The Dissertation Committee for James Louis Levitt certifies that this is the approved version of the following dissertation:

Randomized Algorithms for Revealing Hidden Structure in Data-Sparse Matrices

Committee:

Per-Gunnar Martinsson, Supervisor

George Biros, Co-supervisor

Erik Boman

Tan Bui-Thanh

Robert van de Geijn

Rachel Ward

Randomized Algorithms for Revealing Hidden Structure in Data-Sparse Matrices

by

James Louis Levitt

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

May 2022

Dedicated to my family.

Acknowledgments

I would like to thank Gunnar Martinsson and George Biros for their mentorship and support over the last several years. They have guided me toward interesting problems and taught me so much about research. I thank the other members of my committee, many of whom I have had the pleasure of working with and learning from.

I have benefited greatly from interacting with many labmates and collaborators. In particular, I acknowledge Gokberk Kabacaoglu, Chenhan Yu, Severin Reiz, Birkan Tunc, Bill March, Andreas Mang, Carlos Borges, Chao Chen, Sameer Tharakan, Maggie Myers, Field van Zee, Devangi Parikh, Hongru Yang, and Nicholai Tukanov.

I am fortunate to have had several mentors outside of graduate school. I am grateful to Erik Boman and Siva Rajamanickam, who supervised me during a summer internship at Sandia National Labs in Albuquerque, New Mexico. I thank Robert van de Geijn for introducing me to research in numerical linear algebra as an undergraduate, and for always being willing to lend a helping hand. Richard Newcomb, my high school math teacher, was a great influence in developing my enthusiasm for mathematics.

Finally, I thank my family for their unwavering love and support every step of the way.

Randomized Algorithms for Revealing Hidden Structure in Data-Sparse Matrices

Publication No. _____

James Louis Levitt, Ph.D.
The University of Texas at Austin, 2022

Supervisor: Per-Gunnar Martinsson
Co-supervisor: George Biros

This thesis describes a set of efficient algorithms for handling large dense matrices that have *rank-structure*. To simplify slightly, this means that they can be tessellated into $\mathcal{O}(N)$ submatrices in such a way that each submatrix is either small, or of numerically low rank. A matrix that is rank-structured can be stored economically, and is amenable to fast algorithms for performing arithmetic operations such as matrix-vector multiplication. Rank-structured matrices arise in solvers for elliptic PDEs, for problems in data mining and statistics involving smooth kernel functions, and many more.

Recent algorithms [64, 77] compute approximations to rank-structured matrices by accessing the matrix only through a black-box matrix-vector multiplication routine (e.g., an FMM). The need to operate directly on off-diagonal blocks is avoided by instead using randomized samples of the full matrix. The first part of this thesis presents new algorithms for black-box

randomized compression of rank-structured matrices. These algorithms improve on prior work in terms of both their efficiency and their range of applicability.

In general, a matrix that admits a rank-structured approximation only does so under an appropriate ordering of its rows and columns. Therefore, finding a suitable ordering of the rows and columns of a matrix is an essential step in building a rank-structured approximation. In prior work, ordering techniques were developed for special classes of rank structured matrices, such as those arising from a given sparse matrix, or through a kernel matrix associated with a given set of points. The second part of the thesis describes new techniques that apply to broader classes of rank-structured matrices and achieve a balance between computational cost, storage requirements, and accuracy.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This thesis describes a set of efficient algorithms for handling large dense matrices that have *rank structure*. To simplify slightly, this means that an $N \times N$ matrix can be tessellated into $O(N)$ blocks in such a way that each block is either small or of low numerical rank (cf. Fig. 1.1). This structure allows the matrix to be stored and applied to vectors efficiently, often with cost that scales linearly or close to linearly with N. Sometimes, it is also possible to compute an approximate inverse or LU factorization in linear or close to linear time. Matrices of this type have turned out to be ubiquitous in both engineering and data sciences, and have been the subject of much research in recent decades, going under names such as $\mathcal{H}$-matrices [8, 12, 43]; HODLR matrices [3, 75], Hierarchically Block Separable (HBS) or Hierarchically Semi-Separable (HSS) matrices [16, 17, 102], Recursive Skeletonization [34, 48, 84, 86], and many more.

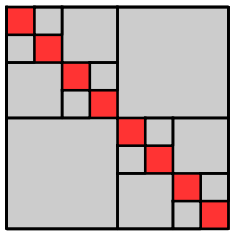


Figure 1.1 *The figure shows an example of a rank-structured matrix. Each off-diagonal block (gray) has low numerical rank, and each diagonal block (red) is small. The tessellation pattern shown is just one example among many possibilities. For matrices of this type, efficient algorithms exist for matrix-vector multiplication, matrix inversion, LU decomposition, etc.*

The original algorithms for rank-structured matrices were developed for accelerating the simulation of physical phenomena such as acoustic and electromagnetic scattering problems, for evaluating electrostatic interactions

in molecular dynamics simulations, and so on. Among the best known early techniques is the celebrated Fast Multipole Method (FMM) of Greengard and Rokhlin [38]. These techniques were later generalized by Hackbusch and co-workers [43], who developed the so called $\mathcal{H}$-matrix techniques to extend the range of supported arithmetic operations to include tasks such as inversion, LU factorization, and matrix-matrix multiplication.

Recent algorithms [64, 77] compute approximations to rank-structured matrices by accessing the matrix only through a black-box matrix-vector multiplication routine (e.g., an FMM). The need to operate directly on off-diagonal blocks is avoided by instead using randomized samples of the full matrix. The first part of this thesis presents new algorithms for black-box randomized compression of rank-structured matrices. These algorithms improve on prior work in terms of both their efficiency and their range of applicability.

The FMM and related techniques based on rank-structured matrices are at this time widely used in traditional scientific computing. Recently, it has been demonstrated that they also hold great promise for applications in machine learning and data science. These environments pose new challenges, however. The second part of this thesis addresses the challenge of finding permutations that reveal rank structure for matrices that are neither sparse nor for which geometric information is available.

1.1 Black-box randomized compression

The black-box randomized compression problem is the following: Suppose that $\mathbf{A}$ is an $N \times N$ matrix that we know is rank-structured, but we do not have direct access to the low-rank factors that define the compressible off-diagonal blocks. Instead, we have access to fast algorithms that given tall thin matrices $\mathbf{\Omega}, \mathbf{\Psi} \in \mathbb{R}^{N \times \ell}$, evaluate the matrix-matrix products

$$\mathbf{Y} = \mathbf{A}\mathbf{\Omega}, \qquad \text{and} \qquad \mathbf{Z} = \mathbf{A}^*\mathbf{\Psi}.$$

The problem is then to recover $\mathbf{A}$ from the information in the set $\{\mathbf{Y}, \mathbf{\Omega}, \mathbf{Z}, \mathbf{\Psi}\}$.

The presented schemes have several important applications. First, they can be used to derive a rank-structured representation of any integral operator for which a fast matrix-vector multiplication algorithm, such as the Fast Multipole Method [38, 40], is available. Such a representation opens the door to a wider range of matrix operations such as LU factorization, matrix inversion, and sometimes even full spectral decompositions. Second, it can greatly simplify algebraic operations involving products of rank-structured matrices. For instance, the perhaps key application of rank-structured matrix algebra is the acceleration of sparse direct solvers, as the dense matrices that arise during LU factorization are often rank-structured. In the course of such a solver, a typical operation would be to form a Schur complement such as $\mathbf{S}_{22} = \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}$ that would arise when the top left block $\mathbf{A}_{11}$ is eliminated from a 2×2 blocked matrix. If $\mathbf{A}_{11}$ is rank-structured, then $\mathbf{A}_{11}^{-1}$ can easily be applied to vectors via an LU factorization. If, additionally, $\mathbf{A}_{12}$ and

$\mathbf{A}_{21}$ are either sparse or rank-structured, then $\mathbf{S}_{22}$ can easily be applied to a vector. The technique described will then enable one to construct a data-sparse representation of $\mathbf{S}_{22}$. In contrast, to directly evaluate the product $\mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}$ is both onerous to code and slow to execute.

Prior work on these problems include the randomized compression algorithm of [76]. Those algorithms require only $\sim k$ samples to compress a matrix with the HBS format (also known as the HSS format), and they have linear complexity when the cost of applying $\mathbf{A}$ to a vector is $\mathcal{O}(N)$. However, they are not true black box algorithms since, in addition to randomized samples, they also require direct evaluation of a small number of entries of the matrix. Another family of algorithms [64, 77], known as *peeling algorithms*, are true black-box algorithms in that they only access $\mathbf{A}$ through matrix-vector products and do not require directly accessing matrix entries. However, they require $\sim k \log N$ samples, and $\sim k^2 N \log N$ additional floating-point operations, so they do not have linear complexity.

To our knowledge, the method presented in Chapter 2 (and also in **??**) is the first linear-complexity black-box algorithm for compressing HBS matrices. The compression algorithm requires only $\sim k$ samples and it has linear complexity when the cost of applying $\mathbf{A}$ to a vector is $\mathcal{O}(N)$. It is inspired by the randomized compression algorithms of [76], which also require only $\sim k$ samples to compress an HBS matrix, but are not truly black-box.

The method described in Chapter 3 (and also in **??**) is inspired by the *peeling algorithm* of [64], which to the best of our knowledge is the first

true black box algorithm described in the literature. The method of [64] has formally the same sample complexity $\ell \sim k \log(N)$ as our method, but involves substantially larger pre-factors. To be precise, [64] is targeted specifically for $\mathcal{H}^1$- and $\mathcal{H}^2$-matrices arising from the discretization of integral equations. Strong admissibility, and regular tree structures are used. In this environment, the method requires $\ell \sim k\, 8^d \log(N)$ matrix-vector products involving $\mathbf{A}$ and $\mathbf{A}^*$, where d is the dimension of space in which the underlying integral equation is defined. In contrast, the method presented here has complexity $\ell \sim k\, 6^d \log(N)$ for fully populated uniform trees. For more general trees, the acceleration over the method of [64] is even more dramatic, since the adaptivity of our method enables it to exploit situations where the matrix arises from a set of points located on a lower dimensional geometric object. This method is not competitive with the one of Chapter 3 in terms of computational cost, but it nonetheless represents an advance over prior peeling algorithms, and is applicable to additional classes of rank-structured matrices.

1.2 Permutation to reveal rank-structure

In general, a matrix that admits a rank-structured approximation only does so under an appropriate ordering of its rows and columns. The second part of this thesis considers the problem of finding permutations of matrices that reveal rank structure. Prior methods are limited to kernel matrices with some associated geometric information or to sparse matrices. We present two new approaches to the permutation problem that apply to previously unaddressed

classes of rank-structured matrices: one for dense symmetric positive definite matrices and one for general dense matrices. In Chapter 4, we describe the Geometry-Oblivious Fast Multipole Method (GOFMM) [107], with which we approach the problem of reordering a symmetric positive-definite matrix by viewing the matrix as a kernel matrix with the linear kernel $\mathcal{K}(\phi_i, \phi_j) = \langle \phi_i, \phi_j \rangle$ applied to a set of Gram vectors $\{\phi_i\}$ whose coordinates are unspecified. While it would be possible to compute coordinates of the Gram vectors, doing so would be prohibitively expensive. Instead, we operate on the Gram vectors implicitly by defining measures of similarity on pairs of Gram vectors expressed in terms of only a small number matrix entries. We use the *geometry-oblivious* similarity measures to generalize techniques developed in the ASKIT algorithm [72] for approximation of rank-structured kernel matrices.

In Chapter 5, we present an approach for reordering matrices to reveal rank structure (assuming such latent structure exists) for general dense matrices. The reordering of the matrix is determined using statistical leverage scores. Leverage scores have a rich history of application in regression diagnostics and, more recently, for randomized low-rank approximation [30, 65, 88]. We use leverage scores to partition certain blocks of a matrix into compressible and incompressible blocks, leading to a reordering of the matrix with low-rank off-diagonal blocks.

1.3 Contributions

A summary of contributions by CSEM area follows.

Area A

- The presented works draw on numerous areas of mathematical theory including cluster analysis (Chapters 4 and 5), optimization (Chapter 5), graph theory (Chapter 3). In particular, randomized linear algebra plays a critical role throughout the thesis.

- The compression schemes apply statistical and linear algebraic techniques to estimate the error in various parts of the approximation. Such techniques are used during the process of compression, e.g., for adaptively determining the rank of a low-rank approximation in order to meet some error tolerance. They are also used for *a posteriori* error estimation of the data-sparse representation.

- In Chapter 3, we analyze the number of vertices the degree of the graphs we construct. These properties are useful in analyzing the complexity of our algorithms.

Area B

- Two algorithms for black-box randomized compression of rank-structured matrices are developed and implemented. Notably, the algorithm of Chapter 2 is, to our knowledge, the first algorithm for compressing HBS matrices that is black-box and has linear-complexity.

- Two algorithms for generalized permutation of rank-structured matrices are developed and implemented.

- The implementations are designed with performance and numerics in mind, and include features to enhance usability, such as the ability to adaptively determining approximation ranks of compressible blocks so as to meet a user-specified error tolerance.

- Experimental results analyze scalability, accuracy, and efficacy of each algorithm.

Area C

- The geometry-aware variant of the GOFMM compression algorithm are applied to matrices arising from problems in electromagnetic scattering.

- The black-box randomized compression algorithms are tested on several applications including discretization of boundary integral equations and frontal matrices in nested dissection.

1.3.1 Published and in-progress works

- Levitt, J. L., Martinsson, P.-G. (2022). Linear-Complexity Black-Box Randomized Compression of Hierarchically Block Separable Matrices. arXiv preprint arXiv:2205.02990.

- Levitt, J. L., Martinsson, P.-G. (2022). Compressing rank-structured matrices with graph coloring. arXiv preprint arXiv:2205.03406.

- Yu, C. D., Levitt, J., Reiz, S., & Biros, G. (2017). Geometry-oblivious FMM for compressing dense SPD matrices. In *Proceedings of the In-*

ternational Conference for High Performance Computing, Networking, Storage and Analysis (pp. 1-14).

- Mang, A., Tharakan, S., Gholami, A., Himthani, N., Subramanian, S., Levitt, J., Azmat, M., Mehl, M., Davatzikos, C., Barth, B., & Biros, G. (2017). SIBIA-GlS: Scalable Biophysics-Based Image Analysis for Glioma Segmentation. *The multimodal brain tumor image segmentation benchmark (BRATS), MICCAI.*

- Levitt, J. L., Boman, E. G., Rajamanickam, S., & Biros, G. (2018). Nonsymmetric algebraic FMM with application to combined field integral equations. In *Center for Computing Research Summer Proceedings 2018, 2018. Technical Report SAND2019-5093R.*

- Levitt, J. L., Biros, G. (2022). Rank-structured approximation of general dense matrices with leverage score clustering. Unpublished manuscript, The University of Texas at Austin, Oden Institute for Computational Engineering and Sciences, Austin, USA.

1.4 Outline

The remainder of this thesis is structured as follows.

Part I addresses the problem of black-box randomized compression of rank-structured matrices. Chapter 2 presents a linear-complexity algorithm for compressing HBS matrices, and Chapter 3 presents peeling algorithms

accelerated with graph coloring for compressing $\mathcal{H}^1$, uniform $\mathcal{H}^1$, and $\mathcal{H}^2$ matrices.

Part II covers the permutation schemes for revealing rank structure. Chapter 4 describes the Geometry-Oblivious Fast Multipole Method for permuting symmetric positive-definite matrices, and Chapter 5 describes the leverage score clustering scheme for general dense matrices. Chapter 6 presents an application of the geometry-aware variant of the GOFMM compression algorithm to matrices arising from problems in electromagnetic scattering.

Part I

Black-Box Randomized Compression of Rank-Structured Matrices

Chapter 2

Linear-Complexity Black-Box Randomized Compression of Hierarchically Semiseparable Matrices[1]

[1]The content in this chapter is based on work done in collaboration with Per-Gunnar Martinsson and published in [60].

A randomized algorithm for computing a compressed representation of a given rank structured matrix $\mathbf{A} \in \mathbb{R}^{N \times N}$ is presented. The algorithm interacts with $\mathbf{A}$ only through its action on vectors. Specifically, it draws two tall thin matrices $\mathbf{\Omega}$, $\mathbf{\Psi} \in \mathbb{R}^{N \times s}$ from a suitable distribution, and then reconstructs $\mathbf{A}$ by analyzing the matrices $\mathbf{A\Omega}$ and $\mathbf{A}^*\mathbf{\Psi}$. For the specific case of a "Hierarchically Block Separable (HBS)" matrix (a.k.a. Hierarchically Semi-Separable matrix) of block rank k, the number of samples s required satisfies $s = O(k)$, with $s \approx 3k$ being a typical scaling. While a number of randomized algorithms for compressing rank structured matrices have previously been published, the current algorithm appears to be the first that is both of truly linear complexity (no $N \log(N)$ factors) and fully black-box in nature (in the sense that no matrix entry evaluation is required).

2.1 Introduction

This work describes a set of efficient algorithms for handling large dense matrices that have *rank structure*. To simplify slightly, this means that an $N \times N$ matrix can be tessellated into $O(N)$ blocks in such a way that each block is either small or of low numerical rank, cf. Fig. 2.2. This structure allows the matrix to be stored and applied to vectors efficiently, often with cost that scales linearly or close to linearly with N. Sometimes, it is also possible to compute an approximate inverse or LU factorization in linear or close to linear time. Matrices of this type have turned out to be ubiquitous in both engineering and data sciences, and have been the subject of much research in recent decades,

going under names such as $\mathcal{H}$-matrices [8, 12, 43]; HODLR matrices [3, 75], Hierarchically Block Separable (HBS) or Hierarchically Semi-Separable (HSS) matrices [16, 17, 102], Recursive Skeletonization [34, 48, 84, 86], and many more.

The specific problem we address is the following: Suppose that $\mathbf{A}$ is an $N \times N$ matrix that we know has HBS structure, but we do not have direct access to the low-rank factors that define the compressible off-diagonal blocks. Instead, we have access to fast "black-box" algorithms that given tall thin matrices $\mathbf{\Omega}, \mathbf{\Psi} \in \mathbb{R}^{N \times s}$, evaluate the matrix-matrix products

$$\mathbf{Y} = \mathbf{A\Omega}, \qquad \text{and} \qquad \mathbf{Z} = \mathbf{A}^*\mathbf{\Psi}.$$

The problem is then to recover $\mathbf{A}$ from the information in the set $\{\mathbf{Y}, \mathbf{\Omega}, \mathbf{Z}, \mathbf{\Psi}\}$. The algorithms described here solve the reconstruction problem using $s = O(k)$ sample vectors, where k is an upper bound on the ranks of the off-diagonal blocks. (The pre-factor hidden in the formula $s = O(k)$ is often modest, with $s \approx 3k$ being representative.)

The scheme presented has several important applications. First, it can be used to derive a rank-structured representation of any integral operator for which a fast matrix-vector multiplication algorithm, such as the Fast Multipole Method [38, 40], is available. Such a representation opens the door to a wider range of matrix operations such as LU factorization, matrix inversion, and sometimes even full spectral decompositions. Second, it can greatly simplify algebraic operations involving products of rank-structured matrices. For instance, the perhaps key application of rank-structured matrix

algebra is the acceleration of sparse direct solvers, as the dense matrices that arise during LU factorization are often rank-structured. In the course of such a solver, a typical operation would be to form a Schur complement such as $\mathbf{S}_{22} = \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}$ that would arise when the top left block $\mathbf{A}_{11}$ is eliminated from a 2×2 blocked matrix. If $\mathbf{A}_{11}$ is rank-structured, then $\mathbf{A}_{11}^{-1}$ can easily be applied to vectors via an LU factorization. If, additionally, $\mathbf{A}_{12}$ and $\mathbf{A}_{21}$ are either sparse or rank-structured, then $\mathbf{S}_{22}$ can easily be applied to a vector. The technique described will then enable one to construct a data-sparse representation of $\mathbf{S}_{22}$. In contrast, to directly evaluate the product $\mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}$ is both onerous to code and slow to execute.

The method we describe is inspired by the randomized compression algorithms of [76]. The algorithms described in that work require only $O(k)$ samples to compress an HBS matrix, and they have linear complexity when the cost of applying $\mathbf{A}$ to a vector is $\mathcal{O}(N)$. However, they are not true black-box algorithms since, in addition to randomized samples, they also require direct evaluation of a small number of entries of the matrix. In contrast, the method presented here is truly black-box.

Remark 1 (Peeling algorithms)**.** *A related class of algorithms for randomized compression of rank-structured matrices is described in [64, 77]. These techniques are "true" black-box algorithms in that they only access the matrix through the black-box matrix-vector multiplication routines, but they require* $O(k \log(N))$ *samples and* $O(k^2 N \log(N))$ *floating point operations, so they do not have linear complexity.*

The manuscript is structured as follows: Section 2.2 surveys some basic linear algebraic techniques that we rely on. Section 2.3 introduces our formalism for HBS matrices. Section 2.4 describes the new algorithm, and analyzes its asymptotic complexity. Section 2.5 describes numerical results.

2.2 Preliminaries

In introducing well-known material, we follow the presentation of [77].

2.2.1 Notation

Throughout the paper, we measure a vector $\mathbf{x} \in \mathbb{R}^n$ by its Euclidean norm $\|\mathbf{x}\| = (\sum_i |x_i|^2)^{\frac{1}{2}}$. We measure a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ with the corresponding operator norm $\|\mathbf{A}\| = \sup_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\|$, and in some cases with the Frobenius norm $\|\mathbf{A}\|_{\text{Fro}} = (\sum_{i,j} |\mathbf{A}(i,j)^2)^{1/2}$. To denote submatrices, we use the notation of Golub and van Loan [35]: If $\mathbf{A}$ is an $m \times n$ matrix, and $I = [i_1, i_2, \ldots, i_k]$ and $J = [j_1, j_2, \ldots, j_k]$, then $\mathbf{A}(I, J)$ denotes the $k \times l$ matrix

$$\mathbf{A}(I,J) = \begin{bmatrix} \mathbf{A}(i_1,j_1) & \mathbf{A}(i_1,j_2) & \ldots & \mathbf{A}(i_1,j_l) \\ \mathbf{A}(i_2,j_1) & \mathbf{A}(i_2,j_2) & \ldots & \mathbf{A}(i_2,j_l) \\ \vdots & \vdots & & \vdots \\ \mathbf{A}(i_k,j_1) & \mathbf{A}(i_k,j_2) & \ldots & \mathbf{A}(i_k,j_l) \end{bmatrix}$$

We let $\mathbf{A}(I,:)$ denote the column submatrix $\mathbf{A}(I,[1,2,\ldots,n])$ and let $\mathbf{A}(:,J)$ denote the analogous row submatrix of $\mathbf{A}$. We let $\mathbf{A}^*$ denote the transpose of $\mathbf{A}$, and we say that matrix $\mathbf{U}$ is *orthonormal* if its columns are orthonormal, $\mathbf{U}^*\mathbf{U} = \mathbf{I}$.

2.2.2 The QR factorization

The *full QR factorization* of a matrix $\mathbf{A}$ of size $m \times n$ takes the form

$$\underset{m \times n}{\mathbf{A}} = \underset{m \times m}{\mathbf{Q}} \; \underset{m \times n}{\mathbf{R}}, \tag{2.1}$$

where $\mathbf{Q}$ is orthonormal and $\mathbf{R}$ is upper-triangular. For a matrix of rank k, the rank-k partial QR factorization of $\mathbf{A}$ is given by

$$\underset{m \times n}{\mathbf{A}} = \underset{m \times k}{\mathbf{Q}(:, 1:k)} \; \underset{k \times n}{\mathbf{R}(1:k, 1:k)},$$

and the last $m - k$ columns of $\mathbf{Q}$ contain an orthonormal basis of the nullspace of $\mathbf{A}$,

$$\text{span}(\mathbf{Q}(:, (k+1):m)) = \mathcal{N}(\mathbf{A}).$$

2.2.3 Randomized compression

Let $\mathbf{A}$ be an $m \times n$ matrix that can be accurately approximated by a matrix of rank k, and suppose we seek to determine a matrix $\mathbf{Q}$ with orthonormal columns (as few as possible) such that $\|\mathbf{A} - \mathbf{Q}\mathbf{Q}^*\mathbf{A}\|$ is small. In other words, we seek a matrix $\mathbf{Q}$ whose columns form an approximate orthornomal basis (ON-basis) for the column space of $\mathbf{A}$. This task can efficiently be solved via the following randomized procedure:

1. Pick a small integer p representing how much "oversampling" is done. ($p = 10$ is often good.)

2. Form an $n \times (k+p)$ matrix $\mathbf{G}$ whose entries are independent and identically distributed (i.i.d.) normalized Gaussian random numbers.

3. Form the "sample matrix" $\mathbf{Y} = \mathbf{AG}$ of size $m \times (k+p)$.

4. Construct an $m \times (k+p)$ matrix $\mathbf{Q}$ whose columns form an ON basis for the columns of $\mathbf{Y}$.

Note that each column of the sample matrix $\mathbf{Y}$ is a random linear combination of the columns of $\mathbf{A}$. We would therefore expect the algorithm described to have a high probability of producing an accurate result when p is a large number. It is perhaps less obvious that this probability depends only on p (not on m or n, or any other properties of $\mathbf{A}$) and that it approaches 1 extremely rapidly as p increases. In fact, one can show that the basis $\mathbf{Q}$ determined by the scheme above satisfies

$$\|\mathbf{A} - \mathbf{QQ}^*\mathbf{A}\| \leq [1 + 11\sqrt{k+p} \cdot \sqrt{\min\{m,n\}}]\sigma_{k+1} \tag{2.2}$$

with probability at least $1 - 6 \cdot p^{-p}$ (see [46, sec. 1.5]), where σ_{k+1} is the $(k+1)$-largest singular value of $\mathbf{A}$. The error bound Eq. (2.2) indicates that the error produced by the randomized sampling procedure can be larger than the theoretically minimal error σ_{k+1} by a factor of $1 + 11\sqrt{k+p} \cdot \sqrt{\min\{m,n\}}$. This crude bound is typically very pessimistic, in particular for matrices whose singular values decay rapidly; cf. [46].

2.2.4 Functions for computing orthonormal bases

We write

$$\mathbf{Q} = \mathtt{qr}(\mathbf{B}, k),$$

for a function call that returns the first k columns of $\mathbf{Q}$ in a QR factorization, and we write

$$\mathbf{Q} = \texttt{null}(\mathbf{B}, k),$$

for a function call that returns the last k columns of $\mathbf{Q}$ in a QR factorization. We only call `null` with inputs $\mathbf{B}, k$ for a matrix $\mathbf{B}$ that is known to have a nullspace of dimension at least k.

2.3 Hierarchically semiseparable matrices

In this section, we review important concepts relating to HBS matrices, based on the presentation of [79]. We introduce a binary tree structure that specifies the tessellation of an HBS matrix, give the precise definition of an HBS matrix, and describe telescoping factorizations of HBS matrices.

2.3.1 A tree structure

Let $I = [1, 2, \ldots, N]$ be a vector of indices corresponding to the rows and columns of an $N \times N$ matrix. We define a tree $\mathcal{T}$, in which each node τ is associated with a contiguous subset of the indices I_τ. To the root node of the tree, we assign the full set of indices I. The two children of the root node are given index vectors $[1, \ldots, \lceil N/2 \rceil]$ and $[\lceil N/2 \rceil + 1, \ldots, N]$. We continue evenly splitting the indices of each node to form two child nodes until we reach a level of the tree in which the size of each node is below some given threshold m. We refer to a node with children as a parent node, and a node with no children as a leaf node. The depth of a node is defined as its distance from the root node,

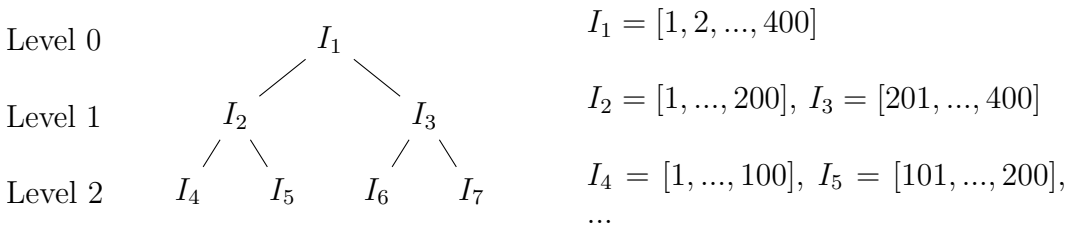


Figure 2.1 *A binary tree structure, where the levels of the tree represent successively refined partitions of the index vector* $[1, 2, ..., 400]$.

and level ℓ of the tree is defined as the set of nodes with depth ℓ, so that level 0 consists of only the root node, level 1 consists of the two children of the root node, and so on. The levels of the tree represent successively finer partitions of I. The depth of the tree is defined as the maximum node depth, denoted by $L \approx \log_2 N/m$. For simplicity, we only consider fully populated binary trees. An example tree structure is depicted in Fig. 2.1.

2.3.2 The HBS matrix format

Let $\mathcal{T}$ be a tree defined on index vector $I = [1, 2, \ldots, N]$. Matrix $\mathbf{A}$ of size $N \times N$ is said to be hierarchically block separable with block rank k with respect to $\mathcal{T}$ if the following conditions are satisfied.

(1) *Assumptions on the ranks of off-diagonal blocks of the finest level*: For every pair of distinct leaf nodes τ and τ', we define

$$\mathbf{A}_{\tau,\tau'} = \mathbf{A}(I_\tau, I_{\tau'})$$

and require that every such matrix have rank of at most k. Additionally, for each leaf node τ there must exist basis matrices $\mathbf{U}_\tau$ and $\mathbf{V}_\tau$ such that for every

leaf node $\tau' \neq \tau$ we have

$$\begin{array}{ccccccc} \mathbf{A}_{\tau,\tau'} & = & \mathbf{U}_\tau & \tilde{\mathbf{A}}_{\tau,\tau'} & \mathbf{V}^*_{\tau'}. \\ m \times m & & m \times k & k \times k & k \times m \end{array} \tag{2.3}$$

(2) *Assumptions on the ranks of off-diagonal blocks of levels* $L-1, L-2, \ldots 1$. The following conditions must hold for each level $\ell = L-1, L-2, \ldots, 1$. For every pair of distinct nodes τ and τ' on level ℓ with children (α, β) and (α', β'), respectively, we define

$$\mathbf{A}_{\tau,\tau'} = \begin{bmatrix} \tilde{\mathbf{A}}_{\alpha,\alpha'} & \tilde{\mathbf{A}}_{\alpha,\beta'} \\ \tilde{\mathbf{A}}_{\beta,\alpha'} & \tilde{\mathbf{A}}_{\beta,\beta'} \end{bmatrix}$$

and require that every such matrix have rank of at most k. Additionally, for each node τ on level ℓ there must exist basis matrices $\mathbf{U}_\tau$ and $\mathbf{V}_\tau$ such that for every node $\tau' \neq \tau$ on level ℓ, we have

$$\begin{array}{ccccccc} \mathbf{A}_{\tau,\tau'} & = & \mathbf{U}_\tau & \tilde{\mathbf{A}}_{\tau,\tau'} & \mathbf{V}^*_{\tau'}. \\ 2k \times 2k & & 2k \times k & k \times k & k \times 2k \end{array}$$

Notably, no assumptions are made on the ranks of the on-diagonal blocks of level L, and those blocks may have full rank. An example tessellation of an HBS matrix showing compressible and incompressible blocks is given in Fig. 2.2.

2.3.3 Telescoping factorizations

We define the following block-diagonal basis matrices.

$$\mathbf{U}^{(\ell)} = \text{diag}(\mathbf{U}_\tau : \tau \text{ is a node on level } \ell), \qquad \ell = 1, 2, \ldots, L$$

$$\mathbf{V}^{(\ell)} = \text{diag}(\mathbf{V}_\tau : \tau \text{ is a node on level } \ell), \qquad \ell = 1, 2, \ldots, L$$

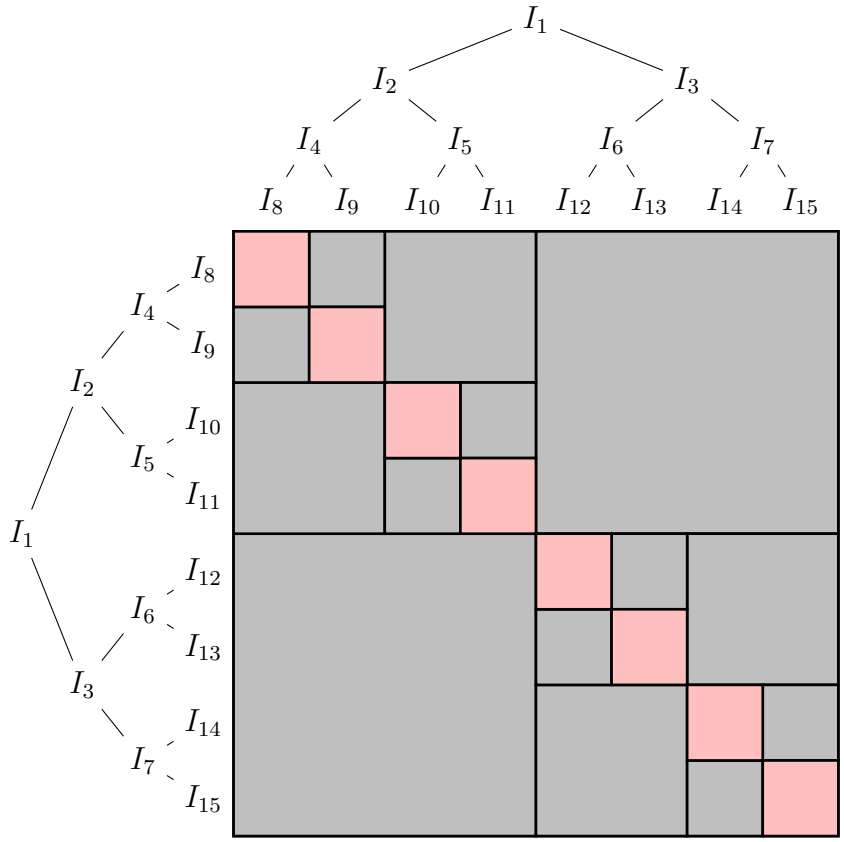


Figure 2.2 *Tessellation of an HBS matrix with depth 3. Low-rank blocks are shown in gray, and blocks that are not necessarily low-rank are shown in pink.*

Then we obtain a factorization of level L of the form

$$\mathbf{A} = \mathbf{U}^{(L)}\tilde{\mathbf{A}}^{(L)}(\mathbf{V}^{(L)})^* + \mathbf{D}^{(L)}, \tag{2.4}$$

where

$$\tilde{\mathbf{A}}^{(L)} = (\mathbf{U}^{(L)})^*\mathbf{A}\mathbf{V}^{(L)}$$

$$\mathbf{D}^{(L)} = \mathbf{A} - \mathbf{U}^{(L)}\tilde{\mathbf{A}}^{(L)}(\mathbf{V}^{(L)})^*.$$

Equation (2.4) can be viewed as a decomposition of $\mathbf{A}$ into a term that "fits" into the low-rank approximation using basis matrices $\mathbf{U}^{(L)}$ and $\mathbf{V}^{(L)}$ and a

discrepancy term $\mathbf{D}^{(L)}$ that does not.

For successively coarser levels $\ell = L-1, L-2, \ldots, 1$, we similarly have

$$\tilde{\mathbf{A}}^{(\ell+1)} = \mathbf{U}^{(\ell)} \tilde{\mathbf{A}}^{(\ell)} (\mathbf{V}^{(\ell)})^* + \mathbf{D}^{(\ell)}, \tag{2.5}$$

where

$$\tilde{\mathbf{A}}^{(\ell)} = (\mathbf{U}^{(\ell)})^* \tilde{\mathbf{A}}^{(\ell+1)} \mathbf{V}^{(\ell)}$$
$$\mathbf{D}^{(\ell)} = \tilde{\mathbf{A}}^{(\ell+1)} - \mathbf{U}^{(\ell)} \tilde{\mathbf{A}}^{(\ell)} (\mathbf{V}^{(\ell)})^*.$$

For the root level, we define

$$\mathbf{D}^{(0)} = \tilde{\mathbf{A}}^{(\ell+1)}.$$

Equations (2.4) and (2.5) define a telescoping factorization of $\mathbf{A}$. For example, a factorization with $L = 3$ takes the form

$$\mathbf{A} = \mathbf{U}^{(3)} (\mathbf{U}^{(2)} (\mathbf{U}^{(1)} \mathbf{D}^{(0)} (\mathbf{V}^{(1)})^* + \mathbf{D}^{(1)}) (\mathbf{V}^{(2)})^* + \mathbf{D}^{(2)}) (\mathbf{V}^{(3)})^* + \mathbf{D}^{(3)}.$$

Algorithm 2.3.1 describes the process of efficiently applying the telescoping factorization to a vector.

It follows from Eq. (2.3) that matrices $\mathbf{D}^{(\ell)}, \ell = 0, 1, \ldots L$, are also block-diagonal. Matrix $\mathbf{D}^{(\ell)}$ can be described in terms of its on-diagonal blocks as

$$\mathbf{D}^{(\ell)} = \mathrm{diag}(\mathbf{D}_\tau : \tau \text{ is a node on level } \ell),$$

where

$$\mathbf{D}_\tau = \mathbf{A}_{\tau,\tau} - \mathbf{U}_\tau \mathbf{U}_\tau^* \mathbf{A}_{\tau,\tau} \mathbf{V}_\tau \mathbf{V}_\tau^*. \tag{2.6}$$

Algorithm 2.3.1 Apply a compressed HBS matrix to a vector: $\mathbf{u} = \mathbf{A}\mathbf{q}$.

Upward pass
for level $\ell = L, L-1, \ldots, 1$ **do**
  for node τ in level L **do**
    if τ is a leaf node **then**
      $\hat{\mathbf{q}}_\tau = \mathbf{V}_\tau^* \mathbf{q}(I_\tau)$
    else
      Let α and β be the children of τ.
      $\hat{\mathbf{q}}_\tau = \mathbf{V}_\tau^* \begin{bmatrix} \hat{\mathbf{q}}_\alpha \\ \hat{\mathbf{q}}_\beta \end{bmatrix}$

Downward pass
for levels $\ell = 0, 1, \ldots, L$ **do**
  if τ is the root node **then**
    Let α and β be the children of τ.
    $\begin{bmatrix} \hat{\mathbf{u}}_\alpha \\ \hat{\mathbf{u}}_\beta \end{bmatrix} = \mathbf{D}_\tau \begin{bmatrix} \hat{\mathbf{q}}_\alpha \\ \hat{\mathbf{q}}_\beta \end{bmatrix}$
  else if τ is a parent node **then**
    Let α and β be the children of τ.
    $\begin{bmatrix} \hat{\mathbf{u}}_\alpha \\ \hat{\mathbf{u}}_\beta \end{bmatrix} = \mathbf{U}_\tau \hat{\mathbf{u}}_\tau + \mathbf{D}_\tau \begin{bmatrix} \hat{\mathbf{q}}_\alpha \\ \hat{\mathbf{q}}_\beta \end{bmatrix}$
  else
    $\mathbf{u}(I_\tau) = \mathbf{U}_\tau \hat{\mathbf{u}}_\tau + \mathbf{D}_\tau \mathbf{q}(I_\tau)$

Remark 2. *A common practice is to define the telescoping factorization with* $\mathbf{D}^{(L)}$ *as the block-diagonal part of* $\mathbf{A}$ *and with* $\tilde{\mathbf{A}}^{(L)}$ *as all but the block-diagonal part of* $\mathbf{A}$*, and to define* $\mathbf{D}^{(\ell)}$ *and* $\tilde{\mathbf{A}}^{(\ell)}$ *similarly for* $\ell = 0, 1, \ldots, L-1$*. That approach, which is taken by [76], necessitates directly accessing entries of the matrix to form the matrices* $\mathbf{D}^{(\ell)}$*. Our definition of the telescoping factorization facilitates compressing the matrix while accessing it only through randomized sampling, as will become clear in later sections.*

2.4 Compressing HSS matrices

In this section we present the main algorithm for compressing an HBS matrix. Let $\mathbf{A}$ be an $N \times N$ HBS matrix with block rank k, and let $r = k + p$, where p represents a small amount of oversampling ($p = 5$ or $p = 10$ are often sufficient). Let $\mathbf{\Omega}$ and $\mathbf{\Psi}$ be $N \times s$ Gaussian test matrices, where $s \geq \max(r + m, 3r)$, and define sample matrices $\mathbf{Y} = \mathbf{A}\mathbf{\Omega}$ and $\mathbf{Z} = \mathbf{A}^*\mathbf{\Psi}$. Our objective is to use the information contained in the test and sample matrices to construct a telescoping factorization of $\mathbf{A}$, as defined in Section 2.3.3.

We begin this section by describing the process of finding the level-L basis matrices $\mathbf{U}^{(L)}$ and $\mathbf{V}^{(L)}$ and the level-L discrepancy matrix $\mathbf{D}^{(L)}$. Next, we describe how to proceed to coarser levels of the tree. Finally, we analyze the asymptotic complexity of the compression algorithm.

2.4.1 Computing basis matrices $\mathbf{U}, \mathbf{V}$

We start by describing the process of computing the basis matrix $\mathbf{U}^{(L)}$ of the finest level, which involves finding for each τ on level L a basis matrix $\mathbf{U}_\tau$ that spans the range of $\mathbf{A}(I_\tau, I'_\tau)$ for every node $\tau' \neq \tau$ on level L. We will compute $\mathbf{U}_\tau$ by applying the randomized algorithm described in Section 2.2.3 to $\mathbf{A}(I_\tau, I_\tau^c)$, where $I_\tau^c = I \setminus I_\tau$ is the set of indices that are not in I_τ. Importantly, the procedure does not require the ability to apply $\mathbf{A}(I_\tau, I_\tau^c)$ to random vectors; rather we compute the randomized samples only using information contained in $\mathbf{\Omega}$ and $\mathbf{Y}$.

We define the following blocks of size $m \times s$ associated with τ.

$$\mathbf{\Omega}_\tau = \mathbf{\Omega}(I_\tau, :)$$

$$\mathbf{Y}_\tau = \mathbf{Y}(I_\tau, :)$$

Since $\mathbf{\Omega}_\tau$ is of size $m \times s$, it has a nullspace of dimension at least $s - m \geq r$, so we can find a set of r orthonormal vectors that belong to its nullspace

$$\mathbf{P}_\tau = \texttt{null}(\mathbf{\Omega}_\tau, r)$$

so that $\mathbf{\Omega}_\tau \mathbf{P}_\tau = \mathbf{0}$.

Then $\mathbf{\Omega}\mathbf{P}_\tau$ is of size $N \times r$ with $(\mathbf{\Omega}\mathbf{P}_\tau)(I_\tau, :) = \mathbf{\Omega}_\tau \mathbf{P}_\tau = \mathbf{0}$, and $(\mathbf{\Omega}\mathbf{P}_\tau)(I_\tau^c, :)$ is a standard Gaussian random matrix of size $(N - m) \times r$. Considering the structure of $\mathbf{\Omega}\mathbf{P}_\tau$, the product $\mathbf{A}\mathbf{\Omega}\mathbf{P}_\tau$ can be viewed as a randomized sample of $\mathbf{A}$, excluding contributions from columns $\mathbf{A}(:, I_\tau)$. Then the rows of $\mathbf{A}\mathbf{\Omega}\mathbf{P}_\tau$ indexed by I_τ contain a randomized sample of $\mathbf{A}(I_\tau, I_\tau^c)$. Moreover, since $(\mathbf{A}\mathbf{\Omega}\mathbf{P}_\tau)(I_\tau, :) = (\mathbf{Y}\mathbf{P}_\tau)(I_\tau, :) = \mathbf{Y}_\tau \mathbf{P}_\tau$, we can obtain that sample inexpensively

by simply multiplying $\mathbf{Y}_\tau \mathbf{P}_\tau$. Then we orthonormalize the sample to find basis matrix $\mathbf{U}_\tau$,

$$\mathbf{U}_\tau = \mathtt{qr}(\mathbf{Y}_\tau \mathbf{P}_\tau, r). \tag{2.7}$$

We repeat the same procedure to find $\mathbf{U}_\tau$ for each node τ on level L. A similar process using $\mathbf{\Psi}$ and $\mathbf{Z}$ yields basis matrices $\mathbf{V}_\tau$,

$$\begin{aligned} \mathbf{Q}_\tau &= \mathtt{null}(\mathbf{\Psi}_\tau, r) \\ \mathbf{V}_\tau &= \mathtt{qr}(\mathbf{Z}_\tau \mathbf{Q}_\tau, r). \end{aligned} \tag{2.8}$$

2.4.2 Computing discrepancy matrix D

Once we have computed $\mathbf{U}^{(L)}$ and $\mathbf{V}^{(L)}$, we next compute $\mathbf{D}^{(L)}$. We proceed by rewriting Eq. (2.6) as

$$\mathbf{D}_\tau = (\mathbf{I} - \mathbf{U}_\tau \mathbf{U}_\tau^*)\mathbf{A}_{\tau,\tau} + \mathbf{U}_\tau \mathbf{U}_\tau^* \mathbf{A}_{\tau,\tau}(\mathbf{I} - \mathbf{V}_\tau \mathbf{V}_\tau^*), \tag{2.9}$$

and deriving formulas for computing $(\mathbf{I} - \mathbf{U}_\tau \mathbf{U}_\tau^*)\mathbf{A}_{\tau,\tau}$ and $\mathbf{U}_\tau \mathbf{U}_\tau^* \mathbf{A}_{\tau,\tau}(\mathbf{I} - \mathbf{V}_\tau \mathbf{V}_\tau^*)$ separately.

For $(\mathbf{I} - \mathbf{U}_\tau \mathbf{U}_\tau^*)\mathbf{A}_{\tau,\tau}$, we first express $\mathbf{Y}_\tau$ as a blocked matrix product

$$\mathbf{Y}_\tau = \sum_{\tau' \text{ in level } \ell} \mathbf{A}_{\tau,\tau'} \mathbf{\Omega}_{\tau'}.$$

Multiplying $(\mathbf{I} - \mathbf{U}_\tau \mathbf{U}_\tau^*)$ and applying Eq. (2.3) gives

$$(\mathbf{I} - \mathbf{U}_\tau \mathbf{U}_\tau^*)\mathbf{Y}_\tau = (\mathbf{I} - \mathbf{U}_\tau \mathbf{U}_\tau^*)\mathbf{A}_{\tau,\tau}\mathbf{\Omega}_\tau.$$

Solving a least-squares problem with $\mathbf{\Omega}_\tau$ gives

$$(\mathbf{I} - \mathbf{U}_\tau \mathbf{U}_\tau^*)\mathbf{A}_{\tau,\tau} = (\mathbf{I} - \mathbf{U}_\tau \mathbf{U}_\tau^*)\mathbf{Y}_\tau \mathbf{\Omega}_\tau^\dagger. \tag{2.10}$$

A similar derivation yields

$$\mathbf{A}_{\tau,\tau}(\mathbf{I} - \mathbf{V}_\tau \mathbf{V}_\tau^*) = ((\mathbf{I} - \mathbf{V}_\tau \mathbf{V}_\tau^*)\mathbf{Z}_\tau \mathbf{\Psi}_\tau^\dagger)^*. \tag{2.11}$$

Substituting Eqs. (2.10) and (2.11) into Eq. (2.9) gives the formula

$$\mathbf{D}_\tau = (\mathbf{I} - \mathbf{U}_\tau \mathbf{U}_\tau^*)\mathbf{Y}_\tau \mathbf{\Omega}_\tau^\dagger + \mathbf{U}_\tau \mathbf{U}_\tau^*((\mathbf{I} - \mathbf{V}_\tau \mathbf{V}_\tau^*)\mathbf{Z}_\tau \mathbf{\Psi}_\tau^\dagger)^*. \tag{2.12}$$

2.4.3 Compressing levels $\ell = L-1, L-2, \ldots, 0$

After compressing level L, we proceed to the next coarser level $L-1$. That is, we seek $\mathbf{U}^{(L-1)}, \mathbf{V}^{(L-1)}$, and $\mathbf{D}^{(L-1)}$ that satisfy Eq. (2.5). We do so by first obtaining randomized samples of $\tilde{\mathbf{A}}^{(L)}$, and then finding $\mathbf{U}^{(L-1)}, \mathbf{V}^{(L-1)}$, and $\mathbf{D}^{(L-1)}$ using the same procedure as for level L.

To compute randomized samples of $\tilde{\mathbf{A}}^{(L)}$, we multiply Eq. (2.4) with $\mathbf{\Omega}$ and rearrange to obtain

$$\mathbf{Y} = \mathbf{A}\mathbf{\Omega} = (\mathbf{U}^{(L)} \tilde{\mathbf{A}}^{(L)} (\mathbf{V}^{(L)})^* + \mathbf{D}^{(L)})\mathbf{\Omega}$$
$$(\mathbf{U}^{(L)})^*(\mathbf{Y} - \mathbf{D}^{(L)}\mathbf{\Omega}) = \tilde{\mathbf{A}}^{(L)}((\mathbf{V}^{(L)})^*\mathbf{\Omega})$$

Then $(\mathbf{U}^{(L)})^*(\mathbf{Y} - \mathbf{D}^{(L)}\mathbf{\Omega})$ contains s randomized samples of $\tilde{\mathbf{A}}^{(L)}$ with test matrix $(\mathbf{V}^{(L)})^*\mathbf{\Omega}$. Then for each node τ on level $L-1$, we define

$$\mathbf{\Omega}_\tau = \begin{bmatrix} \mathbf{V}_\alpha^* \mathbf{\Omega}_\alpha \\ \mathbf{V}_\beta^* \mathbf{\Omega}_\beta \end{bmatrix}$$
$$\mathbf{Y}_\tau = \begin{bmatrix} \mathbf{U}_\alpha^* (\mathbf{Y}_\alpha - \mathbf{D}_\alpha \mathbf{\Omega}_\alpha) \\ \mathbf{U}_\beta^* (\mathbf{Y}_\beta - \mathbf{D}_\beta \mathbf{\Omega}_\beta) \end{bmatrix},$$

where α and β are the children of τ. We define $\mathbf{\Psi}_\tau$ and $\mathbf{Z}_\tau$ analogously. Once we have $\mathbf{\Omega}_\tau, \mathbf{\Psi}_\tau, \mathbf{Y}_\tau, \mathbf{Z}_\tau$ for each τ on level $L-1$, we compute $\mathbf{U}_\tau, \mathbf{V}_\tau$, and

$\mathbf{D}_\tau$ for each node τ on level $L-1$ exactly as before using Eqs. (2.7), (2.8), and (2.12).

This process is applied to successively coarser levels of the tree until the root node is reached. For the root node τ, we have $\mathbf{Y}_\tau = \mathbf{D}^{(0)}\mathbf{\Omega}_\tau$, so we simply solve $\mathbf{D}^{(0)} = \mathbf{Y}_\tau \mathbf{\Omega}_\tau^\dagger$. The full compression procedure is summarized in Algorithm 2.4.1.

Remark 3. *The computation of $\mathbf{D}_\tau$ involves solving least squares problems with Gaussian matrices $\mathbf{\Omega}_\tau$ and $\mathbf{\Psi}_\tau$ of size $m \times s$ or $2r \times s$. Gaussian matrices with nearly square shapes have non-negligible probabilities of being ill-conditioned, but the probabilities quickly become negligible even for slightly rectangular matrices [20,31]. Such concerns can be alleviated by choosing s to be sufficiently large. For the numerical experiments in Section 2.5, we simply use $s = r + m = 3r$.*

Remark 4. *For levels $l = L-1, L-2, ..., 1$, $\mathbf{V}^{(l)}$ depends on the draw of $\mathbf{\Omega}$, and therefore the test matrices defined in Section 2.4.3 may not exactly be standard Gaussian matrices. However, we observe empirically that the dependence is very weak, and the new test matrices behave like Gaussian test matrices.*

2.4.4 Asymptotic complexity

We assume for simplicity that $m = 2r$ and $s = 3r$. Algorithm 2.4.1 requires s matrix-vector products of $\mathbf{A}$ and $\mathbf{A}^*$, and an additional $\mathcal{O}(r^3)$ operations for each node in the tree, of which there are approximately $2N/m$.

Algorithm 2.4.1 Compressing an HBS matrix

Compute randomized samples of $\mathbf{A}$ *and* $\mathbf{A}^*$.
Form Gaussian random test matrices $\mathbf{\Omega}$ and $\mathbf{\Psi}$ of size $N \times s$.
Multiply $\mathbf{Y} = \mathbf{A\Omega}$ and $\mathbf{Z} = \mathbf{A}^*\mathbf{\Psi}$.

Compress level by level from finest to coarsest.
for level $\ell = L, L-1, \ldots, 0$ **do**
  for node τ in level ℓ **do**
    if τ is a leaf node **then**
      $\mathbf{\Omega}_\tau = \mathbf{\Omega}(I_\tau, :), \qquad \mathbf{\Psi}_\tau = \mathbf{\Psi}(I_\tau, :)$
      $\mathbf{Y}_\tau = \mathbf{Y}(I_\tau, :), \qquad \mathbf{Z}_\tau = \mathbf{Z}(I_\tau, :)$
    else
      Let α and β denote the children of τ.
      $$\mathbf{\Omega}_\tau = \begin{bmatrix} \mathbf{V}_\alpha^* \mathbf{\Omega}_\alpha \\ \mathbf{V}_\beta^* \mathbf{\Omega}_\beta \end{bmatrix}, \qquad \mathbf{\Psi}_\tau = \begin{bmatrix} \mathbf{U}_\alpha^* \mathbf{\Psi}_\alpha \\ \mathbf{U}_\beta^* \mathbf{\Psi}_\beta \end{bmatrix}$$
      $$\mathbf{Y}_\tau = \begin{bmatrix} \mathbf{U}_\alpha^* (\mathbf{Y}_\alpha - \mathbf{D}_\alpha \mathbf{\Omega}_\alpha) \\ \mathbf{U}_\beta^* (\mathbf{Y}_\beta - \mathbf{D}_\beta \mathbf{\Omega}_\beta) \end{bmatrix}, \qquad \mathbf{Z}_\tau = \begin{bmatrix} \mathbf{V}_\alpha^* (\mathbf{Z}_\alpha - \mathbf{D}_\alpha^* \mathbf{\Psi}_\alpha) \\ \mathbf{V}_\beta^* (\mathbf{Z}_\beta - \mathbf{D}_\beta^* \mathbf{\Psi}_\beta) \end{bmatrix}$$

    if $\ell > 0$ **then**
      $\mathbf{P}_\tau = \mathtt{null}(\mathbf{\Omega}_\tau, r), \qquad \mathbf{Q}_\tau = \mathtt{null}(\mathbf{\Psi}_\tau, r)$
      $\mathbf{U}_\tau = \mathtt{qr}(\mathbf{Y}_\tau \mathbf{P}_\tau, r), \qquad \mathbf{V}_\tau = \mathtt{qr}(\mathbf{Z}_\tau \mathbf{Q}_\tau, r)$
      $\mathbf{D}_\tau = (\mathbf{I} - \mathbf{U}_\tau \mathbf{U}_\tau^*) \mathbf{Y}_\tau \mathbf{\Omega}_\tau^\dagger + \mathbf{U}_\tau \mathbf{U}_\tau^* \left((\mathbf{I} - \mathbf{V}_\tau \mathbf{V}_\tau^*) \mathbf{Z}_\tau \mathbf{\Psi}_\tau^\dagger \right)^*$
    else
      Find $\mathbf{D}_\tau$ *for root node* τ.
      $\mathbf{D}_\tau = \mathbf{Y}_\tau \mathbf{\Omega}_\tau^\dagger$

Therefore, the total compression time is

$$T_{\text{compress}} = 6rN \times T_{\text{rand}} + 6r \times T_{\text{mult}} + \mathcal{O}(r^2 N) \times T_{\text{flop}},$$

where T_{rand} denotes the time to sample a value from the standard Gaussian distribution, T_{mult} denotes the time to apply $\mathbf{A}$ or $\mathbf{A}^*$ to a vector, and T_{flop} denotes the time to carry out a floating point arithmetic operation.

Remark 5 (Comparison of information efficiency)**.** *If we view each randomized sample as carrying N values worth of information about $\mathbf{A}$, and still assume $m = 2r$ and $s = 3r$, we find that the compression algorithm requires a total of $6rN$ values to reconstruct the matrix. The algorithm of [76] requires only r samples of $\mathbf{A}$ and $\mathbf{A}^*$, but it also requires access to $\sim mN$ matrix entries that form the block-diagonal part of $\mathbf{A}$ as well as $\sim rN$ elements that appear in interpolative decompositions, for a total of $5rN$ values worth of information. Therefore, the algorithm in the present work requires only slightly more information to recover $\mathbf{A}$, while having the advantage of being truly black-box.*

2.5 Numerical experiments

In this section, we present a selection of numerical results. In Sections 2.5.1 to 2.5.4, we report the following quantities for a number of test problems and rank structure formats: (1) the time to compress the operator, (2) the time to apply the compressed representation to a vector, (3) the relative accuracy of the compressed representation, and (4) the storage requirements of the compressed representation measured as the number of values per de-

gree of freedom. For compression time, we report both the total time taken for compression as well as the compression time excluding the time spent by the black-box multiplication routine for computing randomized samples. The algorithms for compressing matrices and applying compressed representations are written in Python, and the black-box multiplication routines are written in MATLAB. The experiments were carried out on a workstation with an Intel Core i9-10900K processor with 10 cores and 128GB of memory.

We measure the accuracy of the compressed matrices using the relative error

$$\frac{\|\tilde{\mathbf{A}} - \mathbf{A}\|}{\|\mathbf{A}\|}$$

computed via 20 iterations of the power method. We also report the maximum leaf node size m and the number r of random vectors per test matrix, which are inputs to the compression algorithm.

2.5.1 Boundary integral equation

We consider a matrix arising from the discretization of the Boundary Integral Equation (BIE)

$$\frac{1}{2}q(\boldsymbol{x}) + \int_{\Gamma} \frac{(\boldsymbol{x} - \boldsymbol{y}) \cdot \boldsymbol{n}(\boldsymbol{y})}{4\pi|\boldsymbol{x} - \boldsymbol{y}|^2}\, q(\boldsymbol{y})\, ds(\boldsymbol{y}) = f(\boldsymbol{x}), \qquad \boldsymbol{x} \in \Gamma, \tag{2.13}$$

where Γ is the simple closed contour in the plane shown in Figure 2.3, and where $\boldsymbol{n}(\boldsymbol{y})$ is the outwards pointing unit normal of Γ at $\boldsymbol{y}$. The BIE (2.13) is a standard integral equation formulation of the Laplace equation with boundary condition f on the domain interior to Γ. The BIE (2.13) is discretized using

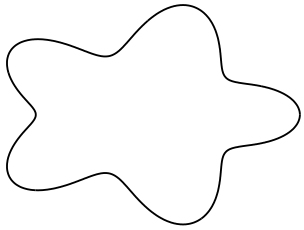


Figure 2.3 *Contour* Γ *on which the BIE (2.13) is defined.*

the Nyström method on N equispaced points on Γ, with the Trapezoidal rule as the quadrature (since the kernel in (2.13) is smooth, the Trapezoidal rule has exponential convergence).

The fast matrix-vector multiplication is in this case furnished by the recursive skeletonization (RS) procedure of [84]. To avoid spurious effects due to the rank structure inherent in RS, we compute the matrix-vector products at close to double precision accuracy, and with an entirely uncorrelated tree structure.

Results are given in Fig. 2.4.

Remark 6. *The problem under consideration here is artificial in the sense that there is no actual need to use more than a couple of hundred points to resolve (2.13) numerically to double precision accuracy. It is included merely to illustrate the asymptotic scaling of the proposed method.*

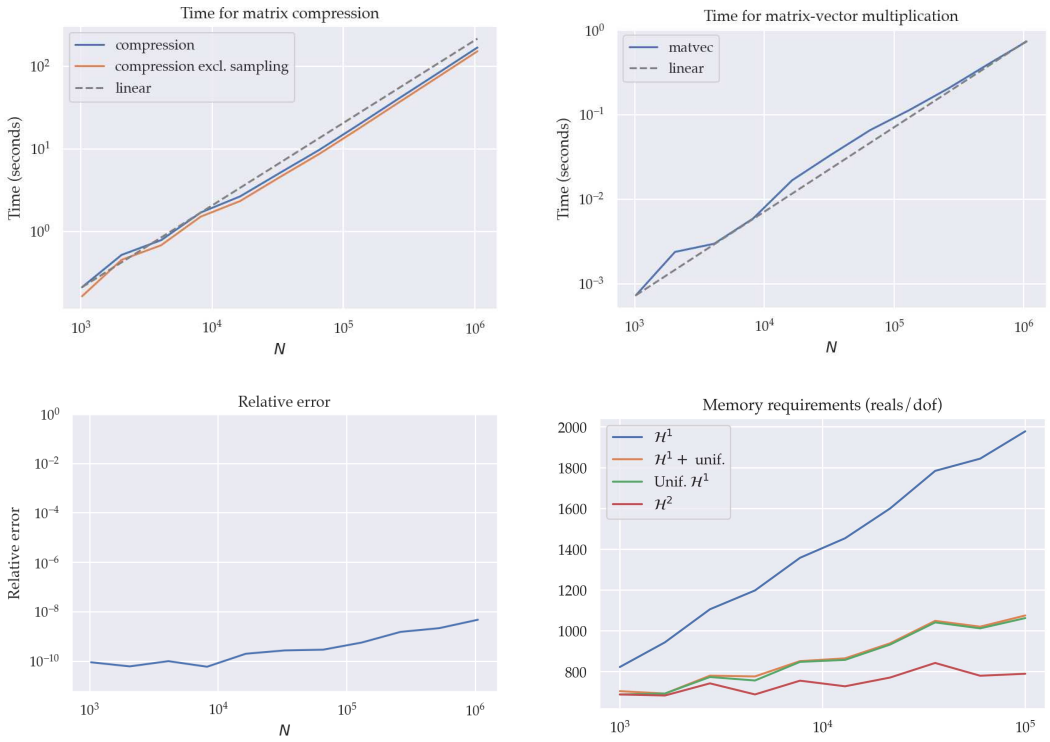


Figure 2.4 *Results from applying the compression algorithm to a double layer potential on a simple contour in the plane. Here* $r = 30$ *and* $m = 60$.

2.5.2 Operator multiplication

We next investigate how the proposed technique performs on a matrix matrix multiplication problem. Specifically, we determine the Neumann-to-Dirichlet operator T for the contour shown in Figure 2.3 using the well known formula

$$T = S\left(\frac{1}{2}I + D^*\right)^{-1},$$

where S is the single layer operator $[Sq](\boldsymbol{x}) = \int_\Gamma -\frac{1}{2\pi}\,\log|\boldsymbol{x}-\boldsymbol{y}|\,q(\boldsymbol{y})\,ds(\boldsymbol{y})$, and where D^* is the adjoint of the double-layer operator $[D^*q](\boldsymbol{x}) = \int_\Gamma \frac{\boldsymbol{n}(\boldsymbol{x})\cdot(\boldsymbol{x}-\boldsymbol{y})}{2\pi|\boldsymbol{x}-\boldsymbol{y}|^2}\,q(\boldsymbol{y})\,ds(\boldsymbol{y})$.

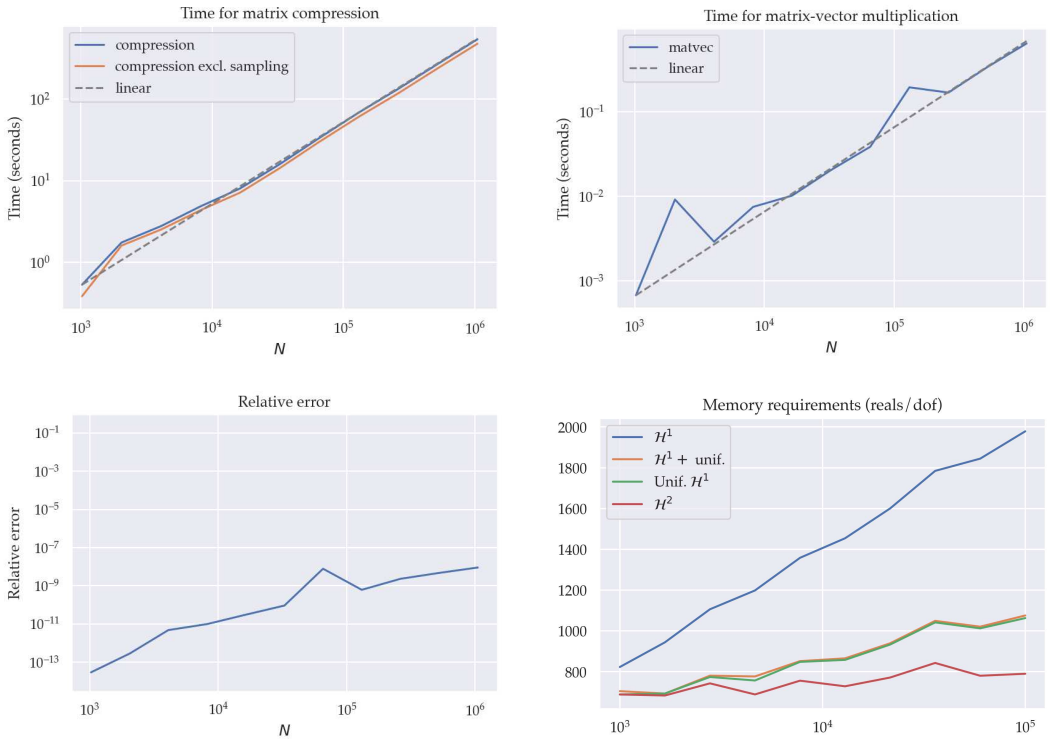


Figure 2.5 *Results from applying the compression algorithm to the Neumann-to-Dirichlet operator. Here* $r = 100$ *and* $m = 200$.

The operators S and D are again discretized using a Nyström method on equispaced points (with sixth order Kapur-Rokhlin [53] corrections to handle the singularity in S), resulting in matrices $\mathbf{S}$ and $\mathbf{D}$. The $\mathbf{S}\left(0.5\mathbf{I} + \mathbf{D}^*\right)^{-1}$ is again applied using the recursive skeletonization procedure of [84].

Results are given in Fig. 2.5.

2.5.3 Fast multipole method

2.5.4 Frontal matrices in nested dissection

Our next example is a simple model problem that illustrates the behavior of the proposed method in the context of sparse direct solvers. The idea here is to use rank structure to compress the increasingly large Schur complements that arise in the LU factorization of a sparse matrix arising from the finite element or finite difference discretization of an elliptic PDE, cf. [80, Ch. 21]. As a model problem, we consider an $N \times N$ matrix $\mathbf{C}$ that encodes the stiffness matrix for the standard five-point stencil finite difference approximation to the Poisson equation on a rectangle. We use a grid with $N \times 51$ nodes. We partition the grid into three sets $\{1, 2, 3\}$, as shown in Fig. 2.6, and then tessellate $\mathbf{C}$ accordingly,

$$\mathbf{C} = \begin{bmatrix} \mathbf{C}_{11} & \mathbf{0} & \mathbf{C}_{13} \\ \mathbf{0} & \mathbf{C}_{22} & \mathbf{C}_{23} \\ \mathbf{C}_{31} & \mathbf{C}_{32} & \mathbf{C}_{33} \end{bmatrix}.$$

The matrix we seek to compress is the Schur complement

$$\mathbf{A} = \mathbf{C}_{33} - \mathbf{C}_{31}\mathbf{C}_{11}^{-1}\mathbf{C}_{31} - \mathbf{C}_{32}\mathbf{C}_{22}^{-1}\mathbf{C}_{23}.$$

In our example, we apply $\mathbf{A}$ to vector by calling standard sparse direct solvers for the left and the right subdomains, respectively.

Results are given in Fig. 2.7.

2.5.5 Summary of observations

- The results exhibit linear scaling of computation time for compressing the operators and for applying the compressed representations to vectors.

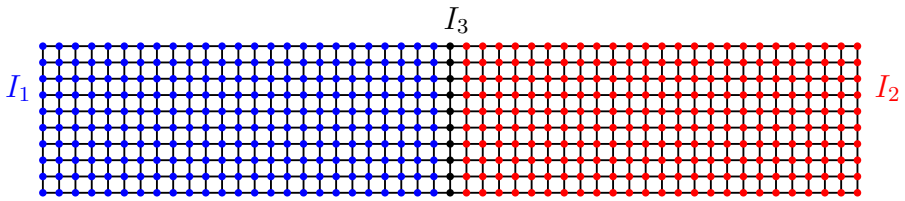


Figure 2.6 *An example of the grid in the sparse LU example described in Section 2.5.4. There are $N \times n$ points in the grid, shown for $N = 10, n = 51$.*

- The approximations achieve high accuracy in every case.
- The asymptotic storage cost, reported as number of floating point numbers per degree of freedom, is depends on r and m, but is independent of the problem size N.

2.6 Conclusions

This paper presents an algorithm for black-box randomized compression of Hierarchically Block Separable matrices. To compress an $N \times N$ matrix $\mathbf{A}$, the algorithm requires only $\mathcal{O}(k)$ samples of $\mathbf{A}$ and $\mathbf{A}^*$, where k is the block rank of $\mathbf{A}$. Numerical experiments demonstrate that the algorithms are accurate and very computationally efficient, with compression time scaling linearly in N when the cost of applying $\mathbf{A}$ and $\mathbf{A}^*$ to a vector is $\mathcal{O}(N)$.

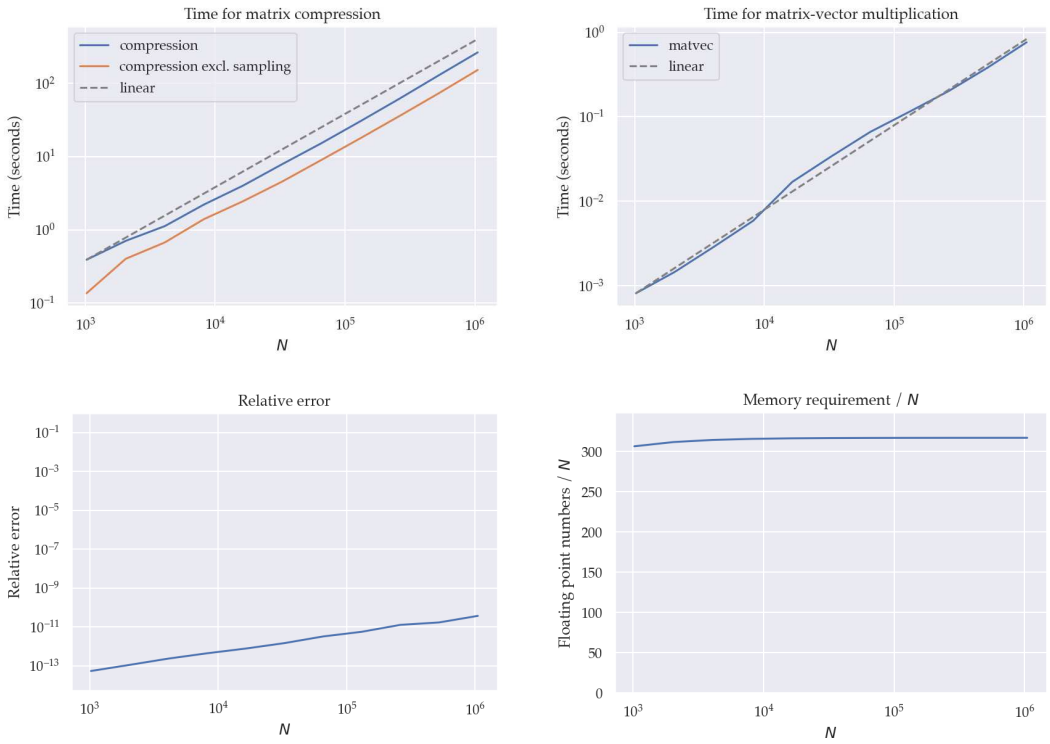


Figure 2.7 *Results from applying the compression algorithm to frontal matrices in the nested dissection algorithm. Here* $r = 30$ *and* $m = 60$.

Chapter 3

Randomized Compression of Rank-Structured Matrices with Graph Coloring[1]

[1]The content in this chapter is based on work done in collaboration with Per-Gunnar Martinsson and published in [61].

A randomized algorithm for computing a data sparse representation of a given rank structured matrix $\mathbf{A}$ (a.k.a. an $\mathcal{H}$-matrix) is presented. The algorithm draws on the randomized singular value decomposition (RSVD), and operates under the assumption that algorithms for rapidly applying $\mathbf{A}$ and $\mathbf{A}^*$ to vectors are available. The algorithm analyzes the hierarchical tree that defines the rank structure using graph coloring algorithms to generate a set of random test vectors. The matrix is then applied to the test vectors, and in a final step the matrix itself is reconstructed by the observed input-output pairs. The method presented is an evolution of the "peeling algorithm" of *L. Lin, J. Lu, and L. Ying, "Fast construction of hierarchical matrix representation from matrixvector multiplication," JCP, 230(10), 2011.* For the case of uniform trees, the new method substantially reduces the pre-factor of the original peeling algorithm. More significantly, the new technique leads to dramatic acceleration for many non-uniform trees since it constructs sample vectors that are optimized for a given tree. The algorithm is particularly effective for kernel matrices involving a set of points restricted to a lower dimensional object than the ambient space, such as a boundary integral equation defined on a surface in three dimensions.

3.1 Introduction

This work describes a set of efficient algorithms for handling large dense matrices that have *rank structure*. To simplify slightly, this means that an $N \times N$ matrix can be tessellated into $O(N)$ blocks in such a way that each

block is either small or of low numerical rank, cf. Figures 3.3 and 3.1. This structure allows the matrix to be stored and applied to vectors efficiently, often with cost that scales linearly or close to linearly with N. Sometimes, it is also possible to compute an approximate inverse or LU factorization in linear or close to linear time. Matrices of this type have turned out to be ubiquitous in both engineering and data sciences, and have been the subject of much research in recent decades, going under names such as $\mathcal{H}$-matrices [8, 12, 43]; HODLR matrices [3, 75], Hierarchically Semi-Separable (HSS) matrices [16, 17, 102], Recursive Skeletonization [34, 48, 84, 86], and many more.

The specific problem we address is the following: Suppose that $\mathbf{A}$ is an $N \times N$ matrix that we know is rank-structured, but we do not have direct access to the low-rank factors that define the compressible off-diagonal blocks. Instead, we have access to fast algorithms that given tall thin matrices $\mathbf{\Omega}, \mathbf{\Psi} \in \mathbb{R}^{N \times \ell}$, evaluate the matrix-matrix products

$$\mathbf{Y} = \mathbf{A}\mathbf{\Omega}, \qquad \text{and} \qquad \mathbf{Z} = \mathbf{A}^*\mathbf{\Psi}.$$

The problem is then to construct two random matrices $\mathbf{\Omega}$ and $\mathbf{\Psi}$ for which $\mathbf{A}$ can be recovered from the information in the set $\{\mathbf{Y}, \mathbf{\Omega}, \mathbf{Z}, \mathbf{\Psi}\}$. The algorithms described here solve the reconstruction problem using $\ell \sim k \log(N)$ sample vectors, where k is an upper bound on the ranks of the off-diagonal blocks. The key novelty is the formulation of a graph coloring problem to analyze the cluster tree that defines the rank structure to build test matrices that are optimized for the specific problem under consideration.

The scheme presented has several important applications. First, it can be used to derive a rank-structured representation of any integral operator for which a fast matrix-vector multiplication algorithm, such as the Fast Multipole Method [38, 40], is available. Such a representation opens the door to a wider range of matrix operations such as LU factorization, matrix inversion, and sometimes even full spectral decompositions. Second, it can greatly simplify algebraic operations involving products of rank-structured matrices. For instance, the perhaps key application of rank-structured matrix algebra is the acceleration of sparse direct solvers, as the dense matrices that arise during LU factorization are often rank-structured. In the course of such a solver, a typical operation would be to form a Schur complement such as $\mathbf{S}_{22} = \mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}$ that would arise when the top left block $\mathbf{A}_{11}$ is eliminated from a 2×2 blocked matrix. If $\mathbf{A}_{11}$ is rank-structured, then $\mathbf{A}_{11}^{-1}$ can easily be applied to vectors via an LU factorization. If, additionally, $\mathbf{A}_{12}$ and $\mathbf{A}_{21}$ are either sparse or rank-structured, then $\mathbf{S}_{22}$ can easily be applied to a vector. The technique described will then enable one to construct a data-sparse representation of $\mathbf{S}_{22}$. In contrast, to directly evaluate the product $\mathbf{A}_{21}\mathbf{A}_{11}^{-1}\mathbf{A}_{12}$ is both onerous to code and slow to execute.

The method we describe is inspired by the "peeling algorithm" of [64], which to the best of our knowledge was the first true black box algorithm described in the literature. The method of [64] has formally the same sample complexity $\ell \sim k \log(N)$ as our method, but involves substantially larger pre-factors. To be precise, [64] is targeted specifically for $\mathcal{H}^1$- and $\mathcal{H}^2$-matrices

arising from the discretization of integral equations. Strong admissibility, and regular tree structures are used. In this environment, the method requires $\ell \sim k\, 8^d \log(N)$ matrix-vector products involving $\mathbf{A}$ and $\mathbf{A}^*$, where d is the dimension of space in which the underlying integral equation is defined. In contrast, the method presented here has complexity $\ell \sim k\, 6^d \log(N)$ for fully populated uniform trees. For more general trees, the acceleration over the method of [64] is even more dramatic, since the adaptivity of our method enables it to exploit situations where the matrix arises from a set of points located on a lower dimensional geometric object. As an illustration, Section 3.5 reports on experiments involving a boundary integral equation defined on a 2D surface in three dimensional space, as well as examples in higher dimensions.

Another advantage of the techniques presented here is that they are not limited to the $\mathcal{H}^1$ structure. To compress uniform $\mathcal{H}^1$ and $\mathcal{H}^2$ matrices, the presented algorithms obtain uniform basis matrices by sampling the interactions of a box with its entire interaction list collectively. This process results in higher quality samples while requiring fewer matrix-vector products compared to existing methods that approximate interactions between boxes separately and then apply a recompression step to obtain uniform basis matrices. More generally, the formulation of a graph coloring problem can be used to design test matrices for any tessellation (e.g., arising from some other geometric or algebraic admissibility condition).

Remark 7 (Linear complexity schemes)**.** *A related class of algorithms for computing a rank-structured matrix by observing its action on certain structured*

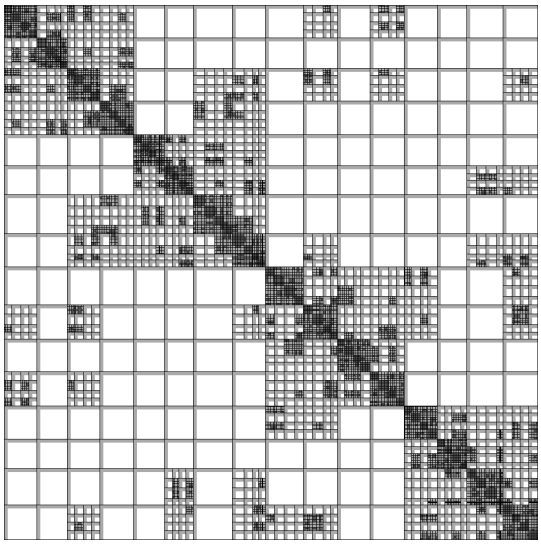


Figure 3.1 *An $\mathcal{H}^1$ matrix for a quadtree over a uniform grid in the plane. Dense blocks are shown in dark gray, and low-rank blocks are represented with a white background and light gray rectangles representing the shapes of the low-rank factors.*

random vectors was described in [76]. These techniques have true linear complexity (no logarithmic terms), and tend to be very fast in practice. However, they are not true black box algorithms, as they require the direct evaluation of a small number of entries of the matrix. In contrast, the method presented here is truly black box, like the methods of [64, 77].

The manuscript is structured as follows: Section 3.2 surveys some basic linear algebraic techniques that we rely on. Section 3.3 introduces our formalism for rank-structured matrices. Section 3.4 describes the new algorithm, and analyzes its asymptotic complexity. Section 3.5 describes numerical results.

3.2 Preliminaries

3.2.1 Notation

Throughout the paper, we measure a vector $\mathbf{x} \in \mathbb{R}^n$ by its Euclidean norm $\|\mathbf{x}\| = (\sum_i |x_i|^2)^{\frac{1}{2}}$. We measure a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ with the corresponding operator norm $\|\mathbf{A}\| = \sup_{\|\mathbf{x}\|=1} \|\mathbf{A}\mathbf{x}\|$, and in some cases with the Frobenius norm $\|\mathbf{A}\|_{\text{Fro}} = (\sum_{i,j} |\mathbf{A}(i,j)^2)^{1/2}$. To denote submatrices, we use the notation of Golub and van Loan [35]: If $\mathbf{A}$ is an $m \times n$ matrix, and $I = [i_1, i_2, \ldots, i_k]$ and $J = [j_1, j_2, \ldots, j_k]$, then $\mathbf{A}(I, J)$ denotes the $k \times l$ matrix

$$\mathbf{A}(I,J) = \begin{bmatrix} \mathbf{A}(i_1,j_1) & \mathbf{A}(i_1,j_2) & \ldots & \mathbf{A}(i_1,j_l) \\ \mathbf{A}(i_2,j_1) & \mathbf{A}(i_2,j_2) & \ldots & \mathbf{A}(i_2,j_l) \\ \vdots & \vdots & & \vdots \\ \mathbf{A}(i_k,j_1) & \mathbf{A}(i_k,j_2) & \ldots & \mathbf{A}(i_k,j_l) \end{bmatrix}$$

We let $\mathbf{A}(I,:)$ denote the column submatrix $\mathbf{A}(I,[1,2,\ldots,n])$ and analogously let $\mathbf{A}(:,J)$ denote a row submatrix of $\mathbf{A}$. We let $\mathbf{A}^*$ denote the transpose of $\mathbf{A}$, and we say that matrix $\mathbf{U}$ is *orthonormal* if its columns are orthonormal, $\mathbf{U}^*\mathbf{U} = \mathbf{I}$.

3.2.2 The QR factorization

The column-pivoted *QR factorization* of a matrix $\mathbf{A}$ of size $m \times n$ takes the form

$$\underset{m \times n}{\mathbf{A}} \quad \underset{n \times n}{\mathbf{P}} \quad = \quad \underset{m \times r}{\mathbf{Q}} \quad \underset{r \times n}{\mathbf{R}}, \tag{3.1}$$

where $r = \min(m,n)$, $\mathbf{Q}$ is orthonormal, $\mathbf{R}$ is upper-triangular, and $\mathbf{P}$ is a permutation matrix. Representing the permutation matrix $\mathbf{P}$ as the vector $J \subset \mathbb{Z}_+^n$ of column indices such that $\mathbf{P} = \mathbf{I}(:,J)$, the factorization Eq. (3.1) can

be expressed as

$$\underset{m \times n}{\mathbf{A}(:, J)} \quad = \quad \underset{m \times r}{\mathbf{Q}} \quad \underset{r \times n}{\mathbf{R}}.$$

For a matrix that is numerically low-rank, a rank-k approximation of $\mathbf{A}$ is given by a "partial QR factorization of $\mathbf{A}$,"

$$\underset{m \times n}{\mathbf{A}(:, J)} \quad \approx \quad \underset{m \times k}{\mathbf{Q}_k} \quad \underset{k \times n}{\mathbf{R}_k}.$$

3.2.3 The singular value decomposition (SVD)

The singular value decomposition of a matrix $\mathbf{A}$ of size $m \times n$ takes the form

$$\underset{m \times n}{\mathbf{A}} \quad = \quad \underset{m \times r}{\mathbf{U}} \quad \underset{r \times r}{\mathbf{\Sigma}} \quad \underset{r \times n}{\mathbf{V}^*}, \tag{3.2}$$

where $r = \min(m, n)$, $\mathbf{U}$ and $\mathbf{V}$ are orthonormal matrices, and $\mathbf{\Sigma}$ is a diagonal matrix with diagonal elements $\{\sigma_j\}_{j=1}^r$ ordered such that $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r \geq 0$. The columns of $\mathbf{U}$ and $\mathbf{V}$, denoted by $\{\mathbf{u}_i\}_{i=1}^r$ and $\{\mathbf{v}_i\}_{i=1}^r$, are the left and right singular vectors of $\mathbf{A}$.

We let $\mathbf{A}_k$ denote the rank-k approximation obtained by truncating the SVD to its first k terms, so that $\mathbf{A}_k = \sum_{j=1}^k \sigma_j \mathbf{u}_j \mathbf{v}_j^*$. It then follows that

$$\|\mathbf{A} - \mathbf{A}_k\| = \sigma_{k+1} \qquad \text{and that} \qquad \|\mathbf{A} - \mathbf{A}_k\|_{\text{Fro}} = \left(\sum_{j=k+1}^{\min(m,n)} \sigma_j^2 \right)^{1/2}.$$

The Eckart–Young theorem asserts that $\mathbf{A}_k$ achieves the smallest possible approximation error of $\mathbf{A}$, in both the operator and Frobenius norms, of any rank-k matrix.

3.2.4 Randomized compression

In this section, we give a brief review of randomized low-rank approximation, following the presentation of [77]. Let $\mathbf{A}$ be an $m \times n$ matrix that can be accurately approximated by a matrix of rank k, and suppose we seek to determine a matrix $\mathbf{Q}$ with orthonormal columns (as few as possible) such that $\|\mathbf{A} - \mathbf{Q}\mathbf{Q}^*\mathbf{A}\|$ is small. In other words, we seek a matrix $\mathbf{Q}$ whose columns form an approximate orthornomal basis (ON-basis) for the column space of $\mathbf{A}$. This task can efficiently be solved via the following randomized procedure:

1. Pick a small integer p representing how much "oversampling" is done. ($p = 10$ is often good.)

2. Form an $n \times (k+p)$ matrix $\mathbf{G}$ whose entries are independent and identically distributed (i.i.d.) normalized Gaussian random numbers.

3. Form the "sample matrix" $\mathbf{Y} = \mathbf{A}\mathbf{G}$ of size $m \times (k + p)$.

4. Construct an $m \times (k + p)$ matrix $\mathbf{Q}$ whose columns form an ON basis for the columns of $\mathbf{Y}$.

Note that each column of the sample matrix $\mathbf{Y}$ is a random linear combination of the columns of $\mathbf{A}$. We would therefore expect the algorithm described to have a high probability of producing an accurate result when p is a large number. It is perhaps less obvious that this probability depends only on p (not on m or n, or any other properties of $\mathbf{A}$) and that it approaches

1 extremely rapidly as p increases. In fact, one can show that the basis $\mathbf{Q}$ determined by the scheme above satisfies

$$\|\mathbf{A} - \mathbf{Q}\mathbf{Q}^*\mathbf{A}\| \leq [1 + 11\sqrt{k+p} \cdot \sqrt{\min\{m,n\}}]\sigma_{k+1} \tag{3.3}$$

with probability at least $1 - 6 \cdot p^{-p}$; see [46, sec. 1.5]. The error bound Eq. (3.3) indicates that the error prodcued by the randomized sampling procedure can be larger than the theoretically minimal error σ_{k+1} by a factor of $1+11\sqrt{k+p}\cdot\sqrt{\min\{m,n\}}$. This crude bound is typically very pessimistic, in particular for matrices whose singular values decay rapidly; cf. [46].

Now, suppose we would like to use the above sampling procedure to compute a low-rank factorization of the form

$$\underset{m \times n}{\mathbf{A}} = \underset{m \times r}{\mathbf{U}} \quad \underset{r \times r}{\mathbf{B}} \quad \underset{r \times n}{\mathbf{V}}, \tag{3.4}$$

where $\mathbf{U}$ and $\mathbf{V}$ have orthonormal columns and $\mathbf{B}$ is a square matrix, not necessarily diagonal. We review two methods of completing this task using randomized sampling. The algorithm summarized in Algorithm 3.2.1 involves two randomized samples, one of $\mathbf{A}$ and one of $\mathbf{A}^*$. It is based on so-called single-view algorithms for matrix compression [83, 98]. The second method [46], summarized in Algorithm 3.2.2, uses a randomized sample of $\mathbf{A}$ to compute the column basis matrix $\mathbf{U}$, and then operates on the product $\mathbf{A}^*\mathbf{U}$ to obtain $\mathbf{B}$ and $\mathbf{V}$. We will use both methods in the algorithms described in Section 3.4.

Algorithm 3.2.1 Compress $\mathbf{A} \approx \mathbf{U}\mathbf{B}\mathbf{V}^*$ via two randomized samples

Form $n \times (k+p)$ Gaussian random matrix $\mathbf{G}_1$ and $m \times (k+p)$ Gaussian random matrix $\mathbf{G}_2$.
Multiply $\mathbf{Y} = \mathbf{A}\mathbf{G}_1$.
Orthonormalize $\mathbf{U} = \mathtt{qr}(\mathbf{Y}, k)$.
Multiply $\mathbf{Z} = \mathbf{A}^*\mathbf{G}_2$.
Orthonormalize $\mathbf{V} = \mathtt{qr}(\mathbf{Z}, k)$.
Solve $\mathbf{B} = (\mathbf{G}_2^*\mathbf{U})^\dagger \mathbf{G}_2^*\mathbf{A}\mathbf{G}_1(\mathbf{V}^*\mathbf{G}_1)^\dagger$.
return $\mathbf{U}, \mathbf{B}, \mathbf{V}$.

Algorithm 3.2.2 Compress $\mathbf{A} \approx \mathbf{U}\mathbf{B}\mathbf{V}^*$ with one randomized and one deterministic sample

Form an $n \times (k+p)$ Gaussian random matrix $\mathbf{G}$.
Multiply $\mathbf{Y} = \mathbf{A}\mathbf{G}$.
Orthonormalize $\mathbf{U} = \mathtt{qr}(\mathbf{Y})$.
Multiply $\mathbf{W} = \mathbf{A}^*\mathbf{Q}$.
Orthonormalize $[\mathbf{V}, \mathbf{B}^*] = \mathtt{qr}(\mathbf{W})$.
return $\mathbf{U}, \mathbf{B}, \mathbf{V}$.

3.2.5 Functions for low-rank factorizations

We introduce the following notation to denote calls to functions that return the results of QR factorizations and singular value decompositions. Function calls to evaluate the full factorizations are written as

$$[\mathbf{Q}, \mathbf{R}, J] = \mathtt{qr}(\mathbf{A}), \qquad [\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}] = \mathtt{svd}(\mathbf{A}).$$

Calls to evaluate the rank-k truncated factorizations are written as

$$[\mathbf{Q}, \mathbf{R}, J] = \mathtt{qr}(\mathbf{A}, k), \qquad [\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}] = \mathtt{svd}(\mathbf{A}, k).$$

We write

$$[\mathbf{Q}, \mathbf{R}] = \mathtt{qr}(\mathbf{A}), \qquad \mathbf{Q} = \mathtt{qr}(\mathbf{A})$$

to compute an unpivoted QR factorization and to compute just the factor $\mathbf{Q}$ via the Gram-Schmidt process, respectively.

3.2.6 The degree of saturation graph coloring algorithm

A vertex coloring of graph is an assigment of a color to each vertex in such a way that no pair of adjacent vertices shares the same color. The problem of finding a vertex coloring with the minimum number of colors is NP-hard, but there exist a number of algorithms for efficiently coloring graphs, though they may produce colorings with more than the minimum number of colors.

Greedy graph coloring algorithms process the vertices in sequence, at each iteration assigning to one vertex the first available color that hasn't already

been assigned to one of its neighbors. In the *degree of saturation* (DSatur) algorithm [13], the choice of which vertex to color at each step is made by selecting from the remaining uncolored vertices the one whose neighbors have the greatest number of distinct colors (the so-called degree of saturation). The procedure is summarized in Algorithm 3.2.3.

Algorithm 3.2.3 Greedy graph coloring with DSatur

Initialize priority queue q of vertices keyed by degree of saturation (initially all zero)
Initialize for each vertex a set of invalid colors (initially all empty)
while q is not empty **do**
    Pop from q the vertex v with highest degree of saturation ▷ $\mathcal{O}(\log |V|)$
    Assign a color to v, creating a new one if necessary ▷ $\mathcal{O}(\deg(G))$
    for each vertex w adjacent to v **do**
        Add the color of v to the set of invalid colors for w ▷ $\mathcal{O}(1)$
        Update the priority of w within q ▷ $\mathcal{O}(\log |V|)$

While the asymptotic complexity of DSatur is often described as quadratic in the number of vertices, the algorithm can be implemented with quasilinear complexity, assuming the degree of the graph is bounded. Summing the costs listed in Algorithm 3.2.3, we find that the asymptotic complexity is

$$T_{\text{color}} \sim \deg(G)|V| \log |V|, \tag{3.5}$$

where $\deg(G)$ denotes the degree of the graph. for the step of assigning a color to vertex v, we note that a greedy coloring algorithm uses no more than $\deg(G) + 1$ colors in the worst case, so checking the existing colors for whether they belong to the set of invalid colors for v requires $\mathcal{O}(\deg(G))$ operations.

3.3 Rank-structured matrices

Let $X = [0,1]^d$ be a d-dimensional hypercube. We introduce a tree of boxes, each level of which represents a partition of X into smaller boxes. Level 0 only contains a single node, which corresponds to X. The boxes belonging to level $l+1$ are obtained by bisecting each of the boxes in level l along each dimension to form 2^d smaller boxes. The 2^d boxes in level $l+1$ obtained by splitting a box in level l are designated as the children of that box, giving rise to the tree structure. Boxes that do not contain any points are omitted from the tree, so the branching factor of the tree may be less than 2^d, depending on the distribution of points. The splitting procedure is applied recursively to boxes that contain more than m points, where m is a prespecified maximum number of points to allow in a leaf box. We let L denote the depth of the tree, and assume that the points are distributed throughout the domain in such a way that $L \sim \log N$.

Two boxes that belong to the same level and have overlapping boundaries are said to be *neighbors*. The neighbors of box τ, of which there are up to 3^d (including τ itself), are stored in the *neighbor list* of τ, denoted by $\mathcal{L}_\tau^{\text{nei}}$. The *interaction list* of τ, denoted by $\mathcal{L}_\tau^{\text{int}}$ contains the children of neighbors of the parent of τ, excluding those that are neighbors of τ. The interaction list of a box contains up to $6^d - 3^d$ boxes.

Let $\{x_i\}_{i=1}^N \subset X$ be a set of points within the domain, and let $\mathbf{A}$ be an

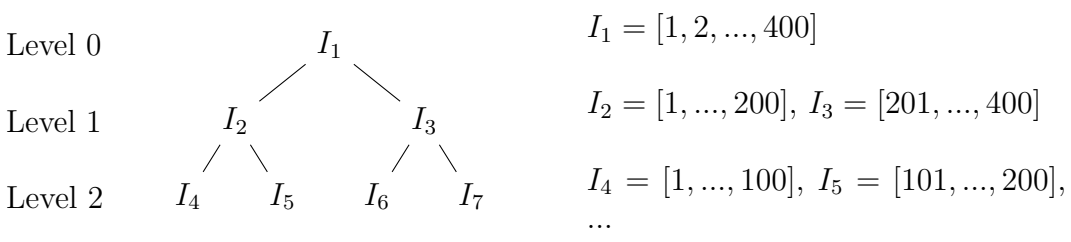


Figure 3.2 *A binary tree structure, where the levels of the tree represent successively refined partitions of the index vector* $[1, ..., 400]$.

N-by-N matrix where

$$\mathbf{A}(i,j) = \mathcal{K}(x_i, x_j) \qquad 1 \le i \le N, 1 \le j \le N,$$

for some kernel function $\mathcal{K}$. That is, the (i,j) entry of $\mathbf{A}$ represents an interaction between points x_i, x_j as defined by $\mathcal{K}$. For each box τ, we assign a list of indices of the points contained by the box, $I_\tau = \{i : x_i \in \tau\}$. Fig. 3.2 shows an example tree structure and index lists.

Matrix $\mathbf{A}$ is said to have $\mathcal{H}^1$ structure if block $\mathbf{A}(I_\alpha, I_\beta)$ is numerically low-rank, for all pairs of boxes α, β belonging to the interaction lists of one another. Such blocks, and the corresponding pairs of boxes, are said to be *admissible*. Fig. 3.3 shows a tessellation of a matrix consisting of the admissible blocks as well as a number of *inadmissible* blocks of interactions between neighboring boxes in level L, which are not necessarily low-rank. Constructing a compressed representation of an $\mathcal{H}^1$ matrix consists of finding low rank approximations

$$\mathbf{A}(I_\alpha, I_\beta) \approx \mathbf{U}_{\alpha,\beta}\mathbf{B}_{\alpha,\beta}\mathbf{V}_{\alpha,\beta}$$

for each admissible pair (α, β) and storing the inadmissible blocks corresponding to neighbor interactions of boxes in level L.

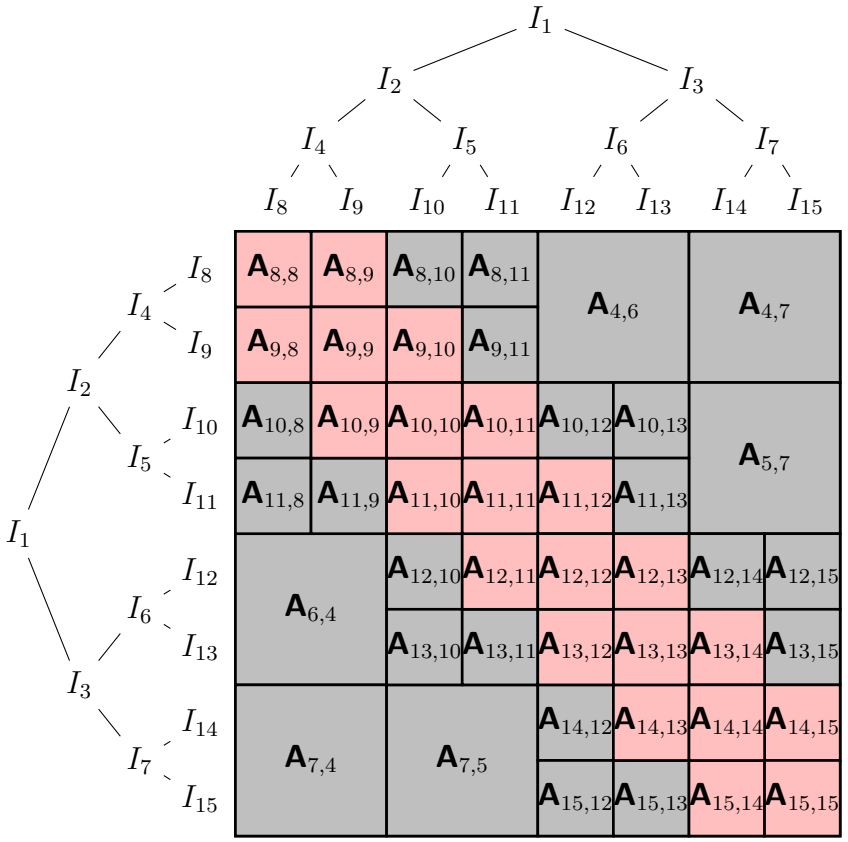


Figure 3.3 *An $\mathcal{H}^1$ matrix with depth 3 based on a grid over $[0,1]$. Admissible blocks are shown in gray, and inadmissible blocks are shown in pink.*

An $\mathcal{H}^1$ matrix $\mathbf{A}$ is said to have uniform $\mathcal{H}^1$ structure if it further satisfies the condition that there exist low-rank basis matrices $\mathcal{U}_\alpha$ spanning $\mathbf{A}\left(I_\alpha, \cup_{\beta\in\mathcal{L}_\alpha^{\text{int}}} I_\beta\right)$ and $\mathcal{V}_\alpha$ spanning $\mathbf{A}\left(\cup_{\beta\in\mathcal{L}_\alpha^{\text{int}}} I_\beta, I_\alpha\right)$ for each box α. Constructing a compressed representation of a uniform $\mathcal{H}^1$ matrix consists of finding the matrices U_α, V_α for each box α and the matrices $\mathbf{B}_{\alpha,\beta}$ such that

$$\mathbf{A}(I_\alpha, I_\beta) \approx \mathcal{U}_\alpha \mathbf{B}_{\alpha,\beta} \mathcal{V}_\beta$$

for each admissible pair (α, β) and storing the inadmissible blocks corresponding

to neighbor interactions of boxes in level L.

A uniform $\mathcal{H}^1$ matrix $\mathbf{A}$ is said to have $\mathcal{H}^2$ structure if there exist low-rank basis matrices $\mathcal{U}_\alpha$ spanning $\mathbf{A}\left(I_\alpha, \left(\cup_{\beta\in\mathcal{L}_\alpha^{\text{nei}}} I_\beta\right)^c\right)$ and $\mathcal{V}_\alpha$ spanning $\mathbf{A}\left(\left(\cup_{\beta\in\mathcal{L}_\alpha^{\text{nei}}} I_\beta\right)^c, I_\alpha\right)$ for each box α. Then the basis matrices of a non-leaf box can be expressed in terms of the basis matrices of its children. For example, if τ is a box with children α, β, then

$$\mathcal{U}_\tau = \begin{bmatrix} \mathcal{U}_\alpha & \mathbf{0} \\ \mathbf{0} & \mathcal{U}_\beta \end{bmatrix} \mathbf{U}_\tau, \tag{3.6}$$

where $\mathcal{U}_\gamma$ is the "long" column basis matrix of size $|\gamma| \times k$ associated with $\gamma \in \{\tau, \alpha, \beta\}$, and $\mathbf{U}_\tau$ is a "short" basis matrix of size $2k \times k$. Analogous relationships must hold for row basis matrices $\mathcal{V}_\tau, \mathcal{V}_\alpha, \mathcal{V}_\beta$. Using such nested basis matrices eliminates the need to explicitly store $\mathcal{U}_\tau$ since it is fully specified by $\mathcal{U}_\alpha, \mathcal{U}_\beta$ and $\mathbf{U}_\tau$. Likewise, if α or β have children of their own, then their basis matrices will be expressed in terms of their own small basis matrices and the basis matrices of their children.

3.4 Compressing rank-structured matrices with graph coloring

3.4.1 $\mathcal{H}^1$ matrix compression

In this section, we present the process of constructing an $\mathcal{H}^1$ representation of a matrix by applying Algorithm 3.2.1 to compute low-rank approximations of the admissible blocks of levels $2, \ldots, L$ and then extracting the inadmissible blocks of level L. We apply A and A^* to a set of carefully

constructed test matrices, and from those products we extract randomized samples of each admissible block. We demonstrate the techniques on a matrix shown in Fig. 3.3, which is based on points in one dimension, but the algorithm generalizes in a straightforward way to points higher dimensions.

3.4.1.1 Compressing level 2

We construct the compressed representation by processing levels of the tree in sequence from the coarsest level to the finest. There are no admissible blocks associated with levels 0 and 1, so we begin by computing low-rank approximations of the 6 admissible blocks associated with level 2 of the tree. To that end, we define four test matrices of size $N \times r$

$$\mathbf{\Omega}_1 = \begin{bmatrix} \mathbf{G}_4 \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \qquad \mathbf{\Omega}_2 = \begin{bmatrix} \mathbf{0} \\ \mathbf{G}_5 \\ \mathbf{0} \\ \mathbf{0} \end{bmatrix}, \qquad \mathbf{\Omega}_3 = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{G}_6 \\ \mathbf{0} \end{bmatrix}, \qquad \mathbf{\Omega}_4 = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{G}_7 \end{bmatrix},$$

where $\mathbf{G}_4, \mathbf{G}_5, \mathbf{G}_6, \mathbf{G}_7$ are random matrices of size $N/4 \times r$ whose entries are drawn from the standard normal distribution. We invoke the black-box matrix-vector product routine to evaluate the products $\mathbf{Y}_i = \mathbf{A}\mathbf{\Omega}_i, i \in \{1, 2, 3, 4\}$, with the following structures. Contained within these products are randomized samples of each admissible block of level 2.

$$\begin{aligned} \mathbf{Y}_1 = \mathbf{A}\mathbf{\Omega}_1 = \begin{bmatrix} * \\ * \\ \mathbf{A}_{6,4}\mathbf{G}_4 \\ \mathbf{A}_{7,4}\mathbf{G}_4 \end{bmatrix}, \qquad \mathbf{Y}_2 = \mathbf{A}\mathbf{\Omega}_2 = \begin{bmatrix} * \\ * \\ * \\ \mathbf{A}_{7,5}\mathbf{G}_5 \end{bmatrix} \\ \mathbf{Y}_3 = \mathbf{A}\mathbf{\Omega}_3 = \begin{bmatrix} \mathbf{A}_{4,6}\mathbf{G}_6 \\ * \\ * \\ * \end{bmatrix}, \qquad \mathbf{Y}_4 = \mathbf{A}\mathbf{\Omega}_4 = \begin{bmatrix} \mathbf{A}_{4,7}\mathbf{G}_7 \\ \mathbf{A}_{5,7}\mathbf{G}_7 \\ * \\ * \end{bmatrix} \end{aligned} \tag{3.7}$$

We then obtain a basis matrix for the column space of each admissible block by orthonormalizing the relevant block of one of the sample matrices.

$$\begin{aligned}
\mathcal{U}_{4,6} &= \mathtt{qr}(\mathbf{Y}_3(I_4,:)) \\
\mathcal{U}_{4,7} &= \mathtt{qr}(\mathbf{Y}_4(I_4,:)) \\
\mathcal{U}_{5,7} &= \mathtt{qr}(\mathbf{Y}_4(I_5,:)) \\
\mathcal{U}_{6,4} &= \mathtt{qr}(\mathbf{Y}_1(I_6,:)) \\
\mathcal{U}_{7,4} &= \mathtt{qr}(\mathbf{Y}_1(I_7,:)) \\
\mathcal{U}_{7,5} &= \mathtt{qr}(\mathbf{Y}_2(I_7,:))
\end{aligned} \tag{3.8}$$

To find a basis matrix for the row space of each admissible block, we follow a similar process using $\mathbf{A}^*$ in place of $\mathbf{A}$. That is, we compute another set of sample matrices $\mathbf{Z}_i = \mathbf{A}^*\boldsymbol{\Psi}_i, i \in \{1,2,3,4\}$, from which we orthonormalize the relevant blocks to obtain row bases $\mathcal{V}_{\alpha,\beta}$ for each admissible pair (α,β).

Finally, we solve for the matrices $\mathbf{B}_{\alpha,\beta}$ as follows. Note that the products $\mathbf{A}_{\alpha,\beta}\mathbf{G}_\beta$ have already been obtained from the samples in Eq. (3.7).

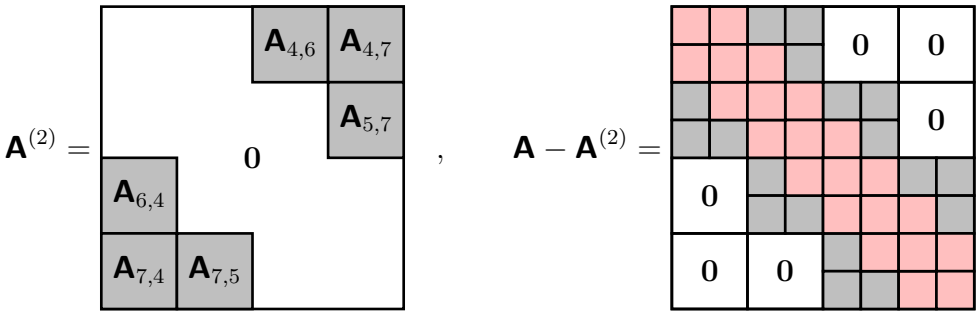
$$\mathbf{B}_{\alpha,\beta} = (\mathbf{G}_\alpha \mathcal{U}_\alpha)^\dagger \, \mathbf{G}_\alpha \mathbf{A}_{\alpha,\beta} \mathbf{G}_\beta \, (\mathcal{V}_\beta \mathbf{G}_\beta)^\dagger \quad \text{for each admissible pair } (\alpha,\beta) \tag{3.9}$$

3.4.1.2 Compressing levels $3, ..., L$

After we have obtained low-rank approximations of the blocks associated with level 2 of the tree, we proceed to level 3. One approach would be to extend the procedure for level 2 by using one test matrix corresponding to each box, for a total of eight test matrices. That approach would grow to be prohibitively

expensive for finer levels of the tree as the number of matrix-vector products required would grow proportionately with the number of boxes. Instead, we present a more efficient procedure, which requires a number of matrix-vector products that is bounded across all levels.

Following [77], we define the level-l truncated matrix $\mathbf{A}^{(l)}$ to be the matrix obtained by replacing with zeros every block of $\mathbf{A}$ corresponding to levels finer than level l. Note that a matrix-vector product involving $\mathbf{A}^{(l)}$ can be computed inexpensively using the low-rank approximations already computed when processing levels $2, \ldots, l-1$. The structures of the level-2 truncated matrix and the difference $\mathbf{A} - \mathbf{A}^{(2)}$ are shown below.



We will sample the admissible blocks of level 3 by applying $\mathbf{A} - \mathbf{A}^{(2)}$ to a set of test matrices subject to certain conditions on their sparsity structure. For example, to isolate a sample of $\mathbf{A}_{8,10}$ (see Fig. 3.3), we must avoid unwanted contributions from $\mathbf{A}_{8,8}, \mathbf{A}_{8,9}, \mathbf{A}_{8,11}$, so we multiply $\mathbf{A} - \mathbf{A}^{(2)}$ with a test matrix whose rows indexed by I_8, I_9, I_{11} are all zeros and whose rows indexed by I_{10} are filled with random values drawn from the standard normal distribution.

The contents of the other rows are irrelevant for the purpose of sampling $\mathbf{A}_{8,10}$ since they will be multiplied with zeros in $\mathbf{A} - \mathbf{A}^{(2)}$. More generally, to sample some admissible block $\mathbf{A}_{\alpha,\beta}$ of level 3, we require a test matrix $\mathbf{\Omega}$ that satisfies the following *sampling constraints*.

$$\begin{aligned} \mathbf{\Omega}(I_\beta, :) &= \mathbf{G}_\beta \\ \mathbf{\Omega}(I_\gamma, :) &= \mathbf{0} \qquad \text{for all } \gamma \in \mathcal{L}_\alpha^{\text{nei}} \cup \mathcal{L}_\alpha^{\text{int}} \setminus \{\beta\} \end{aligned} \tag{3.10}$$

where $\mathbf{G}_\beta$ is a random matrix of size $|I_\beta|$-by-r. If test matrix $\mathbf{\Omega}$ satisfies those sampling constraints, then the rows of the product $\mathbf{A\Omega} - \mathbf{A}^{(2)}\mathbf{\Omega}$ indexed by I_α will contain a randomized sample of the column space of $\mathbf{A}_{\alpha,\beta}$.

To sample all of the admissible blocks, we require a set of test matrices $\{\mathbf{\Omega}_i\}$ such that for every admissible pair (α, β), there is a test matrix within the set that satisfies the constraints (3.10) associated with that pair. Moreover, we would like that set to be as small as possible to minimize cost.

In order to minimize the number of test matrices, we aim to form a small number of groups of compatible sampling constraints. We say that two sets of sampling constraints are *compatible* if it is possible to form a test matrix $\mathbf{\Omega}$ that satisfies both sets of constraints. We then define a *constraint incompatibility graph*, which represents compatibility relationships between pairs of constraint sets. The graph corresponding to the 18 admissible blocks belonging to level 3 is depicted in Figure 3.4.

Definition 1 (Constraint incompatibility graph)**.** *The constraint incompatibility graph for level l of the tree is the graph in which each vertex corresponds to*

a distinct constraint set (3.10)*, and pairs of vertices are connected by an edge if their corresponding constraint sets are incompatible.*

We then compute a vertex coloring of the constraint incompatibility graph using the DSatur algorithm (Algorithm 3.2.3). For a valid coloring of the graph, each subset of vertices sharing the same color represents a mutually compatible collection of sampling constraints. Then for each color, we can define one test matrix that satisfies all of the sampling constraints associated with the vertices of that color. The coloring shown in Figure 3.4 yields test matrices with the following structures.

$$\begin{bmatrix} \boldsymbol{\Omega}_1 & \boldsymbol{\Omega}_2 & \boldsymbol{\Omega}_3 & \boldsymbol{\Omega}_4 & \boldsymbol{\Omega}_5 & \boldsymbol{\Omega}_6 \end{bmatrix} = \begin{bmatrix} \mathbf{G}_8 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{G}_9 & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{G}_{10} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{G}_{11} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{G}_{12} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{G}_{13} \\ \mathbf{G}_{14} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{G}_{15} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \tag{3.11}$$

As in Section 3.4.1.1, we evaluate the samples $\mathbf{Y}_i = \mathbf{A}\boldsymbol{\Omega}_i - \mathbf{A}^{(2)}\boldsymbol{\Omega}_i$ and orthonormalize the relevant blocks to obtain orthonormal bases $\boldsymbol{\mathcal{U}}_{\alpha,\beta}$ of the column spaces of the admissible blocks. A similar process yields orthormal bases $\boldsymbol{\mathcal{V}}_{\alpha,\beta}$ of the row spaces. Finally, we solve for $\mathbf{B}_{\alpha,\beta}$ again using (3.9).

3.4.1.3 Extracting inadmissible blocks of the leaf level

Once we have computed low-rank approximations of the admissible blocks for every level, we finally extract the the inadmissible blocks of the leaf

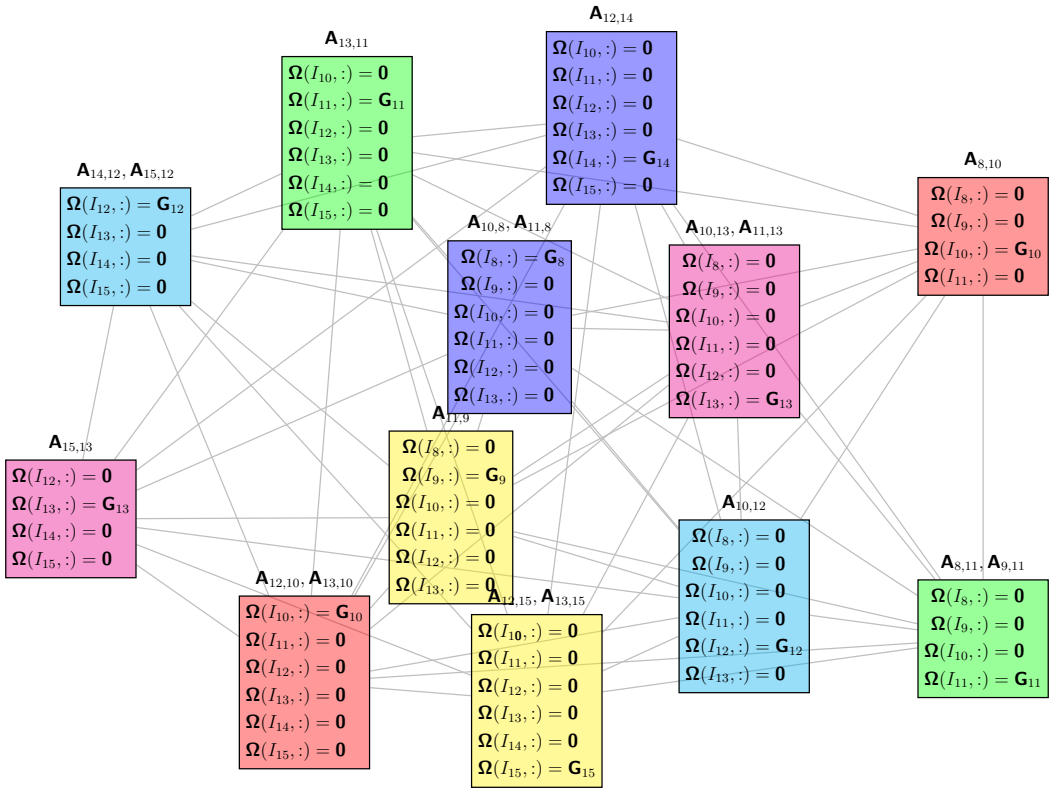


Figure 3.4 *The constraint incompatibility graph corresponding to the 18 admissible blocks belonging to level 3 of the matrix shown in Figure 3.3. Each vertex corresponds to a distinct set of sampling constraints* (3.10)*. Edges connect pairs of vertices that are incompatible. The number of vertices is less than the number of admissible blocks since some admissible blocks share the same set of sampling constraints.*

level. Since the inadmissible blocks are not necessarily low-rank, they cannot be recovered from a small number of randomized samples. Instead, we use test matrices that will multiply the inadmissible blocks with appropriately sized identity matrices. Also, at this point we already have low-rank approximations of the admissible blocks belonging to level L, so the test matrices only need to avoid contributions from other inadmissible blocks, resulting in fewer constraints

than (3.10). To extract inadmissible block $\mathbf{A}_{\alpha,\beta}$ of level L, we require a test matrix $\mathbf{\Omega}$ that satisfies the following sampling constraints.

$$\mathbf{\Omega}(I_\beta, :) = \mathbf{I}$$

$$\mathbf{\Omega}(I_\gamma, :) = \mathbf{0} \quad \text{for all } \gamma \in \mathcal{L}_\alpha^{\text{nei}} \setminus \{\beta\}$$

If test matrix $\mathbf{\Omega}$ satisfies these constraints, then the rows of the product $\mathbf{A\Omega} - \mathbf{A}^{(2)}\mathbf{\Omega}$ indexed by I_α will contain $\mathbf{A}_{\alpha,\beta}$. The graph corresponding to the 22 inadmissible blocks belonging to level 3 is depicted in Figure 3.5, and its coloring produces test matrices with the following structures. The entire process of compressing an $\mathcal{H}^1$ matrix is summarized in Algorithm 3.4.1.

$$\begin{bmatrix} \mathbf{\Omega}_1 & \mathbf{\Omega}_2 & \mathbf{\Omega}_3 \end{bmatrix} = \begin{bmatrix} \mathbf{G}_8 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{G}_9 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{G}_{10} \\ \mathbf{G}_{11} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{G}_{12} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{G}_{13} \\ \mathbf{G}_{14} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{G}_{15} & \mathbf{0} \end{bmatrix}$$

3.4.1.4 General patterns in the test matrices for $\mathcal{H}^1$ compression

In this section, we describe sets of test matrices that are sufficient for compressing any $\mathcal{H}^1$ matrix based on points in arbitrary dimensions. Though sufficient, these sets of test matrices are not necessarily the smallest possible, and the proposed method of constructing problem-specific test matrices via graph coloring often yields a smaller set of test matrices, resulting in fewer matrix-vector products. The test matrices described in this section establish

Algorithm 3.4.1 Randomized compression of an $\mathcal{H}^1$ matrix

for level $l \in [2, ..., L]$ **do**

Compute randomized samples of $\mathbf{A} - \mathbf{A}^{(l)}$

Construct structured random test matrices $\{\mathbf{\Omega}_i\}$ of size $N \times (k+p)$ as in Section 3.4.1.2 or Section 3.4.1.4

for all $\mathbf{\Omega}_i$ **do**

Multiply $\mathbf{Y}_i = \mathbf{A}\mathbf{\Omega}_i - \mathbf{A}^{(l)}\mathbf{\Omega}_i$

Compute orthonormal basis matrices $\mathcal{U}_{\alpha,\beta}$

for all interacting pairs α, β in level l **do**

Identify $\mathbf{Y}_i$ that contains a sample of $\mathbf{A}_{\alpha,\beta}$ in $\mathbf{Y}_i(I_\alpha, :)$

Orthonormalize $\mathcal{U}_{\alpha,\beta} = \mathtt{qr}(\mathbf{Y}_i(I_\alpha, :), k)$

Compute randomized samples of $\mathbf{A}^* - \mathbf{A}^{(l)^*}$

Construct structured random test matrices $\{\mathbf{\Psi}_i\}$ of size $N \times (k+p)$ as in Section 3.4.1.2 or Section 3.4.1.4 (if the rank structure of $\mathbf{A}$ is symmetric, then the test matrices $\{\mathbf{\Omega}_i\}$ can be reused)

for all $\mathbf{\Psi}_i$ **do**

Multiply $\mathbf{Z}_i = \mathbf{A}^*\mathbf{\Psi}_i - \mathbf{A}^{(l)^*}\mathbf{\Psi}_i$

Compute orthonormal basis matrices $\mathcal{V}_{\alpha,\beta}$

for all interacting pairs α, β in level l **do**

Identify $\mathbf{Z}_i$ that contains a sample of $\mathbf{A}^*_{\alpha,\beta}$ in $\mathbf{Z}_i(I_\beta, :)$

Orthonormalize $\mathcal{V}_{\alpha,\beta} = \mathtt{qr}(\mathbf{Z}_i(I_\beta, :), k)$

Solve for $\mathbf{B}$

for all interacting pairs α, β in level l **do**

$\mathbf{B}_{\alpha,\beta} = (\mathbf{G}_\alpha \mathcal{U}_\alpha)^\dagger \mathbf{G}_\alpha \mathbf{A}_{\alpha,\beta} \mathbf{G}_\beta (\mathcal{V}_\beta \mathbf{G}_\beta)^\dagger$

Extract the inadmissible blocks of level L

Construct structured random test matrices $\{\mathbf{\Omega}_i\}$ of size $N \times m$ as in Section 3.4.1.3 or Section 3.4.1.4

for all $\mathbf{\Omega}_i$ **do**

Multiply $\mathbf{Y}_i = \mathbf{A}\mathbf{\Omega}_i - \mathbf{A}^{(L)}\mathbf{\Omega}_i$

for all neighbor pairs α, β in level L **do**

Identify $\mathbf{Y}_i$ that contains $\mathbf{A}_{\alpha,\beta}$

Extract the block $\mathbf{A}_{\alpha,\beta}$ from $\mathbf{Y}_i(I_\alpha, :)$

an upper bound on the number of matrix-vector products that improves on the number given in [64], and they also imply a bound on the chromatic number of the graph, which appears in the estimate of asymptotic complexity.

A general set of test matrices for sampling admissible blocks The structures of the test matrices derived in Sections 3.4.1.2 and 3.4.1.3 exhibit patterns that generalize to finer and higher-dimensional grids. The sampling constraints Eq. (3.10) for admissible block $\mathbf{A}_{\alpha,\beta}$ specify that the rows of the test matrix corresponding to β must be filled with random values, and the rows of the test matrix corresponding to the other boxes in $\mathcal{L}_\alpha^{\text{nei}} \cup \mathcal{L}_\alpha^{\text{int}}$ must be filled with zeros. For a 1-dimensional problem, the indices corresponding to $\mathcal{L}_\alpha^{\text{nei}} \cup \mathcal{L}_\alpha^{\text{int}}$ form 6 contiguous blocks of the test matrix. Accordingly, each of the test matrices defined in Eq. (3.7) has every sixth block filled with random values and the other blocks filled with zeros, a pattern that generalizes to an arbitrarily fine grid in one dimension.

To describe a general set of test matrices for a problem in 2 dimensions, we partition the domain into tiles, each of which covers 6×6 boxes. We define 36 test matrices, each activating a set of boxes that share the same position within their respective tiles. The set of boxes activated by one of the test matrices is shown in Fig. 3.6. The sampling constraints in Eq. (3.10) apply to $\mathcal{L}_\alpha^{\text{nei}} \cup \mathcal{L}_\alpha^{\text{int}}$, which form a square of 6×6 boxes, and they specify that exactly one of those boxes must be activated. Therefore, those constraints will be satisfied by one of the 36 test matrices. By a similar argument, this pattern generalizes

to higher-dimensional problems, requiring at most 6^d test matrices to sample one level of admissible blocks for a problem in d dimensions. Furthermore, since these test matrices satisfy Eq. (3.10), they must correspond to a valid coloring of the graph described in Section 3.4.1.2, bounding the chromatic number of the graph by

$$\chi_{\text{nonunif}} \leq 6^d.$$

A general set of test matrices for extracting inadmissible blocks A similar argument proves that extracting inadmissible blocks of the leaf level can be accomplished using 3^d test matrices. The set of boxes activated by one of the test matrices for a problem in two dimensions is shown in Fig. 3.6. Therefore, the chromatic number of the graph described in Section 3.4.1.3 is bounded by

$$\chi_{\text{leaf}} \leq 3^d.$$

3.4.1.5 Asymptotic complexity

Let $L \sim \log N$ denote the depth of the tree, T_{mult} denote the time to apply $\mathbf{A}$ or $\mathbf{A}^*$ to a vector, and T_{flop} denote the time to execute one floating point operation. There are 2^{dl} boxes belonging to level l of the tree, and the interaction list of each box consists of up to $6^d - 3^d$ other boxes, so there are approximately $(6^d - 3^d)2^{dl}$ admissible blocks associated with level l. The cost of applying the low-rank approximation of an admissible block associated with level l to a vector is $\sim kN/2^{dl}$ operations. Therefore, the cost of applying the

level-l truncated matrix $\mathbf{A}^{(l)}$ (or its transpose) to a vector is

$$T_{A^{(l)}} \sim T_{\text{flop}} \times \sum_{j=0}^{l} (6^d - 3^d) 2^{dj} k \frac{N}{2^{dj}} \sim T_{\text{flop}} \times (6^d - 3^d) lkN.$$

The cost of compressing the admissible blocks associated with level l of the tree is

$$T_l \sim (T_{\text{mult}} + T_{A^{(l)}}) \times 2\chi_{\text{nonunif}} k + T_{\text{flop}} \times (6^d - 3^d) 2^{dl} k^2 \frac{N}{2^{dl}},$$

since we require $\sim \chi_{\text{nonunif}} k$ applications of $\mathbf{A} - \mathbf{A}^{(l)}$ and its transpose, where χ_{nonunif} denotes the chromatic number of the graph described in Section 3.4.1.2, and we require an additional $\sim k^2 N / 2^{dl}$ operations for each admissible block to compute low-rank approximations.

Finally, summing T_l over each level gives the total time to construct an $\mathcal{H}^1$ representation as follows. The cost of extracting inadmissible blocks associated with the leaf level of the tree is omitted as it only contributes a lower order term to the overall cost. We also omit the costs associated with the graph coloring problem since it is only worthwhile for problems that exhibit low-dimensional structure, in which case the costs would be much lower than the worst-case analysis would suggest, and we observe in practice that the cost of graph coloring represents a small portion of the total cost of compression. For problems without low-dimensional structure, one may skip the graph coloring step and instead use the general set of test matrices described in Section 3.4.1.4.

$$T_{\text{compress}} \sim T_{\text{mult}} \times 2\chi_{\text{nonunif}} k \log N + T_{\text{flop}} \times (6^d - 3^d) 2\chi_{\text{nonunif}} k^2 N (\log N)^2$$

3.4.2 Uniform $\mathcal{H}^1$ matrix compression

A uniform $\mathcal{H}^1$ approximation requires for each box τ a column basis matrix $\boldsymbol{\mathcal{U}}_\tau$ and a row basis matrix $\boldsymbol{\mathcal{V}}_\tau$ such that

$$\mathbf{A}_{\alpha,\beta} \approx \boldsymbol{\mathcal{U}}_\alpha \boldsymbol{\mathcal{U}}_\alpha^* \mathbf{A}_{\alpha,\beta} \boldsymbol{\mathcal{V}}_\beta \boldsymbol{\mathcal{V}}_\beta^*$$

for all admissible pairs (α, β). In other words, $\boldsymbol{\mathcal{U}}_\alpha$ must span the column space of $\mathbf{A}\left(I_\alpha, \cup_{\beta \in \mathcal{L}_\alpha^{\text{int}}} I_\beta\right)$, the submatrix of interactions between α and the boxes in its interaction list. Similarly, $\boldsymbol{\mathcal{V}}_\beta$ must span the row space of $\mathbf{A}\left(\cup_{\alpha \in \mathcal{L}_\beta^{\text{int}}} I_\alpha, I_\beta\right)$. The algorithms for compressing $\mathcal{H}^1$ and uniform $\mathcal{H}^1$ matrices have a two important differences. For a uniform $\mathcal{H}^1$ matrix, we use Algorithm 3.2.2, rather than Algorithm 3.2.1 to compute low-rank approximations. Also, instead of sampling the column space of each admissible block $\mathbf{A}_{\alpha,\beta}$ separately, we will sample the interactions between each box and all of the boxes in its interaction list together.

We return to the task of compressing the admissible blocks associated with level 3 of the tree as described in Section 3.4.1.2. As before, we will sample $\mathbf{A} - \mathbf{A}^{(2)}$ with a set of test matrices. To sample those interactions for some box α, we require a test matrix $\boldsymbol{\Omega}$ that satisfies the following sampling constraints.

$$\begin{aligned} \boldsymbol{\Omega}(I_\beta, :) &= \mathbf{G}_\beta \quad \text{for all } \beta \in \mathcal{L}_\alpha^{\text{int}} \\ \boldsymbol{\Omega}(I_\gamma, :) &= \mathbf{0} \quad \text{for all } \gamma \in \mathcal{L}_\alpha^{\text{nei}} \end{aligned} \tag{3.12}$$

If test matrix $\boldsymbol{\Omega}$ satisfies the above sampling constraints, then the rows of the product $\mathbf{A}\boldsymbol{\Omega} - \mathbf{A}^{(2)}\boldsymbol{\Omega}$ indexed by I_α will contain a randomized sample of

the column space of $\mathbf{A}\left(I_\alpha, \cup_{\beta\in\mathcal{L}_\alpha^{\text{int}}} I_\beta\right)$, the submatrix of interactions between α and the boxes in its interaction list.

We have one set of sampling constraints for each box α, and we use them to form the constraint incompatibility graph (Definition 1) shown in Fig. 3.7. As in Section 3.4.1.2, we compute a vertex coloring of the graph. The coloring of the graph in Fig. 3.7 specifies test matrices with the following structures.

$$\begin{bmatrix}\boldsymbol{\Omega}_1 & \boldsymbol{\Omega}_2 & \boldsymbol{\Omega}_3 & \boldsymbol{\Omega}_4 & \boldsymbol{\Omega}_5\end{bmatrix} = \begin{bmatrix} \mathbf{0} & \mathbf{0} & \mathbf{G}_8 & \mathbf{G}_8 & * \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{G}_9 & * \\ \mathbf{G}_{10} & \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{G}_{10} \\ \mathbf{G}_{11} & \mathbf{G}_{11} & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{G}_{12} & \mathbf{G}_{12} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{G}_{13} & \mathbf{G}_{13} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & * & \mathbf{G}_{14} \\ \mathbf{G}_{15} & \mathbf{0} & \mathbf{0} & * & \mathbf{G}_{15} \end{bmatrix}$$

We evaluate the samples $\mathbf{Y}_i = \mathbf{A}\boldsymbol{\Omega}_i - \mathbf{A}^{(2)}\boldsymbol{\Omega}_i$ for each test matrix $\boldsymbol{\Omega}_i$, and orthonormalize the relevant blocks of the sample matrices $\mathbf{Y}_i$ to obtain orthonormal column basis matrix $\boldsymbol{\mathcal{U}}_\alpha$ for each box α. That is, if $\boldsymbol{\Omega}_i$ is the test matrix satisfying the sampling constraints of box α, then

$$\mathbf{Y}_i(I_\alpha, :) = \sum_{\beta\in\mathcal{L}_\alpha^{\text{int}}} \mathbf{A}_{\alpha,\beta}\mathbf{G}_\beta,$$

so we compute

$$\boldsymbol{\mathcal{U}}_\alpha = \mathtt{qr}\left(\mathbf{Y}_i(I_\alpha, :)\right).$$

For the second stage of compression using Algorithm 3.2.2, we must compute $\mathbf{A}_{\alpha,\beta}^*\boldsymbol{\mathcal{U}}_\alpha$ for each admissible block $\mathbf{A}_{\alpha,\beta}$. To do so, we use the procedure

for sampling an $\mathcal{H}^1$ matrix with non-uniform basis matrices (Section 3.4.1.2), but rather than filling blocks of the test matrices with random values, we fill them with appropriately chosen uniform basis matrices. With that modification, we have the following sampling constraints for each admissible pair (α, β).

$$\begin{aligned} \boldsymbol{\Psi}(I_\beta, :) &= \mathcal{U}_\beta \\ \boldsymbol{\Psi}(I_\gamma, :) &= \mathbf{0} \qquad \text{for all } \gamma \in \mathcal{L}_\alpha^{\text{nei}} \cup \mathcal{L}_\alpha^{\text{int}} \setminus \{\beta\} \end{aligned}$$

We construct a suitable set of test matrices via graph coloring and evaluate the products $\mathbf{Z}_i = \left(\mathbf{A} - \mathbf{A}^{(2)}\right)^* \boldsymbol{\Psi}_i$ for each test matrix $\boldsymbol{\Psi}_i$. Then for each admissible pair (α, β), there is some $\mathbf{Z}_i$ such that $\mathbf{Z}_i(I_\alpha, :) = \mathbf{A}_{\alpha,\beta}^* \mathcal{U}_\beta$. With those results, we compute the uniform basis matrices $\mathcal{V}_\alpha$ and the matrices $\mathbf{B}_{\alpha,\beta}$ as follows.

$$\begin{aligned} \mathcal{V}_\beta &= \mathtt{qr}\left(\sum_{\alpha \in \mathcal{L}_\beta^{\text{int}}} \mathbf{Z}_i(I_\alpha, :)\right) = \mathtt{qr}\left(\sum_{\alpha \in \mathcal{L}_\beta^{\text{int}}} \mathbf{A}_{\alpha,\beta}^* \mathcal{U}_\alpha\right) \\ \mathbf{B}_{\alpha,\beta} &= \mathbf{Z}_i(I_\alpha, :)^* \mathcal{V}_\beta = (\mathbf{A}_{\alpha,\beta}^* \mathcal{U}_\alpha)^* \mathcal{V}_\beta \end{aligned}$$

Remark 8. *An alternative approach for uniform $\mathcal{H}^1$ compression [64] is obtained by adding a uniformization step to the $\mathcal{H}^1$ compression algorithm. With that approach, one first computes low-rank approximations $\mathbf{A}_{\alpha,\beta} = \mathcal{U}_{\alpha,\beta} \mathbf{B}_{\alpha,\beta} \mathcal{V}_{\alpha,\beta}^*$ with non-uniform basis matrices. Those low-rank approximations are recompressed to obtain uniform basis matrices, followed by a change of basis.*

$$\mathbf{A}_{\alpha,\beta} \approx \mathcal{U}_\alpha \left(\mathcal{U}_\alpha^* \mathcal{U}_{\alpha,\beta} \mathbf{B}_{\alpha,\beta} \mathcal{V}_{\alpha,\beta}^* \mathcal{V}_\beta\right) \mathcal{V}_\beta^*$$

One disadvantage of that approach is that $\mathcal{U}_\alpha$ and $\mathcal{V}_\alpha$ are based on samples of the factorizations $\mathcal{U}_{\alpha,\beta} \mathbf{B}_{\alpha,\beta} \mathcal{V}_{\alpha,\beta}^*$, which may only be approximate, rather

than direct samples of the blocks $\mathbf{A}_{\alpha,\beta}$ themselves, potentially resulting in lower accuracy. Another difference is that the procedure for sampling a uniform $\mathcal{H}^1$ matrix presented in this section requires fewer matrix-vector products than the procedure for sampling an $\mathcal{H}^1$ matrix, as described in Sections 3.4.1.4 and 3.4.2.1.

3.4.2.1 General patterns in the test matrices for uniform $\mathcal{H}^1$ compression

As in Section 3.4.1.4, we describe a general set of test matrices that is applicable to finer and higher-dimensional grids. To sample the interactions of box α for a problem in 2 dimensions, the blocks of the test matrix corresponding to the boxes in $\mathcal{L}_\alpha^{\text{nei}}$ must be filled with zeros, and the blocks corresponding to the boxes in $\mathcal{L}_\alpha^{\text{int}}$ must be filled with random values Eq. (3.12). The pattern of activated boxes for one test matrix is shown in Fig. 3.8. The complete set of 25 test matrices is obtained by shifting the pattern horizontally and vertically. The pattern generalizes to higher-dimensional problems, requiring at most 5^d test matrices to carry out the first stage of uniform $\mathcal{H}^1$ sampling for a problem in d dimensions. Therefore, we establish the upper bound

$$\chi_{\text{unif}} \leq 5^d.$$

3.4.3 $\mathcal{H}^2$ matrix compression

To obtain an algorithm for computing an $\mathcal{H}^2$ representation, we apply two modifications to the algorithm for computing a uniform $\mathcal{H}^1$ representation.

First, we enrich the basis matrices of each box so that they span the relevant parts of the basis matrices of its parent. Specifically, if box τ is the parent of box α, then when computing the basis matrix $\mathbf{U}_\alpha$, we augment $\mathbf{Y}_i(I_\alpha, :)$, the sample of $\mathbf{A}\left(I_\alpha, \cup_{\beta\in\mathcal{L}_\alpha^{\text{int}}} I_\beta\right)$, with $\mathbf{U}_\tau(I_\alpha, :)\mathbf{\Sigma}_\tau^{\text{in}}\mathbf{G}_k$, a sample of $\mathbf{A}\left(I_\alpha, \left(\cup_{\beta\in\mathcal{L}_\tau^{\text{nei}}} I_\beta\right)^c\right)$, where $\mathbf{G}_k$ is a k-by-k Gaussian random matrix.

$$\mathbf{Y}_i(I_\alpha, ;) = \sum_{\beta\in\mathcal{L}_\alpha^{\text{int}}} \mathbf{A}_{\alpha,\beta}\mathbf{G}_\beta$$

$$[\mathcal{U}_\alpha, \mathbf{\Sigma}_\alpha^{\text{in}}, \sim] = \texttt{svd}\left(\mathbf{Y}_i(I_\alpha, :) + \mathbf{U}_\tau\mathbf{\Sigma}_\tau^{\text{in}}\mathbf{G}_k\right)$$

We use a similarly augmented sample when computing $\mathbf{V}_\alpha$.

$$[\mathcal{V}_\alpha, \mathbf{\Sigma}_\alpha^{\text{out}}, \sim] = \texttt{svd}\left(\sum_{\beta\in\mathcal{L}_\alpha^{\text{int}}} \mathbf{A}_{\beta,\alpha}^*\mathbf{U}_\beta + \mathbf{V}_\tau\mathbf{\Sigma}_\tau^{\text{out}}\mathbf{G}_k\right)$$

Second, after we have computed long basis matrices of its children, we compute the short basis matrix $\mathbf{U}_\tau$ by solving (3.6), and then we can discard $\mathcal{U}_\tau$. Similarly, we compute $\mathbf{V}_\tau$ and discard $\mathcal{V}_\tau$. The process of compressing uniform $\mathcal{H}^1$ and $\mathcal{H}^2$ matrices are summarized in Algorithm 3.4.2. The algorithm for applying a level-truncated approximation of an $\mathcal{H}^2$ matrix to a vector is given in Algorithm 3.4.3.

3.4.3.1 Asymptotic complexity

The asymptotic complexity of the $\mathcal{H}^2$ algorithm is similar to that of the $\mathcal{H}^1$ algorithm. Since we are now using uniform basis matrices, the cost of applying $T_{A^{(l)}}$ is lower.

$$\begin{aligned} T_{A^{(l)}} &\sim T_{\text{flop}} \times \left(2^{dl}k\frac{N}{2^{dl}} + \sum_{j=0}^{l}(6^d - 3^d)k^2 2^{dj}\right) \\ &\sim T_{\text{flop}} \times \left(kN + (6^d - 3^d)k^2 2^{dl}\right) \end{aligned}$$

Algorithm 3.4.2 Randomized compression of a uniform $\mathcal{H}^1$ or $\mathcal{H}^2$ matrix

for level $l \in [2, ..., L]$ **do**

Compute randomized samples of $\mathbf{A} - \mathbf{A}^{(l)}$

Construct structured random test matrices $\{\mathbf{\Omega}_i\}$ of size $N \times (k+p)$ as in Section 3.4.2 or Section 3.4.2.1

for all $\mathbf{\Omega}_i$ **do**

Multiply $\mathbf{Y}_i = \mathbf{A}\mathbf{\Omega}_i - \mathbf{A}^{(l)}\mathbf{\Omega}_i$

Compute uniform orthonormal basis matrices $\mathcal{U}_\alpha$

for all boxes α in level l **do**

Identify $\mathbf{Y}_i$ that contains a sample of $\mathbf{A}(I_\alpha, \cup_{\beta \in \mathcal{L}_\alpha^{\text{int}}} I_\beta)$ in $\mathbf{Y}_i(I_\alpha, :)$

if $\mathbf{A}$ has uniform $\mathcal{H}^1$ structure **then**

Compute an SVD of the sample: $[\mathcal{U}_\alpha, \mathbf{\Sigma}_\alpha^{\text{in}}, \sim] = \texttt{svd}(\mathbf{Y}_i(I_\alpha, :), k)$

else if $\mathbf{A}$ has $\mathcal{H}^2$ structure **then**

Compute an SVD of the augmented sample: $[\mathcal{U}_\alpha, \mathbf{\Sigma}_\alpha^{\text{in}}, \sim] = \texttt{svd}(\mathbf{Y}_i(I_\alpha, :) + \mathcal{U}_\tau \mathbf{\Sigma}_\tau^{\text{in}} \mathbf{G}_k, k)$

Compute randomized samples of $\mathbf{A}^* - \mathbf{A}^{(l)^*}$

Construct structured random test matrices $\{\mathbf{\Psi}_i\}$ of size $N \times (k+p)$ as in Section 3.4.1.2 or Section 3.4.1.4

for all $\mathbf{\Psi}_i$ **do**

Multiply $\mathbf{Z}_i = \mathbf{A}^*\mathbf{\Psi}_i - \mathbf{A}^{(l)^*}\mathbf{\Psi}_i$

Compute uniform orthonormal basis matrices $\mathcal{V}_\beta$

for all boxes β in level l **do**

Identify $\mathbf{Z}_i$ that contains $\mathbf{A}_{\alpha,\beta}^* \mathcal{U}_\alpha$ in $\mathbf{Z}_i(I_\beta, :)$ for $\alpha \in \mathcal{L}_\beta^{\text{int}}$

if $\mathbf{A}$ has uniform $\mathcal{H}^1$ structure **then**

Compute an SVD of the sample: $[\mathcal{V}_\beta, \mathbf{\Sigma}_\beta^{\text{out}}, \sim] = \texttt{svd}(\sum_{\alpha \in \mathcal{L}_\beta^{\text{int}}} \mathbf{Z}_i, k)$

else if $\mathbf{A}$ has $\mathcal{H}^2$ structure **then**

Compute an SVD of the augmented sample: $[\mathcal{V}_\beta, \mathbf{\Sigma}_\beta^{\text{out}}, \sim] = \texttt{svd}(\mathcal{V}_\tau \mathbf{\Sigma}_\tau^{\text{out}} \mathbf{G}_k + \sum_{\alpha \in \mathcal{L}_\beta^{\text{int}}} \mathbf{Z}_{\alpha,\beta}(I_\beta, :), k)$

Compute $\mathbf{B}_{\alpha,\beta}$

for all interacting pairs α, β in level l **do**

Identify $\mathbf{Z}_i$ that contains $\mathbf{A}_{\alpha,\beta}^* \mathcal{U}_\alpha$ in $\mathbf{Z}_i(I_\beta, :)$ for $\alpha \in \mathcal{L}_\beta^{\text{int}}$

$\mathbf{B}_{\alpha,\beta} = \mathbf{Z}_i^*(I_\beta, :)\mathcal{V}_\beta$

Nest the basis matrices

if $\mathbf{A}$ has $\mathcal{H}^2$ structure **then**

for all boxes α in level l **do**

Compute $\mathbf{U}_\alpha, \mathbf{V}_\alpha$ using Eq. (3.6)

Discard $\mathcal{U}_\alpha, \mathcal{V}_\alpha$

Extract the inadmissible blocks of level L in the same way as in Algorithm 3.4.1

Algorithm 3.4.3 Applying an $\mathcal{H}^2$ level-truncated approximation $\mathbf{A}^{(l)}$ to vector $\mathbf{q}$

Build outgoing expansions in level l.
for all boxes τ in level l **do**
  $\hat{\mathbf{q}}_\tau = \mathcal{V}_\tau^* \mathbf{q}(I_\tau)$

Build outgoing expansions for levels coarser than l (upward pass).
for all levels $i \in [l-1, ...2]$ **do**
  for all boxes τ in level i **do**
    for all children α of τ **do**
      $\hat{\mathbf{q}}_\tau(I_\alpha, :) = \mathbf{V}_\tau^*(I_\alpha, :)\hat{\mathbf{q}}_\alpha$

Build incoming expansions for boxes in level 2.
for all boxes α in level 2 **do**
  $\hat{\mathbf{u}}_\alpha = \sum_{\beta \in \mathcal{L}_\alpha^{\text{int}}} \mathbf{B}_{\alpha,\beta}\hat{\mathbf{q}}_\beta$

Build incoming expansions for levels finer than 2 (downward pass).
for all levels $i \in [3, ..., l-1]$ **do**
  for all boxes α in level i **do**
    Let τ be the parent of α
    $\hat{\mathbf{u}}_\alpha = \sum_{\beta \in \mathcal{L}_\alpha^{\text{int}}} \mathbf{B}_{\alpha,\beta}\hat{\mathbf{q}}_\beta + \mathbf{U}_\tau(I_\alpha, :)\hat{\mathbf{u}}_\tau$

Build incoming expansions for level l.
for all boxes α in level l **do**
  $\mathbf{u}(I_\alpha) = \mathcal{U}_\tau \hat{\mathbf{u}}_\tau + \sum_{\beta \in \mathcal{L}_\alpha^{\text{nei}}} \mathbf{A}(I_\alpha, I_\beta)\mathbf{q}(I_\beta)$

The number of matrix-vector products to carry out sampling for uniform basis matrices is also lower.

$$T_l \sim (T_{\text{mult}} + T_{A^{(l)}}) \times (\chi_{\text{unif}} + \chi_{\text{nonunif}})k + T_{\text{flop}} \times (6^d - 3^d)2^{dl}k^2\frac{N}{2^{dl}},$$

Summing T_l over each level, we find that the number of floating point operations is lower than that of Section 3.4.1.5 by a factor of $\mathcal{O}(\log N)$. As in Section 3.4.1.5, we omit the costs associated with graph coloring.

$$\begin{aligned} T_{\text{compress}} \sim\ & T_{\text{mult}} \times (\chi_{\text{unif}} + \chi_{\text{nonunif}})\, k \log N \\ & + T_{\text{flop}} \times k^2 N \left(\left(\chi_{\text{unif}} + \chi_{\text{nonunif}} + 6^d - 3^d\right) \log N \right. \\ & \qquad \left. + (6^d - 3^d)(\chi_{\text{unif}} + \chi_{\text{nonunif}}) \right) \end{aligned}$$

3.5 Numerical experiments

In this section, we present a selection of numerical results. The experiment in Section 3.5.1 demonstrates the ability of the graph coloring approach to exploit low-dimensional structure. In Sections 3.5.2 to 3.5.5, we report the following quantities for a number of test problems and rank structure formats: (1) the time to compress the operator, (2) the time to apply the compressed representation to a vector, (3) the relative accuracy of the compressed representation, and (4) the storage requirements of the compressed representation measured as the number of values per degree of freedom. For compression time, we report both the total time taken for compression as well as the "net time," which does not include the time spent by the black-box multiplication routine.

The algorithms for compressing matrices and applying compressed representations are written in Python, and the black-box multiplication routines are written in MATLAB. The experiments were carried out on a workstation with two Intel Xeon Gold 6254 processors with 18 cores each and 754GB of memory.

The matrices are compressed with the $\mathcal{H}^1$, uniform $\mathcal{H}^1$, and $\mathcal{H}^2$ formats. For the uniform $\mathcal{H}^1$ format, we take two approaches: the "$\mathcal{H}^1 + \text{unif.}$" approach uses the $\mathcal{H}^1$ sampling procedure followed by a uniformization step (based on [64]), and the "unif. $\mathcal{H}^1$" approach uses the technique described in Section 3.4.2, which uses a different sampling procedure to sample entire interaction lists and avoids the uniformization step.

We measure the accuracy of the compressed matrices using the relative error

$$\frac{\|\tilde{\mathbf{A}} - \mathbf{A}\|}{\|\mathbf{A}\|}$$

computed via 20 iterations of the power method. We also report the maximum leaf node size m and the number r of random vectors per test matrix, which are inputs to the compression algorithm.

3.5.1 Exploiting low-dimensional structure

A major advantage of the graph coloring approach is that it tailors the test matrices to the problem at hand, exploiting low-dimensional structure to minimize the number of matrix-vector products. In Sections 3.4.1 to 3.4.3, we establish upper bounds on the chromatic numbers of the graphs in terms of the dimension d of the computational domain. However, if the geometry of the

points exhibits lower-dimensional structure (e.g., a discretization of a surface in three-dimensional space, a machine learning dataset consisting of observations belonging to a high-dimensional feature space), the chromatic numbers may be much lower.

To demonstrate this effect, we color the graphs that arise from sampling one level of admissible blocks of an $\mathcal{H}^1$ matrix based on a uniform grid along a randomly oriented line through $[0,1]^d$ over a range of dimensions d. We perturb the data by adding Gaussian noise to the coordinates of the points in order to simulate geometries that are not perfectly one-dimensional. With zero noise, the points lie exactly on a line. As more and more noise is added, the effective dimensionality of the data increases from one to the full ambient dimension. The results reported in Fig. 3.9 demonstrate that the number of colors increase modestly in the presence of low-dimensional structure and exponentially in the absence of low-dimensional structure.

3.5.2 Boundary integral equation

We consider a matrix arising from the discretization of the Boundary Integral Equation (BIE)

$$\frac{1}{2}q(\boldsymbol{x}) + \int_{\Gamma} \frac{(\boldsymbol{x}-\boldsymbol{y})\cdot\boldsymbol{n}(\boldsymbol{y})}{4\pi|\boldsymbol{x}-\boldsymbol{y}|^2}\, q(\boldsymbol{y})\, ds(\boldsymbol{y}) = f(\boldsymbol{x}), \qquad \boldsymbol{x}\in\Gamma, \tag{3.13}$$

where Γ is the simple closed contour in the plane shown in Figure 3.10, and where $\boldsymbol{n}(\boldsymbol{y})$ is the outwards pointing unit normal of Γ at $\boldsymbol{y}$. The BIE (3.13) is a standard integral equation formulation of the Laplace equation with boundary

condition f on the domain interior to Γ. The BIE (3.13) is discretized using the Nyström method on N equispaced points on Γ, with the Trapezoidal rule as the quadrature (since the kernel in (3.13) is smooth, the Trapezoidal rule has exponential convergence).

The fast matrix-vector multiplication is in this case furnished by the recursive skeletonization (RS) procedure of [84]. To avoid spurious effects due to the rank structure inherent in RS, we compute the matrix-vector products at close to double precision accuracy, and with an entirely uncorrelated tree structure.

Results are given in Fig. 3.13.

Remark 9. *The problem under consideration here is artificial in the sense that there is no actual need to use more than a couple of hundred points to resolve (3.13) numerically to double precision accuracy. It is included merely to illustrate the asymptotic scaling of the proposed method.*

3.5.3 Operator multiplication

We next investigate how the proposed technique performs on a matrix matrix multiplication problem. Specifically, we determine the Neumann-to-Dirichlet operator T for the contour shown in Figure 3.10 using the well known formula

$$T = S \left(\frac{1}{2} I + D^* \right)^{-1},$$

where S is the single layer operator $[Sq](\boldsymbol{x}) = \int_\Gamma -\frac{1}{2\pi} \log |\boldsymbol{x}-\boldsymbol{y}|\, q(\boldsymbol{y})\, ds(\boldsymbol{y})$, and where D^* is the adjoint of the double-layer operator $[D^* q](\boldsymbol{x}) = \int_\Gamma \frac{\boldsymbol{n}(\boldsymbol{x})\cdot(\boldsymbol{x}-\boldsymbol{y})}{2\pi|\boldsymbol{x}-\boldsymbol{y}|^2}\, q(\boldsymbol{y})\, ds(\boldsymbol{y})$.

The operators S and D are again discretized using a Nyström method on equispaced points (with sixth order Kapur-Rokhlin [53] corrections to handle the singularity in S), resulting in matrices $\mathbf{S}$ and $\mathbf{D}$. The $\mathbf{S}\big(0.5\mathbf{I} + \mathbf{D}^*\big)^{-1}$ is again applied using the recursive skeletonization procedure of [84].

Results are given in Fig. 3.12.

3.5.4 Fast multipole method

We consider a kernel matrix representing N-body Laplace interactions in three dimensions, where the interaction kernel is defined by

$$\mathcal{K}(x, y) = \sum_{i \neq j} \frac{c_j}{\|x_i - x_j\|}$$

for a sets of N points $\{x_i\}$ and charges $\{c_i\}$. To simulate a problem with low-dimensional structure, we distribute the points uniformly at random on the surface of the unit sphere. We use an implementation of the fast multipole method included in the Flatiron Institute Fast Multipole Libraries [6] to efficiently apply the matrix to vectors. Results are given in Fig. 3.14.

3.5.5 Frontal matrices in nested dissection

Our next example is a simple model problem that illustrates the behavior of the proposed method in the context of sparse direct solvers. The idea here is to use rank structure to compress the increasingly large Schur complements that arise in the LU factorization of a sparse matrix arising from the finite element or finite difference discretization of an elliptic PDE, cf. [80, Ch. 21].

As a model problem, we consider an $N \times N$ matrix $\mathbf{C}$ that encodes the stiffness matrix for the standard five-point stencil finite difference approximation to the Poisson equation on a rectangle. We use a grid with $N \times 51$ nodes. We partition the grid into three sets $\{1, 2, 3\}$, as shown in Fig. 3.11, and then tessellate $\mathbf{C}$ accordingly,

$$\mathbf{C} = \begin{bmatrix} \mathbf{C}_{11} & \mathbf{0} & \mathbf{C}_{13} \\ \mathbf{0} & \mathbf{C}_{22} & \mathbf{C}_{23} \\ \mathbf{C}_{31} & \mathbf{C}_{32} & \mathbf{C}_{33} \end{bmatrix}.$$

The matrix we seek to compress is the Schur complement

$$\mathbf{A} = \mathbf{C}_{33} - \mathbf{C}_{31}\mathbf{C}_{11}^{-1}\mathbf{C}_{31} - \mathbf{C}_{32}\mathbf{C}_{22}^{-1}\mathbf{C}_{23}.$$

In our example, we apply $\mathbf{A}$ to vector by calling standard sparse direct solvers for the left and the right subdomains, respectively.

Results are given in Fig. 3.15.

3.5.6 Summary of observations

- The results demonstrate quasilinear scaling of computation time for compressing the operators and for applying the compressed representations to vectors.

- The unif. $\mathcal{H}^1$ compression scheme consistently outperforms the $\mathcal{H}^1 +$ unif. scheme. The advantage appears not only in shorter compression times, but also in higher accuracy and lower storage requirements.

- The approximations achieve high accuracy in every case, with the exception of the FMM, for which the accuracy is on par with the accuracy of

the operator.

- The storage costs, reported as number of floating point numbers per degree of freedom, remains roughly constant for the $\mathcal{H}^2$ format, and grows logarithmically for the other formats.

- In every case, the majority of the compression time is spent in the black-box matrix-vector multiplication routines, highlighting the importance of minimizing the number of matrix-vector products.

3.6 Conclusions

This paper presents algorithms for randomized compression of rank-structured matrices. The algorithms only access the matrix via black-box matrix-vector multiplication routines. We formulate a graph coloring problem to design sets of test matrices that are tailored to the given matrix and to minimize the number of matrix-vector multiplications required. Numerical experiments demonstrate that the algorithms are accurate and much more efficient than prior works, particularly when the underlying geometry exhibits low-dimensional structure.

3.7 Appendix

3.7.1 Far-field sampling with tagged test matrices

In this section, we present an approach for efficiently sampling the entire far field of a box. This method can potentially be used as a replacement for

the first stage of sampling, described in Section 3.4.2, in compressing uniform $\mathcal{H}^1$ and $\mathcal{H}^2$ matrices.

Suppose we are interested in sampling the far field of node 9, shown in Fig. 3.3.

We define random matrices $\mathbf{T} \in \mathbb{R}^{8\times 4}$ and $\mathbf{G} \in \mathbb{R}^{N\times r}$ as

$$\mathbf{T} = \begin{bmatrix} t_{8,1} & t_{8,2} & t_{8,3} & t_{8,4} \\ t_{9,1} & t_{9,2} & t_{9,3} & t_{9,4} \\ \vdots & \vdots & \vdots & \vdots \\ t_{15,1} & t_{15,2} & t_{15,3} & t_{15,4} \end{bmatrix}, \qquad \mathbf{G} = \begin{bmatrix} \mathbf{G}_8 \\ \mathbf{G}_9 \\ \vdots \\ \mathbf{G}_{15} \end{bmatrix},$$

and define

$$\begin{bmatrix} \boldsymbol{\Omega}_1 & \boldsymbol{\Omega}_2 & \boldsymbol{\Omega}_3 & \boldsymbol{\Omega}_4 \end{bmatrix} = \begin{bmatrix} t_{8,1}\mathbf{G}_8 & t_{8,2}\mathbf{G}_8 & t_{8,3}\mathbf{G}_8 & t_{8,4}\mathbf{G}_8 \\ t_{9,1}\mathbf{G}_9 & t_{9,2}\mathbf{G}_9 & t_{9,3}\mathbf{G}_9 & t_{9,4}\mathbf{G}_9 \\ \vdots & \vdots & \vdots & \vdots \\ t_{15,1}\mathbf{G}_{15} & t_{15,2}\mathbf{G}_{15} & t_{15,3}\mathbf{G}_{15} & t_{15,4}\mathbf{G}_{15} \end{bmatrix}.$$

To sample the far field of node 9, we must exclude contributions from its near field, which consists of nodes 8, 9, 10. To that end, we find a unit vector z belonging to the nullspace of the first three rows of $\mathbf{T}$.

$$z = \begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} \in \text{Null}\left(\begin{bmatrix} t_{8,1} & t_{8,2} & t_{8,3} & t_{8,4} \\ t_{9,1} & t_{9,2} & t_{9,3} & t_{9,4} \\ t_{10,1} & t_{10,2} & t_{10,3} & t_{10,4} \end{bmatrix}. \right)$$

Then

$$z_1\boldsymbol{\Omega}_1 + z_2\boldsymbol{\Omega}_2 + z_3\boldsymbol{\Omega}_3 + z_4\boldsymbol{\Omega}_4 = \begin{bmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ (z_1t_{11,1} + z_2t_{11,2} + z_3t_{11,3} + z_4t_{11,4})\mathbf{G}_{11} \\ (z_1t_{12,1} + z_2t_{12,2} + z_3t_{12,3} + z_4t_{12,4})\mathbf{G}_{12} \\ \vdots \\ (z_1t_{15,1} + z_2t_{15,2} + z_3t_{15,3} + z_4t_{15,4})\mathbf{G}_{15} \end{bmatrix},$$

and the desired sample is contained in the rows indexed by I_9 of

$$\mathbf{A}(z_1\mathbf{\Omega}_1 + z_2\mathbf{\Omega}_2 + z_3\mathbf{\Omega}_3 + z_4\mathbf{\Omega}_4) = z_1\mathbf{Y}_1 + z_2\mathbf{Y}_2 + z_3\mathbf{Y}_3 + z_4\mathbf{Y}_4.$$

The test matrix above contains zeros in the appropriate positions to exclude the near-field interactions, and it contains random values in the other positions to sample the far-field interactions. Notably, the random values are not distributed according to a standard normal distribution. One may view the blocks as being Gaussian with variances of varying magnitudes, which effectively apply non-uniform weights to blocks of far-field interactions. While this is a deviation from the standard practice of using standard normal random values, we observe empirically that these deviations are inconsequential, and the nonzero parts of the test matrices behave like standard Gaussian test matrices.

The key observation is that in order to obtain a sample of the far field of box τ of level 3, we only need to take an appropriately chosen linear combination of $\{\mathbf{Y}_i(I_\tau, :)\}_{i=1}^4$. Therefore, we only need to take four sets of samples of $\mathbf{A}$, which we can use to obtain samples of the far fields of all of the boxes on a given level.

More generally, for a problem in d dimensions, we must exclude near-field contributions of up to 3^d boxes. Then to ensure that the nullspace of any 3^d rows of $\mathbf{T}$ is nontrivial, we define $\mathbf{T}$ to have 3^d+1 columns, and we have 3^d+1 corresponding test matrices $\mathbf{\Omega}_i$.

These ideas are still in development, but they show promise. For example, this method has the potential to reduce the worst-case cost of the first stage of sampling from 5^d test matrices (cf.Section 3.4.2.1) to $3^d + 1$. Moreover, the method may be applicable in other contexts where one seeks to obtain samples of a matrix that exclude contributions from some blocks.

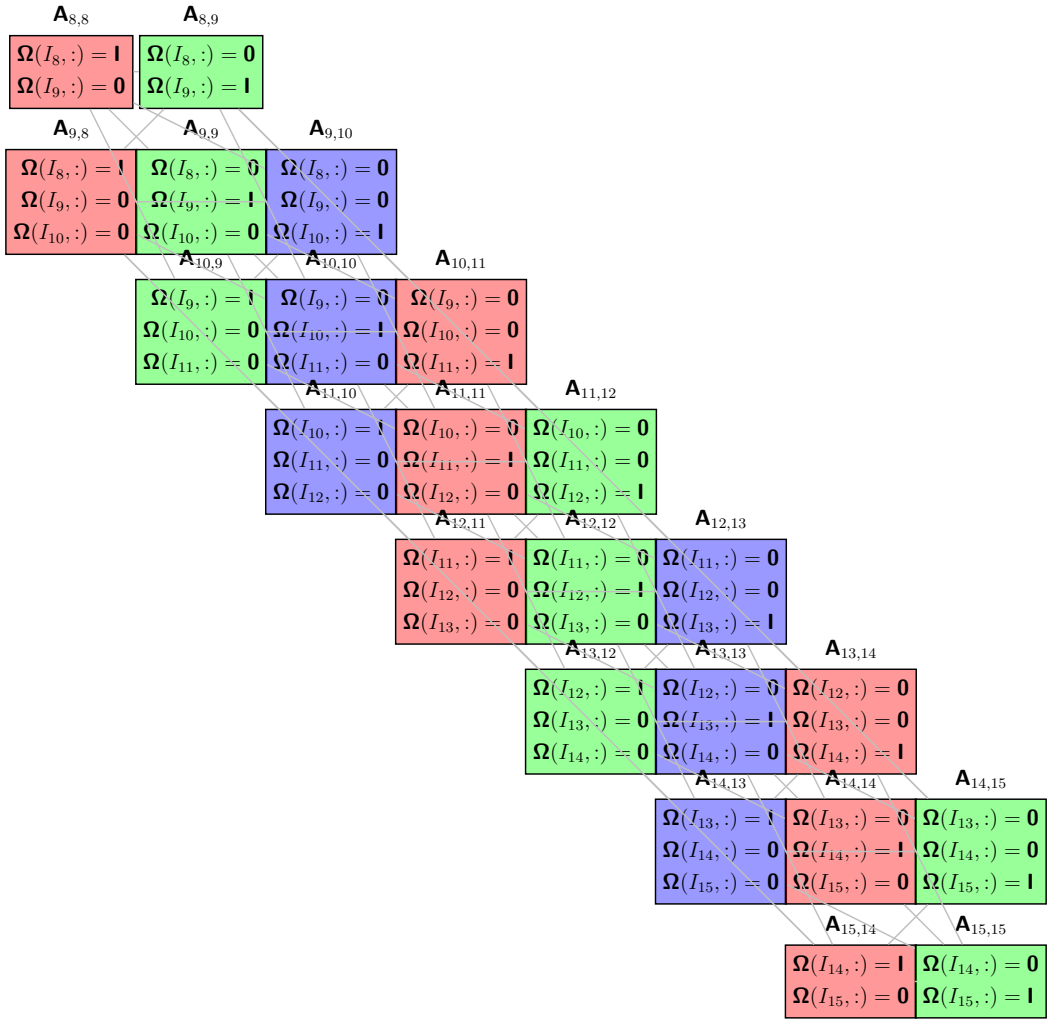


Figure 3.5 *Incompatibility graph for inadmissible blocks belonging to level L.*

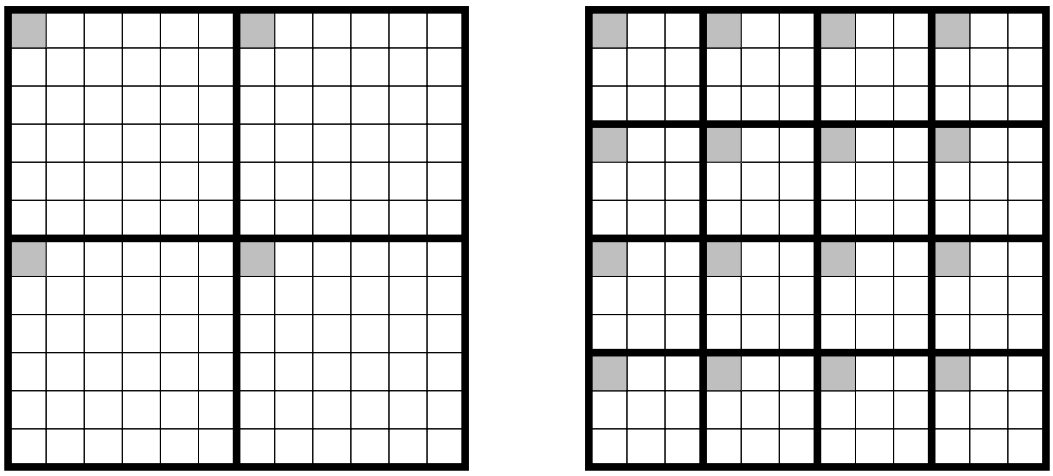


Figure 3.6 *Patterns representing a test matrix for sampling admissible blocks (left) and a test matrix for sampling inadmissible blocks (right) for a problem in 2 dimensions. Blocks of the test matrices corresponding to gray boxes are filled with random values, and those corresponding to white boxes are filled with zeros. The other test matrices are obtained by shifting these pattern horizontally and vertically.*

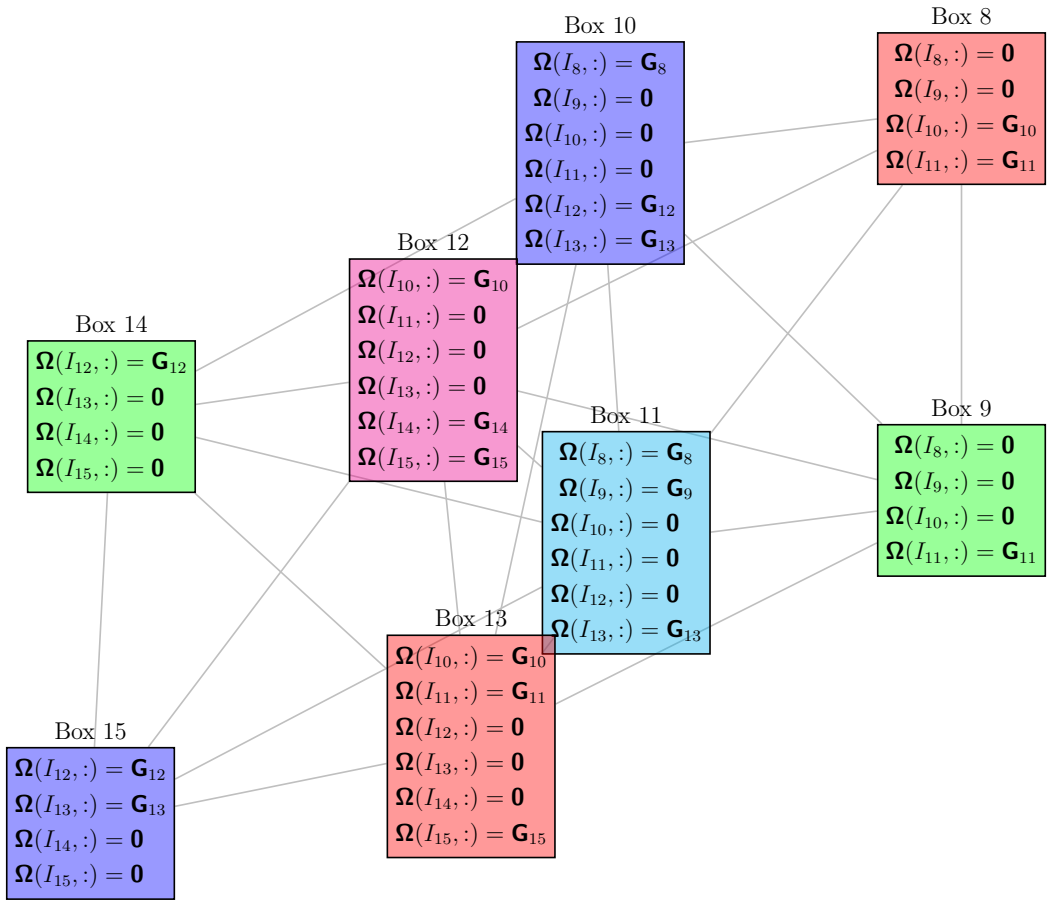


Figure 3.7 *The constraint incompatibility graph corresponding to the 8 boxes on level 3 of the matrix shown in Figure 3.3 along with the constraints* (3.12) *associated with uniform sampling.*

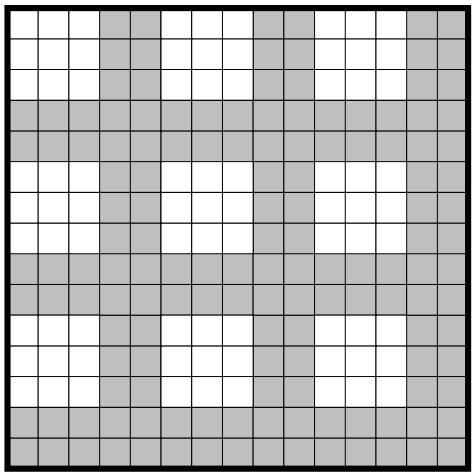


Figure 3.8 *A pattern representing one test matrix for the first stage of uniform* $\mathcal{H}^1$ *sampling for a problem in 2 dimensions. Blocks of the test matrix corresponding to gray boxes are filled with random values, and those corresponding to white boxes are filled with zeros. The other test matrices are obtained by shifting this pattern horizontally and vertically.*

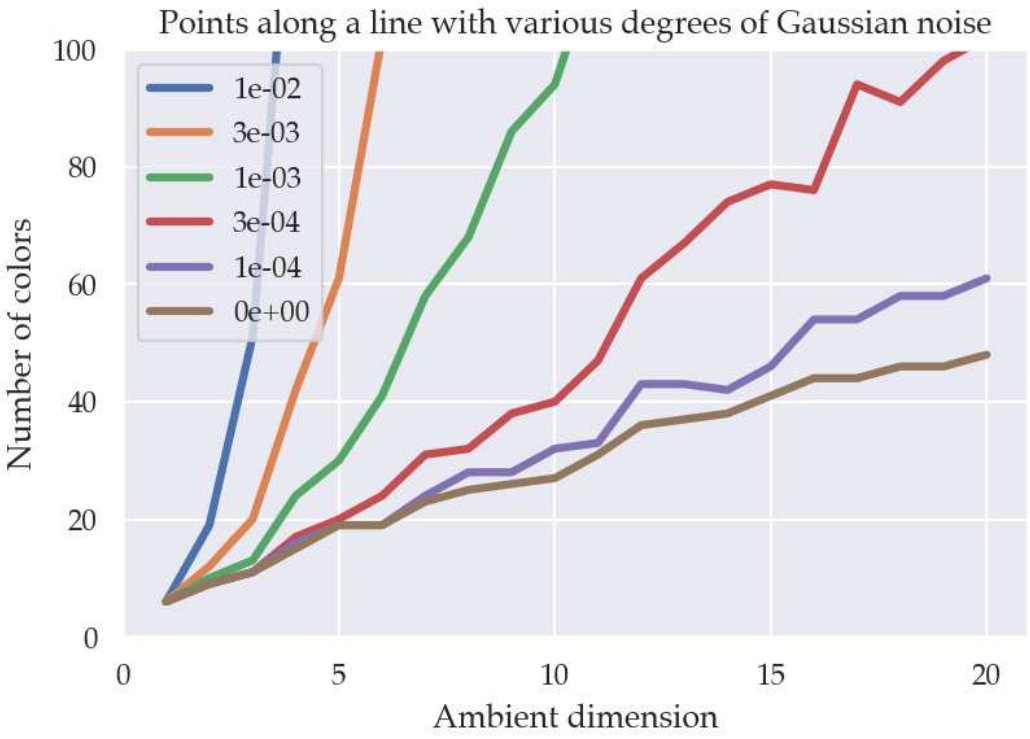


Figure 3.9 *Number of colors for the incompatibility graphs that arise from sampling one level of admissible blocks of an $\mathcal{H}^1$ matrix based on a uniform grid along a randomly oriented line through $[0,1]^d$ with added perturbation over a range of dimensions d.*

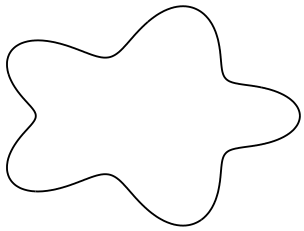


Figure 3.10 *Contour Γ on which the BIE (3.13) is defined.*

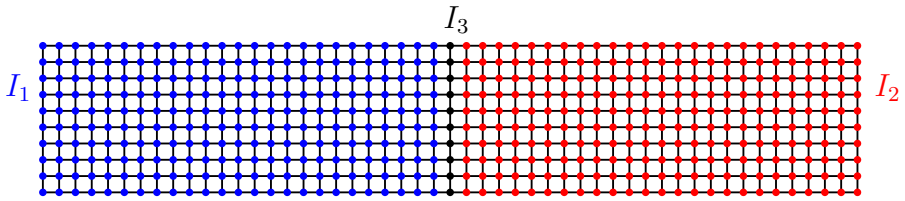


Figure 3.11 *An example of the grid in the sparse LU example described in Section 3.5.5. There are $N \times n$ points in the grid, shown for $N = 10, n = 51$.*

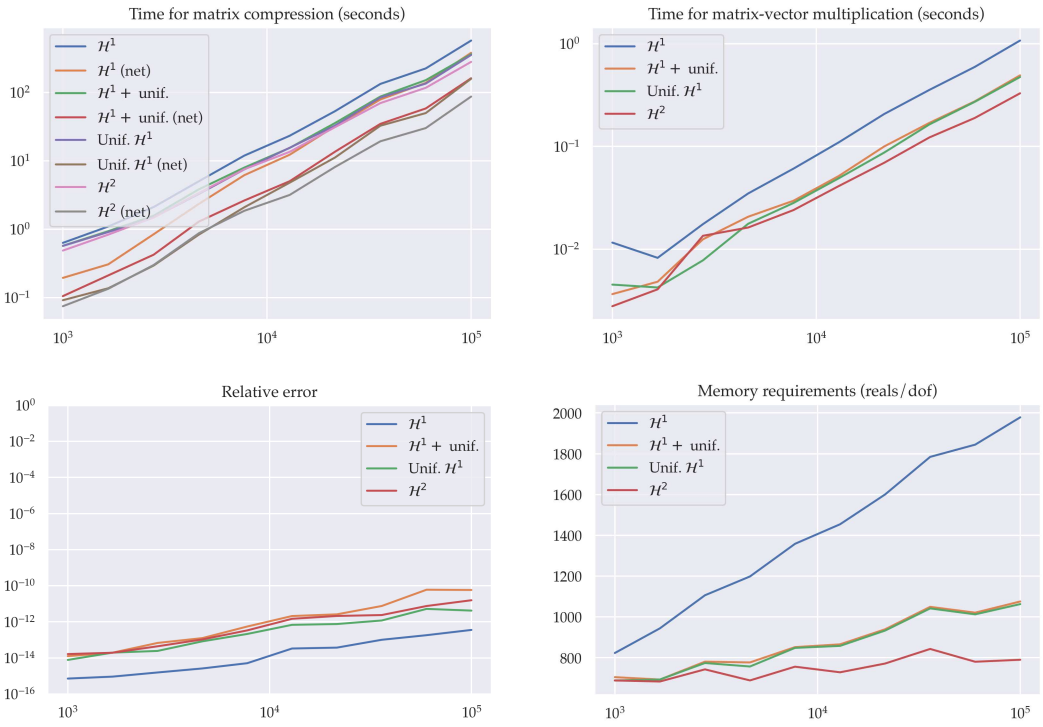


Figure 3.12 *Results from applying peeling algorithms to the Neumann-to-Dirichlet operator. Here $r = 20$ and $m = 200$.*

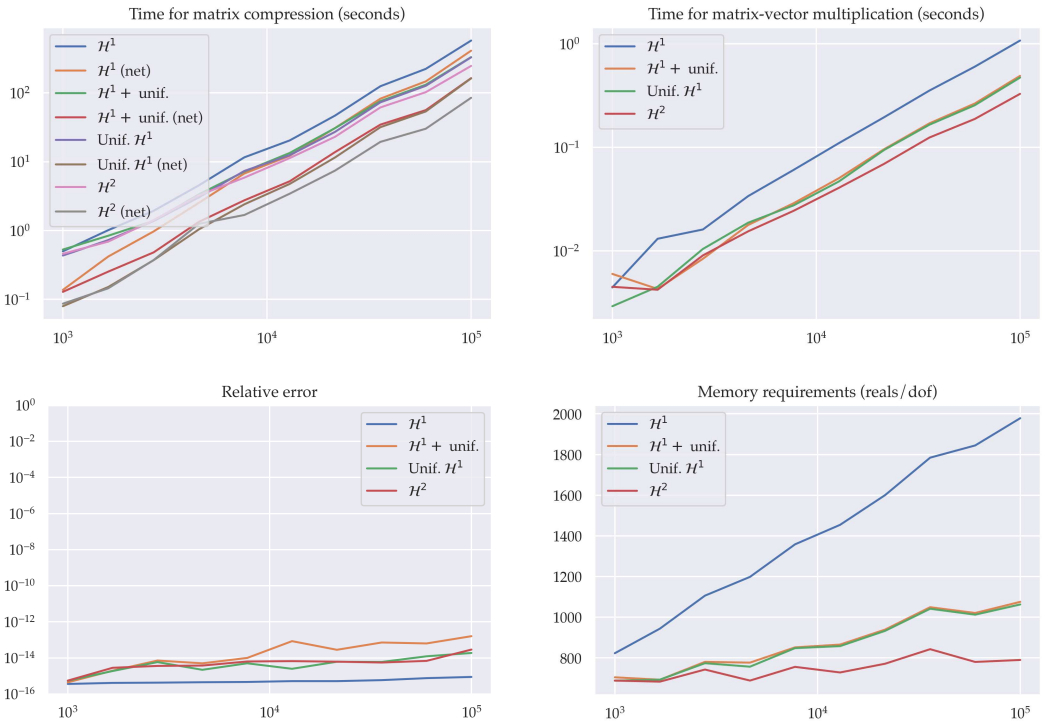


Figure 3.13 *Results from applying peeling algorithms to a double layer potential on a simple contour in the plane. Here* $r = 20$ *and* $m = 200$.

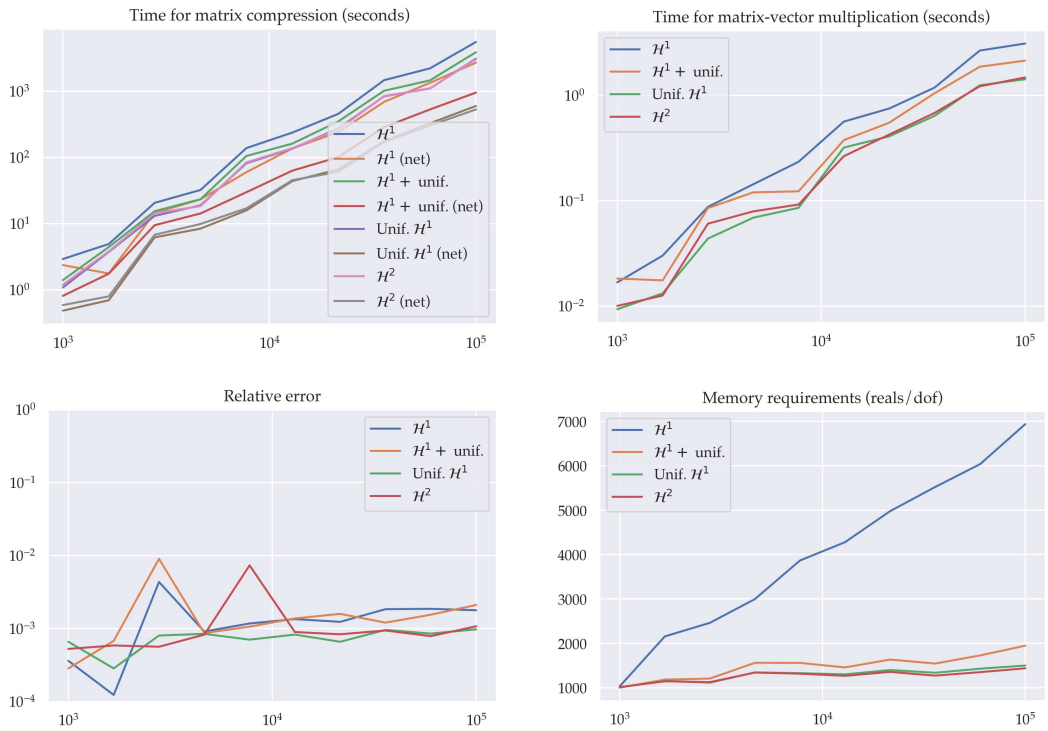


Figure 3.14 *Results from applying peeling algorithms to the 3D fast multipole method operator. Here* $r = 20$ *and* $m = 50$.

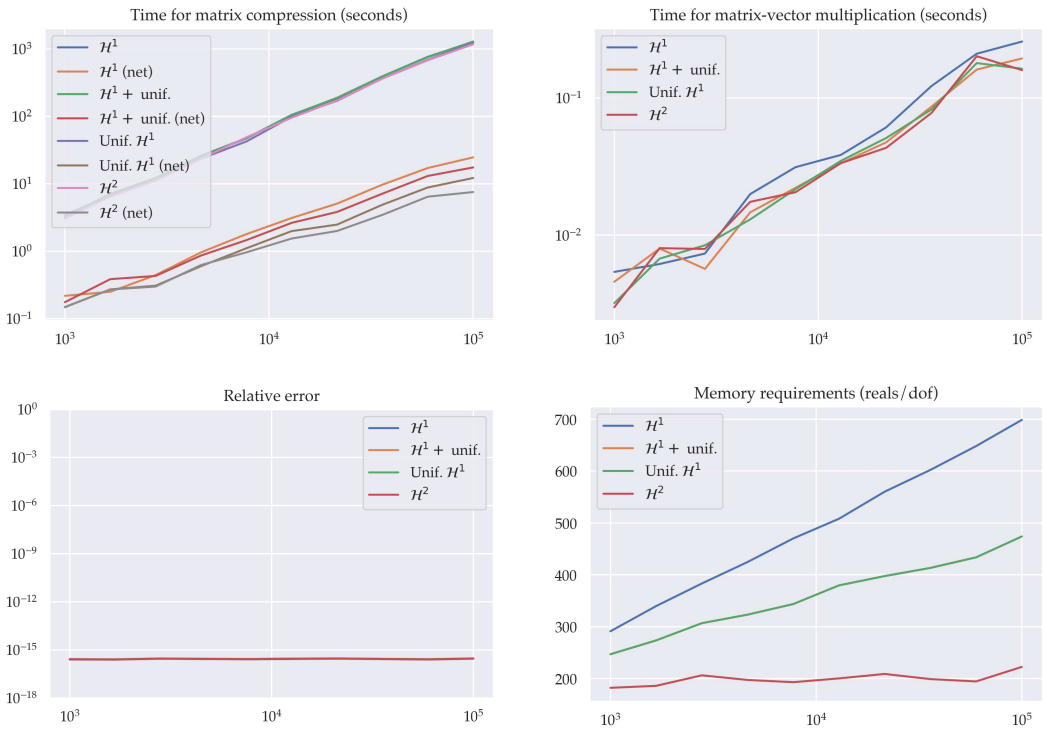


Figure 3.15 *Results from applying peeling algorithms to frontal matrices in the nested dissection algorithm. Here* $r = 10$ *and* $m = 50$.

Part II

Permutation of Rank-Structured Matrices

Chapter 4

Geometry-Oblivious Fast Multipole Method[1]

[1]The content in this chapter was first presented in [107], a work in collaboration with Chenhan Yu, Severin Reiz, and George Biros.

We present GOFMM (geometry-oblivious FMM), a novel method that creates a hierarchical low-rank approximation, or *"compression,"* of an arbitrary dense symmetric positive definite (SPD) matrix. For many applications, GOFMM enables an approximate matrix-vector multiplication in $N \log N$ or even N time, where N is the matrix size. Compression requires $N \log N$ storage and work. In general, our scheme belongs to the family of hierarchical matrix approximation methods. In particular, it generalizes the fast multipole method (FMM) to a purely algebraic setting by only requiring the ability to sample matrix entries. Neither geometric information (i.e., point coordinates) nor knowledge of how the matrix entries have been generated is required, thus the term *"geometry-oblivious."* Also, we introduce a shared-memory parallel scheme for hierarchical matrix computations that reduces synchronization barriers. We present results on the Intel Knights Landing and Haswell architectures, and on the NVIDIA Pascal architecture for a variety of matrices.

4.1 Introduction

We present GOFMM, a novel algorithm for the approximation of dense symmetric positive definite (SPD) matrices. GOFMM can be used for compressing a dense matrix and accelerating matrix-vector multiplication operations. As an example, in Figure 4.1 we report timings for an SGEMM (single-precision matrix-matrix multiplication) operation using an optimized dense matrix library and compare with the GOFMM-compressed version.

Problem statement: Let $K \in \mathbb{R}^{N \times N}$ be a dense SPD matrix, with $K = K^T$ and $x^T K x > 0,\ \forall x \in \mathbb{R}^N,\ x \neq 0$. Since K is dense, it requires $\mathcal{O}(N^2)$ storage and

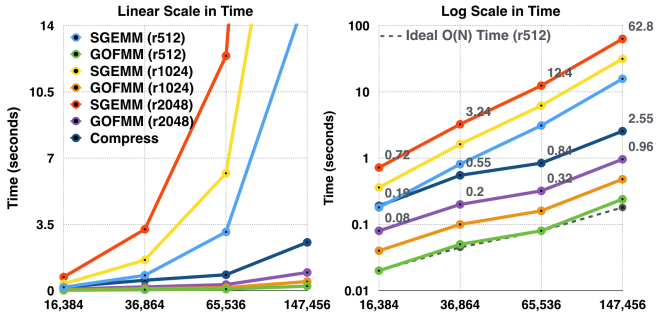


Figure 4.1 *Comparison of* ***runtime in seconds (y-axis)*** *versus* ***problem size*** N ***(x-axis)*** *to multiply test matrix K02 (see §4.3 for details) of size* $N \times N$ *with a matrix of size* $N \times r$*, where* $r = 512, 1024, 2048$*. Results are plotted against a linear scale (left) and a logarithmic scale (right). The top three curves demonstrate* $\mathcal{O}(N^2)$ *scaling of Intel MKL* `SGEMM` *for each value of* r*. The middle curve shows the time for* `GOFMM` *to compress K02, which scales as* $\mathcal{O}(N \log N)$ *in these cases. The bottom three curves show the* $\mathcal{O}(N)$ *scaling of the time for* `GOFMM` *to evaluate the matrix product for each value of* r *after compression is completed. The* `GOFMM` *results reach accuracy of* 1E−2 *to* 4E−4 *in single precision. In these experiments, the crossover problem size (including compression time) is* $N = 16\,384$*, and for* $N = 147\,456$*, we observe an 18× speedup over* `SGEMM`*.*

$\mathcal{O}(N^2)$ work for a matrix-vector multiplication (hereby ***"matvec"***). Assuming the evaluation of a single matrix entry K_{ij} requires $\mathcal{O}(1)$ work, we wish to construct a hierarchical matrix $\tilde{K}$ with the following properties: (1) constructing $\tilde{K}$ requires $\mathcal{O}(N \log N)$ work; (2) a matvec with $\tilde{K}$ also requires $\mathcal{O}(N \log N)$ work; and (3) $\|\tilde{K} - K\| \leq \epsilon \|K\|$, where $0 < \epsilon < 1$ is a user-defined error tolerance. In other words, given any SPD matrix K, our task is to construct a hierarchically low-rank matrix compression $\tilde{K}$ such that the relative error $\|K - \tilde{K}\|/\|K\|$ is small. The only required input to our algorithm is a routine that returns a submatrix K_{IJ}, for arbitrary row and column index sets I and J. For certain matrices, we can achieve these goals with `GOFMM`. Our scheme belongs to the class of hierarchical matrix approximation methods. The constant in the complexity estimate depends on the user-defined

error tolerance, the structure of the underlying matrix, and the `GOFMM` variant. Let us remark and emphasize that our approximation scheme *cannot guarantee both accuracy and work complexity* simultaneously, since an arbitrary SPD matrix may not admit a good hierarchical low-rank matrix approximation (see §4.2).

We say that a matrix $\tilde{K}$ has a ***hierarchically low-rank structure***, i.e., $\tilde{K}$ is an $\mathcal{H}$-matrix [8, 44], if

$$\tilde{K} = D + S + UV, \tag{4.1}$$

where D is ***block-diagonal*** with ***every block being an*** $\mathcal{H}$***-matrix***, U and V are ***low rank***, and S is ***sparse***. At the base case of this recursive definition, the blocks of D are small dense matrices. An $\mathcal{H}$-matrix matvec requires $\mathcal{O}(N \log N)$ work, and the constant in the complexity estimate depends on the rank of U and V. Depending on the construction algorithm, this complexity can go down to $\mathcal{O}(N)$. Although such matrices are rare in real-world applications, it is quite common to find matrices that can be approximated *arbitrarily* well by an $\mathcal{H}$-matrix.

One important observation is that *this hierarchical low-rank structure is not invariant to row and column permutations*. Therefore any algorithm for constructing $\tilde{K}$ must first appropriately permute K before constructing the matrices U, V, D, and S. Existing algorithms rely on the matrix entries K_{ij} being "interactions" (pairwise functions) between ***points*** $\{x_i\}_{i=1}^N$ in $\mathbb{R}^d$ and permute K either by clustering the points (typically using some tree data-structure) or by using graph partitioning techniques (if K is sparse). `GOFMM` requires neither geometric information nor sparsity.

Background and significance Dense SPD matrices appear in scientific computing, statistical inference, and data analytics. They appear in Cholesky and LU factorization [36], in Schur complement matrices for saddle point problems [10], in Hessian operators in optimization [87], in kernel methods for statistical learning [37, 50], and in N-body methods and integral equations [41, 44]. In many applications, the entries of the input matrix K are given by $K_{ij} = \mathcal{K}(x_i, x_j) : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, where $\mathcal{K}$ is a ***kernel function***. Examples of kernel functions are radial basis functions, Green's functions, and angle similarity functions. For such *kernel matrices*, the input is not a matrix, but only the points $\{x_i\}_{i=1}^N$. The points are used to appropriately permute the matrix using spatial data structures. Furthermore, the construction of the sparse correction S uses nearest-neighbor structure of the input points. The low-rank matrices U, V can be either analytically computed using expansions of the kernel function, or semi-algebraically computed using fictitious points (or equivalent points), or using algebraic sampling-based methods that use geometric information. In a nutshell, geometric information is used in all aspects of an $\mathcal{H}$-matrix method.

In many cases however, such points and kernel functions are not available. For example, in dense graphs in data analysis (e.g., social networks, protein interactions). Related matrices include graph Laplacian operators and their inverses. Additional examples include frontal matrices and Schur complements in factorization of sparse matrices; Hessian operators in optimization; and kernel methods in machine learning without points (e.g., word sequences and diffusion on graphs [14, 55]).

Contributions `GOFMM` is inspired by the rich literature of algorithms for matrix

sketching, hierarchical matrices, and fast multipole methods. Its unique feature is that by using only matrix evaluations it generalizes FMM ideas to compressing arbitrary SPD matrices. In more detail, our contributions are summarized below.

- A result from reproducing kernel Hilbert space theory is that any SPD matrix corresponds to a Gram matrix of vectors in some, unknown Gram (or feature) space [50]. Based on this result, the matrix entries are inner products, which we use to define distances. These distances allow us to design an efficient, purely algebraic FMM method.
- The key algorithmic components of `GOFMM` (and other hierarchical matrix and FMM codes) are tree traversals. We test parallel level-by-level traversals, *out-of-order* traversals using `OpenMP`'s advanced task scheduling and an in-house tree-task scheduler. We found that scheduling significantly improves the performance when compared to level-by-level tree traversals. We also use this scheduling to support heterogeneous architectures.
- We conduct extensive experiments to demonstrate the feasibility of the proposed approach. We test our code on 22 different matrices related to machine learning, stencil PDEs, spectral PDEs, inverse problems, and graph Laplacian operators. We perform numerical experiments on Intel Haswell and KNL, Qualcomm ARM, and NVIDIA Pascal architectures. Finally, we compare with three state-of-the-art codes: `HODLR`, `STRUMPACK`, and `ASKIT`.

`GOFMM` also has several additional capabilities. If points and kernel functions (or Green's) function are available, they can be utilized in a similar way to the algebraic

FMM code ASKIT we previously developed [69, 71]. GOFMM currently supports three different measures of distance: geometric point-based (if available), Gram-space ℓ^2 distance, and Gram-space angle distance. GOFMM has support for matvecs with ***multiple right hand sides***, which is useful for Monte-Carlo sampling, optimization, and blocked Krylov methods.

Limitations GOFMM is restricted to SPD matrices. (However, if we are given points, the method becomes similar to existing methods). GOFMM guarantees symmetry of $\tilde{K}$, but if $\|K - \tilde{K}\|/\|K\|$ is large, positive definiteness may be compromised. To reiterate, GOFMM cannot simultaneously guarantee both accuracy and work complexity. This initial implementation of GOFMM supports shared-memory parallelism and accelerators, but not distributed memory architectures. The current version of GOFMM also has several parameters that require manual tuning. Often, the main goal of building $\mathcal{H}$-matrix approximations is to construct a factorization of K, a topic we do not discuss in this paper. Our method requires the ability to evaluate matrix entries and the complexity estimates require that these entries can be computed in $\mathcal{O}(1)$ time. If K is only available through matrix-free interfaces, these assumptions may not be satisfied. Other algorithms, like STRUMPACK, have inherent support for such matrix-free compression.

Related work. The literature on hierarchical matrix methods and fast multipole methods is vast. Our discussion is brief and limited to the most related work.

Low-rank approximations. The most popular approach for compressing arbitrary matrices is a global low-rank approximation using randomized linear algebra. In (4.1), this is equivalent to setting D and S to zero and constructing only U and V. Examples include the CUR [66] factorization, the Nystrom approximation [100], the adaptive cross approximation [9], and randomized rank-revealing factorizations [45,74]. These techniques can also be used for $\mathcal{H}$-matrix approximations when D is not zero. Instead of applying them to K, we can apply them to the off-diagonal blocks of K. FMM-specific techniques that are a mix between analytic and algebraic methods include kernel-independent methods [81, 104] and the black-box FMM [32]. Constructing both U and V accurately and with optimal complexity is hard. The most robust algorithms require $\mathcal{O}(N^2)$ complexity or higher (randomized methods and leverage-score sampling) since they require one to "touch" all the entries of the matrix (or block) to be approximated.

Permuting the matrix. When K is sparse, the method of choice uses graph-partitioning. This doesn't scale to dense matrices because practical graph partitioning algorithms scale at least linearly with the number of edges and thus the construction cost would be at least $\mathcal{O}(N^2)$ [1, 54].

$\mathcal{H}$-matrix methods and software. Treecodes and fast multipole methods originally were developed for N-body problems and integral equations. Algebraic variants led the way to the abstraction of $\mathcal{H}$-matrix methods and the application to the factorization of sparse systems arising from the discretization of elliptic PDEs [2, 8, 39, 44, 49, 101].

Let us briefly summarize the $\mathcal{H}$-matrix classification. Recall the decomposition

$K = D + S + UV$, (4.1). If S is zero the approximation is called a hierarchically off-diagonal low rank (HODLR) scheme. In addition to S being zero, if the $\mathcal{H}$-matrix decomposition of D is used to construct U, V we have a hierarchically semi-separable (HSS) scheme. If S is not zero we have a generic $\mathcal{H}$-matrix; but if the U, V terms are constructed in a nested way then we have an $\mathcal{H}^2$-matrix or an FMM depending on more technical details. HSS and HODLR matrices lead to very efficient approximation algorithms for K^{-1}. However, $\mathcal{H}^2$ and FMM compression schemes better control the maximum rank of the U and V matrices than HODLR and HSS schemes. For the latter, the rank of U and V can grow with N [15] and the complexity bounds are no longer valid. Recently, here have been algorithms to effectively compress FMM and $\mathcal{H}^2$-matrices [25, 105]. One of the most scalable methods is STRUMPACK [33, 78, 90], which constructs an HSS approximation of a square matrix (not necessarily SPD) and then uses it to construct an approximate factorization. For dense matrices STRUMPACK uses the lexicographic ordering. If no fast matrix-vector multiplication is available, STRUMPACK requires $\mathcal{O}(N^2)$ work for compressing a dense SPD matrix, and $\mathcal{O}(N)$ work for the matvec.

4.2 Methods

Given $K \in \mathbb{R}^{N \times N}$, `GOFMM` aims to construct an $\mathcal{H}$-matrix $\widetilde{K}$ in the form of (4.1) such that we can approximate

$$u = Kw \approx \widetilde{K}w, \quad \text{for} \quad w \in \mathbb{R}^N. \tag{4.2}$$

When points $\{x_i\}_{i=1}^N$ are available such that $K_{ij} = \mathcal{K}(x_i, x_j)$, the recursive partitioning on D and the low-rank structure UV use *distances* between x_i and x_j. Existing

METHOD	MATRIX	LOW-RANK	PERM	S
`FMM` [21]	$\mathcal{K}(x_i, x_j)$	EXP	OCTREE	Y
`KIFMM` [104]	$\mathcal{K}(x_i, x_j)$	EQU	OCTREE	Y
`BBFMM` [32]	$\mathcal{K}(x_i, x_j)$	EQU	OCTREE	Y
`HODLR` [4]	K_{ij}	ALG	NONE	N
`STRUMPACK` [90]	K_{ij}	ALG	NONE	N
`ASKIT` [73]	$\mathcal{K}(x_i, x_j)$	ALG	TREE	Y
`MLPACK` [26]	$\mathcal{K}(x_i, x_j)$	EQU	TREE	Y
`GOFMM`	K_{ij}	ALG	TREE	Y

Table 4.1 *We summarize the main features of different $\mathcal{H}$-matrix methods/codes for dense matrices. "**MATRIX**" indicates whether the method requires a kernel function and points—indicated by $\mathcal{K}(x_i, x_j)$—or it just requires kernel entries—indicated by K_{ij}. "**LOW-RANK**" indicates the method used for the off-diagonal low-rank approximations: "EXP" indicates kernel function-dependent analytic expansions; "EQU" indicates the use of equivalent points (restricted to low d problems); "ALG" indicates an algebraic method. "**PERM**" indicates the permutation scheme used for dense matrices: "OCTREE" indicates that the scheme doesn't generalize to high dimensions; "NONE" indicates that the input lexicographic order is used; and "TREE" indicates geometric partitioning that scales to high dimensions. S indicates whether a sparse correction (FMM or $\mathcal{H}^2$) is supported. In §4.4, we present comparisons with `ASKIT`, STRUMPACK, and HODLR.*

FMM methods approximate K_{ij} when x_i and x_j are sufficiently *far* from each other. Otherwise, K_{ij} is not approximated and it is placed either in D or in S. We call this distance-based criterion ***near-far pruning***.

To define such a pruning scheme without $\{x_i\}_{i=1}^N$, we need a notion of distance between two matrix indices i and j. We define such a distance in the next section. With it, we can permute K and define neighbors for each index i. In §4.2.2, we describe a task-based algebraic FMM that only relies on the distance we define. Finally in §4.2.3, we discuss task parallelism and scheduling.

4.2.1 Geometry-oblivious techniques

In this section, we introduce the machinery for using `GOFMM` in a geometry-oblivious manner. Throughout the following discussion, we refer to a set of indices $\mathcal{I} = \{1, \ldots, N\}$, where index i corresponds to the ith row (or column) of the matrix K in the original ordering. Our objective is to find a permutation of $\mathcal{I}$ so that K can be approximated by an $\mathcal{H}$-matrix. The key is to define a distance between a pair of indices $i, j \in \mathcal{I}$, denoted as d_{ij}. Using the distances, we then perform a hierarchical clustering of $\mathcal{I}$, which is used to define the permutation and determine which interactions go into the sparse correction S (using nearest neighbors).

We define three measures of distance including the point-based Euclidean distance (if data points are available), a Gram-space Euclidean distance, and a Gram-space angle distance.

Geometric-ℓ^2. If we are given points $\{x_i\}_{i=1}^N$, then $d_{ij} = \|x_i - x_j\|_2$ is the *geometric ℓ^2 distance*. This will be the *geometry-aware* reference implementation for

cases where points are given.

Gram-ℓ^2 (or *"kernel"* distance). Since K is SPD, it is the *Gram matrix* of some set of ***unknown Gram vectors***, $\{\phi_i\}_{i=1}^N \subset \mathbb{R}^N$ ([92], proposition 2.16, page 44). That is, $K_{ij} = (\phi_i, \phi_j)$, where $(\cdot, \cdot)$ denotes the ℓ^2 inner product in $\mathbb{R}^N$. We define the *Gram ℓ^2 distance* as $d_{ij} = \|\phi_i - \phi_j\|_2$. Computing the kernel distance only requires three entries of K:

$$d_{ij}^2 = \|\phi_i\|^2 + \|\phi_j\|^2 - 2(\phi_i, \phi_j) = K_{ii} + K_{jj} - 2K_{ij}. \tag{4.3}$$

Gram angles (or *"angle"* distance). Our third measure of distance considers angles between Gram vectors, which is based on the standard sine distance (cosine similarity) in inner product spaces. We define the Gram angle distance as $d_{ij} = \sin^2(\angle(\phi_i, \phi_j)) \in [0, 1]$. This expression is chosen so that d_{ij} is small for nearly collinear Gram vectors, large for nearly orthogonal Gram vectors, and d_{ij} is inexpensive to compute. Although the value d_{ij} may seem arbitrary, we only compare values for the purpose of ordering, so any equivalent metric will do. Computing an angle distance only requires three entries of K:

$$d_{ij} = 1 - \cos^2(\angle(\phi_i, \phi_j)) = 1 - K_{ij}^2/(K_{ii}K_{jj}). \tag{4.4}$$

To reiterate for emphasis, d_{ij} define proper distances (metrics) because K is SPD. And with distances, we can apply FMM.

Tree partitioning and nearest neighbor searches. K is permuted using a balanced binary tree. The root node is assigned with the full set of points, and the tree is constructed recursively by splitting a node's points evenly between two

Algorithm 4.2.1 $[\mathtt{l}, \mathtt{r}] = \mathtt{metricSplit}(\alpha)$

$p = \text{argmax}\,(\{d_{ic} | i \in \alpha\})$; $q = \text{argmax}(\{d_{ip} | i \in \alpha\})$;
$[\mathtt{l}, \mathtt{r}] = \text{medianSplit}(\{d_{ip} - d_{iq} | i \in \alpha\})$;

child nodes according to the pairwise distance metric d_{ij}. The splitting terminates at nodes with some pre-determined ***leaf size*** m. The leaf nodes then define a partial ordering of the indices: if leaf α is anywhere to the left of leaf β, then the indices of α precede those of β. We use this ordering to permute rows and columns of K. In the remainder of this paper, we use the notation α, β to refer interchangeably to a node or the set of indices belonging to the node.

In our implementation, we use a metric ball tree [73], which splits data points according to their pairwise distances. For geometric distances, the tree construction costs $\mathcal{O}(N \log N)$. But Gram distances (**kernel** and **angle**) require sampling to avoid $\mathcal{O}(N^2)$ costs. Suppose we use one of the Gram distances to split an interior node α between its left child `l` and right child `r`. We define $c = \frac{1}{n_c} \sum \phi_i$ to be an approximate centroid[2] taken over a small sample of n_c Gram vectors belonging to α. n_c is $\mathcal{O}(1)$. Next, we find the point p that is farthest away in distance from c, and the point q that is farthest away from p. Then we split the indices $i \in \alpha$ on the values $d_{ip} - d_{iq}$, which measures the degree to which i is closer to p than to q. This approach is outlined in Algorithm 4.2.1.

We perform *all nearest neighbors* (`ANN`) search using randomized trees that are constructed in exactly the same way as the metric partitioning tree, except that

[2] Computing the true centroid over all data points would result in $\mathcal{O}(N^2)$ work.

p and q are chosen randomly. The search algorithm is described in [103] and (briefly) in the next section.

4.2.2 Algebraic Fast Multipole Method

$\mathcal{H}$-matrix methods (including algebraic FMM) have two phases: ***compression*** and ***evaluation***. As we discussed in the introduction, K is compressed recursively using a binary tree such that

$$\widetilde{K}_{\alpha\alpha} = \begin{bmatrix} \widetilde{K}_{\mathtt{ll}} & 0 \\ 0 & \widetilde{K}_{\mathtt{rr}} \end{bmatrix} + \begin{bmatrix} 0 & S_{\mathtt{lr}} \\ S_{\mathtt{rl}} & 0 \end{bmatrix} + \begin{bmatrix} 0 & UV_{\mathtt{lr}} \\ UV_{\mathtt{rl}} & 0 \end{bmatrix}, \tag{4.5}$$

where `l` and `r` are ***left*** and ***right child*** of the treenode α. Each node α contains a set of matrix indices and the two children evenly split the indices such that $\alpha = \mathtt{l} \cup \mathtt{r}$. (We overload the notation α, β, `l` and `r` to denote the matrix indices that those treenode own.) In Fig. 4.2, the blue blocks depict S (at all levels) and D (in the leaf level), and the pink blocks depict the UV matrices.

We use ***four tree traversals*** to describe the algorithms in `GOFMM`: postorder (**POST**), preorder (**PRE**), any order (**ANY**), and any order-leaves only (**LEAF**). By ***"task"*** we refer to a computation that occurs when we visit a tree node during a traversal. We list all tasks required by the ***compression*** phase (Algorithm 4.2.2) and ***evaluation*** phase (Algorithm 4.2.7) in Table 4.2.

`GOFMM` **compression** starts by creating the binary metric ball tree in Algorithm 4.2.2 that represents the binary partitioning (and encodes a symmetric permutation of matrix K). This requires the *distance* metric d_{ij} and a preorder traversal (**PRE**) of the first task `SPLI(`α`)` in Table 4.2.

Task	Operations	FLOPS
SPLI(α)	split α into l and r Algorithm 4.2.1	$\lvert\alpha\rvert$
ANN(α)	update $\mathcal{N}_\alpha$ with KNN($K_{\alpha\alpha}$)	m^2
SKEL(α)	$\widetilde{\alpha}$ in Algorithm 4.2.6	$2s^3 + 2m^3$
COEF(α)	$P_{\widetilde{\alpha}\alpha}$ or $P_{\widetilde{\alpha}[\widetilde{\mathtt{lr}}]}$ in Algorithm 4.2.6	s^3
N2S(α)	**if** α is leaf **then** $\widetilde{w}_\alpha = P_{\widetilde{\alpha}\alpha} w_\alpha$ **else** $\widetilde{w}_\alpha = P_{\widetilde{\alpha}[\widetilde{\mathtt{lr}}]}[\widetilde{w}_\mathtt{l}; \widetilde{w}_\mathtt{r}]$	$2msr$ $2s^2r$
SKba(β)	$\forall \alpha \in$ Far(β), $K_{\widetilde{\beta}\widetilde{\alpha}} = K(\widetilde{\beta}, \widetilde{\alpha})$	$ds^2\lvert$Far(β)$\rvert$
S2S(β)	$\widetilde{u}_\beta = \sum_{\alpha \in \mathtt{Far}(\beta)} K_{\widetilde{\beta}\widetilde{\alpha}} \widetilde{w}_\alpha$	$2s^2r\lvert$Far(β)$\rvert$
S2N(β)	**if** α is leaf **then** $u_\beta = P^T_{\widetilde{\beta}\beta} \widetilde{u}_\beta$ **else** $[\widetilde{u}_\mathtt{l}; \widetilde{u}_\mathtt{r}] += P^T_{\widetilde{\beta}[\widetilde{\mathtt{lr}}]} \widetilde{u}_\beta$	$2msr$ $2s^2r$
Kba(β)	$\forall \alpha \in$ Near(β), $K_{\beta\alpha} = K(\beta, \alpha)$	$m^2\lvert$Near(β)$\rvert$
L2L(β)	$u_\beta += \sum_{\alpha \in \mathtt{Near}(\beta)} K_{\beta\alpha} w_\alpha$	$2m^2r\lvert$Near(β)$\rvert$

Table 4.2 *Tasks and their costs in FLOPS. SPLI (tree splitting), ANN (all nearest-neighbors), SKEL (skeletonization), COEF (interpolation) SKba and Kba (caching submatrices) occur in the compression phase. Interactions N2S (nodes to skeletons), S2S (skeletons to skeletons), S2N (skeletons to nodes), and L2L (leaves to leaves) occur in the evaluation phase.*

Algorithm 4.2.2 Compress(K)

1: **for each** randomized tree **do** # iterative neighbor search
2: (**PRE**) SPLI(α) # create a random projection tree
3: (**LEAF**) ANN(α) # search κ neighbors in leaf nodes
4: (**PRE**) SPLI(α) # create a metric ball tree
5: (**LEAF**) LeafNear(β) # build Near(β) using $\mathcal{N}(\beta)$
6: (**LEAF**) FindFar(β,root) # find Far(β) using MortonID
7: (**POST**) MergeFar(α) # merge Far(l), Far(r) to Far(α)
8: (**POST**) SKEL(α) # compute skeletons $\widetilde{\alpha}$
9: (**ANY**) COEF(α) # compute the coefficient matrix P
10: (**ANY**) Kba(β) # optionally evaluate and cache $K_{\beta\alpha}$
11: (**ANY**) SKba(β) # optionally evaluate and cache $K_{\widetilde{\beta}\widetilde{\alpha}}$

Node lists and near-far pruning. GOFMM tasks require that every tree node maintains three lists. For a node α, these lists are the neighbor list $\mathcal{N}(\alpha)$,

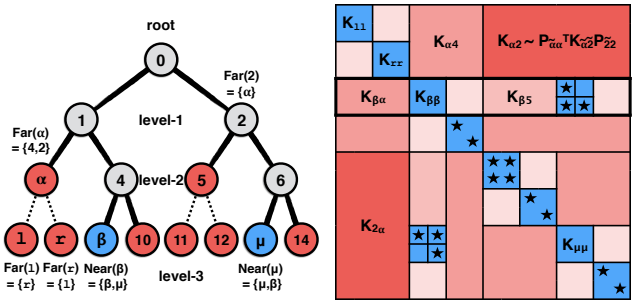


Figure 4.2 *A partitioning tree (left) and corresponding hierarchically low-rank plus sparse matrix (right). The off-diagonal blocks are combinations of low-rank matrices (pink) and sparse matrices (blue). The ★ symbol denotes an entry that cannot be approximated (because the corresponding interaction is between neighbors). The solid edges in the tree mark the path traversed by* `FindFar`$(\beta,0)$. *Since* $K_{\beta\alpha}$ *does not contain any neighbor interactions (★), this traversal adds* α *to* `Far`(β). *In this example,* `FindFar(l,0)` *computes* `Far`$(\mathtt{l}) = \{\mathtt{r}, 4, 2\}$, *and* `FindFar(r,0)` *computes* `Far`$(\mathtt{r}) = \{\mathtt{l}, 4, 2\}$. *Algorithm 4.2.5 (*`MergeFar`*) then moves* `Far`$(\mathtt{l}) \cap Far(\mathtt{r})$ *into* `Far`(α) *so that* `Far`$(\alpha) = \{4, 2\}$, `Far`$(\mathtt{l}) = \{\mathtt{r}\}$ *and* `Far`$(\mathtt{r}) = \{\mathtt{l}\}$.

Algorithm 4.2.3 `LeafNear`(β)

`Near`$(\beta) = \{$`MortonID`$(i) : \forall i \in \mathcal{N}(\beta)\}$

Algorithm 4.2.4 `FindFar`$(\beta =$ `leaf`$, \alpha)$

---

if $\alpha \cap Near(\beta) \neq \phi$ using `MortonID` **then**
  `FindFar`$(\beta$,`l`$)$; `FindFar`$(\beta$,`r`$)$;
else `Far`$(\beta) = Far(\beta) \cup \alpha$;

Algorithm 4.2.5 `MergeFar`(α)

`MergeFar`(`l`); `MergeFar`(`r`);
`Far`$(\alpha) = Far($`l`$) \cap Far($`r`$)$;
`Far`$($`l`$) = Far($`l`$)\backslash Far(\alpha)$; `Far`$($`r`$) = Far($`r`$)\backslash Far(\alpha)$;

---

near interaction list `Near`(α), and far interaction list `Far`(α). Computing these lists requires defining neighbors for indices based on the distance d_{ij} and the Morton ID.

A pair of nodes α and β is said to be *far* if submatrix $K_{\beta\alpha}$ is low-rank

and *near* otherwise. We use neighbor-based pruning [73] to determine the near-far relation. Neighbors are defined based on the specified distance d_{ij}. Recall that we have already defined three different distance metrics in Section 4.2.1. For each i, we search for the κ indices j that result in the smallest d_{ij}. The Morton ID is a bit array that codes the path from the root to a tree node or index i. The Morton ID of an index i is the Morton ID of the leaf node (in `GOFMM` ball metric tree) that contains it. We use `MortonID()` to denote this.

Node neighbor list $\mathcal{N}(\alpha)$: As we discussed, `GOFMM` requires a preprocessing step in which we compute the nearest neighbors for each tree node α. We first construct a list of κ nearest-neighbor for each index $i \in \alpha$ iteratively using a greedy search (steps 1–3 in Algorithm 4.2.2). Then we construct the neighbor list $\mathcal{N}(\alpha)$ by merging all neighbors of $i \in \alpha$. For non-leaf nodes the list is constructed recursively [69].

In each iteration, we create a randomized projection tree [27, 62, 73], and we search for neighbors of i only in the leaf node α that contains i using an exhaustive search [106]. That is, for each $i \in \alpha$, we only search for small d_{ij} where $j \in \alpha$ as well. Due to the randomness, in each iteration leaf node α may be assigned with a different partition, which gradually cover all neighbors during the local exhaustive search. To get a set of approximate neighbors, the iteration stops after reaching 80% accuracy or 10 iterations.

80% and 10 iterations are chosen empirically. In our experiments, we found that more accurate nearest neighbors do not improve the `GOFMM` approximation. More specifically, nearest-neighbors provide a guess of important matrix entries, which

are used in importance sampling and selecting near interactions. In our previous work [73], we show that whether neighbors can improve the accuracy depends on the data (points or Gram vectors) in different scales. Typically, neighbor-pruning works better if the intrinsic dimensionality of the data is low. Consequently, random projection based ANN methods also converge faster [103]. Otherwise, it is likely that neighbors will not improve the accuracy too much. Typically even smaller values are sufficient. We use 80% to be conservative.

Near list of a node $\texttt{Near}(\alpha)$: Leaf nodes α, β are considered near if $\alpha \cap \mathcal{N}(\beta)$ is nonempty (i.e., $K_{\alpha\beta}$ contains at least one neighbor ($\bigstar$) in Fig. 4.2). The `Near` interaction list is defined only for leaf nodes and contains only leaf nodes. For each leaf node β, $\texttt{Near}(\beta)$ is constructed using `LeafNear` (Algorithm 4.2.3). For each neighbor $i \in \mathcal{N}(\beta)$, `LeafNear(`β`)` adds $\texttt{MortonID}(i)$ to $\texttt{Near}(\beta)$. Notice that the size of $\texttt{Near}(\beta)$ determines the number of direct evaluations (blue blocks in Fig. 4.2) in the off-diagonal blocks. To prevent the cost from growing too fast, we introduce a user-defined parameter **budget** such that

$$|\texttt{Near}(\beta)| < \text{budget} \times (N/m). \tag{4.6}$$

While looping over neighbor $i \in \mathcal{N}(\beta)$, instead of directly adding $\texttt{MortonID}(i)$ to $\texttt{Near}(\beta)$, we only mark it with a ballot. Then we insert candidates to $\texttt{Near}(\beta)$ according to their votes until (4.6) is reached. To enforce symmetry of $\tilde{K}$, we loop over all `Near` lists and enforce the following: if $\alpha \in \texttt{Near}(\beta)$ then $\beta \in \texttt{Near}(\alpha)$.

Far list of a node $\texttt{Far}(\alpha)$: $\texttt{Far}(\alpha)$ is constructed in two steps in Algorithm 4.2.2, representing submatrices in the off-diagonal blocks that can be approxi-

mated. First for each leaf node β, we invoke `FindFar`(β, `root`) (Algorithm 4.2.4). Upon visiting α, we check whether α is a parent of any leaf node in `Near`(β) using `MortonID`. If so, we recur to the two children of α; otherwise, we add α to `Far`(β) (i.e. $\mathcal{K}_{\beta\alpha}$ can be approximated). The second step is a postorder traversal on `MergeFar(root)` (Algorithm 4.2.5). This process merges the common nodes from two children lists `Far(l)` and `Far(r)` to create larger off-diagonal blocks for approximation. These common nodes are removed from the children and added to their parent list `Far`(α). In Fig. 4.2, `FindFar` can be identified by the smallest square pink blocks, and `MergeFar` merges small pink blocks into larger blocks.

Low-rank approximation. We approximate off-diagonal matrix blocks with a nested interpolative decomposition (ID) [45]. Let β be the indices in a leaf node and $I = \{1, ..., N\} \backslash \beta$ be the set complement. The *skeletonization* of β is a rank-s approximation of its off-diagonal blocks $K_{I\beta}$ using the ID, which we write as

$$K_{I\beta} \approx K_{I\widetilde{\beta}} P_{\widetilde{\beta}\beta}, \tag{4.7}$$

where $\widetilde{\beta} \subset \beta$ is the *skeleton* of β. $K_{I\widetilde{\beta}} \in \mathbb{R}^{(N-|\beta|)\times s}$ is a column submatrix of $K_{I\beta}$, and $P_{\widetilde{\beta}\beta} \in \mathbb{R}^{s\times|\beta|}$ is a matrix of interpolation coefficients, where s is the approximation rank.

To efficiently compute this approximation, we select a sample subset $I' \subset I$ using neighbor-based importance sampling [73]. We then perform a rank-revealing QR factorization (`GEQP3`) on $K_{I'\beta}$. The skeletons $\widetilde{\beta}$ are selected to be the first s pivots, and the matrix $P_{\widetilde{\beta}\beta}$ is computed by a triangular solve (`TRSM`) using the triangular factor R. The rank s is chosen adaptively such that $\sigma_{s+1}(K_{I'\beta}) < \tau$, where

Algorithm 4.2.6 $[\widetilde{\alpha}, P_{\widetilde{\alpha}\alpha}]$=`Skeleton`$(\alpha)$

if α is leaf **then return** $[\widetilde{\alpha}, P_{\widetilde{\alpha}\alpha}] =$ `ID`(α);
$[\widetilde{\mathtt{l}},] =$ `Skeleton`$(\mathtt{l})$; $[\widetilde{\mathtt{r}},] =$ `Skeleton`$(\mathtt{r})$;
return $[\widetilde{\alpha}, P_{\widetilde{\alpha}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]}] =$ `ID`$([\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}])$;

$\sigma_{s+1}(K_{I'\beta})$ is the estimated $s+1$ singular value and τ is related to a user-specified error tolerance.

For an internal node α, we form the skeletonization in the same way, except that the columns are also sampled using the skeletons of the children of α. That is, the ID is computed for $K_{I'[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]}$, where $[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}] = \widetilde{\mathtt{l}} \cup \widetilde{\mathtt{r}}$ contains the skeletons of the children of α:

$$K_{I[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]} \approx K_{I\widetilde{\alpha}} P_{\widetilde{\alpha}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]}. \tag{4.8}$$

This way, the skeletons are nested: $\widetilde{\alpha} \subset \widetilde{\mathtt{l}} \cup \widetilde{\mathtt{r}}$.

As a consequence of the nesting property, we can use $P_{\widetilde{\mathtt{l}}\mathtt{l}}$ and $P_{\widetilde{\mathtt{r}}\mathtt{r}}$ to construct an approximation of the full block $K_{I\alpha}$:

$$K_{I\alpha} \approx K_{I[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]} \begin{bmatrix} P_{\widetilde{\mathtt{l}}\mathtt{l}} & \\ & P_{\widetilde{\mathtt{r}}\mathtt{r}} \end{bmatrix} \approx K_{I\widetilde{\alpha}} P_{\widetilde{\alpha}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]} \begin{bmatrix} P_{\widetilde{\mathtt{l}}\mathtt{l}} & \\ & P_{\widetilde{\mathtt{r}}\mathtt{r}} \end{bmatrix}. \tag{4.9}$$

Then we have a *telescoping* expression for the full coefficient matrix:

$$P_{\widetilde{\alpha}\alpha} = P_{\widetilde{\alpha}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]} \begin{bmatrix} P_{\widetilde{\mathtt{l}}\mathtt{l}} & \\ & P_{\widetilde{\mathtt{r}}\mathtt{r}} \end{bmatrix}. \tag{4.10}$$

We never explicitly form $P_{\widetilde{\alpha}\alpha}$, but instead use the telescoping expression during evaluation.

Algorithm 4.2.6 computes the skeletonization for all tree nodes with a postorder traversal. There are two tasks for each tree node α listed in Table 4.2: (1)

`SKEL(`α`)` selects $\widetilde{\alpha}$ (in the critical path) and (2) `COEF(`α`)` computes $P_{\widetilde{\alpha}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]}$. Notice that in Algorithm 4.2.2 only `SKEL(`α`)` needs to be executed in postorder (**POST**), but `COEF(`α`)` can be in any order (**ANY**) as long as `SKEL(`α`)` is finished. Such parallelism can only be specified at the task level, which later inspires our task-based parallelism in Section 4.2.3. At the end of the compression, we can optionally evaluate and cache all $K_{\beta\alpha}$ in `Near(`β`)` and all $K_{\widetilde{\beta}\widetilde{\alpha}}$ in `Far(`β`)` by executing `Kba(`β`)` and `SKba(`β`)` in any order. Given enough memory (at least $\mathcal{O}(N)$ for all $K_{\beta\alpha}$ and $\mathcal{K}_{\widetilde{\beta}\widetilde{\alpha}}$), caching can reduce the time spent on evaluating and gathering submatrices.

Evaluation. Following [69], we present Algorithm 4.2.7 a four-step process for computing (4.2). The idea is to approximate each **matvec** $u_\beta += K_{\beta\alpha}w_\alpha$ in `Far(`β`)` using a two-sided ID to accumulate $P_{\widetilde{\beta}\beta}^T K_{\widetilde{\beta}\widetilde{\alpha}} P_{\widetilde{\alpha}\alpha} w_\alpha$, where $P_{\widetilde{\alpha}\alpha}, P_{\widetilde{\beta}\beta}$ are given by the telescoping expression (4.10). For more details, see [69].

Algorithm 4.2.7 `Evaluate(`u,w`)`

1: (**POST**) `N2S(`α`)` # compute skeleton weights $\widetilde{w}$
2: (**ANY**) `S2S(`β`)` # apply skeleton basis $K_{\widetilde{\beta}\widetilde{\alpha}}$
3: (**PRE**) `S2N(`β`)` # accumulate skeleton potentials $\widetilde{u}$
4: (**ANY**) `L2L(`β`)` # accumulate direct **matvec** to u

The first step is to perform a postorder traversal (**POST**) on `N2S(`α`)` (Nodes To Skeletons). This computes the *skeleton weights* $\widetilde{w}_\alpha = P_{\widetilde{\alpha}\alpha} w_\alpha$ for each leaf node, and $\widetilde{w}_\alpha = P_{\widetilde{\alpha}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]}[\widetilde{w}_{\mathtt{l}}; \widetilde{w}_{\mathtt{r}}]$ for each inner node. Recall that in `COEF(`α`)`, we have computed $P_{\widetilde{\alpha}\alpha}$ for each leaf node and $P_{\widetilde{\alpha}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]}$ for each internal node. `S2S(`β`)` (Skeletons to Skeletons) applies the *skeleton basis* $K_{\widetilde{\beta}\widetilde{\alpha}}$ and accumulates *skeleton potentials* $\widetilde{u}$ for each node: $\widetilde{u}_\beta = \sum_{\alpha \in Far(\beta)} K_{\widetilde{\beta}\widetilde{\alpha}} \widetilde{w}_\alpha$. As soon as $\widetilde{w}_\alpha$ are computed in `N2S`, `S2S` can be executed in any order. `S2N(`β`)` (Skeletons To Nodes) performs interpolation

on the left and accumulates $\widetilde{u}$ with a preorder traversal. This uses the transpose of (4.10). For each node β, we accumulate $[\widetilde{u}_{\mathtt{l}}; \widetilde{u}_{\mathtt{r}}] += P^T_{\widetilde{\beta}[\widetilde{\mathtt{lr}}]} \widetilde{u}_\beta$ to its children. In the leaf node, $u_\beta = P^T_{\widetilde{\beta}\beta} \widetilde{u}_\beta$ directly accumulates to the output. These three tasks compute all ***matvec*** for the *far* nodes (pink blocks in Fig. 4.2). All ***matvec*** on $K_{\beta\alpha}$ in $\texttt{Near}(\beta)$ (blue blocks) are computed by `L2L(`β`)` (Leaves To Leaves) and directly accumulated to u_β.

Complexity. The worst case compression cost in Algorithm 4.2.7 is $\mathcal{O}(N^2)$, when $|\texttt{Near}(\alpha)| = (N/m)$ for all α. The best case occurs when each $\texttt{Near}(\alpha)$ only contains α itself. We fix the rank s and leaf size m. The tree has $\mathcal{O}(N/m)$ leaf nodes and $\mathcal{O}(N/m)$ interior nodes, so in the best case, overall `N2S` has $\mathcal{O}(2ms(N/m) + 2s^2(N/m))$ work, `S2S` has $\mathcal{O}(2s^2(N/m))$ work, `S2N` has $\mathcal{O}(2ms(N/m) + 2s^2(N/m))$ work, and `L2` has $\mathcal{O}(2m^2(N/m))$. When s and m are held constant, the total work is $O(N)$ per right hand side. In `GOFMM`, this is controlled by the **budget**.

4.2.3 Shared memory parallelism

In $\mathcal{H}$-matrix methods and FMM, the main algorithmic pattern is a tree traversal. A traversal may exhibit high parallelism at the leaf level, but the parallelism typically diminishes near the root level due to the dependencies. In addition, if the workload per tree node varies, load balancing becomes an issue. Most static scheduling codes employ level-by-level traversals, which introduces unnecessary synchronizations. In `GOFMM`, we observe significant workload variations during the compression (Algorithm 4.2.6) and during the evaluation (tasks `N2S` and `S2N`).

One solution is to exploit parallelism in finer granularity. For example, when

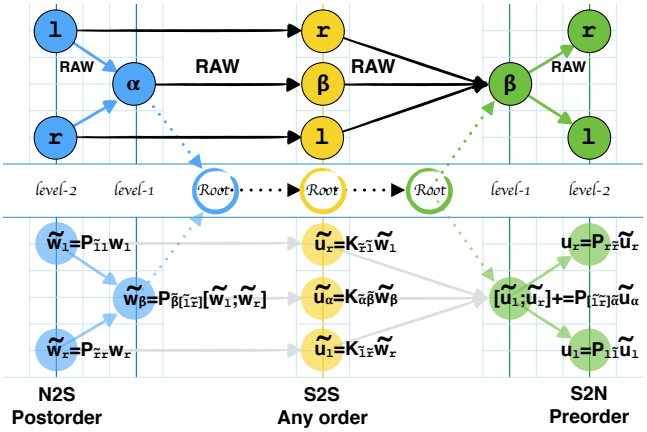


Figure 4.3 *Dependency graph for steps 1–3 of Algorithm 4.2.7 (step 4 is completely independent of steps 1–3). Each tree node denotes a task, and the arrows between nodes imply a dependency. Here* $\texttt{Near}(\alpha)$ *only contains itself (HSS). For example, yellow node* β *has a* ***RAW*** *dependency following blue* α*, because* `S2S(`β`)` *computes* $\widetilde{u}_\beta = \sum_{\alpha \in \texttt{Near}(\beta)} K_{\widetilde{\beta}\widetilde{\alpha}} \widetilde{w}_\alpha$*. When* $\texttt{Near}(\beta)$ *contains more than just itself. The dependencies are unknown at compile time and thus,* `omp task depend` *fails to describe the dependencies between* `N2S` *and* `S2S`*.*

the number of tree nodes in the single tree level is less than the number of cores, we can use multi-threaded BLAS/LAPACK on a single tree node. However, this is insufficient if the workload does not increase significantly (e.g. growing with $|\alpha|$) while approaching the root. (That is, the workload must be within the strong scaling range of BLAS/LAPACK to be efficient).

To partially address these challenges, we abandon the convenient level-by-level traversal and explore an *out-of-order* approach using dynamic scheduling. To this end, we test two approaches and compare them with a level-by-level traversal. In the first approach, we introduce a self-contained runtime system. In the second approach we test the same ideas with `OpenMP`'s `omp task depend` feature.

Dependency analysis. Recursive preorder and postorder traversals in-

herently encode **R**ead/**W**rite dependencies between tree nodes. Following Algorithm 4.2.2 and Algorithm 4.2.7, we can describe dependencies between different tasks. However, due to dynamic granularity of tasks we need a data flow analysis at runtime. For example, dependencies between `N2S` and `S2S` cannot be discovered at compile time, because the **RAW** (read after write) dependencies on $\widetilde{w}_\alpha$ are computed by neighbors $\mathcal{N}(\alpha)$. In order to build dependencies at runtime as a direct acyclic graph (DAG), we perform a *symbolic* execution on Algorithm 4.2.2 and Algorithm 4.2.7. For simplicity, below we just discuss the evaluation phase for the HSS case (the FMM case is more involved).

Figure 4.3 depicts task dependencies (by tasks we mean algorithmic tasks defined in Table 4.2) during the evaluation phase Algorithm 4.2.7 for `N2S`, `S2S` and `S2N` where the off-diagonal blocks are low-rank (HSS) with $S = 0$. This task dependency graph is generated by our runtime using symbolic traversals. The `N2S`, `S2S`, and `S2N` execution order is performed on a binary tree[3].

We use three symbolic tree traversals in Algorithm 4.2.7. In the first traversal (postorder) we find that $\widetilde{w}_{\mathtt{l}}$ is written by `l`. Going from $\widetilde{w}_{\mathtt{l}}$ to $\widetilde{w}_\beta$, we annotate that $\widetilde{w}_{\mathtt{l}}$ is read by β, i.e. $\widetilde{w}_\beta = P_{\widetilde{\beta}[\widetilde{\mathtt{l}}\widetilde{\mathtt{r}}]}[\widetilde{w}_{\mathtt{l}}; \widetilde{w}_{\mathtt{r}}]$. This **RAW** dependency is an edge from `l` to β in the DAG.

Inter-task dependencies are discovered by the *symbolic* execution of the yellow tree. At node β (in yellow), the relation $\widetilde{u}_\alpha = K_{\widetilde{\alpha}\widetilde{\beta}\widetilde{w}_\beta}$ will read $\widetilde{w_\beta}$. Again this is a **RAW** dependency, hence the edge from the blue β to the yellow α. The whole

[3]Execution order from left to right: dependencies are easier to follow if one rotates the page by 90° counter-clockwise

dependency graph for steps 1–3 is built after the green postorder tree traversal. Step 4 in Algorithm 4.2.7 is independent of steps 1–3. Although this runtime data flow analysis has some overhead, the amount is almost negligible ($< 1\%$) compared to the total execution time.

Runtime system. With a dependency graph, scheduling can be done in *static* or *dynamic* fashion. Due to unknown adaptive rank s at compile time, we implement a light-weight dynamic Heterogeneous Earliest Finish Time (HEFT) [97] using `OpenMP` threads. Each worker (thread) in the runtime system can use more than one physical core with either a nested `OpenMP` construct or by employing a device (accelerator) as a slave. Tasks that satisfy all dependencies in the dependency graph will be dispatched to a "ready" queue. Each worker keeps consuming tasks in its own ready queue until no tasks are left.

Although we can estimate a cost for each task[4] in Table 4.2, the execution time of a task on a normal worker (or one with an accelerator) depends on the problem and can only be determined at runtime. The HEFT schedule is implemented using an estimated finish time of all pending tasks in a specific worker's ready queue. Each task dispatched from the dependency graph is assigned to a ready queue such that the maximum estimated finish time of each queue is minimized. For the case where the estimation is inaccurate, we also implement a job stealing mechanism.

Other parallel implementation. We briefly introduce other possible parallel implementations and conduct a strong scaling experiment in §4.4. Here we

[4]We divide costs for tasks by the theoretical peak `FLOPS` of the target architecture and a discount factor. For memory-bound tasks we use the theoretical `MOPS` instead.

implemented parallel level-by-level traversals for all tasks that require preorder and postorder traversals and do not exploit out-of-order parallelism. For tasks that can be executed in any order, we simply use `omp parallel for` with dynamic scheduling. If there are not enough tree nodes in a tree level, we use nested parallelism with inner `OpenMP` constructs and multi-threaded BLAS/LAPACK.

The `omp task` version is implemented using recursive preorder or postorder traversals. Due to the overhead of the deep call stack, this implementation can be much slower than others. Although we tested it, we do not report results because it is not competitive.

We also implemented (and report results for) `omp task depend`, since `OpenMP-4.5` supports task parallelism with dependencies. However there are two issues. First, `omp task depend` requires all dependencies to be known at compile time, which is not the case for the FMM (tasks `N2S` and `S2S`). Second, without knowledge of the estimated finish time, the `OpenMP` scheduler will be suboptimal. Finally for CPU-GPU hybrid architectures, scheduling GPU tasks purely with `omp task` can be very challenging.

CPU-GPU hybrid. GPUs usually offer high computing capacity, but performance can easily be bounded by the PCI-E bandwidth. Because most computations in Algorithm 4.2.2 are complex and memory bound[5], we do not use GPUs for the compression. Instead we only pre-fetch submatrices $K_{\beta\alpha}$ and $K_{\widetilde{\beta}\widetilde{\alpha}}$ to the device memory to overlap with computations on the host (CPUs). During the evaluation,

[5] Although `GEQP3` and `TRSM` can be performed on GPUs with `MAGMA` (`http://icl.cs.utk.edu/magma/`) and `cublas`, we find this inefficient for our methods.

our runtime will decide–depending on the number of `FLOPS`– whether to issue a batch of tasks (up to 8) to the GPU in concurrent (using `stream`). This usually occurs in `N2S` and `S2N` where the size of `cublasXgemm` is bounded by s and m. Furthermore, to hide communication time between CPU and GPU, all arguments of the next task in queue are pre-fetched using asynchronous communication for pipelining. Finally, because a worker with a GPU is usually $50\times$ to $100\times$ more capable than others, we disable job stealing balancing for GPU workers. This optimization prevents the GPU from idling.

Distributed parallelism. In this work, we do not discuss how to parallelize `GOFMM` in a distributed environment. The MPI extension requires new algorithms, which will be discussed in the future work of `GOFMM`. The basic philosophy of MPI parallelism follows [23, 71, 109], which include distributed tree traversal, distributed nearest-neighbor search, local essential trees for reducing communication, and distributed linear algebra operations. New challenges include parallelizing matrix access, integrating the task-scheduling with MPI, accounting for off-diagonal dependencies from other ranks, and load-balancing. Inter-process job stealing may also result in extra communication.

4.3 Experimental Setup

We perform experiments on Haswell, KNL, ARM, and NVIDIA GPU architectures with four different setups to examine the accuracy and efficiency of our methods. We demonstrate (1) the robustness and effectiveness of our geometry-oblivious FMM, (2) the scalability of our runtime system against other parallel schemes, (3) the

accuracy and cost comparison with other software, and (4) the absolute efficiency (in percentage of peak performance).

Implementation and hardware. `GOFMM` is implemented in `C++` and `CUDA`, employing `OpenMP` for shared memory parallelism. The source code of `GOFMM` can be found in the Github repository (`https://github.com/ChenhanYu/hmlp`). Our tests were conducted on TACC's Lonestar 5, (two 12-core, 2.6GHz, Xeon E5-2690 v3 "Haswell"), TACC's Stampede 2 (68-core, 1.4GHz, Xeon Phi 7250 "KNL") and CSCS's Piz Daint (12-core, 2.3GHz, Xeon E5-2650 v3 and NVIDIA Tesla P100).

Matrices We generated 22 matrices emulating different problems. **K02** is a 2D regularized inverse Laplacian squared, resembling the Hessian operator of a PDE-constrained optimization problem. The Laplacian is discretized using a 5-stencil finite-difference scheme with Dirichlet boundary conditions on a regular grid. **K03** has the same setup with the oscillatory Helmholtz operator and 10 points per wave length. **K04**–**K10** are kernel matrices in six dimensions (Gaussians with different bandwidths, narrow and wide; Laplacian Green's function, polynomial and cosine-similarity). **K12**–**K14** are 2D advection-diffusion operators on a regular grid with highly variable coefficients. **K15,K16** are 2D pseudo-spectral advection-diffusion-reaction operators with variable coefficients. **K17** is a 3D pseudo-spectral operator with variable coefficients. **K18** is the inverse squared Laplacian in 3D with variable coefficients. **G01**–**G05** are the inverse Laplacian of the **powersim**, **poli_large**, **rgg_n_2_16_s0**, **denormal**, and **conf6_0-8x8-30** graphs from UFL (`http://yifanhu.net/GALLERY/GRAPHS/search.html`).

K02–**K03**, **K12**–**K14**, and **K18** resemble inverse covariance matrices and Hessian operators from optimization and uncertainty quantification problems. **K04**–**K10** resemble classical kernel/Green function matrices but in high dimensions. **K15**–**K17** resemble pseudo-spectral operators. **G01**–**G05** (N = 15838, 15575, 65536, 89400, 49152) are graphs for which we do not have geometric information. For **K02**–**K18**, we use $N = 65536$ if not specified.

Also, we use kernel matrices from machine learning: **COVTYPE** (100K, 54D, cartographic variables); and **HIGGS** (500K, 28D, physics) [63]; **MNIST** (60K, 780D, digit recognition) [18]. For these datasets, we use a Gaussian kernel with bandwidth h.

`GOFMM` supports both double and single precision. All experiments with matrices **K02**–**K18** and **G01**–**G05** are in single precision. The results for **COVTYPE, HIGGS, MNIST** are in double precision. In the Github repository, we provide a MATLAB script to generate **K02**–**K18**. For real world datasets and graphs, we provide the link to their original sources.

Parameter selection and accuracy metrics. We control m (leaf node size), s (maximum rank), τ (adaptive tolerance), κ (number of neighbors), ***budget*** (a key parameter for amount of direct evaluations and for switching between HSS and FMM) and partitioning (**Kernel**, **Angle**, **Lexicographic**, **geometric**, **random**). We use m =256–512; on average this gives good overall time. The adaptive tolerance τ, reflects the error of the subsampled block and may not correspond to the output error ϵ_2. Depending on the problem, τ may underestimate the rank. Similarly, this may occur in `HODLR`, `STRUMPACK` and `ASKIT`. We use τ between 1E-2 and 1E-7, $s = m$,

$k = 32$ and 3% budget. To enforce a HSS approximation, we use 0% budget. The Gaussian bandwidth values are taken from [70] and produce optimal learning rates.

Throughout we use relative error ϵ_2 defined as the following

$$\epsilon_2 = \|\widetilde{K}w - Kw\|_F / \|Kw\|_F, \text{ where } w \in \mathbb{R}^{N \times r}. \tag{4.11}$$

This metric requires $\mathcal{O}(rN^2)$ work; to reduce the computational effort we instead sample 100 rows of K. In all tables, we use "Comp" and "Eval" to refer the the compression and evaluation time in seconds, and "GFs" to GFLOPS per node.

4.4 Empirical Results

We label all experiments from #1 to #46 in tables and figures. We perform ***strong scaling results*** on a single Haswell and KNL node in Fig. 4.4, comparing different scheduling schemes. In Fig. 4.5, we examine the ***accuracy*** of `GOFMM` for the different matrices; notice that not all 22 matrices admit good hierarchical low-rank structures in the original order (lexicographic). In Fig. 4.6, we ***compare FMM*** ($S \neq 0$ in (4.1)) ***to HSS*** ($S = 0$) and show an example in which increasing direct evaluations in FMM results in higher accuracy and shorter wall-clock time. In Fig. 4.7, we present a comparison between ***five permutation*** schemes; matrix-defined Gram distances work quite well.

For reference, we compare `GOFMM` to ***three other codes***: `HODLR` and `STRUMPACK` ($S = 0$ in these codes) in Table 4.3 and `ASKIT` (high-d FMM) in Table 4.4. The two first codes do not permute K. `ASKIT` is similar to `GOFMM` but uses level-by-level traversals, does not produce a symmetric $\tilde{K}$, and requires points. Finally, we test

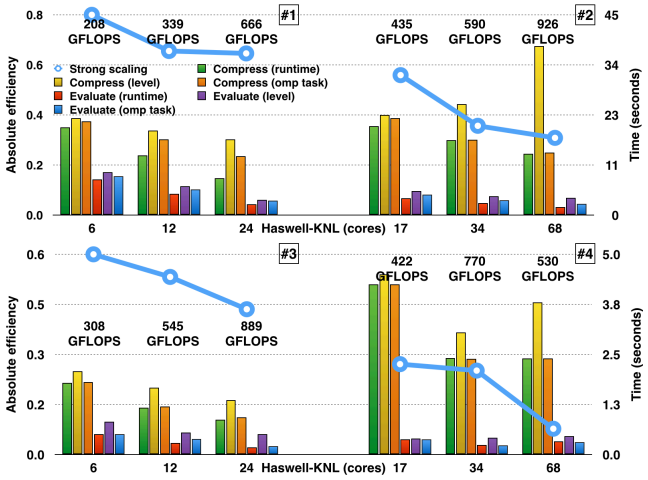


Figure 4.4 *Strong scaling on a single Haswell and KNL node (y-axis, time in seconds on the right, absolute efficiency to the peak* `GFLOPS` *on the left). We use* $s512$, $\tau 1E-5$ *and* $r512$. *#1 and 2 use* ***COVTYPE*** *to create a Gaussian kernel matrix with* $m800$ *and* 12% *budget (*$h = 0.1$*), achieving* $\epsilon_2 = 2\text{E}{-}3$ *with average rank* 487. *#3 and #4 use* ***K02*** *with* $m512$ *and* 3% *budget, achieving* $\epsilon_2 = 5\text{E}{-}5$ *but only with average rank* 35. *We increase the number of cores up to 24 Haswell cores and 68 KNL cores. Each set of experiments contains compression time and evaluation time on three different parallel schemes: wall-clock time, level-by-level and omp tasks. We cannot perform scaling experiments for the hybrid CPU-GPU platform (see Table 4.5 for GPU performance).*

`GOFMM` on ***four different architectures*** in Table 4.5; the performance of `GOFMM` correlates with the performance of BLAS/LAPACK.

Strong scaling (Fig. 4.4). In #1, #2, #3, #4, we use a 24-core Haswell and a 68-core KNL to perform strong scaling experiments. Each set of experiments contains 6 bars including 3 different parallel schemes on both Algorithm 4.2.2 and Algorithm 4.2.7. The blue dot indicates the absolute efficiency (ratio to the peak) of our **evaluation** using dynamic scheduling. #1 and #2 require 12% budget with average rank 487 to achieve 2E−3. This compute-bound problem can reach 65%

peak performance on Haswell and 33% on KNL. However, #3 and #4 only require 3% budget with average rank 35 to achieve 5E−5. As a result, this memory-bound problem does not scale (46% and 8%[6]) very well. In #4, we can even observe slow down from 34-core to 68-core. This is because the wall-clock time is bounded by the task in the critical path; thus, increasing the number of cores does not help.

Throughout, we can observe that the wall-clock time for compression is less than the level-by-level and `omp task` traversals. While the work of `SKEL` is bounded by $2s^3$, parallel `GEQP3` in the level-by-level traversal does not scale (especially on KNL). On the other hand, task based implementations can execute `COEF` and `Kba` out-of-order to maintain the parallelism. Our wall-clock time is better than `omp task` since we use the cost-estimate model for scheduling.

Accuracy (Fig. 4.5). We conduct #5 to examine the accuracy of `GOFMM` (up to single precision). Given $m512$, $s512$ and $r512$, we report relative error ϵ_2 on **K02-18** and **G01-G05** using the **Angle** distance with two tolerances: 1E−2 (in blue) and 1E−5 (in green). Throughout, except for **K06**, **K15**–**K17** (high rank), **K13**, **K14** (underestimating the rank), and **G01**–**G03** (requiring smaller leaf size m), other matrices can usually achieve high accuracy with tolerance 1E−5 (0.9s in compression and 0.2s in evaluation). Our adaptive ID underestimates the rank of **K13** and **K14** such that ϵ_2 is high. By imposing a smaller tolerance 1E−10 (yellow plots), both matrices reach 1E−5 (1s in compression and 0.2s in evaluation).

[6]The average rank of #4 is too small. Except for `L2L` tasks, other tasks can only reach about 5% of the peak during the evaluation. We suspect that MKL' `SGEMM` uses a 30×16 micro-kernel to perform a $30 \times 256 \times 16$ rank-k update each time. For an $m \times k \times n$ `SGEMM` to be efficient, m and n usually need to be at least four times of the micro-kernel size in each way. In #4, many `SGEMM`s have $m < 30$. Still the micro-kernel must compute $2 \times 30 \times 256 \times 16$ FLOPS. These sparse FLOPS are not counted in our experiments.

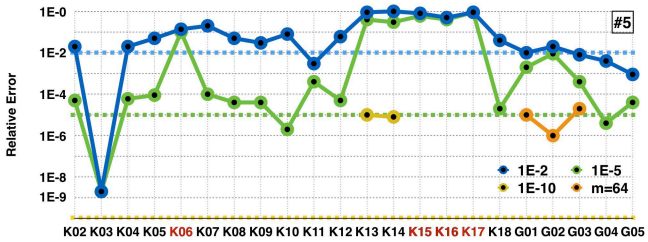


Figure 4.5 *#5, relative error* ϵ_2 *(y-axis, the smaller the better) on all matrices (x-axis) using angle distance. Blue bars use* τ1E−2 *and* 1% *budget (except for* ***K6****,* ***K15****,* ***K16****,* ***K17****, other matrices take* $0.8s$ *to compress and* 0.1 *to evaluate in average). Green bars use* τ1E−5 *and* 3% *budget (in average, compression takes* $1s$ *and evaluation takes* $0.2s$*). Red labels denotes matrices that do not compress.* ***K13*** *and* ***K14*** *have hierarchical low-rank structure, but the adaptive ID underestimates the rank.* ***K13*** *and* ***K14*** *can reach high accuracy (yellow plots) with* τ1E−10 *and* 3% *budget (*$1.0s$ *in compression and* $0.2s$ *in evaluation).*

K6, **K15**–**K17** have high ranks in the off-diagonal blocks; thus they cannot be compressed with $s512$ and 3% budget. **G01**–**G03** requires direct evaluation in the off-diagonal blocks to reach high accuracy. When we reduce the leaf node size from 512 to 64, we can can still reach 1E−5 (orange plots). However, decreasing leaf size to 64 results in a longer wall-clock time ($0.8s$ in evaluation), because small m hurts performance. Overall, we can observe that `GOFMM` can quite robustly discover low-rank plus sparse structure from different SPD matrices. We now investigate how increasing the cost (either with higher rank or more direct evaluations) can improve accuracy.

Comparison between FMM and HSS (Fig. 4.6). We use #6, #7, and #8 to show that even with more evaluations, FMM can be faster than HSS for the same accuracy. For HSS the relative error in #6 (blue plots) plateaus at 5E−4. Further increasing rank from 256 to 512 (or even 1,024) results in $O(s^3)$ work (green

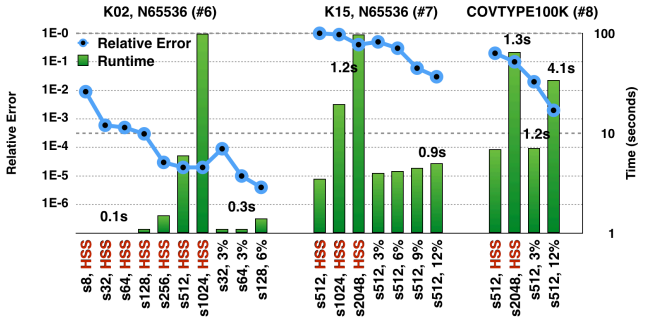


Figure 4.6 *Comparison between HSS and FMM in wall-clock time (seconds, green bars, right y-axis) and accuracy (ϵ_2, blue plots, left y-axis). In #6, #7 and #8, we use* ***K02***, ***K15*** *($m512$) and* ***COVTYPE*** *($m800$) datasets. The fixed rank and budget are labeled on x-axis. The green bar is the total wall-clock time including compression and evaluation on* 512 *right hand sides. For some experiments, we also provide wall-clock time for evaluation to contrast the trade-off of using high rank and high budget.*

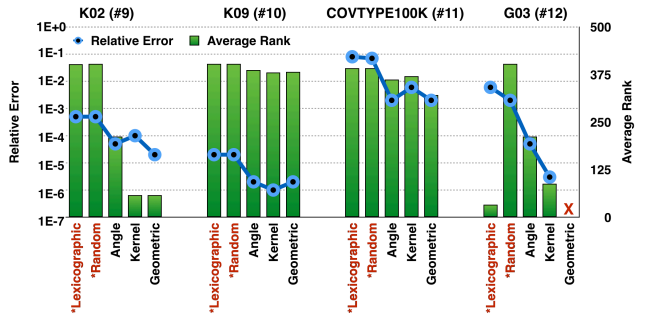


Figure 4.7 *Accuracy (left y-axis) and rank (right, x-axis) comparison:* ***Lexicographic***, ***Random***, ***Kernel 2-norm***, ***Angle*** *and* ***Geometric****. We use τ1E$-$7, $s512$, $m64$. For methods that define* distance, *we use $k32$ and* 3% *budget.* ***G03*** *is a graph Laplacian; thus, using* ***Geometric*** *distance is impossible.*

bars). Using a combination of low-rank ($s64$) and 3% direct evaluation, FMM can achieve higher accuracy with little increment in the evaluation time (compression time remains the same). Similarly, in #8 we can observe that by using $s512$ and 3% budget we achieve better accuracy than the HSS approximation ($s2048$) in less time.

Permutations (Fig. 4.7). Here we test different permutations (#9, #10, #11, and #12) to discuss the different distances in `GOFMM`. In each set of experiments, we present relative error (blue plots) and average rank (green bars) for five different schemes. The first two schemes use lexicographic or random order to recursively permute K. Since there is no *distance* defined, these two schemes can only use HSS approximation. The **Angle** and **Kernel** *distance* use the corresponding Gram distances Section 4.2.1. Finally, we also use standard geometric distance from points. For the last three schemes, we use $\kappa 32$ and 3% budget. Overall, we can observe that the distance metric is important in discovering low-rank structure and improving accuracy. For example, in #9, **Kernel** and **Geometric** show much lower average rank than others. In #10 and #11, although the average ranks are not significantly different, distance-based methods usually have higher accuracy. Finally, we observe for matrix **G03** in #12 where no coordinate information exists, our geometry-oblivious methods can still compress the matrix. Although the lexicographic permutation has very low rank, the error is large. This is because the uniform samples for the low-rank approximation are poor. **Angle** and **Kernel** distance use neighbors for importance sampling, which greatly improves the quality of the low-rank approximation.

Comparison to existing software (Table 4.3, Table 4.4). We compare our methods to `HODLR` [4], `STRUMPACK` [90], and `ASKIT` [73]. Let us summarize some key differences. `HODLR` uses the Adaptive Cross Approximation (ACA, partial pivoted LU) for constructing the low-rank blocks (using the `Eigen` library). Its evaluation requires $O(N \log N)$ work since the U, V matrices are not nested. `STRUMPACK` constructs an HSS representation in $O(N \log N)$ work. This is done by using a

		HODLR			STRUMPACK			GOFMM		
#	case	ϵ_2	Comp	Eval	ϵ_2	Comp	Eval	ϵ_2	Comp	Eval
13	K02	6E−5	0.6	2.7	1E−4	9.2	0.6	2E−5	1.0	0.3
14	K04	6E−5	0.7	2.7	1E−4	507.8	7.8	2E−5	1.0	0.5
15	K07	7E−5	0.9	3.1	2E−4	528.4	8.2	4E−5	0.6	0.2
16	K12	6E−5	0.7	2.7	2E−4	18.8	0.8	1E−4	0.6	0.2
17	K17	1E−1	862.2	37.6	2E−1	663.4	8.2	9E−2	48.8	3.1
18	G03	3E−4	**12.9**	9.7	3E−2	**29.8**	1.3	8E−5	**0.5**	0.8

Table 4.3 *Wall-clock time comparison (in seconds) between* `HODLR`*,* `STRUMPACK`*, and* `GOFMM`*. For* ***K02***–***K12****, we use* $N = 36K$*.* ***K17*** *uses* $N = 32K$*, and* ***G03*** *uses* $N = 65K$*. For all software, we use leaf node size* $m512$ *and* 1024 *right hand sides. We control other parameters (*τ *and* s*) for each software to target the same relative error (*1E−4*).*

	Parameters			ASKIT			GOFMM		
#	case	N	τ	ϵ_2	Comp	Eval	ϵ_2	Comp	Eval
19	K04	36 864	1E−3	2E−4	0.3	2E−2	2E−4	0.6	2E−2
20	K04	36 864	1E−6	8E−7	1.4	4E−2	7E−7	1.0	3E−2
21	K04	65 536	1E−3	2E−4	1.0	4E−2	2E−4	1.2	4E−2
22	K04	65 536	1E−6	7E−7	2.2	8E−2	6E−7	1.7	4E−2
23	K06	36 864	1E−3	4E−2	6.6	6E−2	3E−2	3.3	4E−2
24	K06	36 864	1E−6	2E−2	7.4	6E−2	3E−2	4.8	5E−2
25	K06	65 536	1E−3	4E−2	11.1	1E−1	4E−2	5.7	8E−2
26	K06	65 536	1E−6	5E−2	12.0	1E−1	4E−2	7.7	9E−2

Table 4.4 *Wall-clock time (in seconds) and accuracy* ϵ_2 *comparison with* `ASKIT`*. For both methods, we use* $\kappa = 32$*,* $m = s = 512$ *and* $r1$*.* `ASKIT` *use the* τ *reported in the table, and we adjust the tolerance of* `GOFMM` *to match the accuracy. For all experiments,* `GOFMM` *uses* 7% *budget. The amount of direct evaluation performed by* `ASKIT` *is decided by* κ*.*

randomized ID according to [62]. We used their black-box compression routine with a uniform random distribution and a Householder rank-revealing QR. Once the matrix is compressed, the evaluation time is $O(N)$ per right hand side. `STRUMPACK` supports multiple right hand sides. `ASKIT`'s FMM evaluation has similar complexity as `GOFMM`, but the amount of direct evaluation is only decided by κ. For `GOFMM`, we further introduce the budget to restrict the cost. For all comparisons, we try to match the accuracy by controlling different parameters (τ, s, and κ). Notice that `ASKIT` and `STRUMPACK` support MPI, whereas `GOFMM` does not. We have not used MPI for distributed environment in our experiments.

In Table 4.3, we target final accuracy $\epsilon_2 = 1\text{E}{-}4$. `GOFMM` uses **Angle** distance for neighbor search and tree partitioning. `HODLR` and `STRUMPACK` do not have built-in partitioning schemes for dense matrices. `STRUMPACK` fails to compress **K04** (Gaussian kernel in 6D) and **K07** (Laplace kernel in 6D). This is because the lexicographic order does not admit a good $\mathcal{H}$-matrix approximation. The matrix needs to be permuted. **K17** is difficult to compress with a pure hierarchical low-rank matrix. Finally, **G03** performs better when $S \neq 0$. `HODLR` and `STRUMPACK` must increase the off-diagonal ranks to match the accuracy and thus the cost increases. With a sparse correction S, `GOFMM` is about 25× faster in compression and about 1.5× faster in evaluation.

In Table 4.4, we compare `GOFMM` (with geometric distances) to `ASKIT`. `ASKIT` uses level-by-level traversals in both compression and evaluation. Since `ASKIT` only evaluates a single right hand side, we use $r = 1$. The compression time is inconclusive for #19–#22; the average ranks used in two methods are quite different. The

benefit of *out-of-order* traversal appears in #23–#26 where both methods reach the maximum rank s. The speedup in evaluation is not significant, but `GOFMM` can get up to 2× speedup in compression.

Different architectures. In Table 4.5, we present wall-clock time and `GFLOPS` of `GOFMM` on four architectures for different problems. We want to show that the efficiency of `GOFMM` is portable and only relies on BLAS/LAPACK libraries.

In #27 and #28, we show that a quad-core ARM processor can handle up to 100K fast matrix-multiplication. Because we only have limited memory (2GB) and storage (8GB), in `GOFMM` we compute K_{ij} on the fly (in detail, we compute $K_{\beta\alpha}$ with a `GEMM` using the 2-norm expansion). #27 takes much longer than #28 because the cost of evaluating K_{ij} is proportional to the point dimensions of the dataset (**MNIST** in 780D and `COVTYPE` in 54D). Because there is no active cooling on the board, the ARM processor gets overheated and is forced to reduce its clockrate. That is why we can only reach 30% of peak during the evaluation.

Experiments #29 to #34 are computed in double precision. With 12% budget, our evaluation can reach 68% peak performance on Haswell, 37% on KNL and 38% on a hybrid Haswell-P100 system. The performance degrades in #32–34 because the rank is limited to 256, and 0.3% direct evaluation is not enough to create large `GEMM` calls. For kernel matrices, the `GFLOPS` for compression are usually higher because computing K_{ij} requires floating point operations. For example, compression of **COVTYPE** (in 54D) has higher `GFLOPS` than **HIGGS** (in 28D). This is not only because **COVTYPE** is a dataset with high dimensionality, but we also use a higher rank $s512$ such that `GEQP3` and `TRSM` can be more efficient.

#	Arch	Budget	ϵ_2	Comp	GFs	Eval	GFs
MNIST60K, $h1$, $\kappa32$, $m512$, $s128$, $r256$							
27	ARM	5%	5E-3	285	3	520	12
COVTYPE100K, $h1$, $\kappa32$, $m512$, $s128$, $r256$							
28	ARM	5%	8E-4	71	2	61	10
COVTYPE100K, $h0.1$, $\kappa32$, $m800$, $s512$, $r512$							
29	CPU	12%	2E-3	30	30	4.1	679
30	CPU+GPU	12%	3E-3	33	29	1.7	1952
31	KNL	12%	2E-3	48	25	3.2	1125
HIGGS500K, $h0.9$, $\kappa64$, $m1024$, $s256$, $r512$							
32	CPU	0.3%	2E-1	102	18	3.3	592
33	CPU+GPU	0.3%	2E-1	180	12	1.7	1147
34	KNL	0.3%	2E-1	121	17	2.2	872
K02, $N65536$, $\kappa32$, $m512$, $s512$, $r512$							
35	CPU	3%	9E-5	1	25	0.2	889
36	CPU+GPU	3%	1E-4	2	12	0.1	2175
37	KNL	3%	1E-4	3	11	0.3	530
K15, $N65536$, $\kappa32$, $m512$, $s512$, $r1024$							
38	CPU	10%	2E-1	6.0	81	1.1	1495
39	CPU+GPU	10%	2E-1	7.8	62	0.66	2514
40	KNL	10%	2E-1	9.2	53	1.3	1549
G03, $N65536$, $\kappa32$, $m128$, $s512$, $r512$							
41	CPU	3%	4E-5	4.8	37	0.5	1122
42	CPU+GPU	3%	3E-5	7.9	19	0.53	962
43	KNL	3%	5E-5	11.8	9.1	0.6	741
G04, $N89400$, $\kappa32$, $m512$, $s512$, $r512$							
44	CPU	3%	4E-6	1.8	21	0.3	787
45	CPU+GPU	3%	4E-6	4.0	10	0.13	2277
46	KNL	3%	4E-6	4.2	9	1.5	215

Table 4.5 *Accuracy ϵ_2, wall-clock time (in seconds) and efficiency (in `GFLOPS`) on four architectures. Because our ARM platform only has a 8GB SD card and 2GB DRAM, we only perform kernel matrices (K_{ij} computed on the fly) with small r and s. Note that in the CPU+GPU experiment, the compression is run on the CPU (see Section 4.2.3).*

Finally, we present performance results on several matrices (#35–46) in single precision. With 10% budget in **K15**, our evaluation can reach 75% peak on Haswell, 25% on KNL and 25% on a hybrid Haswell-P100 system. This performance requires large leaf node size m and sufficient direct evaluations (e.g. #35–#46). Since **G03** requires small m, our `GFLOPS` efficiency degrades due to the dependency on the BLAS/LAPACK routines. Notice that $m128$ is not large enough for `GEMM` to reach high performance on KNL and GPUs. For **G04**, we use $m512$ but KNL (#46) does not perform very well. The same problem occurs in Fig. 4.4: the average rank in **G04** is too small. Additionally, we do not observe huge performance degradation on GPUs (#45). This is because we enforce our scheduler to schedule `L2L` tasks to the GPU; thus, tasks with small ranks (`N2S` and `S2N`) are mostly consumed by the host CPU. The comparison between #45 and #46 is a good example that highlights the goal of heterogeneous parallel architectures. CPUs with short vector lengths are suitable for tasks with very low ranks (`N2S` and `S2N`). On the contrary, GPUs are the method of choice for `FLOPS` intensive tasks (`L2L`). Due to very different kinds of tasks, `GOFMM` may require both high throughput (GPU and KNL) and low latency (CPU) units to be efficient. We cannot solve such problems with only one architecture efficiently.

4.5 Conclusions

By using the Gramian vector space for SPD matrices, we defined distances between rows and columns of K using only matrix values. Using the distances, we introduced `GOFMM` and $\mathcal{H}$-matrix scheme that can be used to compress arbitrary SPD

matrices (but without accuracy guarantees). These algorithms are applied black-box for various problems in computational science and we observe that the approach can be very attractive. In `GOFMM` we use a shared-memory runtime system that performs *out-of-order* scheduling in parallel to resolve the dynamic workload due to adaptive ranks and the parallelism-diminishing issue during tree traversals. Our future work will focus on the distributed algorithms and the hierarchical matrix factorization based on our method. We also plan to improve the sampling and pruning quality and to reduce the number of parameters that users need to provide.

Chapter 5

Leverage Score Clustering[1]

[1]The content in this chapter is based on work done in collaboration with George Biros.

We propose RECUR, a novel method for compressing dense matrices. Our method is based on a hierarchical-matrix (H-matrix) approximation. H-matrix approximations have been popular in science and engineering applications. They combine the notion of singular value decomposition (SVD) with appropriate block permutations and recursion. H-matrices are applicable to problems in which the matrix entries correspond to pairwise interactions between sets of points, as for example in kernel matrices. Here we generalize this approximation to arbitrary dense matrices. Our method comprises of a randomized low-rank approximation of permuted blocks along with approximate leverage scores computations that are used to find such permutations. We introduce theoretical analysis, complexity analysis, and experimental results on kernel matrices, neural network Jacobian operators, and other datasets.

5.1 Introduction

Low-rank approximations of matrices are prevalent in machine learning. They can be used for dimensionality reduction, compression, acceleration of matrix operations, and as fundamental blocks in other machine learning tasks. Examples of matrices that require compression include kernel matrices, adjacency matrices of dense graphs, correlation matrices, and Hessians and Jacobian matrices related to optimization problems, to name a few. Here we are interested in hierarchical approximations for matrices that do not necessarily admit a global low rank approximation. Once the matrix is compressed, fast methods for linear algebra operations like linear system and eigenvalue solvers can be applied [44, 46].

In particular, we consider the following problem. Given a dense matrix $A \in \mathbb{R}^{m \times n}$, the cost of matrix vector multiplication operation (matvec) requires $\mathcal{O}(nm)$ work. We wish to construct an approximation to A so that the matvec with the approximate A becomes $\mathcal{O}(n+m)$ up to logarithmic prefactors and a constant that depends on the desired error tolerance. Towards this goal, we propose to use a *hierarchical matrix* (H-matrix) approximation [44]. H-matrix approximation methods are popular on problems that have explicit geometric structure, that is $A_{ij} = a(x_i, x_j)$, where x_i, x_j are points in a vector space, and $a(,)$ is a given pairwise interaction, along with certain growth and decay properties of $a(x_i, x_j)$ as a function of $\|x_i - x_j\|_2$. The geometric structure combined with the properties of $a(,)$ is used to appropriately permute the matrix so that the off-diagonal blocks are easily compressible.

Contributions. Our contributions are summarized as follows

- We present *RECUR*, a new algorithm for matrix approximation.
- We analyze and prove its accuracy and work complexity.
- We present numerical results that demonstrate its efficiency and shortcomings.

The defining feature of rank-structured matrices is the presence of large low-rank off-diagonal blocks that can be efficiently represented with a low-rank approximation. Such structure depends on an appropriate ordering of the rows and columns of the matrix. Therefore, the first step of constructing a rank-structured representation of a matrix is to find an ordering of its rows and columns that produces the

desired structure. RECUR introduces a scheme to find such permutations using block leverage scores and combines it with randomized low-rank approximations of different blocks of the matrix. H-matrices have two main variants: *weak admissibility* and *strong admissibility* methods. Roughly speaking, weak admissibility use low-rank approximations for all off-diagonal blocks; strong admissibility compression allows for dense off-diagonal blocks to improve accuracy. RECUR supports both variants. The main advantage of RECUR over existing H-matrix or other kernel approximation methods is that it is black-box, that is it only requires matrix entries and thus can be applied to any matrix. RECUR has three main parameters: the recursion depth, the error tolerance, and a target compression factor in terms of maximum rank for off-diagonal blocks. RECUR will *not* successfully compress all matrices: if the matrix is not H-matrix compressible then RECUR will fail. For globally low-rank matrices (for the target tolerance), RECUR behaves like a randomized global low-rank approximation.

Related work. RECUR is closely connected to N-body and H-matrix methods for kernel matrices [51, 58, 73, 82, 96, 99]. As we mentioned, all these methods require point coordinates and $a(,)$ that computes matrix entries. When such information is present, these methods should be used instead of RECUR. RECUR is also related to randomized linear algebra methods [82], including row/column sampling methods using leverage scores [24, 29] or randomized projection methods [46]. There are also black box methods for H-matrix approximation, that is, methods that only use matrix entries. For example, peeling algorithms [64] compute an H-matrix

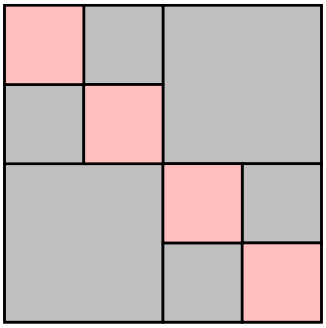


Figure 5.1 *A basic tessellation of a rank-structured matrix. Gray blocks are numerically low-rank, and pink blocks are not low-rank, but are small.*

approximation—but they assume that the matrix has already been appropriately permuted. An exception is GOFMM [108], which implicitly defines point distances based on the matrix entries [108]. GOFMM is the most closely related to RECUR as it both permutes and compresses a matrix and has been applied to many types of problems, including deep neural network Hessians [19]. GOFMM however, is only applicable to symmetric positive definite matrices. RECUR can be applied to arbitrary square or rectangular matrices.

5.2 Preliminaries

ϵ-rank. The ϵ-rank of matrix A, denoted by $\text{rank}(A, \epsilon)$, is the smallest integer k for which there exists a rank-k matrix A_k that satisfies $\|A_k - A\| \leq \epsilon$. For the spectral norm, the ϵ-rank of A equals the number of singular values of A that are greater than ϵ.

Singular value decomposition (SVD). The rank-k truncated SVD of $A \in \mathbb{R}^{m \times n}$ is given by matrices $U \in \mathbb{R}^{m \times k}$, $\Sigma \in \mathbb{R}^{k \times k}$, $V \in \mathbb{R}^{n \times k}$, where U, V have

orthonormal columns, and Σ is diagonal with $\sigma_1 \geq \cdots \geq \sigma_k \geq 0$.

Leverage scores. For matrix $A \in \mathbb{R}^{m \times n}$ with rank-k truncated SVD $A \approx U_k \Sigma_k V_k^T$, the rank-$k$ leverage score of the i^{th} row $A(i,:)$ is given by

$$l_{i,:} = \|U_k^T e_i\|_2^2,$$

and the rank-k leverage score of the j^{th} column $A(:,j)$ is given by

$$l_{:,j} = \|V_k^T e_i\|_2^2.$$

Notation. We refer to a sequence of integers starting from 1 with the shorthand $[n] = 1, ..., n$. A matrix $A \in \mathbb{R}^{m \times n}$ has rows indexed by $[m]$ and columns indexed by $[n]$. We use $A(I_r, I_c)$ to refer to a submatrix of A consisting of entries a_{ij}, where $i \in I_r \subset [m]$ and $j \in I_c \subset [n]$. A colon in place of an index set represents the full set of row or column indices (e.g., $A(:, I_c)$ is the submatrix consisting of columns of A indexed by I_c). Unless specified otherwise, $\|A\|$ refers to the spectral norm of A.

5.3 Algorithms

5.3.1 Selecting a low-rank row submatrix

We propose a technique for selecting within $A \in \mathbb{R}^{m \times n}, m > n$, a row submatrix $A(I_r, :), I_r \subset [m]$ of size $\hat{m} \times n$ with low numerical rank, assuming such a submatrix exists. As we will show, forming the submatrix by excluding rows from A with large leverage scores results in a reduction in the product of singular values by a multiplicative factor. This technique serves as a building block for the algorithms in the following sections.

The technique is based on the principle that rows with high leverage scores are extreme, and therefore, removing those rows leads to a reduction in numerical rank. This idea is formalized in Theorem 1, which asserts that removing a single row with leverage score l_i reduces the product of the singular values by a factor $\sqrt{1-l_i}$. In particular, removing a row with the maximum possible leverage score of 1 guarantees that the remaining submatrix is rank-deficient.

Theorem 1. Given a rank-k matrix $A \in \mathbb{R}^{m\times n}, m \geq n$, with singular value decomposition $A = U\Sigma V^T$, where $U \in \mathbb{R}^{m\times k}, \Sigma \in \mathbb{R}^{k\times k}, V \in \mathbb{R}^{n\times k}$, let $\hat{A} = A(J^c, :)$ be the matrix obtained by removing the rows of A indexed by $J \subset [m]$. Then

$$\prod_{i=1}^{k} \sigma_i(\hat{A}) = \prod_{j=1}^{|J|} \sqrt{1-\sigma_j^2\left(U(J,:)\right)} \prod_{i=1}^{k} \sigma_i(A),$$

where $\sigma_i(A)$ denotes the singular values of A, $\sigma_i(\hat{A})$ denotes the singular values of $\hat{A}$, and $\sigma_j^2\left(U(J,:)\right)$ denotes the singular values of the row submatrix of U indexed by J.

In particular, if $J = \{i\}$ so that $\hat{A}$ is obtained by removing only the i^{th} row from A, then

$$\prod_{i=1}^{k} \sigma_i(\hat{A}) = \sqrt{1-l_i} \prod_{i=1}^{k} \sigma_i(A),$$

where l_i is the leverage score of the i^{th} row of A.

Algorithm 5.3.1 Select a low-rank row submatrix

function LRSubmatrix(A, ϵ, n_r, n_c)
    Randomly initialize I_c with n_c elements of $[n]$
    loop
        $I_r \leftarrow$ LRRowSubmatrix$(A(:, I_c), \epsilon, n_r)$
        $I_c \leftarrow$ LRRowSubmatrix$(A(I_r, :)^T, \epsilon, n_c)$
    return I_r, I_c

function LRRowSubmatrix(A, ϵ, n_r)
    $\{l_i\} \leftarrow$ RowLScores(A, ϵ)
    $I_r \leftarrow$ indices of the n_r rows with smallest leverage scores
    return I_r

function RowLScores(A, ϵ)
    $U, \Sigma, V \leftarrow \mathrm{SVD}(A)$
    $k \leftarrow \#\{\sigma_i : \sigma_i > \epsilon\}$
    for $i \in [m]$ **do**
        $l_i \leftarrow \|U(i, [k])\|_2^2$
    return $\{l_i\}$

A practical algorithm based on these ideas is outlined in function LRRowSubmatrix of Algorithm 5.3.1. Notably, rather than specifying the rank k, we specify some tolerance $\epsilon > 0$, and compute leverage scores with respect to the ϵ-rank of A. Also, rather than recomputing after each selection of a row to remove, we make all of the selections based on the row leverage scores of A computed only once.

5.3.2 Selecting a low-rank submatrix

We incorporate the method for selecting a low-rank row submatrix $A(I_r, :)$ into an algorithm for the more general problem of selecting a low-rank submatrix $A(I_r, I_c)$ indexed over both rows and columns. For a given subset of column indices

I_c, applying LRRowSubmatrix of Algorithm 5.3.1 to the column submatrix $A(:, I_c)$ searches for a set of row indices I_r such that $A(I_r, I_c)$ is a low-rank row submatrix of $A(:, I_c)$. Likewise, for a given set of row indices I_r, applying LRRowSubmatrix to the transposed row submatrix $A(I_r, :)^T$ searches for a set of column indices I_c such that $A(I_r, I_c)$ is a low-rank column submatrix of $A(I_r, :)$. These observations suggest an iterative approach of alternating between a step of fixing I_c and updating I_r and a step of fixing I_r and updating I_c. The procedure for selecting a low-rank submatrix is summarized in LRSubmatrix of Algorithm 5.3.1.

The problem of selecting row and column indices corresponding a low-rank submatrix can be expressed as the following optimization problem.

$$\underset{|I_r|=n_r, |I_c|=n_c}{\arg\min} \quad \operatorname{rank}(A(I_r, I_c), \epsilon)$$

Similarly, the approach of alternating between row and column selections can be expressed as a pair of constrained optimization problems:

$$\underset{|I_r|=n_r}{\arg\min} \operatorname{rank}(A(I_r, I_c), \epsilon) \qquad \text{and} \qquad \underset{|I_c|=n_c}{\arg\min} \operatorname{rank}(A(I_r, I_c), \epsilon).$$

5.3.3 Permuting a matrix to form low-rank off-diagonal blocks

In this section, we address the main problem of interest, which is that of permuting a matrix to form low-rank off-diagonal blocks. That is, given a matrix A, we seek permutation matrices Π_r, Π_c such that the top-right off-diagonal block $(\Pi_r A \Pi_c)(I_t, I_r)$ and the bottom-left off-diagonal block $(\Pi_r A \Pi_c)(I_b, I_l)$ are of low numerical rank, where $I_t = [1, \ldots, r_{\text{mid}}], I_b = [r_{\text{mid}} + 1, \ldots m]$ represent a split of the

Algorithm 5.3.2 Permute A to produce low-rank off-diagonal blocks

```
1: function PERMUTE(A, ϵ)
2:     Initialize I_l, I_r with a random partition of [n]
3:     loop
4:         I_t, I_b ← PERMUTEROWS(A, I_l, I_r, ϵ)
5:         I_l, I_r ← PERMUTECOLUMNS(A, I_t, I_b, ϵ)
6:     return I_t, I_b, I_t, I_b

7: function PERMUTEROWS(A, I_l, I_r, ϵ)
8:     {l_{i,I_l}} ← ROWLSCORES(A(:, I_l, ϵ))
9:     {l_{i,I_r}} ← ROWLSCORES(A(:, I_r, ϵ))
10:    for i ∈ [m] do
11:        s_i ← l_{i,I_r} / Σ_i l_{i,I_r} − l_{i,I_l} / Σ_i l_{i,I_l}
12:    I_t ← {j : s_j < 0}
13:    I_b ← {j : s_j ≥ 0}
14:    return I_t, I_b

15: function PERMUTECOLUMNS(A, I_t, I_b, ϵ)
16:    return PERMUTEROWS(A^T, I_t, I_b, ϵ)
```

row indices into top and bottom parts, and $I_l = [1, \ldots, c_{\text{mid}}], I_r = [c_{\text{mid}} + 1, \ldots m]$ represent a split of the column indices into left and right parts.

As in §5.3.2, we adopt an iterative approach in which we alternate between a step of fixing the column permutation as we update the row permutation and a step of fixing the row permutation as we update the column permutation.

Consider one step of updating the row permutation for a given column permutation. We have two potentially conflicting goals. Considering only the left part of the matrix $A(:, I_l)$, the updated row permutation should move rows of $A(:, I_l)$ with high leverage scores out of the bottom-left off-diagonal block $A(I_b, I_l)$, and

replace them with rows with lower leverage scores. Similarly, for the right part of the matrix, the updated row permutation should move rows of $A(:, I_r)$ with high leverage scores out of the top-right off-diagonal block $A(I_b, I_l)$. and replace them with rows with lower leverage scores.

We balance these two objectives by assigning a single score to each row that indicates whether we prefer to include the in the top or bottom block with the updated row permutation. For each row index i, we compute the leverage score of the i^{th} row of the left block, denoted by l_{i,I_l}, the leverage score of the i^{th} row of the right block, denoted by l_{i,I_r}. Then we define the score s_i as the difference of the normalized leverage scores as follows.

$$s_i = \frac{l_{i,I_r}}{\sum_i l_{i,I_r}} - \frac{l_{i,I_l}}{\sum_i l_{i,I_l}}$$

A large positive value of s_i indicates that the i^{th} row of the right block is much more extreme than the i^{th} row of the left block, so the updated row permutation should place the row in the bottom part of $\Pi_r A \Pi_c$. A large negative value of s_i indicates that the row should be placed in the top part. A value near zero indicates no strong preference.

Once we have computed s_i we define the updated permutation Π_r to be the permutation matrix that sorts the scores s_i in increasing order. Also we update $r_{\text{mid}} \leftarrow \#\{i : s_i < 0\}$ so that the top part of the permuted matrix $(\Pi_r A \Pi_c)(I_t, :)$ consists of the rows with score $s_i < 0$.

Algorithm 5.3.3 Hierarchical low-rank approximation of A

```
1: function COMPRESS(A, ε)
2:     U, Σ, V ← SVD(A)
3:     if the truncated SVD approximates A with sufficiently high accuracy
       and low rank then
4:         Store the truncated SVD of A
5:     else
6:         I_t, I_b, I_l, I_r ← PERMUTE(A, ε)
7:         for A_sub ∈ {A(I_t, I_l), A(I_t, I_r), A(I_b, I_l), A(I_b, I_r)} do
8:             ε_sub ← error tolerance for A_sub
9:             COMPRESS(A, ε_sub)
```

5.3.4 Hierarchical rank-structured approximation

We incorporate the methods of §5.3.3 into an algorithm for constructing a hierarchical rank-structured approximation of a matrix. To compress a matrix, we first check whether it can be approximated with a truncated SVD with sufficiently high accuracy and low rank. If so, we use the truncated SVD to represent the matrix. Otherwise, we permute the matrix with Algorithm 5.3.2, partition the matrix into four blocks, and recursively compress each of the four blocks. The process is summarized in Algorithm 5.3.3.

5.3.5 Error control and rank adaptivity

The relative error tolerance for the approximation of A is specified as an input parameter ϵ. The approximation $\tilde{A}$ must satisfy $\|\tilde{A} - A\|/\|A\| < \epsilon$. Equivalently, we can view the quantity $\epsilon_{\text{abs}} = \epsilon\|A\|$ as an absolute error tolerance, and we can compute it by multiplying ϵ with an estimate of $\|A\|$, which can be obtained, for example, by power iteration. Once we have a global absolute error tolerance ϵ_{abs},

we assign a local absolute error tolerance to each block based on its size. For block $A_b \in \mathbb{R}^{m_b \times n_b}$ of matrix $A \in \mathbb{R}^{m \times n}$, we define the local absolute error tolerance for the approximation $\tilde{A}_b$

$$\|\tilde{A}_b - A_b\| < \epsilon_{\text{abs}} \sqrt{\frac{m_b n_b}{mn}}. \tag{5.1}$$

The above condition is local to block A_b in the sense that it is independent of the other blocks. If each of the block approximations $\tilde{A}_b$ satisfies its local error tolerance condition (5.1), then the approximation of A satisfies the global error bound

$$\|\tilde{A} - A\|_F^2 = \sum_b \|\tilde{A}_b - A_b\|_F^2 < \sum_b \frac{m_b n_b}{mn} \epsilon_{\text{abs}} = \epsilon_{\text{abs}}.$$

When computing a low-rank approximation for some block, we select the rank of the approximation adaptively, choosing the smallest rank that satisfies the block's local error tolerance.

5.3.6 Admissibility

We must decide for each block whether it is admissible, meaning that it will be represented with a low-rank approximation. Instead of being approximated, a large inadmissible block is recursively subdivided into four smaller blocks, and a small inadmissible block is represented as a dense matrix. We use two admissibility conditions: a strong admissibility condition, which prioritizes accuracy, and a weak admissibility condition, which prioritizes compression rate.

For strong admissibility, we use an algebraic condition that designates a block as admissible if the error bound (5.1) is satisfied with $\tilde{A}_b$ of rank at most r, where r is an input parameter that specifies the maximum allowable rank of a

low-rank approximation. The strong admissibility guarantees satisfaction of the error tolerance, but cannot guarantee the presence of low-rank blocks.

The weak admissibility condition designates every off-diagonal block as admissible and applies the strong admissibility condition to on-diagonal blocks. As a result, every off-diagonal block is approximated with a rank-r approximation, even if such an approximation fails to satisfy (5.1) This forces a simple structure and a high degree of compression at the risk of compromising accuracy.

5.3.7 Complexity

In this section, we state asymptotic bounds on the costs of constructing and storing the compressed representation, and of computing an approximate matrix-vector product. Full derivations are provided in the Appendix. We assume that $A \in \mathbb{R}^{n \times n}$, where n is a power of 2, that admissible blocks are approximated to rank r, that large inadmissible blocks are split into four sub-blocks of equal dimensions, and that the recursive splitting proceeds up to a recursion depth of $L = \log(n/m)$, so that the smallest blocks of A are of size m-by-m. Finally, we assume a weak admissibility condition so when an inadmissible block is split into four sub-blocks, at most two of the four sub-blocks is inadmissible.

The total cost of storing the compressed representation consists of $O(nm)$ entries to store the values of the small inadmissible blocks and $O(rn \log(n/m))$ to store the low-rank approximations for the admissible blocks.

The cost for constructing the compressed representation is $O(rn \log(n/m))$ or $O(rn)$, depending on the method of randomized low-rank approximation.

	Leverage		Random		SVD	
	Error	Storage	Error	Storage	Error	Storage
K1	1E−14	0.2	1E−17	0.3	8E−1	0.2
K2	2E−3	0.2	9E−4	1.0	3E−3	0.2
K3	6E−7	0.0	6E−7	0.1	1	0.0
K4	2E−6	0.3	5E−17	1.0	4E−5	0.3
K5	2E−3	0.4	2E−17	1.0	7E−3	0.4
GNH	3E−3	0.1	3E−3	0.3	7E−3	0.1

Table 5.1 *Comparison of compression schemes applied to several test matrices.*

The cost for applying the compressed representation to a vector is $O(rn\log(n))$.

5.4 Experimental Results

In Table 5.1, we compare rank-structured approximations using permutations based on leverage scores with rank-structured approximations using random permutations and with low-rank SVD approximations. The methods are applied to several test matrices, of size 4096-by-4096: K1–K3 and K5 correspond to dense graphs, and K4 and K6 correspond optimization matrices. K6 of size 15680-by-15680. The error tolerance is set to 1E−2 with the exception of K5, for which the error tolerance is 1E−5. The rank of the truncated SVD approximation is chosen to match the storage cost used by the rank-structured approximation based on leverage scores. Error is reported as the relative error of the approximation in the spectral norm. Storage cost is reported as the ratio of the storage cost of the approximation to that of storing the entire dense matrix. The results demonstrate that rank-structured approximations are able to achieve low error and storage cost, even for matrices that are poorly approximated with a low-rank SVD. Furthermore, permutations based on

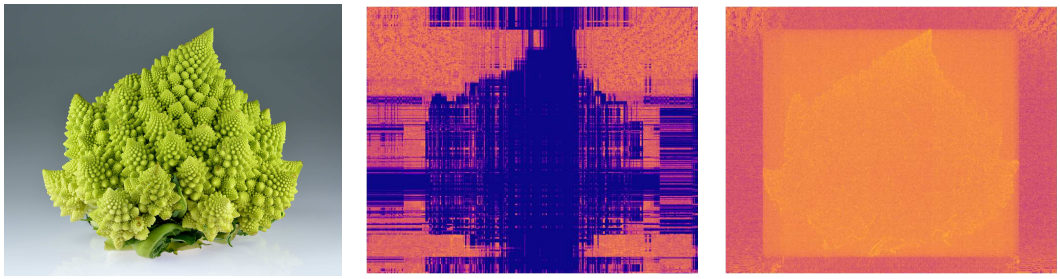


Figure 5.2 *Left: Image of Romanesco broccoli [59]. Center: Logarithm of the reconstruction error of a rank-structured approximation with leverage score permutation. Right: Logarithm of the reconstruction error of an SVD approximation. Darker regions represent relatively lower error.*

leverage scores outperform random permutations, which in some cases fail to achieve any compression at all, and simply fall back to storing the entire matrix.

Figure 5.2 visualizes the error in approximating an image of Romanesco broccoli, represented as a matrix of grayscale pixel intensities. The reconstruction error of the rank-structured approximation is very low in intricate regions that are difficult to compress. The permutation based on leverage scores identifies those regions, arranging them in small blocks that are represented exactly while the compressible regions are arranged into larger low-rank blocks. In contrast, the reconstruction error of the SVD is evenly distributed, with highest errors in the region occupied by the broccoli.

Figure 5.3 visualizes several steps in the iterative permutation of a Gaussian kernel matrix. The (i, j) entry of the matrix is defined by $a_{i,j} = \exp\left(-\|x_i - x_j\|^2/2\sigma^2\right)$, where points $\{x_i\} \subset \mathbb{R}^2$ are drawn from a standard normal distribution in two dimensions, and the bandwidth is $\sigma = 0.5$. The permutation routine only operates on entries of the matrix, not on the coordinates of the points, but the partition

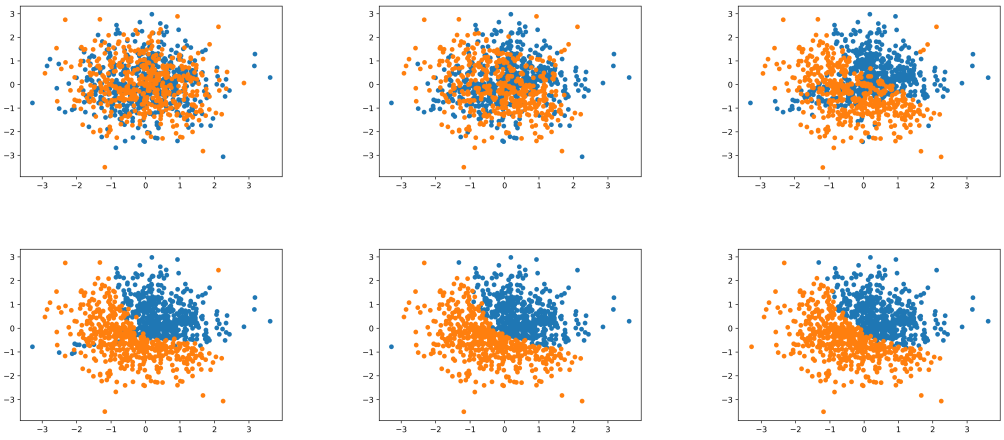


Figure 5.3 *Geometric clustering induced by iterative permutations of a Gaussian kernel matrix.*

of the row indices into top and bottom subsets (or column indices into left and right subsets) induces a clustering of the points. This image demonstrates that the induced clustering overcomes a poor initial cluster assignment and converges to a high-quality clustering within a few iterations. This example is only a demonstration as it is more practical to apply a permutation method based on geometric clustering in cases where such information is available.

5.5 Conclusions

We presented a novel method for approximating dense matrices. The method requires either the ability to sample matrix entries or the ability to perform matrix vector products. Its main features is the automatic permutation of the matrix to expose low-rank structure and its combination with state of the art methods for randomized linear algebra. The next step is to use this H-matrix approximation to

construct solvers for linear systems and spectral/singular value decomposition.

Limitations. One limitation is that the algorithm does not detect early the matrix is not compressible. This could be done by using coherence estimates or by comparing the approximations between the two levels. This is something that we will also pursue, and we believe is not hard to fix. A second limitation is that there is a large class of matrices that admit compression but are not handled by our scheme. For example a dense discrete Fourier matrix admits a telescoping fast factorization that is unrelated to an H-matrix structure. A third limitation is that our focus has been on the theoretical analysis of the algorithm. More effort is needed for a scalable implementation so that we can apply it to larger problem sizes.

5.6 Appendix

5.6.1 Kernel matrices with the covtype dataset

Here we present an analysis of the performance of RECUR for the covtype dataset [11]. The purpose of this experiment is to showcase an example of the sensitivity of RECUR in terms of a familiar parameter, the kernel bandwidth h that appears in Gaussian kernel matrices. The matrices we compress in this example are nonsymmetric, rectangular, and of size 10000×5000. In particular, the matrices are defined as $A_{ij} = a(x_i, y_j)$, where a is a Gaussian kernel parameterized by bandwidth h, and points $\{x_i\}$ and $\{y_j\}$ are mutually exclusive subsets drawn from the covtype dataset. The covtype dataset was preprocessed by normalizing each feature to have unit variance. All matrices were compressed using permutations based on leverage scores, maximum approximation rank of 20, maximum depth of 4, and using strong

Parameters	Storage	RECUR Error	SVD Error
$\epsilon = 0.1, h = 0.005$	0.1	2E−2	2E−1
$\epsilon = 0.1, h = 0.01$	0.1	4E−2	3E−1
$\epsilon = 0.1, h = 0.02$	0.2	3E−2	1E−1
$\epsilon = 0.1, h = 0.05$	0.1	3E−2	3E−2
$\epsilon = 0.1, h = 0.1$	0.0	5E−2	4E−2
$\epsilon = 0.1, h = 0.15$	0.0	9E−2	9E−2
$\epsilon = 0.01, h = 0.005$	0.1	4E−3	2E−1
$\epsilon = 0.01, h = 0.01$	0.1	3E−3	2E−1
$\epsilon = 0.01, h = 0.02$	0.3	1E−3	8E−2
$\epsilon = 0.01, h = 0.05$	0.4	2E−3	4E−3
$\epsilon = 0.01, h = 0.1$	0.1	3E−3	3E−3
$\epsilon = 0.01, h = 0.15$	0.0	5E−3	2E−3
$\epsilon = 0.001, h = 0.005$	0.1	5E−4	2E−1
$\epsilon = 0.001, h = 0.01$	0.2	2E−4	2E−1
$\epsilon = 0.001, h = 0.02$	0.4	1E−4	6E−2
$\epsilon = 0.001, h = 0.05$	0.6	1E−4	1E−3
$\epsilon = 0.001, h = 0.1$	0.2	2E−4	3E−4
$\epsilon = 0.001, h = 0.15$	0.1	2E−4	5E−5

Table 5.2 *Relative error and storage costs for RECUR and truncated SVD approximations applied to nonsymmetric, rectangular Gaussian kernel matrices defined on data drawn from the covtype dataset for a range values for error tolerance ϵ and kernel bandwidth h. The rank of the truncated SVD approximation is chosen so that the resulting approximation has equivalent storage cost to the corresponding RECUR approximation. Storage cost is reported as the ratio of the storage cost of the approximation to that of storing the entire dense matrix.*

admissibility. In Table 5.2, we compare RECUR and with RSVD (randomized SVD), for different target accuracy in terms of the error tolerance ϵ for different values of h. As a reference, the median distance between points in the set is 0.37. We observe that with the exception of large widths RECUR is on par or better with RSVD. In Figure 5.4, we show tessellations of these matrices compressed by RECUR.

The tessellations show that the algorithm produces coarser tessellations for greater values of the error tolerance ϵ, which is expected since numerical ranks decrease as ϵ increases, leading to a more lenient admissibility condition. For the case with $\epsilon = 1, h = 0.15$, the entire matrix is represented with a global low-rank approximation. The ability to adaptively determine the structure of the approximation is a feature that enables efficient compression of matrices with a range of different structures.

For smaller values of the bandwidth h, the Gaussian kernel function is narrower, and, under a suitable ordering of the matrix, the entries in the off-diagonal blocks are small, so the off-diagonal blocks are low-rank. For sufficiently large values of h, all matrix entries approach the same value, leading to a many large, low-rank blocks, or even a single global low-rank approximation. The cases with intermediate values of h are more difficult to compress. The tessellations in Figure 5.4 demonstrate the dependence of the rank structure on the bandwidth.

5.6.2 Omitted theorems and proofs

Theorem 1. Given a rank-k matrix $A \in \mathbb{R}^{m \times n}, m \geq n$, with singular value decomposition $A = U\Sigma V^T$, where $U \in \mathbb{R}^{m \times k}, \Sigma \in \mathbb{R}^{k \times k}, V \in \mathbb{R}^{n \times k}$, let $\hat{A} = A(J^c, :)$ be the

matrix obtained by removing the rows of A indexed by $J \subset [m]$. Then

$$\prod_{i=1}^{k} \sigma_i(\hat{A}) = \prod_{j=1}^{|J|} \sqrt{1 - \sigma_j^2\left(U(J,:)\right)} \prod_{i=1}^{k} \sigma_i(A),$$

where $\sigma_i(A)$ denotes the singular values of A, $\sigma_i(\hat{A})$ denotes the singular values of $\hat{A}$, and $\sigma_j^2\left(U(J,:)\right)$ denotes the singular values of the row submatrix of U indexed by J.

In particular, if $J = \{i\}$ so that $\hat{A}$ is obtained by removing only the i^{th} row from A, then

$$\prod_{i=1}^{k} \sigma_i(\hat{A}) = \sqrt{1 - l_i} \prod_{i=1}^{k} \sigma_i(A),$$

where l_i is the leverage score of the i^{th} row of A.

Proof. Observe that $A^T A = \hat{A}^T \hat{A} + A(J,:)^T A(J,:)$ and $A(J,:) = U(J,:)\Sigma V^T$. Then

$$\begin{aligned}
\prod_{i=1}^{k} \sigma_i(\hat{A}) &= \det\left(V^T \hat{A}^T \hat{A} V\right)^{\frac{1}{2}} \\
&= \det\left(V^T \left(A^T A - A(J,:)^T A(J,:)\right) V\right)^{\frac{1}{2}} \\
&= \det\left(\Sigma \left(I - U(J,:)U(J,:)^T\right) \Sigma\right)^{\frac{1}{2}} \\
&= \det\left(I - U(J,:)U(J,:)^T\right)^{\frac{1}{2}} \det(\Sigma) \\
&= \prod_{j \in J} \sqrt{1 - \sigma_j^2\left(U(J,:)\right)} \prod_{i=1}^{n} \sigma_i(A)
\end{aligned}$$

For the case $J = \{i\}$, we have $\sigma_j^2\left(U(J,:)\right) = \|U(i,:)\|^2 = l_i$. □

Theorem 2. Let $A \in \mathbb{R}^{m \times n}, m > n$ and $A = USV^T$ be the ϵ-truncated SVD of A. Let $A_{-i} \in \mathbb{R}^{(m-1) \times n}$ denote the matrix A with its i^{th} row removed, and define $\alpha = (1 - \sqrt{1 - l_i})/l_i$. Then the singular values of A_{-i} and $(I - \alpha u_{(i)} u_{(i)}^T)S$ are identical.

Proof. The singular values of A_{-i} are the square roots of the eigenvalues of $A_{-i}^T A_{-i}$, and the singular values of $(I - \alpha u_{(i)} u_{(i)}^T)S$ are the square roots of the eigenvalues of $S(I - \alpha u_{(i)} u_{(i)}^T)^2 S$. Equivalence of the singular values of A_{-i} and $(I - \alpha u_{(i)} u_{(i)}^T)S$ then follows from similarity of $A_{-i}^T A_{-i}$ and $S(I - \alpha u_{(i)} u_{(i)}^T)^2 S$:

$$\begin{aligned} A_{-i}^T A_{-i} &= \sum_{j \neq i} A_{(j)} A_{(j)}^T \\ &= A^T (I - e_i e_i^T) A \\ &= VSU^T (I - e_i e_i^T) USV^T \\ &= VS(I - u_{(i)} u_{(i)}^T) SV^T \\ &= VS(I - \alpha u_{(i)} u_{(i)}^T)^2 SV^T. \end{aligned}$$

□

Theorem 2 equates the singular values of $\hat{A}$ to the singular values of the product $(I - \alpha u_{(i)} u_{(i)}^T)S$. It is not essential in our analysis, but it is useful for conceptualizing the effect of removing a row with leverage score l_i. The image of the unit ball in $\mathbb{R}^n$ under A, $\{Ax : x \in \mathbb{R}^n, \|x\| = 1\}$, is an ellipsoid in $\mathbb{R}^m$, and the lengths of its principal semi-axes are equal to the singular values. Theorem 2 implies a relationship between the ellipsoid representing the action of A and that representing the action of $\hat{A}$. Specifically, removing a row with leverage score l_i has the effect of contracting the ellipse by a factor $1 - \sqrt{1 - l_i}$ along some direction. With a sufficiently large contraction, the ellipsoid representing $\hat{A}$ is nearly degenerate, and the $\hat{A}$ is numerically low-rank.

5.6.3 Asymptotic complexity

In this section, we derive asymptotic bounds on the costs of constructing and storing the compressed representation, and of computing an approximate matrix-vector product. We assume that $A \in \mathbb{R}^{n \times n}$, where n is a power of 2, that large inadmissible blocks are split into four sub-blocks of equal dimensions, and that the recursive splitting proceeds up to a depth of $L = \log(n/m)$, so that the smallest blocks of A are of size $m \times m$. We define a tree structure on the blocks of A, Let $b_{l,a}$ denote the number of admissible blocks at level l, $b_{l,i}$ denote the number of inadmissible blocks at level l, and $b_l = b_{l,a} + b_{l,i}$ denote the total number of blocks at level l. Finally, let $0 \leq c \leq 1$ denote an upper bound on the ratio of inadmissible blocks at any level below the root so that

$$\frac{b_{l,i}}{b_l} \leq c, \qquad 1 \leq l \leq L. \tag{5.2}$$

At level $l < L$, each inadmissible block is partitioned into four sub-blocks so that

$$b_{l+1} = 4b_{l,i}. \tag{5.3}$$

We apply (5.2), (5.3), and the observation that $b_1 \leq 4$ to establish the following bounds.

$$b_l \leq 4cb_{l-1} \leq \cdots \leq (4c)^{l-1} b_1 \leq 4^l c^{l-1}$$

$$b_{l,a} \leq b_l \leq 4^l c^{l-1}$$

$$b_{l,i} \leq cb_l \leq 4^l c^l$$

5.6.4 Storage

We first address the storage cost of the matrix approximation associated with admissible blocks. A block at level l has size $\frac{n}{2^l}$-by-$\frac{n}{2^l}$, and its rank-r approximation can be stored as a product of two matrices each with $rn/2^l$ entries. Therefore, the total storage cost of all admissible blocks S^a is bounded by

$$\begin{aligned}
S^a &= \sum_{l=1}^{L} b_{l,a} \frac{2rn}{2^l} \\
&\leq 2rn \sum_{l=1}^{L} 2^l c^{l-1} \\
&\leq \begin{cases} 2rn \sum_{l=1}^{\infty} 2^l c^{l-1} & c < \frac{1}{2} \\ 2rn \sum_{l=1}^{L} 2^l c^{l-1} & c = \frac{1}{2} \\ 2rn \sum_{l=-\infty}^{L} 2^l c^{l-1} & c > \frac{1}{2} \end{cases} \\
&= \begin{cases} \frac{4rn}{1-2c} & c < \frac{1}{2} \\ 4rnL & c = \frac{1}{2} \\ \frac{4rn(2c)^L}{2c-1} & c > \frac{1}{2} \end{cases} \\
&= \begin{cases} \frac{4rn}{1-2c} & c < \frac{1}{2} \\ 4rn \log(n/m) & c = \frac{1}{2} \\ \frac{4rn^{1+\log(2c)}}{(2c-1)m^{\log(2c)}} & c > \frac{1}{2} \end{cases}
\end{aligned}$$

The remaining storage cost consists of inadmissible blocks at level L, whose m-by-m matrix blocks are stored exactly at a total cost S^i bounded by

$$S^i = b_{L,i} m^2 \leq (4c)^L m^2 = \left(\frac{n}{m}\right)^{1+\log(2c)} m^2.$$

The total storage cost of the approximation is given by $S^a + S^i$. Assuming c and m are constants independent of the problem size n, the bound on total storage cost is linear in n for $c < \frac{1}{2}$, quasilinear in n for $c = \frac{1}{2}$, and superlinear in n for $c > \frac{1}{2}$.

5.6.5 Work to Construct the Approximation

The analysis for work complexity is very similar to the analysis for storage. The only difference is that instead of counting $2rn/2^l$ units of storage per block at level l, we count $T_{l,a}$ time to compress the block. The value of the time cost depends on the method for randomized low-rank approximation low-rank approximation. With projection-based randomized approximation using structured test matrices [46], $T_{l,a}$ is $O(r(n/2^l)\log(n/2^l))$, and with sampling-based randomized approximation [28], $T_{l,a}$ is $O(rn/2^l)$. A similar analysis as in §5.6.4 shows that the cost for constructing the compressed representation is $O(rn\log(n/m))$ or $O(rn)$, depending on the method of randomized low-rank approximation.

5.6.6 Work for an Approximate Matrix-Vector Product

Application of the matrix approximation to a vector is essentially a blocked matrix-vector product with admissible blocks represented by low-rank approximations. In the matrix-vector product, each entry stored in the approximation is involved in two floating-point operations, so the total number of operations is twice the number of stored entries (see §5.6.4).

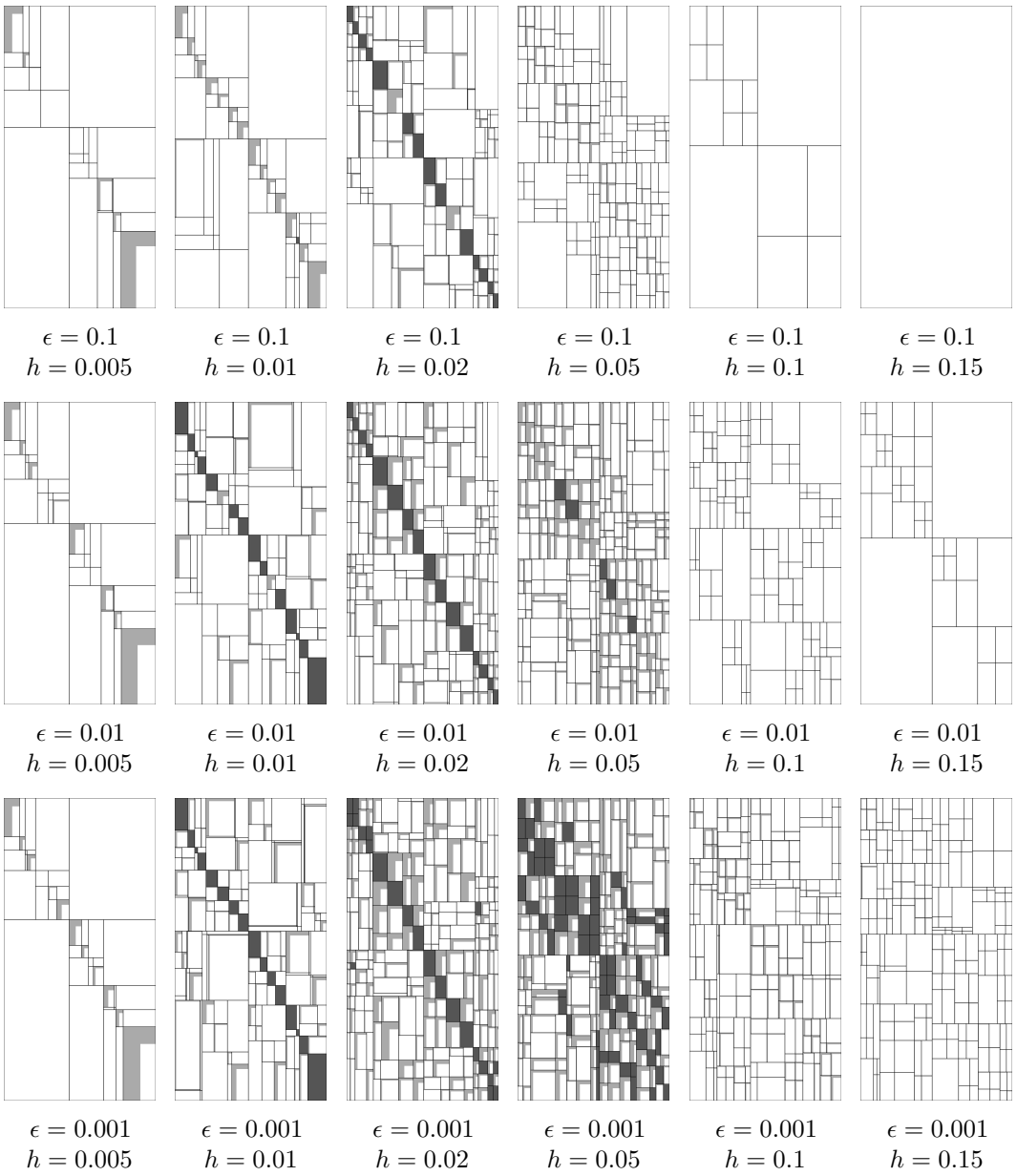


Figure 5.4 *Tessellations of nonsymmetric, rectangular Gaussian kernel matrices defined on data drawn from the covtype dataset for a range values for error tolerance ϵ and kernel bandwidth h. Each column corresponds to a fixed bandwidth. Each row corresponds to a fixed error tolerance. Blocks shown in dark gray are stored as dense blocks since they cannot be represented with low-rank approximations that are of sufficiently low rank and high accuracy. Low-rank blocks are filled in with light gray tiles whose size indicates the size of the factors in the low-rank approximation (blocks that appear to be completely empty in fact have approximations with very low rank).*

Chapter 6

Nonsymmetric Algebraic FMM with Application to Combined Field Integral Equations[1]

[1]The content in this chapter was first presented in *Levitt, J. L., Boman, E. G., Rajamanickam, S., & Biros, G. (2018). Nonsymmetric algebraic FMM with application to combined field integral equations. In Center for Computing Research Summer Proceedings 2018, 2018. Technical Report SAND2019-5093R.*, a work in collaboration with Erik Boman, Sivasankaran Rajamanickam, and George Biros.

Boundary integral equation methods are well-established tools for simulating electromagnetic scattering. One of their shortcomings is that, upon discretization, they result in a linear system with a *dense matrix* K. This problem is well understood and has been addressed with Fast Multipole Methods (FMM), which date back to the early 90s. By introducing a controllable approximation error, such methods turn an $O(N^2)$ matrix-vector product with K to an $O(N)$ matrix-vector product under suitable conditions. Classical FMM schemes are based on analytic expansion of the underlying Green's function and they essentially sparsify or *compress* K. From a software implementation point of view, they are rather intrusive and require careful integration with the discretization code. This makes their use a bit difficult, especially if one wants to experiment with different discretization schemes for the boundary integral equation. An alternative is to use algebraic methods to compress K, the so-called *H-matrix* methods.

Here we test one such method, the geometry-aware variant of GOFMM, on a low-frequency scattering problem discretized using the combined field integral formulation and the method of moments (essentially a Petrov-Galerkin collocation scheme). This scheme gives rise to a non-Hermitian matrix. Current H-matrix methods, including GOFMM, require row and column permutations of the matrix in order to compress K and importance sampling to construct low-rank approximations. To construct these permutations and sample rows and columns we need to know the coordinates of the collocation points. If K were SPD, we could apply the geometry-oblivious variant of GOFMM, which does not require such coordinates. But unfortunately K is not symmetric. In this work we extend the geometry-aware

variant of GOFMM to support nonsymmetric complex matrices and test it on a test matrix generated by the Eiger library developed at Sandia National Laboratories. Since geometric information was unavailable, we use the natural ordering of the matrix (no permutations) and uniform random sampling. Under these conditions, the Eiger matrix K compresses well up to a relative error of 3E-4, which is insufficient for some right-hand sides since the matrix has condition number of 7E-8. Further compression of K may be possible with a better permutation and higher quality sampling that leverage geometric information. To demonstrate the possibility of attaining arbitrary accuracies when the coordinates of the collocation points are available, we also test GOFMM on a synthetic problem that results in a non-symmetric complex matrix related to low-frequency scattering from point scatterers in three dimensions.

6.1 Introduction

Given a perfect electric conductor and an incident electric field, we seek the induced current on the surface of the conductor due to the incident field, which can then be used to find the scattered electric field. The induced current is governed by the combined field integral equation (CFIE) [85], which is a weighted sum of the electric field integral equation (EFIE) and magnetic field integral equation (MFIE). The problem can be solved with either the EFIE or MFIE separately, and while they produce symmetric moment matrices, such approaches encounter problems with homogeneous solutions corresponding to interior resonance entering into the solution. Use of the CFIE avoids such issues at the cost of losing symmetry of the discretized integral equation operator K.

The problem is discretized with the method of moments using Rao-Wilton-Glisson (RWG) basis functions [89] to yield a linear system $Kx = b$, where $K \in \mathbb{C}^{n \times n}$ and $x, b \in \mathbb{C}^n$. The discretized problem is generated using Eiger [52, 57], a parallel code for simulating frequency-domain electromagnetic scattering. There are a number of concerns in formulating the problem relating to the discretization, efficient computation of moment matrix entries, and avoiding singularities, all of which are handled in Eiger and not discussed in this paper.

The matrix K is dense and the system is typically solved using a Krylov method such as GMRES [91], in which the main computational cost is in applying K to a vector. In order to accelerate the algorithm, we construct a hierarchical low-rank approximation (a *compression*) $\tilde{K}$ of K such that $\|\tilde{K} - K\| < \|K\|$. The matrix compression algorithm is an algebraic variant of the Fast Multipole Method (FMM) which takes advantage of hierarchical low-rank structure in the matrix to construct a compressed representation of the matrix with $\mathcal{O}(n \log n)$ work that can be applied to a vector with $\mathcal{O}(n)$ work under suitable conditions on the problem. For certain problems, the compressed matrix-vector product delivers significant speedup over the $\mathcal{O}(n^2)$ uncompressed dense matrix-vector product, leading to a faster overall runtime, even accounting for compression time.

There are a number of related works that use FMM-like algorithms for the fast solution of electromagnetic scattering problems in the engineering literature. Many methods [42, 93–95] have appeared that aim at accelerating matrix-vector products. For a review of nearly optimal and highly accurate methods for frequency-domain FMM methods, see [22]. For a state of the art discretization of integral equations for

electromagnetics, see [5].

As discussed, there are many efficient techniques for accelerating matrix-vector products with K. However, the majority of existing methods require tight integration of the discretization and kernel evaluation codes with the acceleration scheme. Few schemes can be used in black-box fashion and, to our knowledge, no existing software scales to distributed memory architectures. In this project, we aim at exploring black-box schemes that scale on distributed memory architectures. There are numerous works in the literature relating to hierarchical low-rank approximation of rank-structured matrices, and so we only mention those that are most closely related. We implement our scheme using the Geometry-Oblivious Fast Multipole Method (GOFMM) library [107,110]. One of the main contributions of GOFMM is the generalization of the ASKIT FMM [68] to general symmetric positive-definite matrices that do not necessarily represent interactions between pairs of points. However, we do not attempt to generalize the geometry-oblivious features of GOFMM for two reasons: the problem setting of this work is geometry-aware, and there is no guarantee of symmetry and positive-definiteness of K, a requirement for the geometry-oblivious features. Instead, we extend the geometry-aware FMM, as implemented in the GOFMM library, to nonsymmetric matrices.

6.2 Hierarchical Iterative Solver

For matrix $K \in \mathbb{C}^{n \times n}$ and right hand side $b \in \mathbb{C}^n$, in order to solve the linear system $Kx = b$, our first goal is to construct a hierarchical matrix approximation $\tilde{K}$ that satisfies $\|\tilde{K} - K\| < \epsilon_c$, where ϵ_c is a user-defined absolute error tolerance

for the compressed operator. This compressed operator is then used to efficiently compute matrix-vector products inside an iterative solver.

Depending on the matrix and error tolerance, such an approximation may not exist Applicability of this approach requires a matrix that can be permuted to reveal hierarchical low-rank structure and an appropriate choice of error tolerance. Depending on the matrix, an error tolerance that is too small results in very little compression, leading to poor performance.

We begin this section with the hierarchical clustering of points, which defines a reordering of the matrix that seeks to reveal low-rank structure in the off-diagonal blocks. Next, we review the Interpolative Decomposition (ID), the low-rank matrix factorization that we use to approximate off-diagonal blocks. We then describe the two main phases of the algorithm, compression and evaluation, and follow with a comparison with the symmetric algorithm of [107]. Finally, we discuss the use of the compressed operator in accelerating GMRES.

6.2.1 Clustering/reordering

The compressibility of matrix K depends on a proper ordering of points $\{x_i\}$. Therefore, we begin by reindexing the points (reordering the matrix) such that points that are nearby in space have nearby indices. This is done by creating a hierarchical clustering of the points represented by a balanced binary tree. The root node of the tree is assigned the full set of points. The set of points assigned to a node is split into two balanced clusters, and each cluster is assigned to a child node. This process is continued at each level of the tree until the number of points assigned to the leaf

nodes fall within some prescribed maximum leaf size. We split sets of points by their projections along an axis which heuristically estimates the axis along which the points have maximal variation. This method scales well with the dimensionality d of the data [72]. The hierarchical clustering algorithm is outlined in Algorithm 6.2.1.

Algorithm 6.2.1 Construct a cluster tree for points $\{x_i\}$

procedure CLUSTER_TREE($\{x_i\}$)
  α =NEW_NODE($\{x_i\}$)
  if $|\{x_i\}| >$ `max_leaf_size` **then**
    $\bar{x} = \frac{1}{|\{x_i\}|} \sum x_i$ ▷ Compute the centroid
    $x_p = \arg\max_i \|x_i - \bar{x}\|$ ▷ Find the point farthest from $\bar{x}$
    $x_q = \arg\max_i \|x_i - x_p\|$ ▷ Find the point farthest from x_p
    `med` = MEDIAN($\{(x_i, x_p - x_q)\}_i$) ▷ Split points by projections along $x_p - x_q$
    α`.left_child` = CLUSTER_TREE($x_i : (x_i, x_p - x_q) \leq$ `med`)
    α`.right_child` = CLUSTER_TREE($x_i : (x_i, x_p - x_q) >$ `med`)
  return α

6.2.2 Interpolative Decomposition

Consider the task of constructing a low rank approximation of $K_{\alpha\beta} \in \mathbb{C}^{n_\alpha \times n_\beta}$, the block of K corresponding to interactions between tree nodes α and β.

A rank-r *column ID* of a matrix $K_{\alpha\beta}$ takes the form

$$K_{\alpha\beta} \approx K_{\alpha\beta} S^c_{\alpha\beta} P^c_{\alpha\beta}, \tag{6.1}$$

where $S^c_{\alpha\beta} \in \mathbb{R}^{n_\beta \times r}$ is a column submatrix of a permutation matrix and $P^c_{\alpha\beta} \in \mathbb{C}^{r \times n_\beta}$. The product $K_{\alpha\beta} S^c_{\alpha\beta}$ is a column submatrix of $K_{\alpha\beta}$, and matrix $S^c_{\alpha\beta}$ may be viewed as selecting a subset of the columns of $K_{\alpha\beta}$ to use as a basis. The selected columns are referred to as *skeleton columns*. Then the entire decomposition may be viewed as

an expansion of each column of $K_{\alpha\beta}$ in the column basis $K_{\alpha\beta}S^c_{\alpha\beta}$, where matrix $P^c_{\alpha\beta}$ encodes the expansion coefficients. Definitions of the ID in other works often use two terms rather than three, combining the product $K_{\alpha\beta}S^c_{\alpha\beta}$ into a single term. Writing the decomposition with three terms is equivalent and leads to clearer exposition in following sections.

Similarly, a rank-r *row ID* of $K_{\alpha\beta}$ takes the form

$$K_{\alpha\beta} \approx P^r_{\alpha\beta} S^r_{\alpha\beta} K_{\alpha\beta}, \tag{6.2}$$

where $S^r_{\alpha\beta} \in \mathbb{R}^{r \times n_\alpha}$ is a row submatrix of a permutation matrix and $P^r_{\alpha\beta} \in \mathbb{C}^{n_\alpha \times r}$.

A column ID and row ID of $K_{\alpha\beta}$ may be combined to form a *two-sided ID* by a simple substitution:

$$K_{\alpha\beta} \approx K_{\alpha\beta} S^c_{\alpha\beta} P^c_{\alpha\beta} \approx P^r_{\alpha\beta} S^r_{\alpha\beta} K_{\alpha\beta} S^c_{\alpha\beta} P^c_{\alpha\beta} \tag{6.3}$$

Storage of this decomposition only requires $2rn$ values for $P^c_{\alpha\beta}$ and $P^r_{\alpha\beta}$, and $2r$ indices to encode the columns/rows selected by $S^c_{\alpha\beta}$ and $S^r_{\alpha\beta}$.

The algorithm for constructing an ID is given in Algorithm 6.2.2. Given some overall error tolerance ϵ_c, the goal is to construct an ID for submatrix $K_{\alpha\beta}$ with an adaptively-chosen rank r that is less than or equal to the maximum allowed rank s such that the approximation satisfies some accuracy requirement. If such an approximation cannot be found, the matrix block is deemed incompressible. Entries belonging to an incompressible block may still be approximated if they also belong to a different block that is compressible. Otherwise, they are computed exactly. The majority of the computational cost comes from a rank-revealing QR factorization

(RRQR) and a triangular solve. In order to reduce the computational cost, we use a sampled ID, in which only a small subset of the columns or rows are used in computing the results S, P [72]. The sample submatrix is selected using using neighbor-based importance sampling [67].

Algorithm 6.2.2 Construct a sampled Interpolative Decomposition of matrix $K_{\alpha\beta}$

procedure COLUMN_ID($K_{\alpha\beta}$)
  γ =IMPORTANCE_SAMPLE(α)
  Q, R, Π =RRQR($K_{\alpha\gamma}$)
  $r = \min\left(\{i : \|R[i:, i:]\| < \frac{\sqrt{|\alpha||\gamma|}}{n}\epsilon_c\}\right)$
  if $r > s$ **then**
    return `fail`
  $S = \Pi[:, :r]$
  $R_{11} = R[:r, :r]$
  $R_{12} = R[:r, r:]$
  $P = \begin{bmatrix} I & R_{11}^{-1}R_{12} \end{bmatrix} \Pi^{-1}$
  return S, P
procedure ROW_ID($K_{\alpha\beta}$)
  S, P = COLUMN_ID($K_{\alpha\beta}^T$)
  return S^T, P^T

6.2.3 Compression

Constructing interpolative decompositions independently for each block $K_{\alpha\beta}$ would be inefficient and would lead to inefficiency in the evaluation phase. Instead, we construct these approximations using a recursive approach based on the cluster tree. For each tree node β, we define the *column off-diagonal block of* β to be submatrix $K_{\bar{\beta}\beta}$, and the *row off-diagonal block of* β to be submatrix $K_{\beta\bar{\beta}}$, where $\bar{\beta} = \{1, \ldots, n\} \setminus \beta$ denotes the set of points not in β. During compression, we

construct approximations for these column and row off-diagonal blocks. These approximations are later used during evaluation to approximate a compressible block $K_{\alpha\beta}$ with a two-sided ID using the row ID of $K_{\alpha\bar{\alpha}}$ and the column ID of $K_{\bar{\beta}\beta}$.

For a leaf node β, we define the *column skeletonization* of β to be a low-rank approximation of $\bar{\beta}$ using the ID

$$K_{\bar{\beta}\beta} \approx K_{\bar{\beta}\beta} S_{\bar{\beta}\beta} P_{\bar{\beta}\beta}. \tag{6.4}$$

We define the *row skeletonization* of β similarly:

$$K_{\beta\bar{\beta}} \approx P_{\beta\bar{\beta}} S_{\beta\bar{\beta}} K_{\beta\bar{\beta}}. \tag{6.5}$$

For an internal node α, we only compute a column ID using columns that are skeletons of the children of α, That is, a column ID is computed for the submatrix $K_{\bar{\alpha}\alpha'}$, where $\alpha' \subset \alpha$ denotes the set of skeleton columns of the child nodes $\mathtt{l}, \mathtt{r}$ of α:

$$K_{\bar{\alpha}\alpha'} \approx K_{\bar{\alpha}\alpha'} S_{\bar{\alpha}\alpha'} P_{\bar{\alpha}\alpha'}, \qquad \text{where } K_{\bar{\alpha}\alpha'} = K_{\bar{\alpha}\alpha} \begin{bmatrix} S_{\bar{\mathtt{l}}\mathtt{l}} & \\ & S_{\bar{\mathtt{r}}\mathtt{r}} \end{bmatrix}. \tag{6.6}$$

This will be combined with the IDs of the children of α to implicitly define an ID for the entire off-diagonal block of α. The skeletons satisfy the *nesting property*: the skeleton columns of α are a subset of the skeleton columns of its children. The nesting property is important for achieving $\mathcal{O}(n)$ work complexity in the evaluation stage.

Due to the nesting property, we can use (6.6) together with the column IDs

of the child nodes to implicitly define a column ID of the full block $K_{\bar{\alpha}\alpha}$:

$$\begin{aligned} K_{\bar{\alpha}\alpha} &\approx K_{\bar{\alpha}\alpha} \begin{bmatrix} S_{\bar{\mathtt{l}}\mathtt{l}} & \\ & S_{\bar{\mathtt{r}}\mathtt{r}} \end{bmatrix} \begin{bmatrix} P_{\bar{\mathtt{l}}\mathtt{l}} & \\ & P_{\bar{\mathtt{r}}\mathtt{r}} \end{bmatrix} \\ &\approx K_{\bar{\alpha}\alpha'} \begin{bmatrix} P_{\bar{\mathtt{l}}\mathtt{l}} & \\ & P_{\bar{\mathtt{r}}\mathtt{r}} \end{bmatrix} \\ &\approx K_{\bar{\alpha}\alpha'} S_{\bar{\alpha}\alpha'} P_{\bar{\alpha}\alpha'} \begin{bmatrix} P_{\bar{\mathtt{l}}\mathtt{l}} & \\ & P_{\bar{\mathtt{r}}\mathtt{r}} \end{bmatrix} \\ &\approx K_{\bar{\alpha}\alpha} \begin{bmatrix} S_{\bar{\mathtt{l}}\mathtt{l}} & \\ & S_{\bar{\mathtt{r}}\mathtt{r}} \end{bmatrix} S_{\bar{\alpha}\alpha'} P_{\bar{\alpha}\alpha'} \begin{bmatrix} P_{\bar{\mathtt{l}}\mathtt{l}} & \\ & P_{\bar{\mathtt{r}}\mathtt{r}} \end{bmatrix} \end{aligned}$$

This defines a column ID of the full off-diagonal block $K_{\bar{\alpha}\alpha} = S_{\bar{\alpha}\alpha} P_{\bar{\alpha}\alpha}$ with telescoping expressions for $S_{\bar{\alpha}\alpha}$ and $P_{\bar{\alpha}\alpha}$:

$$S_{\bar{\alpha}\alpha} = \begin{bmatrix} S_{\bar{\mathtt{l}}\mathtt{l}} & \\ & S_{\bar{\mathtt{r}}\mathtt{r}} \end{bmatrix} S_{\bar{\alpha}\alpha'}, \qquad P_{\bar{\alpha}\alpha} = P_{\bar{\alpha}\alpha'} \begin{bmatrix} P_{\bar{\mathtt{l}}\mathtt{l}} & \\ & P_{\bar{\mathtt{r}}\mathtt{r}} \end{bmatrix} \tag{6.7}$$

We never explicitly form $P_{\bar{\alpha}\alpha}$, but instead use the telescoping expression to apply $P_{\bar{\alpha}\alpha}$ to a vector during evaluation. The row skeletonization of interior node α is carried out analogously, using the row skeletons of the children of α.

Compression consists of skeletonizing each node in the cluster tree. The tree nodes are skeletonized following a post-order tree traversal in order to satisfy data dependencies. The compression algorithm is outlined in Algorithm 6.2.3.

6.2.4 Evaluation

Once the compressed representation has been constructed, we seek to apply the compressed matrix to *weight vector* w to compute the *product vector* $u \approx Kw$. We refer to this step as *evaluation*. Suppose the block $K_{\alpha\beta}$ may be approximated with a two-sided ID as in (6.3) using the skeletonizations constructed in §6.2.3. We seek to apply it to the appropriate subvector w_β and add the contribution $K_{\alpha\beta} w_\beta$ to

Algorithm 6.2.3 Compress the matrix by skeletonizing each tree node

procedure COMPRESS()
  SKELETONIZE(`root`)
procedure SKELETONIZE(α)
  if IS_LEAF_NODE(α) **then**
    $S_{\bar{\alpha}\alpha}, P_{\bar{\alpha}\alpha}$ = COLUMN_ID($K_{\bar{\alpha}\alpha}$)
    $S_{\alpha\bar{\alpha}}, P_{\alpha\bar{\alpha}}$ = ROW_ID($K_{\alpha\bar{\alpha}}$)
  else
    SKELETONIZE(α`.left_child`)
    SKELETONIZE(α`.right_child`)
    α' = α`.left_child.skeleton` $\cup$ α`.right_child.skeleton`
    $S_{\bar{\alpha}\alpha'}, P_{\bar{\alpha}\alpha'}$ = COLUMN_ID($K_{\bar{\alpha}\alpha'}$)
    $S_{\alpha'\bar{\alpha}}, P_{\alpha'\bar{\alpha}}$ = ROW_ID($K_{\alpha'\bar{\alpha}}$)

u_α. This is done in three steps: we first compute the *skeleton weights* $\tilde{w}_\beta = P_{\bar{\beta}\beta} w_\beta$, then the *skeleton products* $\tilde{u}_\alpha = S_{\alpha\bar{\alpha}} K_{\alpha\beta} S_{\bar{\beta}\beta} \tilde{w}_\beta$, and finally add to the products $u_\alpha \mathrel{+}= P_{\alpha\bar{\alpha}} \tilde{u}_\alpha$. Computing the skeleton products consists of a single matrix-vector product using the submatrix of $K_{\alpha\beta}$ selected by $S_{\alpha\bar{\alpha}}$ and $S_{\bar{\beta}\beta}$, rather than separately applying the three matrices.

We make a few observations that lead to an efficient implementation. First, the matrix blocks whose columns are indexed by β all belong to the column off-diagonal block of β, so they share the same skeleton weights $P_{\bar{\beta}\beta} w_\beta$. Similarly, the matrix blocks whose rows are indexed by α all apply the same matrix $P_{\alpha\bar{\alpha}}$ to their skeleton products, so we may sum their skeleton products before applying $P_{\alpha\bar{\alpha}}$ and apply $P_{\alpha\bar{\alpha}}$ a single time to the summed skeleton products. With this perspective, we define the skeleton weights of node β to be the skeleton weights associated with all sub-blocks of its column off-diagonal blocks, and the skeleton products of α to be the skeleton products summed over the sub-blocks of its row off-diagonal block.

We also observe that as a consequence of the nesting property and telescoping expression (6.7), there is significant overlap in the computation of an interior node's skeleton products and the computation of its children's skeleton products. That is, once the skeleton products of the children are known, the skeleton products of the parent may be computed with a single additional matrix-vector product. Similarly, in computing products, rather than applying the full telescoping expression to its skeleton products, a parent node α may apply a single matrix-vector product and "pass down" the partial result $P_{\alpha'\tilde{\alpha}}\tilde{u}_\alpha$ to its children, which add the appropriate parts to their skeleton products.

The evaluation algorithm is outlined in Algorithm 6.2.4. In order to satisfy data dependencies, the computation of skeleton weights is structured as a post-order tree traversal, the computation of skeleton products is structured as a visitation of tree nodes in any order, and the computation of products is structured as a pre-order tree traversal. Under suitable assumptions on the low-rank structure of K, the computational complexity of evaluation is $\mathcal{O}(n)$. Detailed cost breakdown along with some other discussion can be found in [107].

6.2.5 Accelerated GMRES

6.2.5.1 Analysis

To accelerate GMRES, we construct a compressed operator $\tilde{K}$ and substitute it for the uncompressed operator K throughout GMRES. This leads to two different kinds of residual: we define the *relative residual of x in the compressed operator* to be $\|\tilde{K}x - b\|/\|b\|$ and the *relative residual of x in the uncompressed operator* or

Algorithm 6.2.4 Apply the compressed matrix to a vector w.

procedure EVALUATE(w)
  COMPUTE_SKELETON_WEIGHTS(`root`)
  COMPUTE_SKELETON_PRODUCTS()
  COMPUTE_PRODUCTS(`root`)

procedure COMPUTE_SKELETON_WEIGHTS(α)
  if IS_LEAF_NODE(α) **then**
    $\tilde{w}_\alpha = P_{\tilde{\alpha}\alpha} w_\alpha$
  else
    $\mathtt{l} = \alpha$`.left_child`
    $\mathtt{r} = \alpha$`.right_child`
    COMPUTE_SKELETON_WEIGHTS(`l`)
    COMPUTE_SKELETON_WEIGHTS(`r`)
    $\tilde{w}_\alpha = P_{\tilde{\alpha}\alpha'} \begin{bmatrix} \tilde{w}_\mathtt{l} \\ \tilde{w}_\mathtt{r} \end{bmatrix}$

procedure COMPUTE_SKELETON_PRODUCTS()
  for $\alpha \in$ `tree` **do**
    $\tilde{u}_\alpha = \sum_{\beta \in \alpha\mathtt{.interaction\_list}} S_{\alpha\bar{\alpha}} K_{\alpha\beta} S_{\bar{\beta}\beta} \tilde{w}_\beta$

procedure COMPUTE_PRODUCTS(α)
  if IS_LEAF_NODE(α) **then**
    $u_\alpha = P_{\alpha\bar{\alpha}} \tilde{u}_\alpha$
  else
    $\mathtt{l} = \alpha$`.left_child`
    $\mathtt{r} = \alpha$`.right_child`
    $\begin{bmatrix} \tilde{w}_\mathtt{l} \\ \tilde{w}_\mathtt{r} \end{bmatrix}$ += $P_{\alpha'\bar{\alpha}} \tilde{u}_\alpha$
    COMPUTE_PRODUCTS(`l`)
    COMPUTE_PRODUCTS(`r`)

true residual to be $\|Kx - b\|/\|b\|$. These two residuals may be different, and in the accelerated algorithm, only the residual in the compressed operator is available. The following theorem gives upper bounds for the solution error and the true residual in terms of the residual in the compressed operator, the error in the compression, and the condition number of K.

Theorem 3. For matrix K and right hand side b, let x be the solution of $Kx = b$, and $\tilde{x}$ be an approximate solution of $\tilde{K}\tilde{x} \approx b$ with residual $r = \tilde{K}\tilde{x} - b$. Assume K is nonsingular, $\|\tilde{K} - K\| < \epsilon_c\|K\|$, $\|r\| < \epsilon_r\|b\|$, and $\epsilon_c \texttt{cond}(K) < 1$. Then

$$\frac{\|\tilde{x} - x\|}{\|x\|} < \frac{(\epsilon_r + \epsilon_c)\,\texttt{cond}(K)}{1 - \epsilon_c \texttt{cond}(K)}$$

and

$$\frac{\|K\tilde{x} - b\|}{\|b\|} < \epsilon_r + \epsilon_c \texttt{cond}(K) \frac{1 + \epsilon_r \texttt{cond}(K)}{1 - \epsilon_c \texttt{cond}(K)}.$$

Proof. First, we bound $\|\tilde{x}\|/\|x\|$. Define $\delta K = \tilde{K} - K$. It follows from the definitions that

$$\begin{aligned} (K + \delta K)\,\tilde{x} &= b + \delta b \\ \left(I + K^{-1}\delta K\right)\tilde{x} &= x + K^{-1}r \\ \tilde{x} &= \left(I + K^{-1}\delta K\right)^{-1}\left(x + K^{-1}r\right) \end{aligned}$$

Matrix $I + K^{-1}\delta K$ is invertible since $\|K^{-1}\delta K\| < \epsilon_c \texttt{cond}(K) < 1$. Taking norms and using the Neumann series bound of $\left(I + K^{-1}\delta K\right)^{-1}$,

$$\begin{aligned} \|\tilde{x}\| &\leq \frac{\|x\| + \|K^{-1}r\|}{1 - \|K^{-1}\delta K\|} \\ \frac{\|\tilde{x}\|}{\|x\|} &\leq \frac{1 + \epsilon_r \texttt{cond}(K)}{1 - \epsilon_c \texttt{cond}(K)} \end{aligned} \tag{6.8}$$

To bound $\|\tilde{x} - x\|/\|x\|$, we observe from the definitions that $\tilde{x} - x = K^{-1}r - K^{-1}\delta K\tilde{x}$. Taking norms,

$$\begin{aligned}\frac{\|\tilde{x} - x\|}{\|x\|} &\leq \frac{\|K^{-1}r\|}{\|x\|} + \frac{\|K^{-1}\delta K\tilde{x}\|}{\|x\|} \\ &\leq \epsilon_r \texttt{cond}(K) + \epsilon_c \texttt{cond}(K)\frac{\|\tilde{x}\|}{\|x\|}\end{aligned}$$

Combining this with 6.8 yields the first inequality.

To bound $\|K\tilde{x} - b\|/\|b\|$, we observe from the definitions that $K\tilde{x} = r - \delta K\tilde{x}$. Taking norms,

$$\begin{aligned}\frac{\|K\tilde{x} - b\|}{\|b\|} &\leq \frac{\|r\|}{\|b\|} + \frac{\|\delta K\tilde{x}\|}{\|b\|} \\ &\leq \epsilon_r + \epsilon_c \frac{\|K\|\|x\|}{\|b\|}\frac{\|\tilde{x}\|}{\|x\|} \\ &< \epsilon_r + \epsilon_c \texttt{cond}(K)\frac{\|\tilde{x}\|}{\|x\|}.\end{aligned}$$

Combining this with 6.8 yields the second inequality.

□

6.2.5.2 Trilinos Implementation

Trilinos [47] is a large open-source software project consisting of a collection of packages for solving large-scale numerical problems arising in computational science and engineering. The Belos package [7] provides a number of iterative solvers, including an implementation of GMRES, which we combine with the hierarchical compression algorithm to create an accelerated linear solver. Trilinos is designed to support modularity of components via the use of abstract interfaces. In particular, the GMRES implementation operates on abstract classes for operator and multivector

types. Swapping out different implementations of the abstract classes does not require any changes to the solver, and the user is free to implement any operator and multivector types that satisfy the abstract interfaces. The operator interface is defined in the Belos source file `BelosOperator.hpp` and the multivector interface in `BelosMultiVec.hpp`. Simple unoptimized example implementations are located in `MVOPTester/` in the Belos test directory.

We implement a new operator to support hierarchical approximation of operator K. The new operator type creates a compressed representation of a given matrix when its constructor is called. Subsequent calls to apply the operator execute the compressed matrix-vector product of Algorithm 6.2.4.

Compared to the reference algorithm that uses an uncompressed matrix-vector product, the accelerated algorithm has an additional startup cost associated with compression. In order to achieve speedup in total runtime, the savings due to the compressed matrix-vector product must overcome the cost of compression. For suitable problems, the compressed matrix-vector product is asymptotically faster, even accounting for compression, and therefore will be faster given a large enough problem. We empirically compare the accuracy and convergence of the accelerated algorithm against the reference algorithm in §6.3.

6.3 Experiments

The compression and evaluation algorithms of §6.2 were implemented in the GOFMM codebase and integrated with the GMRES solver of Trilinos. Experiments were conducted on a server equipped with dual Intel Xeon Platinum 8160 CPUs for

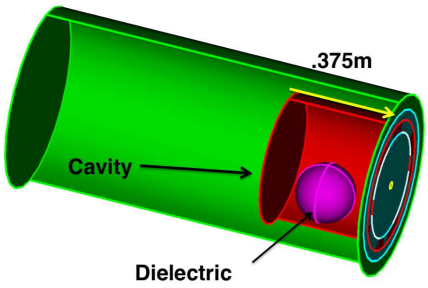


Figure 6.1 *Geometry of the thin-slot EM scattering problem [56].*

a total of 48 physical cores and 192 GB of memory. Though the GOFMM library includes support for distributed-memory parallelism, we run these experiments only on a single compute node, using OpenMP for shared-memory parallelism.

6.3.1 Performance Impact of Nonsymmetry

We first examine the performance overhead incurred by the additional work for nonsymmetric compression. The nonsymmetric algorithm computes two approximations for each node β: a column ID of column off-diagonal block $K_{\bar{\beta}\beta}$ and a row ID of row off-diagonal block $K_{\beta\bar{\beta}}$. The symmetric algorithm only has to compute one of these and takes the transpose to get the other. We compare the results for the symmetric and nonsymmetric algorithms both applied to a symmetric problem. In these experiments, the operator is a kernel matrix $K \in \mathbb{R}^{n\times n}, n = 50000$, with entries defined by $K_{ij} = \mathcal{K}(x_i, x_j)$, where $\mathcal{K}$ is a Gaussian kernel function with bandwidth $h = 0.05$ and points $\{x_i\}$ are randomly drawn from a uniform distribution over the unit hypercube in four dimensions. Matrices such as this one appear in kernel methods in machine learning and their hierarchical rank structure has been

demonstrated, for example in citemarch2017far. The distribution of points and kernel bandwidth parameter are chosen so that the matrix is compressible, but not trivially so.

Table 6.1 shows performance results of the symmetric and nonsymmetric algorithms for various settings of the maximum approximation rank and leaf size.

When the approximation rank and leaf size are large, the cost of compression is dominated by the low-rank approximation of off-diagonal blocks, and the nonsymmetric compression takes roughly twice as long as the symmetric compression. As the approximation rank and leaf size are decreased, the low-rank approximation represents a smaller portion of the total compression time, and the penalty for nonsymmetric compression is much less than a factor of two. Setting these parameters too small results in poor performance due to increased costs of other steps. For the following experiments we use an approximation rank and leaf size of 256 unless otherwise noted.

While the nonsymmetric algorithm requires additional work for compression, it does not require any additional work for evaluation of the matrix-vector product. As expected, we observe no significant difference in evaluation times between the two algorithms.

6.3.2 Electromagnetic scattering on thin-slot geometry

Next, we consider a linear system generated by Eiger from an electromagnetic scattering problem on the thin-slot geometry shown in Figure 6.1 with $n = 15565$ unknowns. We compute approximate solutions to the system using GMRES with a

Table 6.1 *Timings of symmetric and nonsymmetric algorithms for compressed matrix-vector product of a (symmetric) Gaussian kernel matrix with a random vector ($n = 50000, \epsilon_c = 1\text{e}{-3}$, budget = 0.05). The compressed algorithms use the same value for both the maximum rank and maximum leaf-size parameters. The time for an uncompressed matrix-vector product is given for reference. Timings in seconds are reported separately for compression(Algorithm 6.2.3) and evaluation (Algorithm 6.2.4).*

	Rank/leaf size	Compress time	Evaluate time	Rel. error
Symmetric	128	7.51	2.54	1.1e-3
Nonsymmetric	128	7.70	2.50	1.7e-3
Symmetric	256	1.42	0.62	1.5e-2
Nonsymmetric	256	1.62	0.58	1.4e-2
Symmetric	512	2.54	0.42	3.2e-2
Nonsymmetric	512	3.64	0.40	3.8e-2
Symmetric	1024	9.84	0.41	7.1e-2
Nonsymmetric	1024	19.80	0.44	6.0e-2
Uncompressed			2.15	

compressed operator and with an uncompressed operator. The two algorithms were run until the relative residual was less than 1e−5. In the algorithm that uses the compressed operator, the convergence test uses the relative residual in the compressed operator $\|\tilde{K}\tilde{x} - b\|/\|b\|$. After the algorithm terminates, either due to satisfaction of the convergence condition or reaching the maximum number of iterations, the true relative residual is computed using the uncompressed operator $\|K\tilde{x} - b\|/\|b\|$.

Table 6.2 shows results for a right-hand side that was generated by applying the operator to a random vector drawn from a uniform distribution over $[-1, 1]^n$. For this problem, the algorithm using the compressed operator took 0.35 seconds for compression and 0.72 seconds for 102 iterations of GMRES to reach a relative residual of 1e−5 in the compressed operator, corresponding to a relative residual

Table 6.2 *GMRES applied to the linear system $Kx = b$, where K is the matrix from the thin-slot scattering problem and b is set to be product of K applied to a random vector drawn from a uniform distribution over $[0, 1]^n$. The table reports compression time in seconds, GMRES time in seconds (excluding compression), the number of GMRES iterations, and relative residual in the uncompressed operator.*

	Compress time	GMRES time	# iter	True residual
Compressed	0.35	0.72	102	3e-4
Uncompressed	0	6.28	101	1e-5

of 3e−4 in the uncompressed operator. GMRES using the uncompressed operator completed after 101 iterations in 6.28 seconds, taking 8.8× longer per GMRES iteration and 5.9× longer in total runtime than the compressed algorithm. The two algorithms took nearly the same number of iterations to converge, but the GMRES iterations of the compressed algorithm were much faster.

Table 6.3 shows results for a given right-hand side of practical interest. In this case, both algorithms reached the limit of 200 iterations before convergence of the residual. For the compressed algorithm, the relative residual in the compressed operator reached 1e−2, but the true residual was 1.5. This discrepancy is caused by a large condition number ($\texttt{cond}(K) = 6.7\text{e}8$) compared to the relative error of the approximation $\left(\|K - \tilde{K}\|/\|K\| = 2.7\text{e}{-4}\right)$, and by the particular choice of b. The analysis in §6.2.5.1 shows that the discrepancy can be controlled by ensuring $\epsilon_c \texttt{cond}(K)$ is much less than 1, which is not satisfied.

A possible remedy would be to compress the matrix to higher accuracy, but attempts to tighten the error tolerance parameter failed to produce a more accurate compression. The cause of this failure is likely to be poor quality of the

Table 6.3 *GMRES applied to the linear system $Kx = b$, where K is the matrix from the thin-slot scattering problem and b is a given right-hand side of practical interest. The table reports compression time in seconds, GMRES time in seconds (excluding compression), the number of GMRES iterations, and relative residual in the uncompressed operator.*

	Compress time	GMRES time	# iter	True residual
Compressed	0.36	2.1	200	1.5
Uncompressed	0	12.3	200	6e-3

samples used in the sampled ID, which were selected at random due to the absence of geometric information. Geometric information can be easily incorporated into the compression algorithm, but it must first be extracted during construction of the discretized problem, and such information was not available at the time of writing. To demonstrate this, we run the algorithm with sampling and without sampling over a range of values for the error tolerance parameter as shown in Table 6.4. The error decreases along with the error tolerance for the unsampled algorithm but not for the sampled algorithm. The unsampled algorithm gives better convergence for this problem, but it is not practical due to its expensive compression phase, which requires $\mathcal{O}(n^2)$ operations. For example, with $\epsilon_c/\|K\| = 1\text{e}{-9} \approx \text{cond}(K)$, the compression time without sampling is 70.5 seconds, which is enough time to run thousands of GMRES iterations using the uncompressed operator. Though we cannot know the extent to which geometric information would improve sampling, it has been observed that the sampling scheme we use generally works well on a range of practical problems, and uniform random sampling generally performs poorly [67].

Table 6.4 *Comparison of compression accuracy using a sampled ID versus an unsampled ID for the scattering problem. The first column reports the relative error tolerance, and the second and third columns report the relative error of the compressed operator constructed using either sampled IDs or unsampled IDs.*

Rel. error tolerance	Sampled ID rel. error	Unsampled ID rel. error
1e-4	3e-4	3e-4
1e-5	3e-4	1e-4
1e-6	1e-3	3e-5
1e-7	1e-3	6e-7
1e-8	2e-3	2e-8
1e-9	4e-3	9e-10
1e-10	4e-3	7e-11

6.3.3 Helmholtz kernel

We consider another problem for which geometric information is available. In these experiments, we use a kernel matrix defined by the three-dimensional Helmholtz kernel

$$\mathcal{K}(x_i, x_j) = \frac{e^{-ik\|x_i - x_j\|_2}}{\|x_i - x_j\|_2}, \tag{6.9}$$

where the points $\{x_i\}_{i=1}^n$ are drawn from a uniform distribution on $[0,1]^3$. A large wave number k in the kernel function produces a highly oscillatory kernel function, resulting in a matrix that is difficult to compress. We solve the regularized linear least squares problem $(DK + \lambda I)x = b$, where D is a diagonal matrix with diagonal entries drawn randomly from a uniform distribution on $[0,1]$ and $\lambda \in \mathbb{R}$ is a regularization term. The purpose of applying D is to make the matrix $DK + \lambda I$ nonsymmetric, and the regularization term is chosen to control the rate of convergence of GMRES. In these experiments, $n = 32768, \lambda = 300$.

Table 6.5 *Matrix-vector product $(DK + \lambda I)x$, where K is the Helmholtz kernel matrix and x is drawn from a uniform distribution over $[0, 1]^n$. The first column reports the relative error tolerance, the second and third columns report timings in seconds for compression and evaluation, and the fourth column reports the relative error of the compressed operator.*

Rel. error tolerance	Compress time	Evaluate time	Rel. error
1e-1	1.51	0.024	4e-1
1e-2	1.53	0.025	6e-2
1e-3	1.76	0.028	2e-2
1e-4	1.84	0.031	1e-3
1e-5	1.92	0.048	9e-5
1e-6	2.05	0.073	6e-6
1e-7	2.23	0.116	1e-6
1e-8	2.33	0.253	2e-7
1e-9	2.69	0.369	4e-8
1e-10	3.10	0.577	1e-8
Uncompressed		0.252	

Table 6.5 shows performance and accuracy of the matrix-vector product for a range of error tolerances. As the error tolerance is decreased, the approximation error of the compressed operator decreases accordingly, and costs increase for both compression and evaluation. Nearly every case demonstrates speedup over the uncompressed matrix-vector product.

Table 6.6 shows performance and accuracy of GMRES applied to the linear system $(DK + \lambda I)x = b$, where b is defined to be the product of $DK + \lambda I$ applied to a random vector drawn from a uniform distribution over $[0, 1]^n$. The problem is solved using an compressed operator for a range of error tolerances and with the uncompressed operator. We report the final relative residual both in the compressed operator and in the uncompressed operator. In cases with larger error tolerance, the

Table 6.6 *GMRES applied to the linear system $(DK + \lambda I)x = b$, where K is the Helmholtz kernel matrix and b is set to be product of $DK + \lambda I$ applied to a random vector drawn from a uniform distribution over $[0, 1]^n$. The table reports the relative error tolerance, compression time in seconds, GMRES time in seconds (excluding compression), relative residual in the compressed operator, and the true residual (the relative residual in the uncompressed operator).*

Rel. err. tol.	Compress time	GMRES time	Comp. resid.	True resid.
1e-1	1.5	3.0	6e-4	2e-1
1e-2	1.5	3.3	3e-5	4e-2
1e-3	1.8	3.8	2e-5	2e-2
1e-4	1.8	4.7	2e-5	3e-3
1e-5	1.9	5.7	2e-5	4e-4
1e-6	2.0	11.5	2e-5	7e-5
1e-7	2.2	16.8	1e-5	2e-5
1e-8	2.3	30.0	2e-5	2e-5
1e-9	2.7	36.8	2e-5	2e-5
1e-10	3.1	45.4	2e-5	2e-5
Uncompressed		25.6	2e-5	2e-5

residual in the uncompressed operator remains large even after convergence in the compressed operator due to error in the compression. For more accurate compressed operators, this discrepancy diminishes, and the two residuals reach agreement.

6.4 Conclusions and Future Work

We present an algorithm for compressing and applying rank-structured nonsymmetric matrices such as those arising from applying the method of moments to the combined field integral equation. We integrate the compression scheme with an iterative solver and apply it to an electromagnetic scattering problem, demonstrating significant speedup over an iterative solver that uses direct matrix-vector products.

We also explore cases for which the algorithm performs poorly, for example, in the absence of geometric information and for a system with a right-hand side that is difficult to solve. Directions for further development include tighter integration between the GOFMM and Eiger (or the new Gemma) codes so that the problem can be solved in a matrix-free manner, applying a preconditioner or scaling rows and columns to solve the problem faster and more robustly, the use of inexact Krylov methods for further speedup, and using the distributed-memory algorithm [110] to solve much larger problems.

Chapter 7

Conclusion

This thesis describes a set of efficient algorithms to address two problems that arise in working with large, dense, rank-structured matrices. Part I addresses the problem of black-box randomized compression of rank-structured matrices. Like the randomized singular value decomposition, these algorithms compute approximations to rank-structured matrices by accessing the matrix only through black-box matrix-vector multiplication routines. Chapter 2 presents a linear-complexity algorithm for compressing HBS matrices, and Chapter 3 presents peeling algorithms accelerated with graph coloring for compressing $\mathcal{H}^1$, uniform $\mathcal{H}^1$, and $\mathcal{H}^2$ matrices. Part II addresses the problem of finding permutations of matrices that reveal rank structure. Chapter 4 describes the Geometry-Oblivious Fast Multipole Method for permuting symmetric positive-definite matrices, and Chapter 5 describes the leverage score clustering scheme for general dense matrices. Finally, Chapter 6 presents an application of the geometry-aware variant of the GOFMM compression algorithm to matrices arising from problems in electromagnetic scattering.

7.1 Future work

The contributions in Part I open up a number of opportunities for futher research. The techniques developed for sampling off-diagonal blocks of interest may be applicable in algorithms for compressing additional classes of data-sparse matrices. While we demonstrate empirically that the black-box compression schemes achieve high accuracy, rigorous theoretical error analysis (e.g., how approximation error for individual blocks relates to approximation of the entire matrix, and how error accumulates across levels of the tree) would be useful to users. We expect

that the algorithm of Chapter 2 can be generalized to the $\mathcal{H}^2$ format, which is subject to a strong admissibility condition. Moreover, given that most of the work in that algorithm is performed for nodes in the finest levels of the tree, and that the data dependencies have a tree-like structure, we expect that the algorithm can be parallelized efficiently and cleanly. The algorithm of Chapter 3 requires many fewer samples than prior works, but it may be possible to achieve even further acceleration.

The contributions in Part II open up techniques for working with rank-structured matrices to broader classes of matrices. A stronger theoretical understanding of which problems are amenable to permutation to reveal rank structure and which problems are not would be of practical value. For the algorithms in Chapter 5, there may be opportunity to improve the efficiency of the algorithms by exploring recent techniques for computing approximations to the leverage scores at less expense than for computing them exactly.

Bibliography

[1] Emmanuel Agullo, Eric Darve, Luc Giraud, and Yuval Harness. *Nearly optimal fast preconditioning of symmetric positive definite matrices*. PhD thesis, Inria Bordeaux Sud-Ouest, 2016.

[2] Sivaram Ambikasaran. *Fast Algorithms for Dense Numerical Linear Algebra and Applications*. PhD thesis, Stanford University, 2013.

[3] Sivaram Ambikasaran and Eric Darve. An\ mathcal o (n\ log n) fast direct solver for partial hierarchically semi-separable matrices. *Journal of Scientific Computing*, 57(3):477–501, 2013.

[4] Sivaram Ambikasaran and Eric Darve. An $(n \log n)$ fast direct solver for partial hierarchically semi-separable matrices. *Journal of Scientific Computing*, 57(3):477–501, 2013.

[5] Francesco P Andriulli, Kristof Cools, Ignace Bogaert, and Eric Michielssen. On a well-conditioned electric field integral operator for multiply connected geometries. *IEEE transactions on antennas and propagation*, 61(4):2077–2087, 2013.

[6] Travis Askham, Zydrunas Gimbutas, Leslie Greengard, Libin Lu, Jeremy Magland, Dhairya Malhotra, Mike O'Neil, Manas Rachh, Vladimir Rokhlin,

and Felipe Vico. Flatiron institute fast multipole libraries. `https://github.com/flatironinstitute/FMM3D`.

[7] Eric Bavier, Mark Hoemmen, Sivasankaran Rajamanickam, and Heidi Thornquist. Amesos2 and belos: Direct and iterative solvers for large sparse linear systems. *Scientific Programming*, 20(3):241–255, 2012.

[8] Mario Bebendorf. *Hierarchical matrices*. Springer, 2008.

[9] Mario Bebendorf and Sergej Rjasanow. Adaptive low-rank approximation of collocation matrices. *Computing*, 70(1):1–24, 2003.

[10] Michele Benzi, Gene H Golub, and Jörg Liesen. Numerical solution of saddle point problems. *Acta numerica*, 14(1):1–137, 2005.

[11] Jock A Blackard and Denis J Dean. Comparative accuracies of artificial neural networks and discriminant analysis in predicting forest cover types from cartographic variables. *Computers and electronics in agriculture*, 24(3):131–151, 1999.

[12] Steffen Börm. *Efficient numerical methods for non-local operators*, volume 14 of *EMS Tracts in Mathematics*. European Mathematical Society (EMS), Zürich, 2010. $\mathcal{H}^2$-matrix compression, algorithms and analysis.

[13] Daniel Brélaz. New methods to color the vertices of a graph. *Communications of the ACM*, 22(4):251–256, 1979.

[14] Nicola Cancedda, Eric Gaussier, Cyril Goutte, and Jean Michel Renders. Word sequence kernels. *Journal of Machine Learning Research*, 3:1059–1082, March 2003.

[15] S. Chandrasekaran, P. Dewilde, M. Gu, and N. Somasunderam. On the numerical rank of the off-diagonal blocks of Schur complements of discretized elliptic PDEs. *SIAM Journal on Matrix Analysis and Applications*, 31(5):2261–2290, 2010.

[16] Shiv Chandrasekaran, Patrick Dewilde, Ming Gu, William Lyons, and Timothy Pals. A fast solver for hss representations via sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 29(1):67–81, 2007.

[17] Shiv Chandrasekaran, Ming Gu, and Timothy Pals. A fast ulv decomposition solver for hierarchically semiseparable representations. *SIAM Journal on Matrix Analysis and Applications*, 28(3):603–622, 2006.

[18] Chih-Chung Chang and Chih-Jen Lin. Libsvm: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):27, 2011.

[19] Chao Chen, Severin Reiz, Chenhan D. Yu, Hans-Joachim Bungartz, and George Biros. Fast approximation of the gauss–newton hessian matrix for the multilayer perceptron. *SIAM Journal on Matrix Analysis and Applications*, 42(1):165–184, 2021.

[20] Zizhong Chen and Jack J Dongarra. Condition numbers of gaussian random

matrices. *SIAM Journal on Matrix Analysis and Applications*, 27(3):603–620, 2005.

[21] H. Cheng, Leslie Greengard, and Vladimir Rokhlin. A fast adaptive multipole algorithm in three dimensions. *Journal of Computational Physics*, 155:468–498, 1999.

[22] Hongwei Cheng, William Y Crutchfield, Zydrunas Gimbutas, Leslie F Greengard, J Frank Ethridge, Jingfang Huang, Vladimir Rokhlin, Norman Yarvin, and Junsheng Zhao. A wideband fast multipole method for the helmholtz equation in three dimensions. *Journal of Computational Physics*, 216(1):300–325, 2006.

[23] D Yu Chenhan, William B March, and George Biros. An n log n parallel fast direct solver for kernel matrices. In *Parallel and Distributed Processing Symposium (IPDPS), 2017 IEEE International*, pages 886–896. IEEE, 2017.

[24] Michael B Cohen, Cameron Musco, and Christopher Musco. Input sparsity time low-rank approximation via ridge leverage score sampling. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1758–1777. SIAM, 2017.

[25] P. Coulier, H. Pouransari, and E. Darve. The inverse fast multipole method: using a fast approximate direct solver as a preconditioner for dense linear systems. *ArXiv e-prints*, 2016.

[26] Ryan R. Curtin, James R. Cline, Neil P. Slagle, William B. March, P. Ram,

Nishant A. Mehta, and Alexander G. Gray. MLPACK: A scalable C++ machine learning library. *Journal of Machine Learning Research*, 14:801–805, 2013.

[27] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 537–546. ACM, 2008.

[28] Petros Drineas, Ravi Kannan, and Michael W Mahoney. Fast monte carlo algorithms for matrices ii: Computing a low-rank approximation to a matrix. *SIAM Journal on computing*, 36(1):158–183, 2006.

[29] Petros Drineas, Malik Magdon-Ismail, Michael W Mahoney, and David P Woodruff. Fast approximation of matrix coherence and statistical leverage. *The Journal of Machine Learning Research*, 13(1):3475–3506, 2012.

[30] Petros Drineas, Michael W Mahoney, and Shan Muthukrishnan. Relative-error cur matrix decompositions. *SIAM Journal on Matrix Analysis and Applications*, 30(2):844–881, 2008.

[31] Alan Edelman and Brian D Sutton. Tails of condition number distributions. *SIAM journal on matrix analysis and applications*, 27(2):547–560, 2005.

[32] William Fong and Eric Darve. The black-box fast multipole method. *Journal of Computational Physics*, 228(23):8712–8725, 2009.

[33] Pieter Ghysels, Xiaoye S. Li, François-Henry Rouet, Samuel Williams, and Artem Napov. An efficient multicore implementation of a novel HSS-structured

multifrontal solver using randomized sampling. *SIAM Journal on Scientific Computing*, 38(5):S358–S384, 2016.

[34] Adrianna Gillman, Patrick M Young, and Per-Gunnar Martinsson. A direct solver with o (n) complexity for integral equations on one-dimensional domains. *Frontiers of Mathematics in China*, 7(2):217–247, 2012.

[35] Gene H Golub and Charles F Van Loan. *Matrix computations*. JHU press, 2013.

[36] Lars Grasedyck, Ronald Kriemann, and Sabine Le Borne. Parallel black box-lu preconditioning for elliptic boundary value problems. *Computing and visualization in science*, 11(4):273–291, 2008.

[37] A.G. Gray and A.W. Moore. N-body problems in statistical learning. *Advances in neural information processing systems*, pages 521–527, 2001.

[38] L Greengard and V Rokhlin. A fast algorithm for particle simulations. *J. Comp. Phys*, 73:325–348, 1987.

[39] Leslie Greengard, Denis Gueyffier, Per-Gunnar Martinsson, and Vladimir Rokhlin. Fast direct solvers for integral equations in complex three-dimensional domains. *Acta Numerica*, 18(1):243–275, 2009.

[40] Leslie Greengard and Vladimir Rokhlin. A new version of the fast multipole method for the laplace equation in three dimensions. *Acta numerica*, 6:229–269, 1997.

[41] Greengard, L. Fast Algorithms For Classical Physics. *Science*, 265(5174):909–914, 1994.

[42] Han Guo, Yang Liu, Jun Hu, and Eric Michielssen. A butterfly-based direct integral-equation solver using hierarchical lu factorization for analyzing scattering from electrically large conducting objects. *IEEE Transactions on Antennas and Propagation*, 65(9):4742–4750, 2017.

[43] Wolfgang Hackbusch. A sparse matrix arithmetic based on h-matrices. part i: Introduction to h-matrices. *Computing*, 62(2):89–108, 1999.

[44] Wolfgang Hackbusch. *Hierarchical Matrices: Algorithms and Analysis*. Springer Series in Computational Mathematics 49. Springer-Verlag Berlin Heidelberg, 1 edition, 2015.

[45] N. Halko, P.-G. Martinsson, and J.A. Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM Review*, 53:217–288, 2011.

[46] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

[47] Michael A Heroux and James M Willenbring. A new overview of the trilinos project. *Scientific Programming*, 20(2):83–88, 2012.

[48] Kenneth L Ho and Leslie Greengard. A fast direct solver for structured linear systems by recursive skeletonization. *SIAM Journal on Scientific Computing*,

34(5):A2507–A2532, 2012.

[49] Kenneth L Ho and Leslie Greengard. A fast direct solver for structured linear systems by recursive skeletonization. *SIAM Journal on Scientific Computing*, 34(5):A2507–A2532, 2012.

[50] Thomas Hofmann, Bernhard Schölkopf, and Alexander J Smola. Kernel methods in machine learning. *The annals of statistics*, pages 1171–1220, 2008.

[51] Cho-Jui Hsieh, Si Si, and Inderjit S Dhillon. Fast prediction for large-scale kernel machines. In Z. Ghahramani, M. Welling, C. Cortes, N.D. Lawrence, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 27*, pages 3689–3697. Curran Associates, Inc., 2014.

[52] William A Johnson et al. Eiger: An open-source frequency-domain electromagnetics code. In *Antennas and Propagation Society International Symposium, 2007 IEEE*, pages 3328–3331. IEEE, 2007.

[53] Sharad Kapur and Vladimir Rokhlin. High-order corrected trapezoidal quadrature rules for singular functions. *SIAM Journal on Numerical Analysis*, 34(4):1331–1356, 1997.

[54] George Karypis and Vipin Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[55] Risi Imre Kondor and John Lafferty. Diffusion kernels on graphs and other discrete input spaces. In *ICML*, volume 2, pages 315–322, 2002.

[56] Joseph D. Kotulski. Computational electromagnetics at sandia national laboratories - current code capability. 10 2015.

[57] William L Langston, Joseph Kotulski, Rebecca Coats, Roy Jorgenson, S Adam Blake, Salvatore Campione, Aaron Pung, and Brian Zinser. Massively parallel frequency domain electromagnetic simulation codes. In *Applied Computational Electromagnetics Society Symposium (ACES), 2018 International*, pages 1–2. IEEE, 2018.

[58] Dongryeol Lee, Piyush Sao, Richard Vuduc, and Alexander G Gray. A distributed kernel summation framework for general-dimension machine learning. *Statistical Analysis and Data Mining*, 2013.

[59] Ivar Leidus. Romanesco broccoli (brassica oleracea), 2021.

[60] James Levitt and Per-Gunnar Martinsson. Linear-complexity black-box randomized compression of hierarchically block separable matrices. *arXiv preprint arXiv:2205.02990*, 2022.

[61] James Levitt and Per-Gunnar Martinsson. Randomized compression of rank-structured matrices accelerated with graph coloring. *arXiv preprint arXiv:2205.03406*, 2022.

[62] Edo Liberty, Franco Woolfe, Per-Gunnar Martinsson, Vladimir Rokhlin, and Mark Tygert. Randomized algorithms for the low-rank approximation of matrices. *Proceedings of the National Academy of Sciences*, 104(51):20167–20172, 2007.

[63] M. Lichman. UCI machine learning repository, 2013.

[64] Lin Lin, Jianfeng Lu, and Lexing Ying. Fast construction of hierarchical matrix representation from matrix–vector multiplication. *Journal of Computational Physics*, 230(10):4071–4087, 2011.

[65] Michael W Mahoney and Petros Drineas. Cur matrix decompositions for improved data analysis. *Proceedings of the National Academy of Sciences*, 106(3):697–702, 2009.

[66] M.W. Mahoney and P. Drineas. Cur matrix decompositions for improved data analysis. *Proceedings of the National Academy of Sciences*, 106(3):697, 2009.

[67] William B March and George Biros. Far-field compression for fast kernel summation methods in high dimensions. *Applied and Computational Harmonic Analysis*, 43(1):39–75, 2017.

[68] William B March, Bo Xiao, Sameer Tharakan, D Yu Chenhan, and George Biros. A kernel-independent fmm in general dimensions. In *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pages 1–12. IEEE, 2015.

[69] William B. March, Bo Xiao, Sameer Tharakan, Chenhan D. Yu, and George Biros. A kernel-independent FMM in general dimensions. In *Proceedings of SC15*, The SCxy Conference series, Austin, Texas, November 2015. ACM/IEEE.

[70] William B. March, Bo Xiao, Sameer Tharakan, Chenhan D. Yu, and George Biros. Robust treecode approximation for kernel machines. In *Proceedings of the 21st ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 1–10, Sydney, Australia, August 2015.

[71] William B. March, Bo Xiao, Chenhan Yu, and George Biros. An algebraic parallel treecode in arbitrary dimensions. In *Proceedings of IPDPS 2015*, 29th IEEE International Parallel and Distributed Computing Symposium, Hyderabad, India, May 2015.

[72] William B March, Bo Xiao, Chenhan D Yu, and George Biros. Askit: an efficient, parallel library for high-dimensional kernel summations. *SIAM Journal on Scientific Computing*, 38(5):S720–S749, 2016.

[73] William B. March, Bo Xiao, Chenhan D. Yu, and George Biros. Askit: An efficient, parallel library for high-dimensional kernel summations. *SIAM Journal on Scientific Computing*, 38(5):S720–S749, 2016.

[74] P.-G. Martinsson, V. Rokhlin, and M. Tygert. A randomized algorithm for the decomposition of matrices. *Applied and Computational Harmonic Analysis*, 2010.

[75] Per-Gunnar Martinsson. A fast direct solver for a class of elliptic partial differential equations. *Journal of Scientific Computing*, 38(3):316–330, 2009.

[76] Per-Gunnar Martinsson. A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix. *SIAM Journal on*

Matrix Analysis and Applications, 32(4):1251–1274, 2011.

[77] Per-Gunnar Martinsson. Compressing rank-structured matrices via randomized sampling. *SIAM Journal on Scientific Computing*, 38(4):A1959–A1986, 2016.

[78] Per-Gunnar Martinsson. Compressing rank-structured matrices via randomized sampling. *SIAM Journal on Scientific Computing*, 38(4):A1959–A1986, 2016.

[79] Per-Gunnar Martinsson. *Fast direct solvers for elliptic PDEs*. SIAM, 2019.

[80] Per-Gunnar Martinsson. *Fast Direct Solvers for Elliptic PDEs*, volume CB96 of *CBMS-NSF conference series*. SIAM, 2019.

[81] Per-Gunnar Martinsson and Vladimir Rokhlin. An accelerated kernel-independent fast multipole method in one dimension. *SIAM Journal on Scientific Computing*, 29(3):1160–1178, 2007.

[82] Per-Gunnar Martinsson and Joel Tropp. Randomized numerical linear algebra: Foundations & algorithms. *arXiv preprint arXiv:2002.01387*, 2020.

[83] Per-Gunnar Martinsson and Joel A Tropp. Randomized numerical linear algebra: Foundations and algorithms. *Acta Numerica*, 29:403–572, 2020.

[84] P.G. Martinsson and V. Rokhlin. A fast direct solver for boundary integral equations in two dimensions. *J. Comp. Phys.*, 205(1):1–23, 2005.

[85] Joseph R Mautz and Roger F Harrington. H-field, e-field, and combined field solutions for bodies of revolution. Technical report, SYRACUSE UNIV NY DEPT OF ELECTRICAL AND COMPUTER ENGINEERING, 1977.

[86] Victor Minden, Kenneth L Ho, Anil Damle, and Lexing Ying. A recursive skeletonization factorization based on strong admissibility. *Multiscale Modeling & Simulation*, 15(2):768–796, 2017.

[87] K. R. Muske and J. W. Howse. A Lagrangian method for simultaneous nonlinear model predictive control. In L.T. Biegler, O. Ghattas, M. Heinkenschloss, and B. van Bloemen Waanders, editors, *Large-Scale PDE-constrained Optimization: State-of-the-Art*, Lecture Notes in Computational Science and Engineering. Springer-Verlag, 2001.

[88] Dimitris Papailiopoulos, Anastasios Kyrillidis, and Christos Boutsidis. Provable deterministic leverage score sampling. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 997–1006, 2014.

[89] Sadasiva Rao, D Wilton, and Allen Glisson. Electromagnetic scattering by surfaces of arbitrary shape. *IEEE Transactions on antennas and propagation*, 30(3):409–418, 1982.

[90] François-Henry Rouet, Xiaoye S. Li, Pieter Ghysels, and Artem Napov. A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization. *ACM Transactions in Mathematical Software*, 42(4):27:1–27:35, June 2016.

[91] Yousef Saad and Martin H Schultz. Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on scientific and statistical computing*, 7(3):856–869, 1986.

[92] Bernhard Schölkopf and Alexander J Smola. *Learning with kernels: support vector machines, regularization, optimization, and beyond*. MIT press, 2002.

[93] Xing Q Sheng, J-M Jin, Jiming Song, Weng C Chew, and C-C Lu. Solution of combined-field integral equation using multilevel fast multipole algorithm for scattering by homogeneous bodies. *IEEE transactions on antennas and propagation*, 46(11):1718–1726, 1998.

[94] Jiming Song, Cai-Cheng Lu, and Weng Cho Chew. Multilevel fast multipole algorithm for electromagnetic scattering by large complex objects. *IEEE Transactions on Antennas and Propagation*, 45(10):1488–1493, 1997.

[95] JM Song and Weng Cho Chew. Multilevel fast-multipole algorithm for solving combined field integral equations of electromagnetic scattering. *Microwave and Optical Technology Letters*, 10(1):14–19, 1995.

[96] Balaji Vasan Srinivasan, Qi Hu, and Ramani Duraiswami. GPUML: Graphical processors for speeding up kernel machines. In *Workshop on High Performance Analytics-Algorithms, Implementations, and Applications, Siam Conference on Data Mining*, 2010.

[97] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE*

transactions on parallel and distributed systems, 13(3):260–274, 2002.

[98] Joel A Tropp, Alp Yurtsever, Madeleine Udell, and Volkan Cevher. Practical sketching algorithms for low-rank matrix approximation. *SIAM Journal on Matrix Analysis and Applications*, 38(4):1454–1485, 2017.

[99] Ruoxi Wang, Yingzhou Li, Michael W Mahoney, and Eric Darve. Structured block basis factorization for scalable kernel matrix evaluation. *arXiv preprint arXiv:1505.00398*, 2015.

[100] Christopher Williams and Matthias Seeger. Using the Nyström method to speed up kernel machines. In *Proceedings of the 14th Annual Conference on Neural Information Processing Systems*, number EPFL-CONF-161322, pages 682–688, 2001.

[101] Jianlin Xia, Shivkumar Chandrasekaran, Ming Gu, and Xiaoye S Li. Fast algorithms for hierarchically semiseparable matrices. *Numerical Linear Algebra with Applications*, 17(6):953–976, 2010.

[102] Jianlin Xia, Shivkumar Chandrasekaran, Ming Gu, and Xiaoye S Li. Superfast multifrontal method for large structured linear systems of equations. *SIAM Journal on Matrix Analysis and Applications*, 31(3):1382–1411, 2010.

[103] Bo Xiao and George Biros. Parallel algorithms for nearest neighbor search problems in high dimensions. *SIAM Journal on Scientific Computing*, 38(5):S667–S699, 2016.

[104] Lexing Ying, George Biros, and Denis Zorin. A kernel-independent adaptive fast multipole method in two and three dimensions. *Journal of Computational Physics*, 196(2):591–626, 2004.

[105] Rio Yokota, Huda Ibeid, and David Keyes. Fast multipole method as a matrix-free hierarchical low-rank approximation. *Computing Research Repository*, abs/1602.02244, 2016.

[106] Chenhan D Yu, Jianyu Huang, Woody Austin, Bo Xiao, and George Biros. Performance optimization for the k-nearest neighbors kernel on x86 architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 7. ACM, 2015.

[107] Chenhan D Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious fmm for compressing dense spd matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–14, 2017.

[108] Chenhan D. Yu, James Levitt, Severin Reiz, and George Biros. Geometry-oblivious fmm for compressing dense spd matrices. In *Proceedings of SC17*, The SCxy Conference series, Denver, Colorado, November 2017. ACM/IEEE.

[109] Chenhan D. Yu, William B. March, Bo Xiao, and George Biros. INV-ASKIT: a parallel fast direct solver for kernel matrices. In *Proceedings of the IPDPS16*, Chicago, USA, May 2016.

[110] Chenhan D Yu, Severin Reiz, and George Biros. Distributed-memory hierarchical compression of dense spd matrices. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 15. IEEE Press, 2018.