-center leads to improved results. Through our experiments, we have shown that using NESTA-LASSO in the Pareto root-finding method leads to results comparable to those of currently available state-of-the-art methods. Moreover, PARNES performs consistently well in all our experiments.

Bibliography

[1] Manya V. Afonso, Jos M. Bioucas-Dias, and Mrio A. T. Figueiredo. “Fast frame-based image deconvolution using variable splitting and constrained optimization”. In: *IEEE/SP 15th Workshop on Statistical Signal Processing* (2009), pp. 109–112.

[2] Manya V. Afonso, Jos M. Bioucas-Dias, and Mrio A. T. Figueiredo. “Fast Image Recovery Using Variable Splitting and Constrained Optimization”. In: *IEEE Transactions on Image Processing* 19.9 (2010), pp. 2345–2356.

[3] P. Amestoy and R. Tilch. “Solving the compressible Navier-Stokes equations with finite elements using a multifrontal method”. In: *Impact of Computing in Science and Engineering* 1 (1989), pp. 93 –107.

[4] C. Ashcraft. “A vector implementation of multifrontal method for large sparse, symmetric positive definite linear systems”. In: *Tech Report ETA-TR-52, Engineering Technology Applications Division, Boeing Computer Services, Seattle, WA* (1987).

[5] C. Ashcraft et al. “Progress in sparse matrix methods for large linear systems on vector supercomputers”. In: *Internat. J. Supercomput. Appl.* 1 (1987), pp. 10 –29.

[6] Jonathan Barzilai and Jonathan M. Borwein. “Two point step size gradient method”. In: *IMA Journal of Numerical Analysis* 8.1 (1988), pp. 141–148.

[7] Mario Bebendorf. “Efficient Inversion of the Galerkin Matrix of General Second Order Elliptic Operators with Non-smooth Coefficients”. In: *Math. Comp* 74 (2005), pp. 1179–1199.

[8] Amir Beck and Marc Teboulle. “Fast iterative shrinkage-thresholding algorithm for linear inverse problems”. In: *SIAM Journal of Imaging Sciences* 2.1 (2009), pp. 183–202.

[9] Stephen Becker, Jerome Bobin, and Emmanuel Candes. “NESTA: a fast and accurate first-order method for sparse recovery”. In: *SIAM Journal of Imaging Sciences* 4.1 (2011), pp. 1–39.

[10] R.E. Benner, G. R. Montry, and G. G. Weigand. “Concurrent multifrontal methods: shared memory, cache, and front width issues”. In: *Internat. J. Supercomput. Appl.* 1 (1987), pp. 26 –44.

[11] E. van den Berg et al. “Algorithm 890: SPARCO: a testing framework for sparse reconstruction”. In: *ACM Transaction on Mathematical Software* 35.4 (2009), p. 16.

[12] Ewout van den Berg and Michael P. Friedlander. “Probing the Pareto frontier for basis pursuit solutions”. In: *SIAM Journal of Scientific Computing* 31.2 (2009), pp. 890–912.

[13] Dimitri P. Bertsekas. *Nonlinear Programming.* Belmont, MA, 1999.

[14] Dario A Bini, Luca Gemignani, and Victor Y Pan. “Fast and stable QR eigenvalue algorithms for generalized companion matrices and secular equations”. In: *Numer. Math* 100 (), pp. 373–408.

[15] Ernesto G. Birgin, Jos É Mario Martínez, and Marcos Raydan. “Nonmonotone spectral projected-gradient methods on convex sets”. In: *SIAM Journal of Optimization* 10.4 (2000), pp. 1196–1211.

[16] Jerome Bobin, Jean-Luc Starck, and Roland Ottensamer. “Compressed sensing in astronomy”. In: *IEEE Journal of Selected Topics Signal Processing* 2.5 (2008), pp. 718–726.

[17] Emmanuel Candes, Justin Romberg, and Terence Tao. “Stable signal recovery from incomplete and inaccurate measurements”. In: *Communications on Pure and Applied Mathematics* 59.8 (2006), pp. 1207–1223.

[18] Emmanuel Candes and Terence Tao. “The Dantzig selector: statistical estimation when p is much larger than n”. In: *Annals of Statistics* 35.6 (2007), pp. 2313–2351.

[19] Tony Chan. “Rank revealing QR factorizations”. In: *Numer. Math* 88/89 (1987), pp. 67–82.

[20] S. Chandrasekaran and M. Gu. “A fast and stable solver for recursively semiseparable systems of equations”. In: *Structured Matrices in Mathematics, Computer Science and Engineering, II, V. Olshevsky, ed., Contemp. Math. 281* AMS, Providence, RI (2001), pp. 39–53.

[21] S. Chandrasekaran, M. Gu, and T. Pals. “A fast ULV decomposition solver for hierarchically semiseparable representations”. In: *SIAM Journal on Matrix Analysis and Applications* 28 (2006), pp. 603–622.

[22] S Chandrasekaran, M Gu, and T Pals. “Fast and stable algorithms for hierarchically semi-separable representations. Technical Report”. In: (2004).

[23] S Chandrasekaran et al. “Fast stable solvers for sequentially semiseparable linear systems of equations and least squares problems. http://cobalt.et.tudelft.nl/ allejan/publications.html, preprint”. In: (2001).

[24] S. Chandrasekaran et al. “Some fast algorithms for sequentially semiseparable representation”. In: *SIAM Journal on Matrix Analysis and Applications* 27 (2005), pp. 341–364.

[25] Scott Shaobing Chen, David L. Donoho, and A. Saunders Michael. “Atomic decomposition by basis pursuit”. In: *SIAM Journal of Scientific Computing* 20.1 (1998), pp. 33–61.

[26] A. R. Conn et al. "Performance of a multifrontal scheme for partially separable optimization". In: *Tech. Report 88/4* Department of Combinatorics and Optimization, University of Waterloo, Ontario, Canada (1989).

[27] Ron S. Dembo, Stanley C. Eisenstat, and Trond Steihaug. "Inexact newton methods". In: *SIAM Journla of Numerical Analysis* 19.2 (1982), pp. 400–408.

[28] David L. Donoho. "For most large underdetermined systems of linear equations the ℓ_1-norm solution is also the sparsest solution". In: *Communications on Pure and Applied Mathematics* 59.6 (2006), pp. 797–829.

[29] J. Duchi et al. "Efficient Projections onto the ℓ_1–Ball for Learning in High Dimensions". In: *Proceedings of the 25th International Conference on Machine Learning, ICML 2008*. 2008.

[30] I. S. Duff and J. K. Reid. "The multifrontal solution of indefinite sparse symmetric linear equations". In: *ACM Bans. Math. Software* 9 (1983), pp. 302–325.

[31] Bradley Efron et al. "Least Angle Regression". In: *Annals of Statistics* 32.2 (2004), pp. 407–499.

[32] Y. Eidelman and I. Gohberg. "On a new class of structured matrices". In: *Integral Equations Operator Theory* 34 (1999), pp. 293 –324.

[33] Y. Eidelman and I. Gohberg. "On a new class of structured matrices". In: *Integral Equations Operator Theory* 34 (1999), pp. 293–324.

[34] Mrio A. T. Figueiredo, Robert D. Nowak, and Stephen J. Wright. "Gradient projection for sparse reconstruction: Application to compressed sensing and other inverse problems". In: *IEEE Journal of Selected Topics in Signal Processing* 1.4 (2007), pp. 586–597.

[35] Mrio A. T. Figueiredo, Robert D. Nowak, and Stephen J. Wright. "Sparse reconstruction by separable approximation". In: *IEEE Transaction on Signal Processing* 57.7 (2009), pp. 2479–2493.

[36] Jerome Friedman et al. "Pathwise coordinate optimization". In: *Annals of Applied Statistics* 1.2 (2007), pp. 302–332.

[37] Jean jacques Fuchs. "On sparse representations in arbitrary redundant bases". In: *IEEE Transaction of Information Theory* 1344 (2004).

[38] Michael R. Garey and David S. Johnson. *D.S.: Computers and Intractability. A guide to the theory of NP-completeness*. New York, NY: W. H. Freeman, 1979.

[39] Alan George. "Nested dissection of a regular finite element mesh". In: *SIAM J. Numer. Anal* 10 (1973).

[40] I. Gohberg, T. Kailath, and I. Koltrachttitle. "Linear complexity algorithms for semiseparable matrices". In: *Integral Equations and Operator Theory* 8 (1985), pp. 780–804.

[41] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Baltimore, MD: The Johns Hopkins University Press, 1996.

[42] L. Grasedyck, R. Kriemann, and S. Le Borne. “Domain-decomposition based $\mathcal{H}$-*LU preconditioners*”. In: *Domain Decomposition Methods in Science and Engineering XVI, O.B.Widlund and D.E.Keyes (eds.), Springer LNCSE* 55 (2006), pp. 661–668.

[43] L. Grasedyck, R. Kriemann, and S. Le Borne. “Parallel black box domain decomposition based $\mathcal{H}$-*LU preconditioning*, Technical Report 115”. In: (2005).

[44] M. Gu. “Randomized sampling I: Low-rank matrix approximations”. In: *SIAM Journal on Matrix Analysis and Applications* (submitted, 2011).

[45] M. Gu. “Randomized sampling II: Subspace iterations”. In: *SIAM Journal on Matrix Analysis and Applications* (submitted, 2011).

[46] M. Gu and S. C. Eisenstat. “Efficient algorithms for computing a strong-rank revealing QR factorization”. In: *SIAM J. Sci. Comput.* 17 (1996), pp. 848–869.

[47] Jr. H. P. Starr. “On the Numerical Solution of One-Dimensional Integral and Differential Equations, Ph.D. thesis, Department of Computer Science, Yale University”. In: (1992).

[48] W. Hackbusch. “A sparse matrix arithmetic based on $\mathscr{H}$*-matrices. Part I: Introduction to* $\mathscr{H}$*-matrices*”. In: *Computing* 62 (1999), pp. 89–108.

[49] W. Hackbusch and S. Börm. “Data-sparse approximation by adaptive $\mathscr{H}^2$*-matrices*”. In: *Computing* 69 (2002), pp. 1–35.

[50] W. Hackbusch, L. Grasedtck, and S. Borm. “An introduction to hierarchical matrices”. In: *Math. Bohem.* 127 (2002), pp. 229 –241.

[51] W. Hackbusch and B. Khoromskij. “A sparse matrix arithmetic based on $\mathscr{H}$*-matrices*, Part II: Application to multi-dimensional problems”. In: *Computing* 64 (2000), pp. 21–47.

[52] W. Hackbusch, B. Khoromskij, and S. Sauter. “On $\mathscr{H}^2$*-matrices*”. In: *Lectures on Applied Mathematics, Bungartz H, Hoppe RHW, Zenger C (eds). Springer: Berlin* (2000), pp. 9–29.

[53] W. Hackbusch and B. N. Khoromskij. “A sparse H-matrix arithmetic. Part-II: Application to multi-dimensional problems”. In: *Computing* 64 (2000), pp. 21 –47.

[54] Elaine T. Hale, Wotao Yin, and Yin Zhang. “A fixed-point continuation method for ℓ_1-regularized minimization with applications to compressed sensing”. In: *Rice University Technical Report* (2007).

[55] Elaine T. Hale, Wotao Yin, and Yin Zhang. “Fixed-point continuation for ℓ_1-min: Methodology and convergence”. In: *SIAM Journal of Opt.* 19.3 (2008), pp. 1107 – 1130.

[56] N. Halko, P. G. Martinsson, and J. A. Tropp. “Finding structure with randomness probabilistic algorithms for constructing approximate matrix decompositions”. In: *SIAM Review* 53 (2011), pp. 217–288.

[57] Gilles Hennenfent and Felix J. Herrmann. “Simply denoise: wavefield reconstruction via jittered undersampling”. In: *Geophysics* 73.3 (2008), pp. V19–V28.

[58] Gilles Hennenfent and Felix J. Herrmann. “Sparseness-constrained data continuation with frames: Applications to missing traces and aliased signals in 2/3-D”. In: *SEG Technical Program Expanded Abstracts* 24.1 (2005), pp. 2162–2165.

[59] B. M. Irons. “A frontal solution program for finite element analysis”. In: *Internat. J. Numer. Methods Engrg.* 2 (1970), pp. 5 –32.

[60] George Karypis and Vipin Kumar. *METIS - Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0.* Tech. rep. 1995.

[61] Shengguo Li et al. “New Efficient and Robust HSS Cholesky Factorization of SPD Matrices”. In: *SIAM J. Matrix. Anal. Appl.* 33 (2012), pp. 886–904.

[62] E. Liberty et al. “Randomized algorithms for the low-rank approximation of matrices”. In: *Proc. Natl. Acad. Sci. USA* 104 (2007), pp. 20167–20172.

[63] R. Lipton, D. Rose, and R. Tarjan. “Generalized nested dissection”. In: *SIAM J. Numer. Anal.* 16 (1979), pp. 346 –358.

[64] J. W. H. Liu. “The multifrontal method for sparse matrix solution: Theory and practice”. In: *SIAM Review* (1992), pp. 82 –109.

[65] J. W. H. Liu. “The role of elimination trees in sparse factorization”. In: *SIAM J. Matrix Anal. Appl.* 18 (1990), pp. 134 –172.

[66] R. Lucas. “Solving planar systems of equations on distributed-memory multiprocessor”. In: *Ph.D. thesis* Department of Electrical Engineering, Stanford University, Stanford CA (1987).

[67] P. G. Martinsson. “A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix”. In: *SIAM. J. Matrix Anal. Appl.* 32 (2011), pp. 1251–1274.

[68] P. G. Martinsson and V. Rokhlin. “A fast direct solver for boundary integral equations in two dimensions”. In: *J. Comput.Phys.* 205 (2005), pp. 1–23.

[69] P. G. Martinsson, V. Rokhlin, and M. Tygert. “A randomized algorithm for the approximation of matrices”. In: *Appl. Comput. Harmon. Anal.* 30 (2011), pp. 47–68.

[70] Balas K. Natarajan. “Sparse approximate solutions to linear systems”. In: *SIAM Journal of Computing* 24.2 (1995), pp. 227–234.

[71] Yu Nesterov. “A method for solving the convex programming problem with convergence rate $O(1/k^2)$”. In: *Doklady Akademii Nauk SSSR* 269.3 (1983), pp. 543–547.

[72] Yu Nesterov. "Gradient methods for minimizing composite objective function". In: *Core Discussion Paper 2007/76* (2007).

[73] Yu Nesterov. "Smooth minimization of non-smooth functions". In: *Mathematical Programming* 103.1 (2005), pp. 127–152.

[74] Michael R. Osborne, Brett Presnell, and Berwin A. Turlach. "A new approach to variable selection in least squares problems". In: *IMA Journal of Numerical Analysis* 20.3 (2000), pp. 389–403.

[75] Michael R. Osborne, Brett Presnell, and Berwin A. Turlach. "On the LASSO and its dual". In: *Journal of Computational and Graphical Statistics* 9.2 (2000), pp. 319–337.

[76] Stanislav P. Ponomarev. "Submersions and preimages of sets of measure zero". In: *Siberian Mathematical Journal* 28.1 (1987), pp. 153–163.

[77] J. K. Reid. "TREESOLVE, a Fortran package for solving large sets of linear finite element equations". In: *Tech Report CSS 155* Computer Sciences and Systems Division, AERE Harwell, Oxfordshire, UK (1984).

[78] Ralph T. Rockafellar. *Convex Analysis.* Princeton, NJ: Princeton University Press, 1970.

[79] V. Rokhlin. "Rapid solution of integral equations of scattering theory in two dimensions". In: *J. Comput.Phys.* 86 (1990), pp. 414–439.

[80] Jerome Romberg. "Imaging via compressive sensing". In: *IEEE Transaction on Signal Processing* 25.2 (2008), pp. 14–20.

[81] R. Schreiber. "A new implementation of sparse Gaussian elimination". In: *ACM Trans. Math. Software* 8 (1982), pp. 256–276.

[82] S. Teng. "Fast nested dissection for finite element meshes". In: *SIAM J. Matrix Anal. Appl.* 18 (1997), pp. 552–565.

[83] "The multifronal solution of indefinite sparse symmetric linear equations". In: *ACM Trans. Math. Software* 9 (1983), pp. 302–323.

[84] "The restricted isometry property and its implications for compressed sensing". In: ().

[85] Robert Tibshirani. "Regression shrinkage and selection via the LASSO". In: *Journal of the Royal Statistical Society: Series B* 58.1 (1996), pp. 267–288.

[86] Joel A. Tropp. "Just relax: Convex programming methods for identifying sparse signals in noise". In: *IEEE Transaction on Information Theory* 52.3 (2006), pp. 1030–1051.

[87] Paul Tseng. "On accelerated proximal gradient methods for convex-concave optimization". In: *Preprint* (2008).

[88] R. Vandebril, M. Van Barel, and N. Mastronardi. "Matrix Computations and Semiseparable Matrices, Volume I: Linear Systems". In: *Johns Hopkins University Press* (2008).

[89] R. Vandebril et al. "A bibliography on semiseparable matrices". In: *Calcolo* 42 (2005), pp. 249–270.

[90] S. Wang et al. "Massively parallel structured multifrontal solver for time-harmonic elastic waves in 3D anisotropic media". In: *Geophys. J. Int.* 191 (2012), pp. 346 –366.

[91] Zaiwen Wen et al. "A fast algorithm for sparse reconstruction based on shrinkage, subspace optimization and continuation". In: *SIAM Journal of Scientific Computing* 32.4 (2010), p. 1832.

[92] J. Xia. "Efficient structured multifrontal factorization for general large sparse matrices". In: *SIAM J. Sci. Computing* (2012).

[93] J. Xia. "Fast direct solvers for structured linear systems of equations". In: *Ph.D. thesis* Department of Applied Mathematics, University of California, Berkeley, Berkeley CA (2006).

[94] J. Xia. "On the complexity of some hierarchical structured matrices". In: *SIAM Journal on Matrix Analysis and Applications* (to appear, 2012).

[95] J. Xia. "Randomized Sparse Direct Solvers". In: *SIAM J. Matrix Anal. Appl.* (to appear 2012).

[96] J. Xia and M. Gu. "Robust approximate Cholesky factorization of rank-structured symmetric positive definite matrices". In: *SIAM Journal on Matrix Analysis and Applications* 31 (2010), pp. 2899–2920.

[97] J. Xia, Y. Xi, and M. Gu. "A superfast structured solver for Toeplitz linear systems via randomized sampling". In: *SIAM Journal on Matrix Analysis and Applications* (submitted, 2011).

[98] J. Xia et al. "Fast algorithm for hierarchically semiseparable matrices". In: *Numer. Linear Algebra Appl.* 17 (2010), pp. 953–976.

[99] J. Xia et al. "Superfast multifrontal method for large structured linear systems of equations". In: *SIAM J. Matrix. Anal. Appl.* 31 (2009), pp. 1382–1411.

[100] Wotao Yin et al. "Bregman iterative algorithms for l_1 minimization with applications to compressed sensing". In: *SIAM Journal of Imaging Sciences* 1.1 (2008), pp. 143–168.

[101] Yao-Liang Yu. "Nesterov's optimal gradient method". In: *http://webdocs.cs.ualberta.ca/~yaoliang/Non-smooth\%20Optimization.pdf* (2009).