A TALE OF TWO TENSORS: USING HIERARCHICAL AND BLOCK LOW RANK MATRICES TO MAKE PRECONDITIONERS AND SAVE STORAGE

A thesis

submitted by

Mitchell T. Scott

in partial fulfillment of the requirements

for the degree of

Master of Science

in

Mathematics

TUFTS UNIVERSITY

May 2023

Advisor: Professor Misha E. Kilmer

Abstract

Hierarchical matrices are commonly encountered while solving discretized fractional differential equations, such as those arising in the modeling of turbulence, financial markets, and continuum mechanics. Unfortunately, such matrices are dense (i.e., a high percentage of the entries in the matrices are non-zero), so that the cost of storing and accessing elements in the matrices limits the size of the problems that can be handled on current computer architectures. However, such matrices are known to have so-called hierarchical structure: various off-diagonal sub-blocks have low-rank. We take advantage of this recursive hierarchical structure to accumulate these low-rank blocks into tensors at multiple levels. For each of these third order tensors, we form approximated tensor decompositions and use the tensor approximations to form a matrix approximation. Lastly, we discuss how to leverage this hierarchical and Kronecker structure to construct a preconditioner for these systems.

To those who never stopped believing in me, even when I had.

Acknowledgements

First, I need to acknowledge my advisor, Misha Kilmer, for her unwavering support and counseling. She has always been so willing to answer my silly questions as my numerical analysis professor even before and definitely during this thesis process. She has helped me understand what good research is and can look like. Her willingness to carve out time out of her extremely busy schedule to answer panicked emails or give me feedback has meant a lot. I appreciate her compassion and understanding when I had a busy week and wasn't able to give results, or when the results I had were less than promising. On those tough days, our mutual enthusiasm of this project kept me going. It has been great to have the opportunity to see Misha, in so many different facets of academia – my professor, my research advisor, and the professor for whom I am a TA. The lessons you have imparted on me will stick with me as an academic.

Thank you to everyone who took time to come to the weekly research meetings and be on my thesis committee — James Adler, Xiaozhe Hu, and Arvind Saibaba. The recommendations in papers, GitHub repos, books, and your individualized lectures as well as your incredible patience have reinforced my awe for computational math and allowed me to grow as a mathematician. Additional thanks to James, Liz, and JC for the opportunities to present my research publicly. I also need to thank Sebastian Bozlee, Robert Lemke Oliver, and Kasso Okoudjou for being great professors and mathematical mentors.

Thank you to Sarah, Noah, and Chris. They have been the first line of defense for graduate students and their support keeps the department functional, fed, and fulfilled.

The graduate student community here is truly unmatched in terms of support and camaraderie; thank you all for making my time here so special. I want to thank my OGSM mentees— Kate and Miranda— they mentored me a lot, more than I ever did them, never letting me forget my humanity. I am also so appreciative of the huddle room 515 and Torie for our early morning Analysis study-turned-rant sessions as we embarked on the tumultuous PhD Admissions and Master's thesis processes together. We made it! I want to thank Alex for being a friendly face in a large crowd and teaching me about the life work balance and protecting ourselves in academia.

Life as a graduate student can also revolve only about school, so I am thankful for those who gave me opportunities to turn my brain off and have fun outside of JCC. I want to thank Alicia and Drew for sticking by me for so long and always reminding me of how far I've come and never letting me forget all my other non-scholarly attributes. Many thanks to the Backman family for always being so welcoming and supportive, truly providing a home away from home. I am appreciative of Michael who indulged my math stories any time, anywhere and talked about math departments of old. Thanks to Abdel for always making sure that schedules worked out so I could put school first.

Lastly, I am eternally grateful for my parents, Tim and Rhonda. They answer the phone whenever I call (and on stressful weeks, that's quite a feat), and they are always in my corner. They sacrificed so much to get me to where I am, and as I get older, I am only starting to comprehend the time, the money, and mostly the emotional labor. They taught me a metric for success from a very young age, a metric I still use today. I hope they see that I'm being the "best little Mitchell [I] can be".

Contents

List of Tables **viii**

List of Figures **ix**

List of Algorithms **xi**

1 Introduction to Tensors and Discretized Partial Differential Equations **2**

1.1 Introduction to Matrices 2

1.1.1 Matrix Structures 4

1.1.2 Norms and Error 6

1.2 Introduction to Tensors 7

1.2.1 Turning Matrices into Tensors and Back Again 8

1.2.2 Tensor Ranks and Decompositions 10

1.3 Introduction to discretized PDEs 14

1.3.1 Preconditioning 15

1.3.2 Hierarchical Matrices ($\mathcal{H}$ - matrices) 17

1.3.3 Fractional Partial Differential Equations 18

2 Research Question - how to approximate and precondition the dense matrix in an efficient way? **21**

2.1 What has been done 21

2.2 Leverage the approximation 25

2.3 Leverage the Structure 28

3 Research and Experimental Design **30**

3.1 Overview 30
3.2 Hierarchical Matrix Algorithm for 2^L-sized matrices 30
3.3 Alternative Algorithms for Matrices not of 2^L. 33
3.3.1 Padding 34
3.3.2 Truncating 35

4 Research Results 40

4.1 Storage 40
4.1.1 Naive 40
4.1.2 Low Rank Matrices 41
4.1.3 Tensor-based Methods 42
4.1.4 Additional Storage Considerations 44
4.2 Approximation 47
4.3 Speed of Mat-vec 50
4.4 Preconditioning 50

5 Summary, Implications, and Conclusions 54

5.1 Summary 54
5.2 Future Work 55

A 57

A.1 List of Notation 57
A.2 MATLAB Code 58
A.2.1 hmat2ten 58

Bibliography 59

List of Tables

3.1 Using our toy problem, three categories are found 33

4.1 Storage accounting for toy problem ($\theta = 0.613, \alpha = 1.5$) using method in [25]. 41

4.2 Generalization of storage accounting for toy problem, using hierarchical low rank blocks 42

4.3 Storage Accounting for toy problem ($\theta = 0.613, \alpha = 1.5$) using `hmat2ten` with truncation ranks `lev2` $\in \mathbb{R}^{47\times47\times4}$, `lev3` $\in \mathbb{R}^{32\times32\times10}$ 43

4.4 Generalizeation of Storage accounting for toy problem, using tensor based methods on low rank blocks 43

4.5 Comparing relative errors between the proposed tensor based methods and literature using a similar compression benchmark. 47

List of Figures

1.1 A visualization of the different slices of a tensor, $\mathcal{A} \in \mathbb{R}^{m \times p \times n}$. Note that each of these slices is a matrix in its own right. 8
1.2 The original matrix $\mathbf{M}$ is mapped into a tensor $\mathcal{T}$, which is then approximated using a tensor decomposition, $\widehat{\mathcal{T}}$. Lastly, $\widehat{\mathcal{T}}$ is mapped back to a matrix $\widehat{\mathbf{M}}$. 9
1.3 The bijective mapping between an $m \times n$ matrix and an $m \times 1 \times n$ tensor. The forward function is called "twist" while the inverse function is "squeeze". 9
1.4 We can twist these submatrices into lateral slices of a tensor. Similarly, we can twist the lateral slices back into a matrix. 10
1.5 The three mode$-k$ unfoldings of a third order tensor, $\mathcal{A}$. 11
1.6 On the left, a non-overlapping Jacobi block pattern is presented. On the right, an overlapping Jacobi block pattern is presented. 17
2.1 The mesh discretization for fPDEs needs to be adaptive, which breaks the Toeplitz structured problem. 23
2.2 When a matrix has block structure, it can be approximated a sum of Kronecker products. 26
2.3 When a matrix has block structure, it can be approximated by orthogonal matrices, left and right multiplied by, a block-rank structured matrix. 26
2.4 A way to approximate our matrix of interest would be to ignore the strong diagonal blocks, and map the other off-diagonal (hopefully low-rank) blocks into a tensor to be decomposed. 27

2.5 (1.) The whole matrix A, (2.) Subdivision of whole matrix A into four subblocks A_{11}, A_{12}, A_{21}, and A_{22}, (3.) Subdivision of $A.11$ into four subblocks $A.11_{11}, A.11_{12}, A.11_{21}$, and $A.11_{22}$, (4.) Subdivision of $A.11.11$ into four subblocks $A.11.11_{11}, A.11.11_{12}, A.11.11_{21}$, and $A.11.11_{22}$, (5.) The block $A.11.11.11$ failing the geometric stopping criterion, stored as full, (6.) All the blocks in $A.11.11$ failing the geometric stopping criteria, stored as full, (7.) Subdivision of $A.11.12$ into four subblocks $A.11.12_{11}, A.11.12_{12}, A.11.12_{21}$, and $A.11.12_{22}$, (8) The block $A.11.12.11$ satisfying the algebraic rank stopping criterion, twisted into a tensor for later compression, (9) All the blocks in $A.11.12$ stored according to their structure, (10) Due to symmetry and strong diagonal dominance, all the blocks in $A.11$ stored according to their structure, $A.12$ subdivided into four subblocks, (11) The block $A.12.11$ satisifying the algebraic rank stopping condition, twisted into a different tensor for later compression, (12) Repeating this process recursively unto all entries of A have been stored. 29

3.1 For $\alpha = 1.5$, the resulting matrix in $\mathbb{R}^{256\times256}$ visualized in terms of the relative magnitude of the entries. 32

3.2 The original dense matrix can be split into disjoint parts– the block diagonal parts (darkest blue), the subblocks that turn low-rank after three levels (medium blue), and the subblocks that turn low-rank after two levels (lightest blue) 32

3.3 This is the levels diagram for the adaptive mesh of size $2^8 \times 2^8$. Each small block is 32×32 and the bigger blocks on the off diagonals are 64×64. Once these blocks are found, they are twisted into a tensor at each level. 33

3.4 The $\mathbf{A}_{(6,3)}$ submatrix is placed in the correct location by $\mathbf{E}^1_{(6,3)}$, and all white blocks are zero. 38

3.5 The $\mathbf{A}_{(1,3)}$ submatrix is placed in the correct location by $\mathbf{E}^2_{(1,3)}$, and all white blocks are zero. 39

3.6 The original dense matrix is approximated by the sum of the exact full blocks, the approximation of the low rank blocks at level 3, and the approximation of the low rank blocks at level 2. 39

4.1 Using that our toy problem matrix is SPD, we can reformulate the matrix to include nonoverlapping blocks along the main diagonal, which are SPD, so to save storage, we can perform a Cholesky factorization and store one of the Cholesky factors. This cuts our storage in half. 47

4.2 As the fractional index, α, and the size of the problem, n, increase, the condition number of the resulting full matrix increases exponentially ($\theta = 0.613$). 51

4.3 By altering the refinement index, θ from 0.613 to 0.8, the trends of the condition number of the full stiffness matrix increasing with n, α are still seen. 51

5.1 An adaptive mesh that hase four levels of low-rank hierarchical structure 56

List of Algorithms

1 `tr`- Higher Order Singular Value Decomposition 13

2 Sequentially `tr`- Higher Order Singular Value Decomposition 13

3 Turning BLR matrices into Tensors 25

4 Turning $\mathcal{H}$-matrices into Tensors 31

5 Turning $\mathcal{H}$-matrices into Padded Tensors 34

6 Turning $\mathcal{H}$-matrices into Truncated Tensors 35

7 Turning Tensors into $\mathcal{H}$-matrices 36

A Tale of Two Tensors: Using Hierarchical and Block Low Rank Matrices to Make Preconditioners and Save Storage

Chapter 1

Introduction to Tensors and Discretized Partial Differential Equations

Many fields - like financial mathematics and fluid dynamics - can be accurately modeled by fractional partial differential equations (fPDEs). However, like most interesting problems, there is either no analytic solution to these fPDEs, or it is not feasible without using a computational method. One such method is to take the discretization matrix and see if we can extract and exploit any latent structure by taking this matrix into a higher level tensor. This hidden structure is used to make computational methods solve the problem at hand 1.) using fewer floating-point operations `flops`, and 2.) using less memory. The following thesis is an attempt to investigate a tensor-based numerical method to solve these fPDE problems.

1.1 Introduction to Matrices

Definition 1.1.1 (Matrix) *A matrix* $\mathbf{M}$ *is a two dimensional rectangular array of numbers. We say that* $\mathbf{M} \in \mathbb{F}^{m\times n}$*, if* $\mathbf{M}$ *has* m *rows and* n *columns, and each of the* m_{ij}*, or the element of* $\mathbf{M}$ *in the* i^{th} *row and the* j^{th} *column, all belong to some field* $\mathbb{F}$*. In this research, we exclusively use* $\mathbb{F} = \mathbb{R}$*.*

Matrices have many operations one can perform on them, such as addition, scalar multiplication, matrix multiplication, which are all well known. A lesser known operation, and one that is of great interest to this project is the "Kronecker Product".

Definition 1.1.2 (Kronecker Product) *Let* $\mathbf{A} \in \mathbb{R}^{m\times p}, \mathbf{B} \in \mathbb{R}^{n\times \ell}$*. Then the*

Kronecker Product $\mathbf{A} \otimes \mathbf{B} \in \mathbb{R}^{(mn)\times(p\ell)}$ *is denoted as*

$$\mathbf{A} \otimes \mathbf{B} = \begin{pmatrix} a_{11}\mathbf{B} & a_{12}\mathbf{B} & \cdots & a_{1p}\mathbf{B} \\ a_{21}\mathbf{B} & a_{22}\mathbf{B} & \cdots & a_{2p}\mathbf{B} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1}\mathbf{B} & a_{m2}\mathbf{B} & \cdots & a_{mp}\mathbf{B} \end{pmatrix} \tag{1.1}$$

It is straight forward to show just from the definition of Kronecker product how the product of two Kronecker Product matrices interact:

Corollary 1.1.3 *Assuming the dimensions of the following matrices work out, so that for* $\mathbf{A}, \mathbf{B}, \mathbf{C}, \mathbf{D}$, *we have*

$$(\mathbf{A} \otimes \mathbf{B})(\mathbf{C} \otimes \mathbf{D}) = (\mathbf{AC}) \otimes (\mathbf{BD}) \tag{1.2}$$

Another operation that is useful is that of turning a matrix into a vector, which will elicit useful properties.

Definition 1.1.4 (Vectorization) *Assume we have some matrix* $\mathbf{A} \in \mathbb{R}^{m\times n}$. *By* *vectorizing, or turning it into a vector, we see that by stacking the column vectors, we can get a vector,* $\boldsymbol{vec}(v) \in \mathbb{R}^{mn}$.

This concept of vecotrizing a matrix combined with Kronecker products allows us the following property.

Corollary 1.1.5

$$\mathit{vec}(\mathbf{AXB}^T) = (\mathbf{B} \otimes \mathbf{A})\mathit{vec}(\boldsymbol{X}) \tag{1.3}$$

Another useful operation we can perform on a matrix is a factorization, especially the singular value decomposition.

Definition 1.1.6 (Singular Value Decomposition) *Let* $\boldsymbol{M} \in \mathbb{R}^{m\times n}$ *be a rank-r matrix. Then the singular value decomposition (SVD) is* $\boldsymbol{M} = \mathbf{U\Sigma V}^T$, *where* $\boldsymbol{U} \in \mathbb{R}^{m\times m}$, $\boldsymbol{V}^T \in \mathbb{R}^{n\times n}$ *are both real, orthogonal matrices, namely* $\mathbf{U}^T\mathbf{U} = \mathbf{I}_m, \mathbf{V}^T\mathbf{V} = \mathbf{I}_n$.

These are known as the left and right singular vectors, respectively. Lastly, $\mathbf{\Sigma} \in \mathbb{R}^{m\times n}$ *is rectangular diagonal matrix with entries* $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_r > 0$. *These are called the singular values of* $\boldsymbol{M}$. *It is important to note that the SVD always exists, when* $\mathbf{M} \in \mathbb{R}^{m\times n}$.

It is expensive to compute the singular value decomposition (SVD) algorithm [12], so we wouldn't want to actually carry out the entire algorithm. When the singular values decay quickly, we can approximate the rank r matrix $\mathbf{M}$ with an approximation rank $k, k \leq r$ matrix $\widehat{\mathbf{M}}$. This allows us to only deal with the first k left and right singular vectors, and the largest k singular values.

Definition 1.1.7 (Rank-*k* Approximation) *Let* $k < r \leq \min\{m,n\}$. *Then the rank-k approximation of a matrix* $\mathbf{M} \in \mathbb{R}^{m\times n}$ *is* $\widehat{\mathbf{M}} \approx \mathbf{U}_k\mathbf{\Sigma}_k\mathbf{V}_k^T$, *where* $\mathbf{U}_k \in \mathbb{R}^{m\times k}, \mathbf{\Sigma} \in \mathbb{R}^{k\times k}, \mathbf{V}_k^T \in \mathbb{R}^{k\times n}$.

While this is certainly an approximation of the original matrix, we can actually say something stronger. In fact, according to 1.1.8, this is the best rank-k approximation to the matrix, $\mathbf{M}$.

Theorem 1.1.8 (Eckart-Young, 1936) *Let* $A \in \mathbb{R}^{m\times n}$ *be a rank* $r-$*matrix. The best rank k-matrix, where* $k < n$ *is the k largest singular values, accompanied with the k largest singular vectors. Moreover,*

$$\|\mathbf{A} - \mathbf{A}_k\| = \begin{cases} \sigma_{k+1}, & \text{for the } \|\cdot\|_2 \text{ norm.} \\ \sqrt{\sum_{i=k+1}^{r} \sigma_i^2}, & \text{for the } \|\cdot\|_F \text{ norm.} \end{cases} \tag{1.4}$$

1.1.1 Matrix Structures

This whole project is about extracting structure from matrices so that we can exploit that structure. Let's define some common structure that appears in my work.

Definition 1.1.9 (Toeplitz Matrix) *This is also known as "constant diagonal"* *matrix. Let* $\mathbf{A} \in \mathbb{R}^{m \times n}$*, then* $\mathbf{A}$ *is called* <u>*Toeplitz*</u> *if it meets the following criteria:*

$$\mathbf{A} = \begin{pmatrix} a_0 & a_{-1} & a_{-2} & \cdots & \cdots & a_{-(n-1)} \\ a_1 & a_0 & a_{-1} & \ddots & & a_{-(n-2)} \\ a_2 & a_1 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & a_{-1} & a_{-2} \\ \vdots & & \ddots & a_1 & a_0 & a_{-1} \\ a_{m-1} & \cdots & \cdots & a_2 & a_1 & a_0 \end{pmatrix} \tag{1.5}$$

While finding structure in the elements is great, we also want to find structure on more of a block level, where "blocks" are submatrices of the overall matrix.

Definition 1.1.10 (Block Toeplitz) *The definition of the "Block Toeplitz" matrix is the exact same except we are replacing scalars with matrices themselves. Let* $\mathbf{A} \in \mathbb{R}^{mp \times nl}$*, with* $m \times n$ *blocks, each of these blocks* $\mathbf{A}_i$ *is of the size* $p \times l$*, then* $\mathbf{A}$ *is called a* <u>*Block Toeplitz*</u> *matrix if it meets the following criteria:*

$$\mathbf{A} = \begin{pmatrix} \mathbf{A}_0 & \mathbf{A}_{-1} & \mathbf{A}_{-2} & \cdots & \mathbf{A}_{-(n-1)} \\ \mathbf{A}_1 & \mathbf{A}_0 & \mathbf{A}_{-1} & \ddots & \mathbf{A}_{-(n-2)} \\ \mathbf{A}_2 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \mathbf{A}_1 & \mathbf{A}_0 & \mathbf{A}_{-1} \\ \mathbf{A}_{m-1} & \cdots & \mathbf{A}_2 & \mathbf{A}_1 & \mathbf{A}_0 \end{pmatrix} \tag{1.6}$$

Remark 1.1.11 It is not hard to see that we can write this block matrix using an alternative formulation with the concept of a Kronecker product. For example, the last block matrix could also be

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & 0 & 1 & 0 \\ 0 & \cdots & 0 & 0 & 1 \end{pmatrix} \otimes \mathbf{A}_0 + \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \ddots & 0 \\ 0 & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & 0 & 0 & 1 \\ 0 & \cdots & 0 & 0 & 0 \end{pmatrix} \otimes \mathbf{A}_{-1} + \cdots \tag{1.7}$$

$$= \sum_i \mathbf{E}_i \otimes \mathbf{A}_i \tag{1.8}$$

where $\mathbf{E}_i \in \mathbb{R}^{m \times n}$ are matrices that "place" the blocks matrices where they need to be. This means that the block Toeplitz matrix $\mathbf{A}$ can be thought of as a sum of Kronecker products with those blocks. While the example above was demonstrated with a block Toeplitz matrix, this general framework works for any block matrix (even if the blocks are different sizes) just by changing the $\mathbf{E}_i$ "placement matrices".$\diamond$

Definition 1.1.12 (Symmetric Matrix) *A matrix* $\mathbf{A} \in \mathbb{R}^{n \times n}$ *is* <u>*symmetic*</u> *if and only if* $\mathbf{A} = \mathbf{A}^T$*, or for every* i, j *in the row space of* $\mathbf{A}$*, then* $a_{ij} = a_{ji}$*.*

Definition 1.1.13 (Positive Definite Matrix) *A symmetric matrix* $\mathbf{A} \in \mathbb{R}^{n \times n}$ *is* <u>*positive definite*</u> *if for all nonzero real-valued vectors* $\vec{z}$*, then* $\vec{z}^T \mathbf{A} \vec{z} > 0$*.*

Remark 1.1.14 If a matrix is both symmetric and positive definite, we will call them SPD. A property of SPD matrices is that all eigenvalues $\lambda_i > 0$, which means that it is invertible, as it is of full rank. $\diamond$

1.1.2 Norms and Error

Definition 1.1.15 (Frobenius Norm) *Let* $\mathbf{A}$ *be an* $m \times n$ *matrix with* $\operatorname{rank} r$*. The* <u>*Frobenius norm*</u> *of this matrix is the square root of the sums of the absolute squares of all of the elements, or the square root of the trace of* $\mathbf{A}^T \mathbf{A}$*, or the square root of the sums of the singular values of the matrix. Mathematically,*

$$\|\mathbf{A}\|_F := \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} |a_{ij}|^2} \tag{1.9}$$

$$:= \sqrt{\operatorname{tr}(\mathbf{A}^T \mathbf{A})} \tag{1.10}$$

$$:= \sqrt{\sum_{i=1}^{r} \sigma_i^2(\mathbf{A})} \tag{1.11}$$

Definition 1.1.16 (Spectral Norm) *Let* $\mathbf{A}$ *be an* $m \times n$ *matrix. The* <u>*spectral norm*</u> *of this matrix is the square root of the largest eigenvalue of* $\mathbf{A}^T \mathbf{A}$ *or the largest* *singular value of* **A**. *Mathematically, we have*

$$\|\mathbf{A}\|_2 := \sqrt{\lambda_{max}(\mathbf{A}^T\mathbf{A})} \tag{1.12}$$

$$:= \sigma_{max}(\mathbf{A}) \tag{1.13}$$

Remark 1.1.17 This is called the spectral norm because it is related to the spectral radius of a matrix, which is the largest magnitude of an eigenvalue, $|\lambda_{\text{max}}|$. $\diamond$

Lemma 1.1.18 *By application of the above definitions paired with the definition of Kronecker products, one can show that for matrices* $\mathbf{A} \in \mathbb{R}^{m\times n}, \mathbf{B} \in \mathbb{R}^{p\times \ell}$, *we have*

$$\|\mathbf{A}\otimes\mathbf{B}\|_2 = \|\mathbf{A}\|_2\|\mathbf{B}\|_2 \tag{1.14}$$

$$\|\mathbf{A}\otimes\mathbf{B}\|_F = \|\mathbf{A}\|_F\|\mathbf{B}\|_F \tag{1.15}$$

Theorem 1.1.19 (Sherman-Morrison-Woodbury Formula, 1949) *Let* $\mathbf{A} \in \mathbb{R}^{n\times n}$ *be an invertible matrix, and let* $\mathbf{U} \in \mathbb{R}^{n\times k}, \mathbf{V} \in \mathbb{R}^{k\times n}$, *where* $k < n$. *If we want to update* **A** *by* $\mathbf{U}\mathbf{V}^T$, *then assuming* $\left(\mathbf{I}_k + \mathbf{V}\mathbf{A}^{-1}\mathbf{U}\right)$ *is invertible, we have a numerically cheap way of computing* $\left(\mathbf{A} + \mathbf{U}\mathbf{V}^T\right)^{-1}$ *if we know what* $\mathbf{A}^{-1}$ *is already, which is*

$$\left(\mathbf{A} + \mathbf{U}\mathbf{V}^T\right)^{-1} = \mathbf{A}^{-1} - \mathbf{A}^{-1}\mathbf{U}\left(\mathbf{I}_k + \mathbf{V}\mathbf{A}^{-1}\mathbf{U}\right)^{-1}\mathbf{V}\mathbf{A}^{-1} \tag{1.16}$$

Definition 1.1.20 (Absolute and Relative Error) *Let* $\hat{\mathbf{A}}$ *be a matrix approximation to the matrix* **A**. *Then the* <u>*absolute error*</u>, $\|\mathbf{A} - \hat{\mathbf{A}}\|$ *is just the distance between the matrix and its approximation using whatever norm we are using. The* <u>*relative error*</u> *is the absolute error inversely scaled by the norm of the original matrix, namely* $\frac{\|\mathbf{A}-\hat{\mathbf{A}}\|}{\|\mathbf{A}\|}$.

1.2 Introduction to Tensors

Although the definition of a "tensor" might have different meanings to different fields, we will motivate our definition by examples. A scalar c, is a 0-way, or 0th order tensor, a vector $\vec{v}$ is a 1-way, or 1st order tensor, a matrix, **M** is just a 2-way

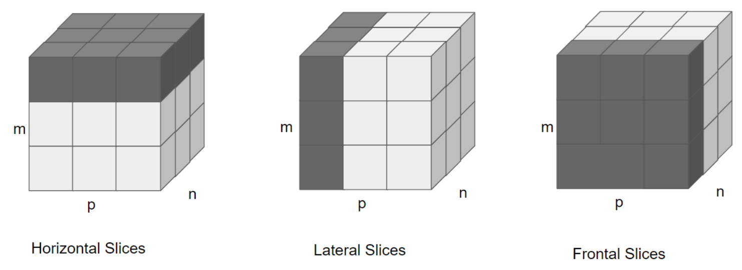

Figure 1.1: A visualization of the different slices of a tensor, $\mathcal{A} \in \mathbb{R}^{m \times p \times n}$. Note that each of these slices is a matrix in its own right.

or 2nd order tensor. For convention, we use the word "tensor" to mean a 3+ order tensor, and refer to anything less than this by their more common names– scalars, vectors, or matrices.

Definition 1.2.1 (Tensor) *With that definition let $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ be a d-way array, with elements $a_{i_1, i_2, \cdots, i_d}$.*

Sometimes we want to access an entire row of a matrix, say the i^{th} row of a matrix $\mathbf{M}$, we will denote that as $M_{i:}$, where the : means all the elements. Similarly, if we wanted to talk about the j^{th} column, we would denote that as $M_{:j}$. This same notation is used for accessing elements of a tensor.

Definition 1.2.2 (Slices of a Third order Tensor) *A slice of a 3rd-order tensor is simply a matrix, where one index is held fixed, and access all of the other elements in the other two dimensions.*

If we want to denote the i^{th} horizontal slice of a tensor $\mathcal{A}$, we'd write $\mathcal{A}_{i::}$, the j^{th} lateral slice, we'd write $\mathcal{A}_{:j:}$, and k^{th} frontal slice, we'd write $\mathcal{A}_{::k}$. The three slices (for the third order tensor we are dealing with) are visualized in Figure 1.1.

1.2.1 Turning Matrices into Tensors and Back Again

A lot of the matrices that we encounter coming from natural and practical sources are abundant in structure. However, this structure may not be inherent from the beginning, especially when we are dealing with large dense matrices. These redundances motivated researchers like Kilmer and Saibaba to come up with a methodology to

$$\begin{tikzcd}
\mathbf{M} \arrow[r, "\text{Matrix-to-tensor}"] & \mathcal{T} \arrow[d, "\texttt{tr-HOSVD}"] \\
\widehat{\mathbf{M}} \arrow[u, "\approx"] & \widehat{\mathcal{T}} \arrow[l, "\text{Tensor-to-matrix}"]
\end{tikzcd}$$

Figure 1.2: The original matrix $\mathbf{M}$ is mapped into a tensor $\mathcal{T}$, which is then approximated using a tensor decomposition, $\widehat{\mathcal{T}}$. Lastly, $\widehat{\mathcal{T}}$ is mapped back to a matrix $\widehat{\mathbf{M}}$.

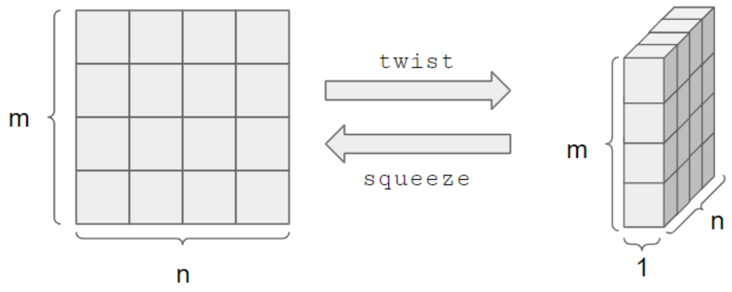

Figure 1.3: The bijective mapping between an $m \times n$ matrix and an $m \times 1 \times n$ tensor. The forward function is called "twist" while the inverse function is "squeeze".

exploit this latent structure [18]. The example that was presented was a symmetric positive definite matrix which was also block Toeplitz, so dividing the entire matrix into these blocks, then converting to a tensor allowed the researchers to isolate only the non-redudant information necessary to reconstruct the matrix. Then, they compressed the tensor representation and mapped the tensor representation back into a matrix, thereby uncovering other latent structure. An outline of this procedure is given in Figure 1.2.

Once we have these blocks of structure, we can take them and convert them into lateral slice of a tensor. This process is bijective. We do this by performing a bijective function on these sub blocks. To turn a matrix into a tensor, we "twist" it into a higher dimension, and to turn a tensor into a matrix, "squeeze" out a dimension. This bijective action is visualized in Figure 1.3. So we take these blocks, twist them into lateral slices and then concatenate these lateral slices to form a tensor as seen in Figure 1.4.

Even though Figure 1.4 gives a visual understanding of the bijective mapping in [18], it fails to demonstrate the novelty. If the first matrix in the Figure had no

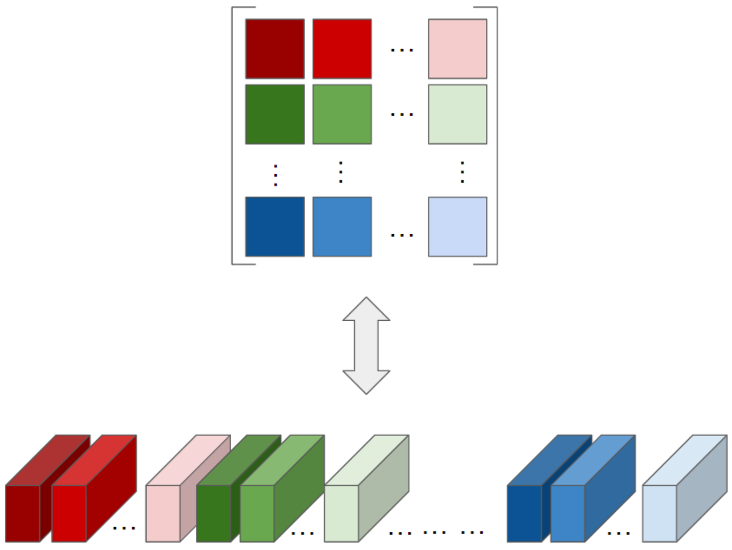

Figure 1.4: We can twist these submatrices into lateral slices of a tensor. Similarly, we can twist the lateral slices back into a matrix.

underlying block structure, then this is exactly what would happen, but this is not the case we are dealing with. In fact, for a block Toeplitz matrix, most of the blocks are repeated and don't need to be included in the tensor. Recall that a block Toeplitz matrix is constant along the diagonals, so it is uniquely specified by $mn - 1$ blocks- the p blocks along the first row, and the n blocks along the first column, exlcuding the repetition of the (1,1) block. The method in [18] was designed specifically for block-structured matrices, so the tensor to be decomposed stores fewer entries then the (possibly dense) starting matrix. The tensor stores only non-redundant blocks, arising from the block structure, symmetry arguments, etc. Our goal will be to generalize their approach for a special class of dense matrices.

1.2.2 Tensor Ranks and Decompositions

For matrices, the rank is well defined and can be computed in polynomial time in the size of the matrix. For tensors, however, there is more than one notion of rank, and which rank depends on the decomposition used. Once we make a tensor, it is time to find the rank and illuminate it. This is done through a tensor decomposition. While there are many different kinds of tensor decompositions, we are only going to talk about the CANDECOMP/PARAFAC (CP) decompositon [15] and the `tr-HOSVD`,

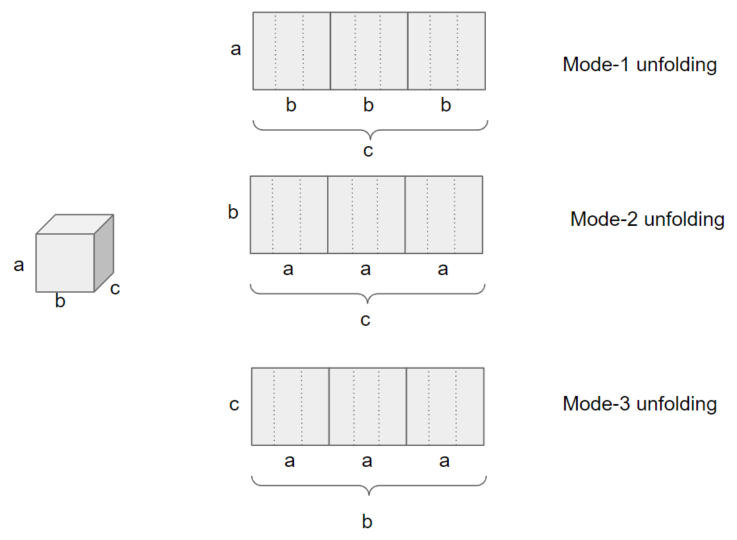

Figure 1.5: The three mode$-k$ unfoldings of a third order tensor, $\mathcal{A}$.

or the truncated-Higher Order Singular Value Decomposition [11]. First we need to define a few more properties that arise from higher dimensions. While we now have a way of talking about these slices, we can use that to reorder the structure and make tensors matrices by unfolding them, and matrices into tensors by refolding them. We do that through tensor unfoldings, sometimes called "matricization". We see the different ways of unfolding a third order tensor in Figure 1.5.

Definition 1.2.3 (k^{th}-mode Tensor Unfolding) *Let $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times \cdots \times n_d}$ be a d-way tensor. Then the k^{th}-mode unfolding is defined as*

$$\mathbf{A}_{(k)} \in \mathbb{R}^{n_k \times n_1 n_2 \cdots n_{k-1} n_{k+1} \cdots n_d} \tag{1.17}$$

Note that the ordering ultimately doesn't matter as they will just be different permutations of the same tensor. As long as each unfolding is performed, the work is consistent.

Definition 1.2.4 (mode-k product) *The <u>mode-k product</u> is a way of denoting a tensor-matrix product, where the tensor is unfolded in the k^{th} mode and left multiplied* *by a matrix, assuming matrix dimensions match. Mathematically,*

$$\mathcal{A} \times_i \mathbf{U} \coloneqq \mathbf{U}\mathbf{A}_{(i)} \tag{1.18}$$

Now that we have defined this operation, the following higher-order SVD follows nicely. This is the natural generalization of the standard SVD which we already defined in 1.1.6.

Definition 1.2.5 (CP decomposition [15]) *The CP decomposition, which stands for CANDECOMP (canonical decomposition)/PARAFAC (parallel factors) is expressing the tensor as a finite sum of rank - one tensors. For a tensor $\mathcal{X} \in \mathbb{R}^{I \times J \times K}$, we define the CP decomposition as*

$$\mathcal{X} \approx \sum_{r=1}^{R} \mathbf{a}_r \circ \mathbf{b}_r \circ \mathbf{c}_r \tag{1.19}$$

where $0 < R < \infty, \mathbf{a}_r \in \mathbb{R}^I, \mathbf{b}_r \in \mathbb{R}^J, \mathbf{c}_r \in \mathbb{R}^K$ for $r = 1, 2, \cdots, R$, and $\circ$ is the standard vector outer product.

Definition 1.2.6 (HOSVD [11, 18]) *Once we have unfolded the tensors along modes 1,2 and 3*, we perform an SVD computation on all of the matricizations, keeping the left singular vectors in each case, denoted $\mathbf{U}, \mathbf{V}, \mathbf{W}$ for the first, second, and third matricization, respectively. Then the HOSVD is performed by computing the core tensor $\mathcal{G}$ by*

$$\mathcal{G} \coloneqq \mathcal{A} \times_1 \mathbf{U}^T \times_2 \mathbf{V}^T \times_3 \mathbf{W}^T \tag{1.20}$$

Once we have the core tensor, we can truncated it in any mode possible, or we can keep it full rank, and then we "undo" the process to get a tensor approximation, namely

$$\widehat{\mathcal{A}} \approx \widehat{\mathcal{G}} \times_1 \widehat{\mathbf{U}} \times_2 \widehat{\mathbf{V}} \times_3 \widehat{\mathbf{W}} \tag{1.21}$$

*Since the rest of the paper deals only with third order tensors, we just say modes 1,2, and 3, but this idea is easily extendible to any dimension tensor.

The standard algorithm of `tr`-HOSVD is given below in pseudocode in Algorithm 1, and the modified algorithm of sequentially `tr`-HOSVD is given in Algorithm 2. While both are presented for third order tensors with truncation ranks (r_1, r_2, r_3), the algorithms are easily extendable to whatever order tensor you have.

Algorithm 1 `tr`- Higher Order Singular Value Decomposition

```
Require: $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, truncation ranks $r_i, i = 1, 2, 3$
  $A_i \leftarrow$ matricization of $\mathcal{A}$ along mode $i$
  $U_i \leftarrow$ SVD($A_i$, econ)
  $\mathcal{G} \leftarrow \mathcal{A} \times_i U_i^T$                              ▷ Construct Core
  $\widehat{\mathcal{A}} \leftarrow \mathcal{G} \times_i U_i$                   ▷ Construct Approximation
  $\widehat{\mathcal{A}} \leftarrow \widehat{\mathcal{A}}(1:r_1, 1:r_2, 1:r_3)$              ▷ Truncate
  return $\widehat{\mathcal{A}}$
```

Algorithm 2 Sequentially `tr`- Higher Order Singular Value Decomposition

```
Require: $\mathcal{A} \in \mathbb{R}^{n_1 \times n_2 \times n_3}$, truncation ranks $r_i, i = 1, 2, 3$
  $A_i \leftarrow$ matricization of $\mathcal{A}$ along mode $i$
  $U_i \leftarrow$ SVD($A_i$, econ)
  $U_i \leftarrow U_i(:, 1:r_i)$                               ▷ Sequentially Truncate
  $\widehat{\mathcal{G}} \leftarrow \mathcal{A} \times_i U_i^T$                              ▷ Construct Core
  $\widehat{\mathcal{A}} \leftarrow \widehat{\mathcal{G}} \times_i U_i$                   ▷ Construct Approximation
  return $\widehat{\mathcal{A}}$
```

Remark 1.2.7 Depending on when you truncate the core, you get two different outcomes. If you truncate the core before you use it to form the tensor approximation, that is called <u>sequentially `tr-HOSVD`</u>. However, if you truncate the final tensor approximation, that is called just a `tr-HOSVD`. Sequentially truncating a tensor results in less memory and suprisingly similar results. ◇

When wanting to construct a low-rank approximation of a tensor, it is imporant to know the rank or calculate it easily. However, approximating the tensor by a sum of rank 1 tensors[†] (which is the CP decomposition among other names) is NP-hard [14]. In fact, in a more general sense, tensor rank is NP-complete for finite fields and NP-hard over $\mathbb{R}$ and $\mathbb{C}$ [16], which is not good since our problems are all over $\mathbb{R}$. Other properties related to tensors, like finding or appoximating eigenvalues, finding

[†]A rank-1 tensor for a third order tensor is a three-way outer product among three vectors

or appoximating singular values, or approximating the spectral norm of a tensor within a certain accuracy are also known NP-hard [14].

It would be nice to know that this technique is not costing us anything in terms of approxation error, because then that would not be advantagous. From the schema presented in Figure 1.2, we have the following:

Lemma 1.2.8 (Kilmer, Saibaba, 2021) *Let* $\mathbf{A} \in \mathbb{R}^{(\ell m)\times(qn)}$ *and let* $\mathcal{T}_{\mathcal{E}}[\cdot]$ *and* $\mathcal{M}_{\mathcal{E}}[\cdot]$ *be the associated tensor-to-matrix and matrix-to-tensor mappings respectively. Let* $\mathbf{\mathfrak{X}} = \mathcal{T}_{\mathcal{E}}[\mathbf{A}]$ *and let* $\mathbf{\mathfrak{T}} \approx \mathcal{T}_{\mathcal{E}}[\mathbf{A}]$ *be a tensor approximation computed using any appropriate method. Then the error in the matrix approximation* $\widehat{\mathbf{A}} = \mathcal{M}_{\mathcal{E}}[\mathbf{\mathfrak{T}}]$ *satisfies*

$$\|\mathbf{A} - \widehat{\mathbf{A}}\|_F = \|\mathbf{\mathfrak{X}} - \mathbf{\mathfrak{T}}\|_F.$$

While this theorem is not going to be directly applicable to our matrix approximations because our matrix-to-tensor mapping is hierarchical, it motivates us to come up with a similar result for our matrix-to-tensor approximations.

1.3 Introduction to discretized PDEs

While most of this research is based in numerical linear algebra and tensor decompositions, the matrices that we are dealing with come from a discretized fractional PDE. The following definitions give an approach to how these matrices are actually constructed. Fractional PDEs are just a specific type of differential equation, so we would expect that the solution should have some smoothness, but how is this actually computed on a computer. Since computers are finite-memory machines, a "mesh" or "grid" is constructed. partitioning the domain, and the solution is computed at those specific meshpoints using a finite element schema. Now, let's touch briefly on what meshes can be.

If we want the error to be small, we would take $n \to \infty$. If we wanted to solve this using a direct solver, we would need n^3 operations, and n^2 pieces of memory. That's bad! We see just from looking at this matrix, that there is some structure, so using this structure we can speed up matrix-vector products (mat-vecs).

1.3.1 Preconditioning

As was just stated solving a linear system $\mathbf{A}\vec{u} = \vec{f}$ simply by computing $\mathbf{A}^{-1}$ and applying it to $\vec{f}$ is a direct solve that requires $\mathcal{O}(n^3)$ flops; however, this method is numerically instable and is very rarely used in practice. The closest might be performing the matrix factorization $\mathbf{A} = \mathbf{PLU}$, and then performing the respective forward and backward substitutions to solve the system. To combat the direct methods, iterative methods are often the solver of choice because of the reduced cost in each iteration. The golden standard of iterative methods are Krylov subspaces methods as at each iteration, the cost is simply a matvec ($\mathbf{A} \times (\mathbf{A}^{i-1}\vec{x})$ for the i^{th} iteration). Then the total cost of solving the linear equation is just the number of iterations until convergence is met times the cost of the matvec.

A common approach for these iterative methods is matrix splitting, where you take the matrix $\mathbf{A} = \mathbf{M} - \mathbf{N}$, where $\mathbf{M}$ is not only invertible but also easy to invert. For example, in Jacobi's method, $\mathbf{M} = \mathbf{D}$ which is just the diagonal entries, which is easy to invert as you can just take the inverse of each element. Another example is Gauss-Seidel's method where $\mathbf{M} = \mathbf{D} - \mathbf{U}$, where $\mathbf{U}$ is the strict upper triagular parts of the matrix. Since this is a triangular system, the forward triangular solve is used. This means we are hoping that we have fast convergence which comes from the condition number of the matrix $\mathbf{M}^{-1}\mathbf{A}$ to be smaller. That is the point of preconditioning - to converge to the final solution faster.

Definition 1.3.1 (Preconditioner) *Assuming* $\mathbf{P}, \mathbf{A}$ *are SPD, like our case, a* *preconditioner* $\mathbf{P}$ *of* $\mathbf{A}$ *is a matrix such that* $cond(\mathbf{P}^{-1}\mathbf{A}) \leq cond(\mathbf{A})$

Definition 1.3.2 (Left Preconditioned System) *If we are trying to solve* $\mathbf{A}x = b$*, then the* *left preconditioned* *system is*

$$\mathbf{P}^{-1}(\mathbf{A}x - b) = 0 \tag{1.22}$$

A candidate for a possible choice of preconditioner (not a solver for our fPDE) might be a stationary iterative method.

Definition 1.3.3 (Stationary Iterative Methods) *A stationary iterative method is one where the iteration scheme can be manipulated into*

$$\vec{x}^{(k)} = \mathbf{B}\vec{x}^{(k+1)} + \vec{c} \tag{1.23}$$

The examples above of Jacobi and Gauss-Seidel are called stationary iterative methods since $\mathbf{B} = \mathbf{M}^{-1}\mathbf{N}$*, and* $\mathbf{B} = (\mathbf{D} - \mathbf{U})^{-1}\mathbf{L}$ *respectively, where* $\mathbf{L}$ *is the strict lower triangular part of* $\mathbf{A}$*.*

Remark 1.3.4 While of course we don't use preconditioners for something as simple as a stationary iterative method, we can use the idea to make a preconditioner. These methods might be beneficial in the long run because they are guaranteed to converge for strongly diagonally dominant systems. ◊

The reason that matrix splitting and stationary methods are mentioned is that the method proposed in this thesis (Chapter 3) can be thought of as a matrix splitting, and we can leverage that to use as a possible preconditioner. Mathematically, we have that our matrix splitting preconditioner, $\mathbf{P}^{-1}$ could be applied to solve our system $\mathbf{P}^{-1}\vec{v}$ would be a few steps of the stationary iterative method with our matrix approximation. (To see how this relates to a matrix splitting, please consult Figure 3.6.)

Definition 1.3.5 (Block Jacobi Preconditioning) *As mentioned above, the Jacobi, or block, preconditioner is just approximating the matrix by taking only the diagonal entries, which is easy to invert. However, most scalar entried subroutines have a block analogue; Jacobi is not unique. The block Jacobi preconditioner is taking the blocks along the main diagonal and using those to approximate the matrix. Since we are ignoring all the other blocks, we just have to invert the matrix subblocks along the diagonal. One could easily see how this would work for non-overlapping block, but some alterations have been shown for overlapping diagonal blocks1.6.*

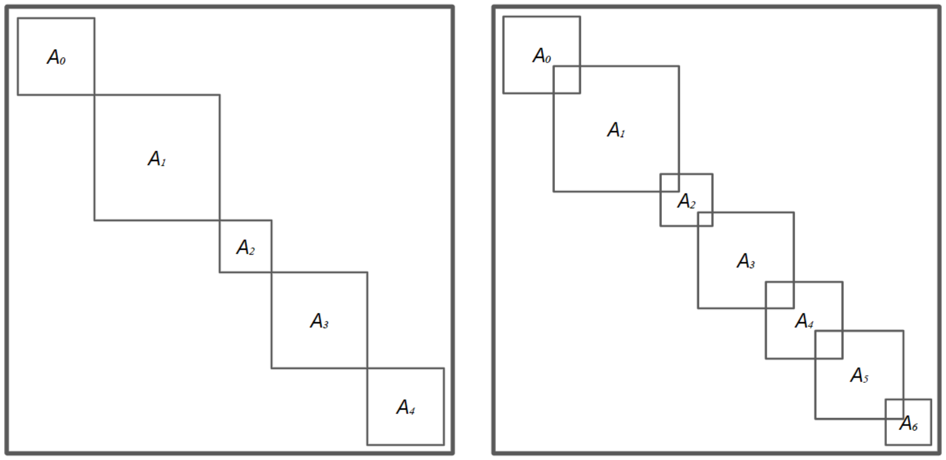

Figure 1.6: On the left, a non-overlapping Jacobi block pattern is presented. On the right, an overlapping Jacobi block pattern is presented.

Remark 1.3.6 This might be benefical to the system we are analyzing later as we have an SPD system that is strongly diagonally dominant. Lastly, with Kronecker products, it is well know that if $\mathbf{B}$ has some sort of entry-wise structure, then regardless of what structure $\mathbf{A}$ possesses, $\mathbf{A} \otimes \mathbf{B}$ will inherit the block analogue of whatever entry-wise structure $\mathbf{B}$ has. Further discussion of Block Jacobi preconditioning is relegated to section 4.4. $\diamondsuit$

1.3.2 Hierarchical Matrices ($\mathcal{H}$ - matrices)

As a general matrix $\mathbf{A}$ modeling real world problems get bigger and bigger, even on the scale of $n = 1{,}000{,}000$, the number of operations needed to do basic matrix operations $\mathbf{A}\vec{x}, \mathbf{A} * \mathbf{B}, \mathbf{A} + \mathbf{B}$ require either $\mathcal{O}\big(n^2\big)$ or $\mathcal{O}\big(n^3\big)$ operations naively. This motivates the need for a different matrix representation that can allow these operations to be performed more quickly [13].

Definition 1.3.7 (Hierarchical Matrix) *A* <u>*hierarchcial matrix*</u> *or $\mathcal{H}$-matrix is a representation of a matrix (typically non-dense) of size n such that matrix operations on a wide class matrices can take place in either $\mathcal{O}(n)$ or $\mathcal{O}\big(n \log^k n\big)$ (almost-linear) time, where k is a tunable approximation parameter.*

Remark 1.3.8 Although $\mathcal{O}(n)$ and $\mathcal{O}\big(n \log^k n\big)$ are not the same asymptotic behavior, since $\log^k n$ grows very slowly, and the fact that the constant $c_1 \gg c_2$ preceeding these two asymptotics could allow for $c_1 n$ to take longer than $c_2 n \log^k n$, and in practice they are very similar [2]. $\diamond$

Since matrices are very rarely globally low rank, the way that the $\mathcal{H}$-matrix finds these blocks of low rank is through subdivision of the matrix and it is called cluster tree construction, trying to find "admissible blocks" or low rank blocks, as this thesis will call them. This leads to a hierarchical structure of the cluster tree, the ability to find these low-rank blocks, factorize them, and still overall approximate the original matrix well, as illustrated by the following two lemmas.

Lemma 1.3.9 (Local Matrix Approximation Error [5]) *The elementwise error for the matrix entries G_{ij} approximated by the degenerate kernel $\tilde{g}$ in the admissible block $t \times s$ (and g in the other blocks) is bounded by*

$$|\mathbf{G}_{ij} - \widehat{\mathbf{G}_{ij}}| \leq \frac{3}{2} n^{-2} 3^{-k} \tag{1.24}$$

Lemma 1.3.10 (Global Approximation Error [5]) *The approximation error $\left\|\mathbf{G} - \hat{\mathbf{G}}\right\|_F$ in the Frobenius norm for the matrix $\hat{\mathbf{G}}$ built by the degenerate kernel $\tilde{g}$ in the admissible blocks $t_\nu \times s_\nu$, and g in the inadmissible blocks is bounded by*

$$\left\|\mathbf{G} - \widehat{\mathbf{G}}\right\|_F \leq \frac{3}{2} n^{-1} 3^{-k} \tag{1.25}$$

Because of these factorizations, and the structure in which they are constructed, $\mathcal{H}$-matrices can take a dense matrix and turn it into a data-sparse representation.

1.3.3 Fractional Partial Differential Equations

Definition 1.3.11 (Uniform and Adaptive mesh) *A <u>uniform mesh</u> is an example of taking a domain $[a, b]$ and dividing it into equal spaced pieces. So for example, the uniform mesh, consisting of n nodes of a domain $[a, b] \in \mathbb{R}$ is $a + ih$, where $h = (b - a)/n$, and $i = 0, \cdots, n - 1$. An <u>adaptive mesh</u> is one that changes based on the problem at hand. If there is an area of great change and the uniform mesh doesn't* *meet the requirements to accurately capture the change, then we subdivide that area adaptively so that we can capture the true nature of the solution.*

Ultimately the fractional PDE equation that we want to solve is

$$\mathcal{D}_x^\alpha u(x) = f(x), x \in (b, c) \tag{1.26}$$

where $f(x) \in L^2([b, c])$. We now define what the fractional differential operator is.

Definition 1.3.12 (Integral definition) *The fractional integral of order $\alpha \in \mathbb{R}^+ \times i\mathbb{R}$, for function $f(x)$:*

$$({}_b\mathcal{I}_x^\alpha f)(x) = \frac{1}{\Gamma(\alpha)} \int_b^x \frac{f(s)}{(x-s)^{1-\alpha}} \, \mathrm{d}s\,, \quad x > b \tag{1.27}$$

Definition 1.3.13 (Riesz fractional derivative) *The Riesz fractional derivative of order $\alpha \in (1, 2)$ for the function $f(x)$:*

$$\begin{aligned} \mathcal{D}_x^\alpha f(x) &= -\frac{1}{2\cos\alpha\pi/2\Gamma(2-\alpha)} \frac{\mathrm{d}^2}{\mathrm{d}x^2} \int_b^c |x-\xi|^{1-\alpha} f(\xi) \, \mathrm{d}\xi & (1.28) \\ &= -\frac{1}{2\cos\alpha\pi/2} \left[{}_b^{RL}\mathcal{D}_x^\alpha f(x) + {}_x^{RL}\mathcal{D}_c^\alpha f(x) \right] & (1.29) \end{aligned}$$

Remark 1.3.14 It is easy to see that if $\alpha = 2$, then we have the normal Laplacian kernel, and as $\alpha \nearrow 2$, the resulting solution gets less smooth, which results in harder computational problems. ◇

For generality, we are solving equation (1.26) over one dimension with Dirichlet boundary conditons, namely $u(b) = u(c) = 0$. To discretize this fPDE, we are using the standard Galerkin method, where this is projected onto a finite dimensional space which is a subset of the Sobolev space $H_0^1([b, c])$.

$$\int_b^c \mathcal{D}_x^\alpha u(x) \phi_i(x) \, \mathrm{d}x = \int_b^c f(x) \phi_i(x) \, \mathrm{d}x \tag{1.30}$$

Now integrating by parts, and recognizing that this fPDE has homogeneous

Dirichlet boundary conditions, we find a weak solution of this equation is

$$\int_b^c \left[\frac{1}{2C_\alpha} \frac{\mathrm{d}}{\mathrm{d}x} \int_b^c |x-\xi|^{1-\alpha} u(\xi)\,\mathrm{d}\xi \right] v'(x)\,\mathrm{d}x = \int_b^c f(x)v(x)\,\mathrm{d}x\,, \quad \forall v \in \mathcal{V} \tag{1.31}$$

, where $C_\alpha = \cos(\alpha\pi/2)\Gamma(2-\alpha)$.

Now we discretized the system using whatever discretizatoin schema we need $u_h = \sum_{j=1}^N u_j\phi_j \in \mathcal{V}$. Now we have the coefficient vector of all the u_h, which can be rewritten as a linear system $\mathbf{A}\vec{u} = \vec{f}$, where

$$\mathbf{A}_{ij} := \int_b^c \left[\frac{1}{2C_\alpha} \frac{\mathrm{d}}{\mathrm{d}x} \int_b^c |x-\xi|^{1-\alpha} \phi_j(\xi)\,\mathrm{d}\xi \right] \phi_i'(x)\,\mathrm{d}x \tag{1.32}$$

$$f_i := \int_b^c \phi_i(x) f(x)\,\mathrm{d}x \tag{1.33}$$

It is not hard to verify that this matrix has all nonzero elements, so this naturally leads to some sort of approximation schema.

To elucidate any confusion, on the uniform mesh $\{0, \frac{1}{7}, \frac{2}{7}, \cdots, \frac{6}{7}, 1\}$, we obtain the following $\mathbf{A} \in \mathbb{R}^{8\times 8}$:

$$\mathbf{A} = \begin{pmatrix}
3.7391 & -1.4081 & -0.2967 & -0.0694 & -0.0309 & -0.0170 & -0.0106 & -0.0071 \\
-1.4081 & 3.7391 & -1.4081 & -0.2967 & -0.0694 & -0.0309 & -0.0170 & -0.0106 \\
-0.2967 & -1.4081 & 3.7391 & -1.4081 & -0.2967 & -0.0694 & -0.0309 & -0.0170 \\
-0.0694 & -0.2967 & -1.4081 & 3.7391 & -1.4081 & -0.2967 & -0.0694 & -0.0309 \\
-0.0309 & -0.0694 & -0.2967 & -1.4081 & 3.7391 & -1.4081 & -0.2967 & -0.0694 \\
-0.0170 & -0.0309 & -0.0694 & -0.2967 & -1.4081 & 3.7391 & -1.4081 & -0.2967 \\
-0.0106 & -0.0170 & -0.0309 & -0.0694 & -0.2967 & -1.4081 & 3.7391 & -1.4081 \\
-0.0071 & -0.0106 & -0.0170 & -0.0309 & -0.0694 & -0.2967 & -1.4081 & 3.7391
\end{pmatrix}$$

Chapter 2

Research Question - how to approximate and precondition the dense matrix in an efficient way?

In this thesis, we hope to address the issue of numerical solutions of fPDEs as efficiently as possible. This means we need to address the issue of storage and computations with the matrix. We aim to improve on the approximation to the matrix, which is storage efficient but better captures features of the original matrix. We do this via tensor decomposition.

2.1 What has been done

As humanity seeks to better model the world around us, we see that the traditional partial differential equations (PDEs) do not accurately suffice on all natural phenomena, but fractional PDEs (fPDEs) have been growing in popularity and research. Some examples of useful applications of fPDEs include those in the fields of physics, anomalous diffusion, poroelastic/viscoelastic processes, fluid dynamics, signal processing, electromagnetics, and economic/financial models [1]. The known numerical methods to solve fPDEs all lead to very dense matrices that have no nonzero elements. This means that $\mathcal{O}\big(n^2\big)$ elements have to be stored in memory, and to directly solve this linear system, $\mathcal{O}\big(n^3\big)$ operations have to be used. As problems are getting larger and larger to mitigate errors, the process of storing and solving these systems become prohibitively expensive. New techniques must be developed that make these computational problems feasible.

One way to overcome the fact that the matrices arising from fPDE discretization are dense is to use the Toeplitz stucture that is achieved from a uniform mesh discretization over the domain. Toeplitz matrices lend themselves nicely to be solved using Krylov subspace methods such as Conjugate Gradient and GMRES, which can lead to fast matrix-vector products [7]. These methods are well-studied and known to be well preconditioned by circulant matrices, which keep the Toeplitz structure and add periodicity. Then one can use the fast Fourier Transform (FFTs) to perform a Toeplitz matrix-vector product in $\mathcal{O}(n \log n)$ operations, where n is the size of the matrix (related to the spacial grid) and $\mathcal{O}(n)$ in memory [8]. These methods are fast to apply and show promising clustering of the spectrum of the preconditioner applied to the orginal matrix, thus speeding up convergence [6].

However, these methods all are depedent on the Toeplitz structure of the matrix. In certain cases, this uniform mesh isn't suitable to deal with singularities that happen on the boundary, so adaptive meshes have to be used, which breaks the Toeplitz structure. For example, in Figure 2.1, which is an adaptive mesh used to solve some fPDEs, it is easy to see that there is a smaller mesh size around the boundaries, then a wider mesh size around one quarter into the domain, then a return to a tighter discretization in the middle half [25]. This clearly isn't Toeplitz as we are now dealing with an adaptive grid, so the distances between nodes will not be uniform, so investigations into how to combat or work around these singularities should take place. One work around might be to use known polynomials that can combat this singularity, but this requires some knowledge of the solution and where the singularity is [9]. This begs the question; what methods are available to us to use less data and solve the problem faster if the mesh is non-uniform or adaptive as to mitigate the unknown singularities of the problem?

One such way might be to use a hierarchical matrix ($\mathcal{H}$-matrix), briefly introduced in 1.3.2, which has become a great tool of dealing with dense matrices especially those arising in the discretization of PDEs in a data sparse, not necessarily actually sparse way [24]. They can take $\mathcal{O}(n^2)$ amount of storage and represent that as an approximation using only $\mathcal{O}(nk \log n)$, where k is a parameter depending on how good of an appoximation you want. Similarly, they can perform matrix operations, like the multiplication of two general $n \times n$ matrices, which typically is done

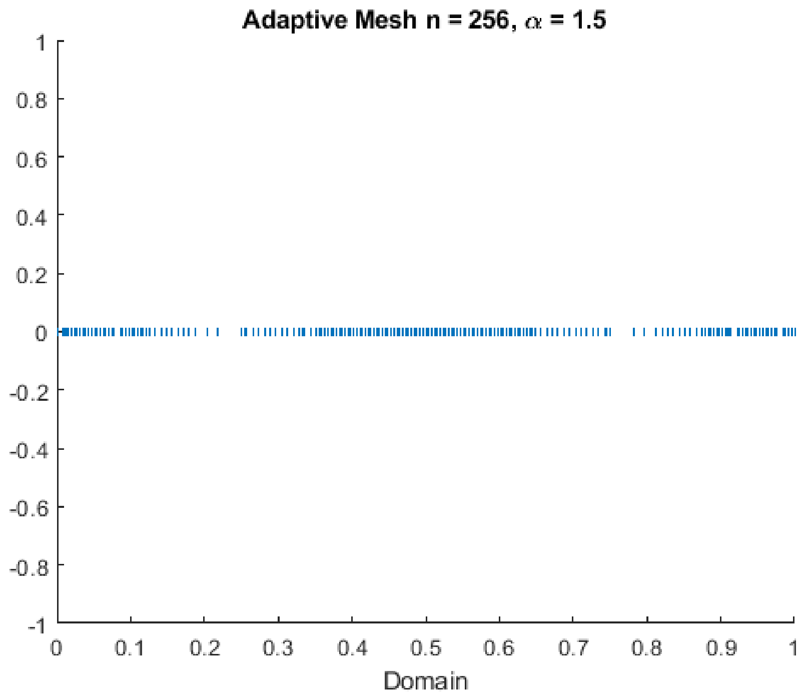

Figure 2.1: The mesh discretization for fPDEs needs to be adaptive, which breaks the Toeplitz structured problem.

in $\mathcal{O}\big(n^3\big)$ time, and perform them in $\mathcal{O}\big(n \log^k n\big)$ time [13]. This provides a powerful starting block for numerical methods based on these dense matrices.

While there are many ideas on how to solve these fractional linear system, like geometric multigrid [25], matrix splitting [10], and block preconditioning [4], many of these methods rely on Kronecker product representations of the preconditioners. While there are many different ways to think of these tensor product representations, one could think of them in dimensions higher than just the two dimensional matrices.

There are quite a few references of using higher order structures, such as tensors, to help solve these preconditioning problems in a variety of applications from image debluring to integral equations and PDEs [17, 20]. Tensors provide a great structure to study these problems that naturally live in higher dimension where they are, and not that data projected on something of a lower dimension. The use of tensor based numerical methods for these fPDEs have come about in either solving the original problem [23] or preconditioners for the problem [3, 22]. Most of these methods are based on the Higher Order SVD method, but Bertaccini is based on Tensor

Train-GMRES. Tensor-train [21] really shines in a very high number of dimensions because it is linear in all of those dimensions, but for our problem, a small number of dimensions is satisfactory, so we will use the HOSVD algorithm.

Because HOSVD works will in a small number of dimensions, it is well studied. In fact, there are applications to reducing the storage requirement of the densely populated structure, as we are truncating after the most significant contributions in each dimension. There are also many nice properties that are observed in both the tensor and matrix setting. For example, using a Kronecker product framework of turning a matrix into a tensor and decomposing the tensor using HOSVD, then converting back into a block matrix has the same Frobenius error as performing a decomposition on the original matrix [18]. This also has the advantage of dealing with the higher dimensional data in its more natrual environment. While this method does a great job of turning dense matrices into approximations using the HOSVD, the method is limited by only dealing with block low rank matrices. However, what if the matrices have a slightly modified structure, of not block low rank, but hierachically low rank (or data-sparse representations)?

The scope of this thesis is to answer exactly that. These fPDEs are dense matrices, and using the Toeplitz structure doesn't account for the singularies that may arise even with smooth initial data. Therefore, the well studied solutions and preconditioners of Toeplitz matrices cannot be used. There is hierarchical structure that can and should be exploited. This thesis develops a novel tensor based technique that both approximate the matrix better using less stroage than known methods and preconditioner the matrix so that the solutions can be found much faster. The method can approximate the original stiffness matrix in a much more sparse representation using these hierarchial off-diagonal blocks of low rank, which uses less data and closer approximates the dense matrix. This method also is found to be an effective preconditioner for how small it is in the number of elements in storage.

2.2 Leverage the approximation

The methodology for turning matrices into tensors when they are all the same dimension $m \times n$, is documented in [18] to uncover possible additional latent structure, such as block low rank structure. For convenience, the algorithm is presented below [Algorithm 3].

Algorithm 3 Turning BLR matrices into Tensors

```
Require: $\mathbf{A}$
  Partition $\mathbf{A}$ into block matrices $\mathbf{A}_{ij} \in \mathbb{R}^{m \times n}$
  Encode the position of $\mathbf{A}_{ij}$ by $\mathbf{E}_{ij}$ “placement matrix"
  for All nonredundant off-block diagonal submatrices do
      Twist $\mathbf{A}_{ij}$ into $\mathcal{A}$ lateral slices of the tensor
      Perform tr-HOSVD on $\mathcal{A}$ to get $\widehat{\mathcal{A}}$
      Squeeze the lateral sliced of$\widehat{\mathcal{A}}$ into matrices placed in the $(i, j)$ position by $\mathbf{E}_{ij}$
  end for
  return $\widehat{\mathbf{A}}$
```

This generic framework works really well if the structure is based on block low rank as the Kronecker structure that we are encoding by $\mathbf{E}_{ij}$ causes us to inherit some structure from the original matrix to our approximation causing some savings. However, our problem has hierarchical structure so we want to find a way to perform this same principle on multiple tensors of varying block sizes.

Two main classes of structured matrices were considered in [18] – block Toeplitz (with possible Toeplitz blocks) and block low-rank. For specificity, please consult their paper, as we are trying to generalize their results.

Starting with the Block Toeplitz case, if $\mathfrak{T} \approx \mathcal{T}_{\mathcal{E}}[\mathbf{A}]$, both the CP and Tucker decompositions can be shown to correspond to

$$\widehat{\mathbf{A}} = \mathcal{M}_{\mathcal{E}}[\mathfrak{T}] = \sum_{j=1}^{\tau} \mathbf{C}_j \otimes \mathbf{D}_j$$

where $\tau = r$ is the rank of the CP approx, or $\tau = r_2$ is the mode-2 truncation index. As was previously noted, the entry-wise structure of $\mathbf{C}_j$ comes from the block-wise structure of the original matrix. Similarly, if we have $\mathbf{D}_j$ with low rank (small r for CP, and small r_1, r_3 for Tucker), then serious storage savings can occur. A diagram

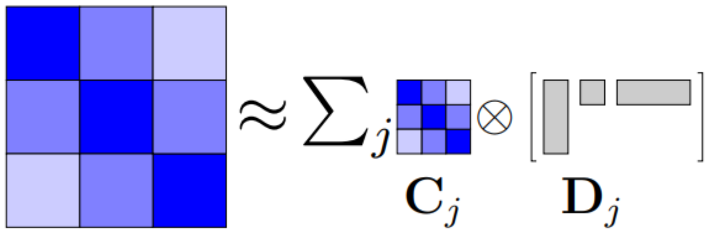

Figure 2.2: When a matrix has block structure, it can be approximated a sum of Kronecker products.

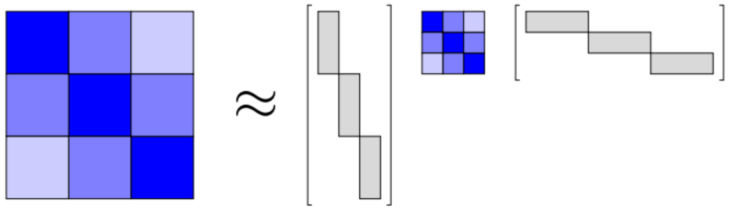

Figure 2.3: When a matrix has block structure, it can be approximated by orthogonal matrices, left and right multiplied by, a block-rank structured matrix.

of what this can look like is seen in Figure 2.2.

This paper also extrapolates this finding to a more broad class of matrices– those with no block structure other than low rank blocks. Regardless of the tensor method used (see [19] for more details about tensors decompositions than those presented here), like rank r in CP and rank (r_1, r_2, r_3) in Tucker, the approximate tensor $\mathfrak{T}$ when mapped back to a matrix has the following structure:

$$\mathcal{M}_{\mathcal{E}}[\mathfrak{T}] = (\mathbf{I}_\ell \otimes \mathbf{U})\texttt{struct}_{\mathcal{E}}\left(\mathbf{G}_k\right)_{k=1}^{p}(\mathbf{I}_q \otimes \mathbf{W}^\top). \tag{2.1}$$

where $\mathbf{G}_k$ is $\tau_1 \times \tau_2$, $\mathbf{U}$, $\mathbf{W}$ can be found with τ_1, τ_2 orthonormal columns and $\tau_1 = \tau_2 = r$ (CP) or $\tau_1 = r_1, \tau_2 = r_3$ (Tucker).

The `struct` function here is the sum of placement matrices Kronecker product-ed with the sub-blocks of the original matrix. This allows for the product of (the sum of) Kronecker products which have the structure mentioned in the introduction.

However, there is a common similarity between the BLR structure and the hiarchical structure, which is the low rank off diagonal blocks. This is going to

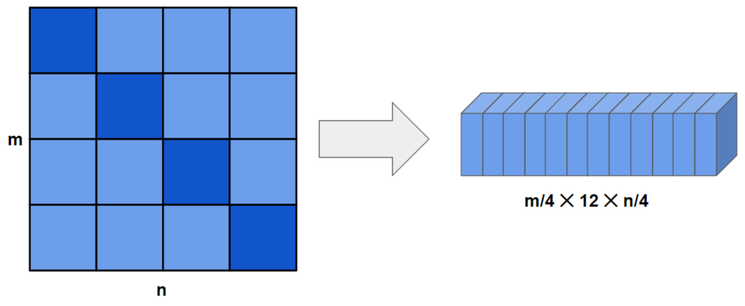

Figure 2.4: A way to approximate our matrix of interest would be to ignore the strong diagonal blocks, and map the other off-diagonal (hopefully low-rank) blocks into a tensor to be decomposed.

make the idea easily extendible to our problem where we keep the most important parts and find a way to approxmate the rest. With the block low rank framework, it is easy to fix a size of submatrices, divide the matrix into these subblocks, and twist these into a tensor. Because of the inclusion of all the blocks, or all of the blocks off the main diagonal (strict upper triangular if you are using symmetrical arguments), one knows that there will be room for compression, since the full block diagonal blocks are excluded.

Naively, we have

$$\mathbf{A} \approx \text{blockdiag}(\mathbf{A}) + \sum_i \mathbf{E}_i \otimes \widehat{\mathbf{A}}_i \tag{2.2}$$

where the $\widehat{\mathbf{A}}_i$ are the approximations of $\mathbf{A}_i$ resulting from the tensor decomposition, and $\mathbf{E}_i$ are the weighted placement matrices so that the blocks consisting of all of the similar subblocks are accounted for and, placing them where they should be so that $\|\mathbf{E}_i\|_F^2 = 1, \forall i$.

However, this process of finding these low rank features is opaque for the hierarchical matrix. To access these low-rank off diagonal blocks, a recursive splitting was performed. The first concept is having both a geometric and algebraic stopping criteria, where if the block gets below a certain threshold, and it is still not low rank, it most likely never will be, so the entire block is stored, which means there is very little compression that can happen. To combat this, the other criteria is an algebraic rank of the subblocks, meaning at each step the subblock rank is checked and if it falls below a user-defined threshold, then it is considered low rank. The entire matrix $\mathbf{A}$ is initally considered, and the questions of is it geometrically small or have small algebraic rank are asked. If the answer is “no", then the matrix is subdivided into four block matrices- $\mathbf{A}_{11}, \mathbf{A}_{12}, \mathbf{A}_{21}$, and $\mathbf{A}_{22}$. Then the process is repeated until one or both of the conditions are satisfied. This process teases out the blocks that become considered low rank, and is visualized in Figure 2.5.

2.3 Leverage the Structure

With the loss of Toeplitz structure, a preconditioner for this type of problem is not as well studied. However, based on the hierarchical structure of the matrix, we see that the most important parts of the matrix are contained in the main diagonal blocks. As you get farther away from the central diagonal, the rank in the off diagonal blocks gets smaller and smaller so we can get larger blocks that have a fixed rank. This is reminiscent of a block Jacobi preconditioner 1.3.5. However, as opposed to completely ignoring the off diagonal blocks, and since we have already computed an approximation, we are going to use the approximation as the preconditioner to get very well conditioned systems even as n gets large and $\alpha \nearrow 2$.

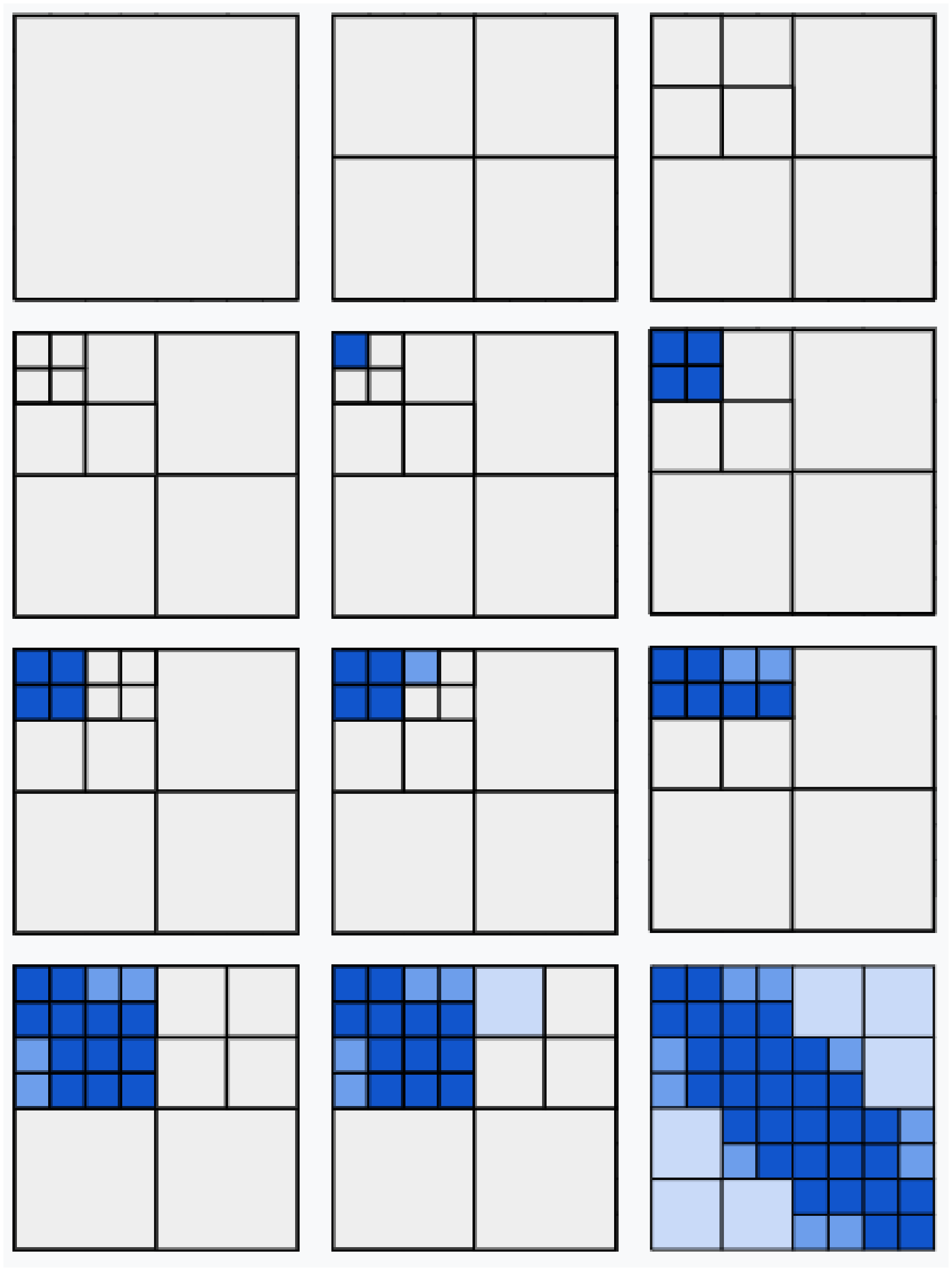

Figure 2.5: (1.) The whole matrix A, (2.) Subdivision of whole matrix A into four subblocks A_{11}, A_{12}, A_{21}, and A_{22}, (3.) Subdivision of $A.11$ into four subblocks $A.11_{11}, A.11_{12}, A.11_{21}$, and $A.11_{22}$, (4.) Subdivision of $A.11.11$ into four subblocks $A.11.11_{11}, A.11.11_{12}, A.11.11_{21}$, and $A.11.11_{22}$, (5.) The block $A.11.11.11$ failing the geometric stopping criterion, stored as full, (6.) All the blocks in $A.11.11$ failing the geometric stopping criteria, stored as full, (7.) Subdivision of $A.11.12$ into four subblocks $A.11.12_{11}, A.11.12_{12}, A.11.12_{21}$, and $A.11.12_{22}$, (8) The block $A.11.12.11$ satisfying the algebraic rank stopping criterion, twisted into a tensor for later compression, (9) All the blocks in $A.11.12$ stored according to their structure, (10) Due to symmetry and strong diagonal dominance, all the blocks in $A.11$ stored according to their structure, $A.12$ subdivided into four subblocks, (11) The block $A.12.11$ satisifying the algebraic rank stopping condition, twisted into a different tensor for later compression, (12) Repeating this process recursively unto all entries of A have been stored.

Chapter 3

Research and Experimental Design

3.1 Overview

This entire experiemental design process can be summarized by dividing the matrix up into two categories– those subblocks that are advantagous to compress and those that are not– turning those low-rank matrices into tensors at different levels depending on the size of the low rank subblocks, performing a tensor compression based on the HOSVD decomposition, and then mapping these approximated tensors back to a matrix.

3.2 Hierarchical Matrix Algorithm for 2^L-sized matrices

First, we need to solve the fPDE over the unit square on an adaptive mesh. Once the code is run for that, two matrices are output (for each iteration of the mesh), which are `A_full` and `A`, which are the full dense matrix storage and the hierarchical representation of the full matrix, respectively. This hiearchical matrix representaion is based on Taylor expanding the kernel of the fPDE. The adaptive code is set up to subdivide the matrix in half both by rows and columns and then check the stopping conditions. We will access the code's recursive nature to partion the dense matrix into tensors.

Because of the hierarchical structure, there is no guarentee that these submatrices are going to be the same size, so we have included a part to check how many levels one has gone to get a low rank subblock, which is related to the size of the subblock as everything is partitioned into 4 at each step. Keeping track of these levels, we are able to keep subblocks of the same size together into the same tensor. While most of the matrices we are working with only require the construction of tensors at two different sizes, that is not a requirement, and we hope to build a method that can accommodate as many levels of these subblocks tensors as the code requires.

Algorithm 4 Turning $\mathcal{H}$-matrices into Tensors

```
Require: A, a hierarchically structured approximation matrix, A_full, the full matrix
                                    ▷ These come from running the compare_adaptive code
  if flag == 0 then
      Repeat on A_11
      Repeat on A_12
      Repeat on A_21
      Repeat on A_22
  else if flag == 1 then                                ▷ These have low-rank structure
      See how many levels we have gone down, denoted by super script $\ell$
      Twist A_full$^\ell$[row_start : row_start + n_row - 1, col_start : col_start + n_col - 1]
  else if flag == -1 then                          ▷ These do not have low-rank structure
      Store the entire block
  end if
  return A_full$^\ell$, $\forall \ell$
```

This algorithm works really well if we have a matrix of size $2^L \times 2^L$, as when the matrix or submatrix that we are looking at doesn't have low rank, so to solve that we take the submatrix we are looking at and divide it in half rows and columns, so it is still of the form $2^{L-1} \times 2^{L-1}$, so we can keeping doing this procedure until the stop criteria are met. These criteria are both geometric (the size of the submatrix) as well as algebraic (the rank of the submatrices). In fact, this leads to our first toy problem of the thesis. Under the correct parameters in the original code ($\theta = 0.613, \texttt{rk} = 20, \alpha = 1.5$), we get a matrix that is in $\mathbb{R}^{256\times 256}$. To visualize the magnitudes of the entries, Figure 3.1, where the yellow diagonal has entries on the order of magnitude of 10^2, and the blue off diagonal entries have order of magnitude of 10^{-7}.

When we apply the algorithm 4, we are able to find the indices and flags for different off-diagonal blocks for the actual matrix `A_full` (not the $\mathcal{H}$-matrix approximation) as seen in Figure 3.3. To be more explicit, in this toy example one can count that there are 30 "full" blocks (denoted by the darkest blue, along the main block tri-diagonal) that are all $\mathbb{R}^{32\times 32}$, which is due to the geometric stopping condition, 10 "level 3" low-rank blocks (denoted by the medium color blue, between the main

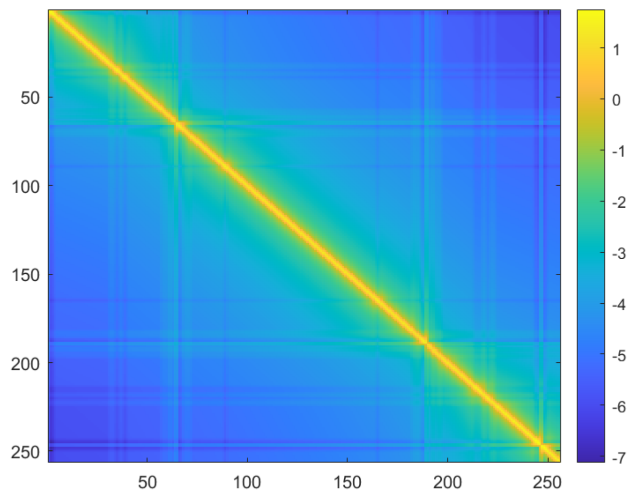

Figure 3.1: For $\alpha = 1.5$, the resulting matrix in $\mathbb{R}^{256\times256}$ visualized in terms of the relative magnitude of the entries.

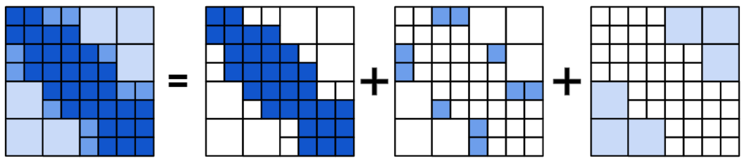

Figure 3.2: The original dense matrix can be split into disjoint parts– the block diagonal parts (darkest blue), the subblocks that turn low-rank after three levels (medium blue), and the subblocks that turn low-rank after two levels (lightest blue)

tridiagonal and outer blocks) that are all $\mathbb{R}^{32\times32}$ rank = 20, which is due to the algebraic stopping condition, and 6 "level 2" low-rank blocks (denoted by the lightest color blue, all the way off the diagonal) that are all $\mathbb{R}^{64\times64}$, rank = 20 (summarized in Table 3.1).

Once we have all the indices and flags for the different levels at which the subblocks become low rank, we can consider this a "matrix splitting" problem as demonstrated in Figure 3.2. We have what we will denote as "blockdiag($\mathbf{A}$)" which is the first term in the RHS (darkest blue) plus the terms consisting of the subblocks that turn low rank at the different levels. The white blocks indicate the subblocks that are all zero, as they have already been accounted for at another term.

However, after twisting each of the block matrices within the same category and concatenating them, we have three third order tensors. Starting with the "full"

Category	Color	Number of blocks	Size	Location
Full	Dark blue	30	$\mathbb{R}^{32\times 32}$	Main diagonal to tri-diagonal
Level 3	Medium blue	10	$\mathbb{R}^{32\times 32}$	Between main and off-diagonal
Level 2	Light blue	6	$\mathbb{R}^{64\times 64}$	Off diagonal corners

Table 3.1: Using our toy problem, three categories are found

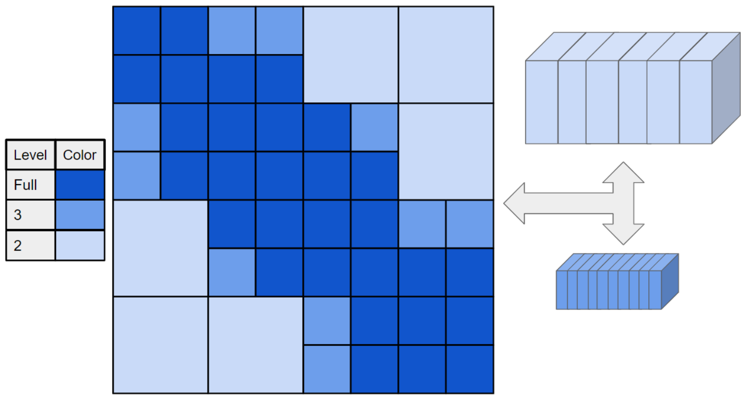

Figure 3.3: This is the levels diagram for the adaptive mesh of size $2^8 \times 2^8$. Each small block is 32×32 and the bigger blocks on the off diagonals are 64×64. Once these blocks are found, they are twisted into a tensor at each level.

matrix blocks, we could get a "full" tensor $\in \mathbb{R}^{32\times 30\times 32}$. Nevertheless, this is frivilous to do as we assert that there is no compression possible for the "full" tensor. We also get the medium blue "level 3" matrix blocks which get mapped into a tensor, `lev3` $\in \mathbb{R}^{32\times 10\times 32}$ and the light blue "level 2" matrix blocks which get mapped into a tensor, `lev2` $\in \mathbb{R}^{64\times 6\times 64}$.

3.3 Alternative Algorithms for Matrices not of 2^L.

The one problem with adaptive meshes is that they are not going to be of the size 2^L, which means that the first algorithm that we were investigating will not work on all (in fact most) of the discretized meshes. This means that we have to find a way to overcome this challenge. If we started with a 2^L matrix, then every subsequent halving would also have that same structure, and we didn't have to worry about having non-square block matrices. Now let's consider if $n = 129$, where we do not have to go past the first splitting where we would get block matrices that are $\mathbf{A}_{11} \in \mathbb{R}^{64\times 64}$, $\mathbf{A}_{12} \in \mathbb{R}^{64\times 65}$, $\mathbf{A}_{21} \in \mathbb{R}^{65\times 64}$, and $\mathbf{A}_{22} \in \mathbb{R}^{65\times 65}$. However, one thing is nice is we know how the partition is performed in general. The left-to-right partition of an $n \times n$ matrix will always partition the matrix so the left child contains indices $(:, 1 : \lfloor \frac{n}{2} \rfloor)$, which is all the rows and the first $\lfloor \frac{n}{2} \rfloor$ columns, and the right child will contain the indices $(:, \lfloor \frac{n}{2} \rfloor + 1 : \texttt{end})$. The top-to-bottom partition works the same way where the top child will inherit the first $\lfloor \frac{n}{2} \rfloor$ rows, and the bottom child will inherit the last $n - \left(\lfloor \frac{n}{2} + 1\right)$ rows. Comparing these two partitions we know that these numbers are going to differ at most by 1, since any number is either even or one away from being even. If we can use this, then we can easily find an algorithm that can work for any adaptive mesh that we desire.

3.3.1 Padding

Naturally, one way we can handle this problem is pad the smaller number by a row (column) of all zeros so that the number of rows and columns are the same size, which we will call “padded". This is presented below in Algorithm 5.

Algorithm 5 Turning $\mathcal{H}$-matrices into Padded Tensors

```
Require: A, a hierarchically structured approximation matrix, A_full, the full matrix
                              ▷ These come from running the compare_adaptive code
  if flag == 0 then
      Repeat on A_11
      Repeat on A_12
      Repeat on A_21
      Repeat on A_22
  else if flag == 1 then                          ▷ These have low-rank structure
      See how many levels, $\ell$ we have gone down
      maxi ← max(n_row,n_col)
      A_pad ← zeros(maxi)
      A_pad$^\ell$[1:n_row,1:n_col] ← A_full[rowstart : row_start + n_row - 1, col_start : col_start + n_col - 1]
      Twist A_pad$^\ell$ into a tensor
  else if flag == -1 then                    ▷ These do not have low-rank structure
      Store the entire block
  end if
  return A_pad$^\ell$, $\forall \ell$
```

To make this more concrete, if we run the code for a slightly different refinement ratio, $\theta = 0.614$ now, we end up with an adaptive mesh discretization leading to a stiffness matrix of $n = 257$. Since the parameter didn't change that much, we have the same number of blocks that turn low-rank at all levels, but the dimension of those blocks are different, causing the tensors to be different dimensions as well. Following the know partition pattern for this problem, in the padded case, we would get `full_ten` $\in \mathbb{R}^{33\times33\times30}$, `lev2` $\in \mathbb{R}^{65\times65\times6}$, and `lev3` $\in \mathbb{R}^{33\times33\times10}$.

3.3.2 Truncating

An equally valid way of dealing with the problem of matrices not of $n = 2^L$ is simply truncate the greater number between the rows and columns so that they equal the lesser number. This would then mean we would have to remove either the last row or column of the subblock so that it fits with all of the other subblocks in a tensor. After all, we are going be using HOSVD to get a truncation rank that is smaller than the original problem, so it should not matter that much. This algorithm is presented in Algorithm 6.

Algorithm 6 Turning $\mathcal{H}$-matrices into Truncated Tensors

```
Require: A, a hierarchically structured approximation matrix, A_full, the full matrix
                                   ▷ These come from running the compare_adaptive code
  if flag == 0 then
      Repeat on A_11
      Repeat on A_12
      Repeat on A_21
      Repeat on A_22
  else if flag == 1 then                          ▷ These have low-rank structure
      See how many levels$\ell$ we have gone down
      mini ← min(nrow,ncol)
      Twist A_full$^\ell$[rowstart : rowstart + mini - 1, colstart : colstart + mini - 1]
  else if flag == -1 then                    ▷ These do not have low-rank structure
      Store the entire block
  end if
  return A_full$^\ell$, $\forall\ell$.
```

Using the same example matrix as above for $n = 257$, the truncated algorithm we would get `full_ten` $\in \mathbb{R}^{32\times32\times30}$, `lev2` $\in \mathbb{R}^{64\times64\times6}$, and `lev3` $\in \mathbb{R}^{32\times32\times10}$. Of course this matches the dimensions of $n = 256$ because we are chopping off the one row and column that causes it to be nonsquare.

Once these matrices are turned into tensors by whichever way is necessary, then they can be truncated by the `tr-HOSVD` algorithm. These tensor approximations will have the same dimensions as the orginal tensors, so in the case of our toy problem, the arguments for the mapping algorithm are `lev2`, a level 2 tensor of dimension $\mathbb{R}^{64\times 64\times 6}$ and `lev3`, a level 3 tensor of dimension $\mathbb{R}^{32\times 32\times 10}$. The outputs are $\widehat{\texttt{lev2}}, \widehat{\texttt{lev3}}$, which have dimension $\mathbb{R}^{64\times 64\times 6}$ and $\mathbb{R}^{32\times 32\times 10}$, respectively*.

This fact was instrumental in mapping these approximation tensors to the original location in the matrix. We are able to record either the indices or the Kronecker structured matrix that places these slices back where they should. So running through the size of the frontal slices on all levels of tensors we are able to squeeze those slices back into matrix subblocks, placing them by the information that we saved from the original tensorization, which is presented in Algorithm 7. Since the tensor and tensor approximations have the same dimensions, we are able to uniquely place the correct number of blocks (frontal slices) with the same dimension (dimension of those frontal slices) back where we retrieved them from, resulting in a tensor-based low rank approximation of the original matrix.

Algorithm 7 Turning Tensors into $\mathcal{H}$-matrices

```
Require: $\hat{\mathcal{T}}^\ell$ the approximated tensors , $idx^\ell$, keeping track of Kronecker structure
  for those tensors $\forall \ell$
  for every $\ell$ do
      for $j \leftarrow 1 : size(\hat{\mathcal{T}}^\ell)$ do
          tblrappx$(idx_j(1), idx_j(2)) \leftarrow \hat{\mathcal{T}}^\ell(:,:,j)$
      end for
  end for
  return tblrappx, a tensor-based low rank approximation
```

But what does this tensor-based low rank approximation look like? As stated, because the blocks around the main tridiagonal are full, we don't try to compress those, so these subblocks will be the same as the original. We do compression only on those off-diagonal subblocks that are found to be low-rank. For our problem, we only have constructed (and approximated) two tensors, $\widehat{\mathcal{T}}^1, \widehat{\mathcal{T}}^2$, and then these will

*A keen observer might have noticed that before we twisted subblocks of the matrix into lateral slices, whereas here we are concatenating them as frontal slices. This is just an artifact of the code and nothing to worry about as these tensors we constructed are unique up to permutation.

get matricized into $\widehat{\mathbf{A^1}}, \widehat{\mathbf{A^2}}$, so we can write $\mathbf{A} \approx \widehat{\mathbf{A}}$.

$$\begin{aligned}
\mathbf{A} &\approx \text{blockdiag}(\mathbf{A}) + \widehat{\mathbf{A^1}} + \widehat{\mathbf{A^2}} && (3.1)\\
&\approx \text{blockdiag}(\mathbf{A}) + \sum_{\substack{i\in \\ size(A^1,3)}} \mathbf{E}_i \otimes \widehat{\mathbf{A^1_i}} + \sum_{\substack{j\in \\ size(A^2,3)}} \mathbf{E}_j \otimes \widehat{\mathbf{A^2_j}} && (3.2)
\end{aligned}$$

where the $\mathbf{E}_i$ are the respective Kronecker "placement" matrices and since $\hat{\mathbf{A}}^{\mathbf{1}} \in \mathbb{R}^{32\times32}$, the $\mathbf{E}_i \in \mathbb{R}^{8\times8}$, so that we are summing matrices that are all 256×256. Similarly, the $\mathbf{E}_j \in \mathbb{R}^{4\times4}$ are the Kronecker matrices for the $\hat{\mathbf{A}}^{\mathbf{2}} \in \mathbb{R}^{64\times64}$. Lastly, the phrase "blockdiag" here is a little ambigous. Certainly the main diagonal blocks are included in this function, but depending on the parameters of the fPDE, not much else can be known. It could be as thin as block tridiagonal, or as thick as block pentadiagonal. But the amount of blocks are not constant throughout the problem. In an abuse of notation "blockdiag" is used to mean the blocks on and surrounding the main block diagonal that is found to be full rank.

These placement matrices might be a little ambiguous, so to elucidate this concept, we will look at where a specific placement matrix will place the block of interest. First, let's consider $\mathbf{E}^1 \in \mathbb{R}^{8\times8}$, with only one nonzero element. First it makes sense that it is 8×8 because when you Kronecker product it with a block matrix $\mathbf{A}_{6,3} \in \mathbb{R}^3 2 \times 32$, we end up adding a $(8)32 \times (8)32$ matrix to the original 256×256 matrix, so matrix addition is well defined. If we wanted to place the $\mathbf{A}_{(6,3)}$ matrix

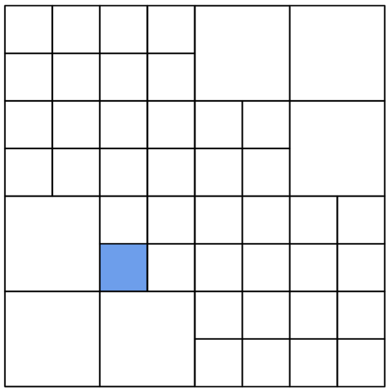

Figure 3.4: The $\mathbf{A}_{(6,3)}$ submatrix is placed in the correct location by $\mathbf{E}^1_{(6,3)}$, and all white blocks are zero.

block, the placement matrix is:

$$\mathbf{E}^1_{(6,3)} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

This block placement is represented pictorially in Figure **??**, where the medium blue block is placed in the $(6, 3)$ position, and all the white blocks are the zero block matrix.

Similarly, we can perform this operation on the largest light blue matrix block, $\mathbf{A}_{(1,3)} \in \mathbb{R}^{64\times64}$ by the placement matrix $\mathbf{E}^2_{(1,3)}$, represented pictorially in Figure **??**.

$$\mathbf{E}^2_{(1,3)} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

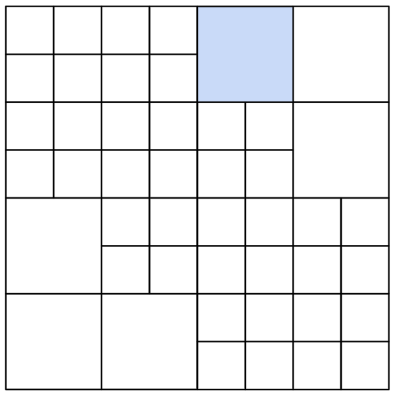

Figure 3.5: The $\mathbf{A}_{(1,3)}$ submatrix is placed in the correct location by $\mathbf{E}^{2}_{(1,3)}$, and all white blocks are zero.

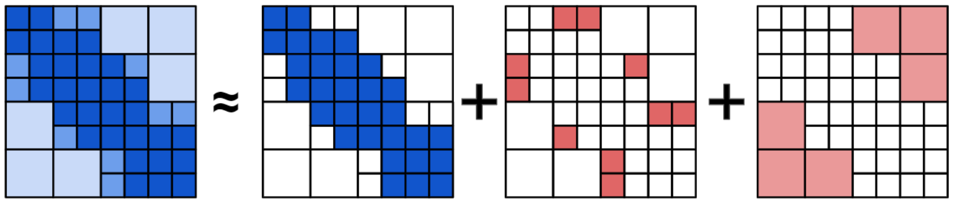

Figure 3.6: The original dense matrix is approximated by the sum of the exact full blocks, the approximation of the low rank blocks at level 3, and the approximation of the low rank blocks at level 2.

These sums of Kronecker products might not be clear so using Figure 3.6, one can see that the matrix can be rewritten as an approximation. The original matrix that we want to approximate is on the left hand side of the equation. The left hand side now consists of the sum of three matrices– the first is the full blocks and are the same dark blue color as the original since they are never compressed, the second is the approximation of the low rank blocks at level 3 (originally they were medium blue, but since they now are approximated they are medium red), and the last is the approximation of the low rank blocks at level 2, colored light red to indicate that these are an approximation to the original light blue blocks.

This means that even though we have a matrix with hierarchical structure, we are able to still construct the approximation as the block diagonal "full rank" blocks plus the sum of Kronecker products of the first collection of low rank blocks plus the sum of Kronecker products of the second collection of low rank blocks.

Chapter 4

Research Results

4.1 Storage

To quantify the results of this proposed method, it is best to compare the three options' storage and compression requirements as well as the relative error between the approximation to prove novelty. These three methods are: naively storing the entire dense matrix, approximating the hierarchical structure by Taylor expanding the differential kernel, and approximating the matrix using the methods developed in this thesis.

The metrics that will be used are called compression ratio, and data saving percent, where

$$\text{Compression Ratio} := \frac{\text{Uncompressed Size}}{\text{Compressed Size}} \tag{4.1}$$

$$\text{Data Saving percentage} := 1 - \frac{\text{Compressed Size}}{\text{Uncompressed Size}}, \tag{4.2}$$

where $\underline{\text{uncompressed size}}$ is the starting amount of elements to store, and $\underline{\text{compressed size}}$ is the amount of stored elements in the not-to-lossy approximation.

4.1.1 Naive

First, to naively store every element in an $n \times n$ dense matrix, like the one achieved by discretizing the fPDE would result in n^2 elements. In our relatively small case of $n = 256$, that is already storage of 65,536 elements. It is conceivable that the adaptive mesh for a larger problem has at least 1,000 gridpoints, which results in 1,000,000 elements. Since we are storing all of the elements, that is the compressed size is the uncompressed size, the compression ratio is 1:1, and the data saving percentage is 0%. This is not ideal.

4.1.2 Low Rank Matrices

Another way would be to differentiate which subblocks of the matrices are considered low rank, and which are not. This process has been discussed before by recursive subdivision and can be pictorally seen in Figure 2.5. This method designates the blocks around the main diagonal as full rank, so that means they need to be stored in full. Next to these are the blocks that are “barely" low rank, and there is a low rank factorization, and lastly the farthest block from the main diagonal are low rank blocks and benefit the most from storing them in a factorization. A detailed analysis of storage for the test problem ($\alpha = 1.5, \theta = 0.613, n = 256, \texttt{rk} = 20$) is done below in table 4.1.

Category	Number of blocks	Size of blocks	Total
Full	30	(32×32)	30720
Level 3	10	$2 \times (32 \times 20)$	12800
Level 2	6	$2 \times (64 \times 20)$	15360
			58880

Table 4.1: Storage accounting for toy problem ($\theta = 0.613, \alpha = 1.5$) using method in [25].

The 2 outside of the parentheses in the "size of blocks" column of table 4.1 comes from the fact that when there is a factorization of $\mathbf{C} \in \mathbb{R}^{n\times n}$ which has rank r such that $\mathbf{C} = \mathbf{A}\mathbf{B}^T$ and $\mathbf{A} \in \mathbb{R}^{n\times r}, \mathbf{B}^T \in \mathbb{R}^{r\times n}$, which is why we need to store two matrices of those size. This demonstrates that this method requires 58880 elements of storage and if every element was stored, 65,536 would be needed, which means this method gives a compression ratio of 1.112:1 and data saving percentage of 10.16%.

It is also advantagous to note that the rank condition of $r = 20$ is a parameter that can be changed, so if we set a different rank stopping criterion, then the compression ratio would change. It is interesting to note however, that the size and structure of the problem do not change, just the rank of the factorization $\mathbf{C} = \mathbf{A}\mathbf{B}^T$. For example, if we set the rank condition to be r, we still have 30 full sized blocks, ten level 3 blocks, and six level 2 blocks. And because the problem is still $n = 256$, that doesn’t change either, just the size of the factorization. Generalizing the results for this method in a generic rank, we get table 4.2. Doing the math, we are able to get a data saving percentage of approximately $0.53125 - 0.02148r\%$, where the 0.53125% comes strictly from the storage of all full blocks along the main diagonal. More detailed analyses for lowering storage for the main blocks are performed in Section 4.1.4.

Category	Number of blocks	Size of blocks	Total
Full	30	(32×32)	30720
Level 3	10	$2 \times (32 \times r)$	$640r$
Level 2	6	$2 \times (64 \times r)$	$768r$
			$30720 + 1408r$

Table 4.2: Generalization of storage accounting for toy problem, using hierarchical low rank blocks

4.1.3 Tensor-based Methods

Another way that the we can find an approximation of the dense matrix is using the tensor based approach that we have discussed in detail in chapter 3. For the $n = 256$ toy problem, we will analyze the storage requirements for the standard $\mathcal{H}$-matrix to tensor algorithm (4). The main analysis transfers nicely over to the other adaptations of algorithm (namely algorithms 5,6).

First, just like the method mentioned above, we do not want to try to compress the blocks that are deemed full rank and incompressible, so we will leave them at bay. Through the 2^L sized matrix method presented (Algorithm 4), we already have constructed tensors of all the blocks that turn "low-rank" in the same size step and that approximation can be written as a sum of Kronecker products (Eqn. 3.2). Since we are dealing with an inital problem that is exactly $n = 2^L$, we leave discussion of the edge cases for later. Since we are trying to approximate the same hierarchical matrix of our toy problem in a different way, it is no suprise that there are 30 blocks along the main tridiagonal that are considered full rank, 10 blocks that are considered low rank in level 3, and six blocks that are considered low rank in level 2 (darkest blue, medium blue, and light blue in Figure 3.3, respectively). For the `tr-HOSVD` algorithm, we have to store not only the truncated core tensor but also the factor matrices to be able to recreated the approximation. To mirror table

4.1, an accounting of the storage used for this new method is presented in table 4.3.

Category	Subcategory	Size of blocks	Total
Full	30	(32×32)	30720
Level 2	`lev2` core	$47 \times 47 \times 4$	8836
	`lev2` Factor matrix	64×47	3008
	`lev2` Factor matrix	64×47	3008
	`lev2` Factor matrix	6×4	24
Level 3	`lev3` core	$32 \times 32 \times 10$	10240
	`lev3` Factor matrix	32×32	1024
	`lev3` Factor matrix	32×32	1024
	`lev3` Factor matrix	10×10	100
			57984

Table 4.3: Storage Accounting for toy problem ($\theta = 0.613, \alpha = 1.5$) using `hmat2ten` with truncation ranks `lev2` $\in \mathbb{R}^{47\times47\times4}$, `lev3` $\in \mathbb{R}^{32\times32\times10}$

The storage accounting for a general problem with only two different sized tensors with truncation ranks for `lev2` of r_1, r_2, r_3 and truncation ranks for `lev3` of k_1, k_2, k_3 is performed in table 4.4.

Category	Subcategory	Size of blocks	Total
Full	30	(32×32)	30720
Level 2	`lev2` core	$r_1 \times r_2 \times r_3$	$r_1 r_2 r_3$
	`lev2` Factor matrix	$64 \times r_1$	$64r_1$
	`lev2` Factor matrix	$6 \times r_2$	$6r_2$
	`lev2` Factor matrix	$64 \times r_3$	$64r_3$
Level 3	`lev3` core	$k_1 \times k_2 \times k_3$	$k_1 k_2 k_3$
	`lev2` Factor matrix	$32 \times k_1$	$32k_1$
	`lev2` Factor matrix	$10 \times k_2$	$10k_2$
	`lev2` Factor matrix	$32 \times r_3$	$32k_3$

Table 4.4: Generalizeation of Storage accounting for toy problem, using tensor based methods on low rank blocks

There are two more proposed tensor methods whose storage should be considered – trucation and padding. For a concrete example, consider a total matrix of size $n = 257$. This example is so close to $n = 256$, so that most of the previous analysis is still valid, i.e. two levels of tensors are constructed. The following is how the two methods would store the information: For the trunction method, after the first subdivision we have $\mathbf{A}_{11} \in \mathbb{R}^{128\times128}, \mathbf{A}_{12} \in \mathbb{R}^{129\times128}, \mathbf{A}_{21} \in \mathbb{R}^{128\times129}, \mathbf{A}_{22} \in \mathbb{R}^{129\times129}$; however it is easy to see that these methods are going to lead to different sized blocks being put in the same tensor, which is impossible.

4.1.4 Additional Storage Considerations

While storage of the individual elements is important, there are additional storage features especially for placement of these blocks that need to be considered. Storing the elements needed for subblocks is important, but if we place these elements in the wrong location, catastrophic consequences can (and will) happen. For the hierarchical matrix representation, the information is stored in a recursive cell array. But if it wasn't, the information can be summarized with two bits of information. At each level, after the subdivision into four smaller blocks, the information about which side of the horizontal division and which side of the vertical division the new subblock is in. This means all the entries in $\mathbf{A}_{12}$ need to be stored as well as $\{0, 1\}$ which says that these elements are in the top half and the right half of the matrix. This means at each level of recursion, we have to store 2 bits. For example `A.11.12.21` has 6 bits of information $\{0, 0, 0, 1, 1, 0\}$ to dictate that it located in the lower left quadrant of the upper right quadrant of the upper left quadrant. The tensor based method also has to allocate bits to placement of the blocks in the form of placement matrices that are Kronecker producted to the necessary block entries. Using the same example above, the entries in $\mathbf{A}_{12}$ could be placed by the following schema

$$\begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} \otimes \mathbf{A}_{12} \tag{4.3}$$

$$\left(\begin{pmatrix} 0 & 1 \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 0 \end{pmatrix} \right) \otimes \mathbf{A}_{12} \tag{4.4}$$

We see that this requires 4 pieces of information to place the entries where they need to go. However, this has introduced a new form of structure into our problem, that will be utilized later. For the example of placing `A.11.12.21` could look something like

$$\left(\begin{pmatrix}0 & 0 & 1 & 0 & 0 & 0 & 0 & 0\end{pmatrix} \otimes \begin{pmatrix}0\\1\\0\\0\\0\\0\\0\\0\end{pmatrix}\right) \otimes \mathbf{A}_{11.12.21} \tag{4.5}$$

$$\left(\begin{pmatrix}1 & 0\end{pmatrix} \otimes \begin{pmatrix}0 & 0 & 1 & 0\end{pmatrix}\right) \otimes \left(\begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\\0\\0\end{pmatrix}\right) \otimes \mathbf{A}_{11.12.21} \tag{4.6}$$

$$\left(\begin{pmatrix}1 & 0\end{pmatrix} \otimes \left(\begin{pmatrix}0 & 1\end{pmatrix} \otimes \begin{pmatrix}1 & 0\end{pmatrix}\right)\right) \otimes \left(\begin{pmatrix}1\\0\end{pmatrix} \otimes \left(\begin{pmatrix}1\\0\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix}\right)\right) \otimes \mathbf{A}_{11.12.21} \tag{4.7}$$

This example shows that using Kronecker product structure nested inside of Kronecker structure has ways of reducing the number of entries needed to properly place the subblocks. As opposed to the hierarchical way, this needs 12 pieces of information to place it, but the structure can be exploited.

There are also symmetry advantages that were not taken advantage of, but would greatly save storage. First, the adaptive mesh discretization still has symmetric positive definite structure. In fact this can be visualized in Figures 3.1 and 3.3. This means that for the hierarchical formulation, we have stored both `A_12.A_12` and `A_21.A_21`, even though they both represent the same 64×64 subblock and are just transposes of each other. Other extensions of this fact are that $\operatorname{rank} \mathbf{C} = \operatorname{rank} \mathbf{C}^T$ and if $\mathbf{C} = \mathbf{A}\mathbf{B}^T$ then surely $\mathbf{C}^T = \mathbf{B}\mathbf{A}^T$, so any properties we would like can be found by only considering one of these blocks. For the toy problem considering symmetry, instead of just storing 30 full rank blocks, 10 level 3 blocks, and 6 level 2 blocks, only

19, 5, and 3 blocks would have to be stored to get the same result. This advantage also applies to the tensor based methods that we proposed, as we would only have to put half of the total blocks into our tensors to compress them.

Considering the symmetry of the low-rank off diagonal blocks does help lower storage costs, but the largest proportion of storage is still coming from storing the entire blocks along the block diagonal of the matrix. This storage of the full blocks along the block diagonal occurs in both [25] and this new proposed method. To combat this, we utilize the SPD structure of our problem. Since the entire matrix is SPD, we can easily apply the definition to see that any pricipal submatrix is also SPD. This means that the full blocks and principal submatrices along the main diagonal have a Cholesky factorization, $\mathbf{A}_{11} = \mathbf{L}_{11}^T\mathbf{L}_{11}$, which cuts the storage of these blocks in half as we only have to store the $\mathbf{L}$ matrix, which is an lower triangular matrix. Since both methods treat the main diagonal blocks the same way, only one analysis is needed. We see that we can view the full rank diagonals matrices into larger submatrices that are now still symmetric and positive definite. There are a few that cannot be encorporated into a larger block matrix, so we will just store them as a whole. Then we have block matrices along the diagonal that are all SPD, so that means there is a Cholesky factorization for those. If the size of the submatrix is n, that means the lower triangular Cholesky only requires $\frac{n(n+1)}{2}$ elements of storage.

To be explicit, let's use our toy matrix as an example of how these considerations can save storage in Figure 4.1. We have the four 32×32 matrices now thought of as a SPD 64×64, which means that we only have to store $\mathbf{L}_1$ using 2080 elements of storage, Similarly, we have 9 32×32 that can be thought of as a 96×96 SPD principal submatrix, and only have to store $\mathbf{L}_2, \mathbf{L}_3$ using 4656 elemetns of storage each. Lastly, there was no way to incorporate some of the blocks, so we can store those 4 blocks in their entirety so that requires $32 \times 32 = 1024$ elements of storage each. Lastly, the white squares in the matrix are there to denote that because of symmetry arguments, they do not need to be stored.

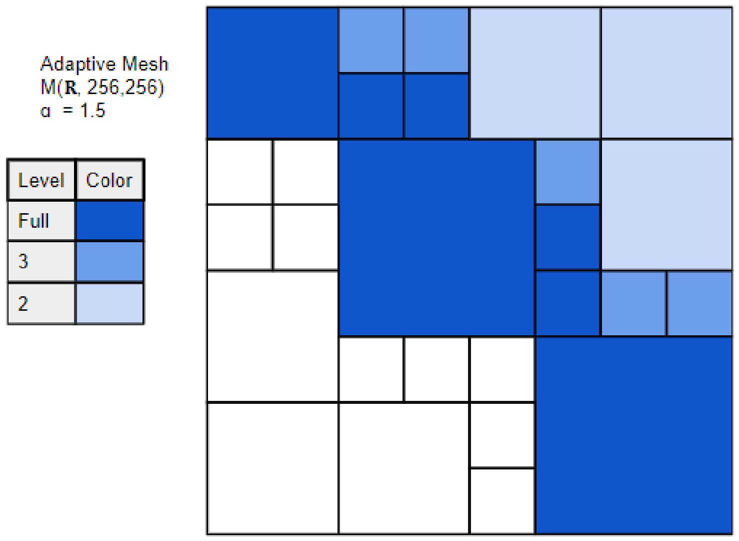

Figure 4.1: Using that our toy problem matrix is SPD, we can reformulate the matrix to include nonoverlapping blocks along the main diagonal, which are SPD, so to save storage, we can perform a Cholesky factorization and store one of the Cholesky factors. This cuts our storage in half.

4.2 Approximation

While storage considerations are an important way to compare methods, another comparison consideration is to see how the various methods' approximation to the original problem compare. Using the same information that is presented in table 4.1, 4.3, we compare how the hierarchical low rank approximation compares to the full matrix arising from the discretization, as well as how the tensor based low rank approximation compares as well from the relative Frobenius norm in table 4.5.

Method	**Literature [25]**	**Proposed**
Data Saving (%)	10.16%	11.52%
Rel Error	8.9805×10^{-12}	5.2210×10^{-6}

Table 4.5: Comparing relative errors between the proposed tensor based methods and literature using a similar compression benchmark.

In summary for the toy problem, we do not see the same level of approximation using consistant compression benchmarking. However, we are being unfair to our method in this comparison as all simulations were run using the rank of the subblocks to be `rk` = 20. This naturally will cause the relative error to be very low since this means we are storing all 32×32 blocks that become low rank after three levels of recursion to be $\mathbf{C} = \mathbf{A}\mathbf{B}^T$, where $\mathbf{A}, \mathbf{B} \in \mathbb{R}^{32\times 20}$. This means for the blocks that we are approximating, isn't really an approximating. Additional testing of how the ratio of rank and size of the subblocks will be performed to find a way to level the playing field between the two methods in our comparison. Our method allows for varying levels of compression and is much more flexible in terms of storage than the rigidity that the method in [25] provided as there is no way to change storage requirements, other than changing the `rk`, but this can lead to problems with convergence between the hierarchical matrix approximation and the true stiffness matrix as well as unstable meshes as n increases for larger problems; both are problems with the code to solve the fPDE, and further discussion is outside of the scope of this thesis. Also, this $n = 256$ matrix, which has been extensively studied in this thesis may just be too small to notice the large amount of potential savings that are possible. It would be interesting to see how these methods work for larger sized problems.

As was stated, these two alternative formuations should be the same as the first formulation for a matrix 2^L.

Lastly, it should be noted that this method that was developed does not cost us anything in the approximation of the original matrix in the Frobenius norm.

Lemma 4.2.1 (S., 2023) *Let* $\mathbf{A}$ *be a matrix that can be expressed as*

$$\begin{aligned} \mathbf{A} &= blockdiag(\mathbf{A}) + \sum_k \mathbf{E}_k^1 \otimes \mathbf{A}^1 + \sum_k \mathbf{E}_k^2 \otimes \mathbf{A}^2 && (4.8) \\ &= blockdiag(\mathbf{A}) + \sum_\ell \left(\sum_k \mathbf{E}_k^\ell \otimes \mathbf{A}^\ell \right) && (4.9) \end{aligned}$$

and let $\widehat{\mathbf{A}}$ *be an approximation to* $\mathbf{A}$ *, where*

$$\widehat{\mathbf{A}} = blockdiag(\mathbf{A}) + \sum_\ell \left(\sum_k \mathbf{E}_k^\ell \otimes \widehat{\mathbf{A}}^\ell \right) \qquad (4.10)$$

such that the $\widehat{\mathbf{A}}^\ell$ *are the sum of Kronecker products that come from the matrix-to-tensor mapping of the* $\mathbf{A}^\ell$ *as the blocks, compressing, and then a tensor-to-matrix mapping.*

Then

$$\left\|\mathbf{A} - \widehat{\mathbf{A}}\right\|_F^2 = \sum_{\ell} \|\mathcal{X}^{\ell} - \mathcal{T}^{\ell}\|_F^2. \tag{4.11}$$

Remark 4.2.2 To give some intuition for the coming proof, consider the pictorial reprentations of the exact matrix splitting and the approximation of the matrix splitting (Figures 3.2,3.6, repsectively). If we notice that the blockdiag($\mathbf{A}$) are unchanged, then first two terms will cancel exactly. Then for the next two terms, since the subblocks are all disjoint restrictions of the index set, we can consider then that the difference between $\left\|\mathbf{A}^{\ell} - \widehat{\mathbf{A}^{\ell}}\right\|_F, \forall \ell$ are just the tensor-to-matrix mapping, and since this is bijective, we have $\left\|\mathcal{T}_{\mathcal{E}}[\mathbf{A}^{\ell}] - \mathcal{T}_{\mathcal{E}}[\widehat{\mathbf{A}}^{\ell}]\right\|_F = \left\|\mathcal{T}_{\mathcal{E}}[\mathbf{A}^{\ell}] - \widehat{\mathcal{T}_{\mathcal{E}}}[\mathbf{A}^{\ell}]\right\|_F$, which is just the Frobenius norm difference between the tensorized original and the tensor approximation. Therefore, it follows that the square of the absolute error of the matrix approximation is the sum of the squares of the absolute error of the tensor approximation. ◊

Proof: Using the powerful framework that was presented in [18], we can generalize their proof for 1.2.8. It is important to note that we can ignore the block diagonal term since if we were to map those blocks to a tensor, there is no truncation, and the bijective mapping would place those exact blocks in the exact same place. In short, $\mathcal{M}_{\mathcal{E}}[\mathcal{T}_{\mathcal{E}}[\text{blockdiag}(\mathbf{A})]] = \text{blockdiag}(\mathbf{A})$. For the low rank tensors, let's have p_{ℓ} lateral slices in $\mathcal{X}^{\ell}$, for all ℓ levels of tensors, resulting from the matrix-to-tensor mapping. Also, it is important to note that $\mathbf{E}_k$ are all of the placement matrices, with only one non-zero element, as we are assuming no block structure. Then we have

$$\left\|\mathbf{A} - \widehat{\mathbf{A}}\right\|_F^2 = \|\mathbf{A} - \mathcal{M}_{\mathcal{E}}[\mathcal{T}]\|_F^2 \tag{4.12}$$

$$= \left\|\sum_{\ell}\left(\sum_{k=1}^{p_{\ell}} \mathbf{E}_k^{\ell} \otimes \mathtt{sq}(\mathcal{X}_{:k:}^{\ell}) - \sum_{k=1}^{p_{\ell}} \mathbf{E}_k^{\ell} \otimes \mathtt{sq}(\mathcal{T}_{:k:}^{\ell})\right)\right\|_F^2 \tag{4.13}$$

$$= \sum_{\ell}\sum_{k=1}^{p_{\ell}} \left\|\mathbf{E}_k \otimes \left(\mathtt{sq}(\mathcal{X}_{:k:}^{\ell}) - \mathtt{sq}(\mathcal{T}_{:k:}^{\ell})\right)\right\|_F^2 \tag{4.14}$$

$$= \sum_{\ell} \sum_{k=1}^{p_\ell} \|\mathbf{E}_k\|_F^2 \left\|\mathcal{X}^{\ell}_{:k:} - \mathcal{T}^{\ell}_{:k:}\right\|_F^2 \tag{4.15}$$

$$= \sum_{\ell} \sum_{k=1}^{p_\ell} \left\|\mathcal{X}^{\ell}_{:k:} - \mathcal{T}^{\ell}_{:k:}\right\|_F^2 \tag{4.16}$$

$\square$

4.3 Speed of Mat-vec

While a detailed analysis of speed is beyond the scope of this thesis, a few advantages of this Kronecker-based methodology are presented. Recall Eq 1.3, which says that $(\mathbf{A} \otimes \mathbf{B})\vec{x} \equiv \texttt{vec}(\mathbf{B}\mathbf{X}\mathbf{A}^T)$. Assume $\mathbf{A} \in \mathbb{R}^{n \times n}, \mathbf{B} \in \mathbb{R}^{m \times m}$, then to do the matvec on the RHS, would take $\mathcal{O}\big(m^2 n + n^2 m\big)$; however, if you naively performed the LHS on $\mathbf{A} \otimes \mathbf{B} \in \mathbb{R}^{mn \times mn}$, and is we assume that they are dense, this matvec would be $\mathcal{O}\big((mn)^2\big)$, which is much slower, especially for larger matrices.

4.4 Preconditioning

As was mentioned earlier, another goal of this tensor based approximation is to not only require less storage but to be a good candidate for preconditioning. The main idea here is as the fPDE has more nodes in the mesh, or as $\alpha \nearrow 2$, the full matrix gets more ill-conditioned, so we are trying to develop a method that combats that so the system can converge to the solution faster. The effect of the size of the problem, n, and the fractional index, α on the condition number of the resulting full matrix is visualized in Figure 4.2.

While we change n for each problem and run the fractional index $\alpha = 1.1, 1.2, \cdots, 1.9$, one parameter that we have not altered is θ, which is refinement ratio. We did not expect changing the parameter to change the trends of the condition number of the full matrix, but to affirm that the same numerical experiment was run as above with $\theta = 0.8$ instead, and is visualized in Figure 4.3.

As expected, the trends are unchanged between the two Figures 4.2 and 4.3 although the matrices with $\theta = 0.8$ seem to be more ill-conditioned, and it took longer computational time.

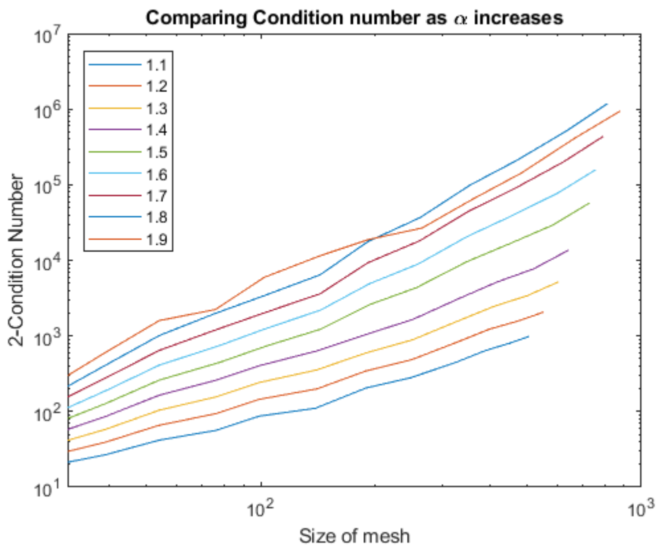

Figure 4.2: As the fractional index, α, and the size of the problem, n, increase, the condition number of the resulting full matrix increases exponentially ($\theta = 0.613$).

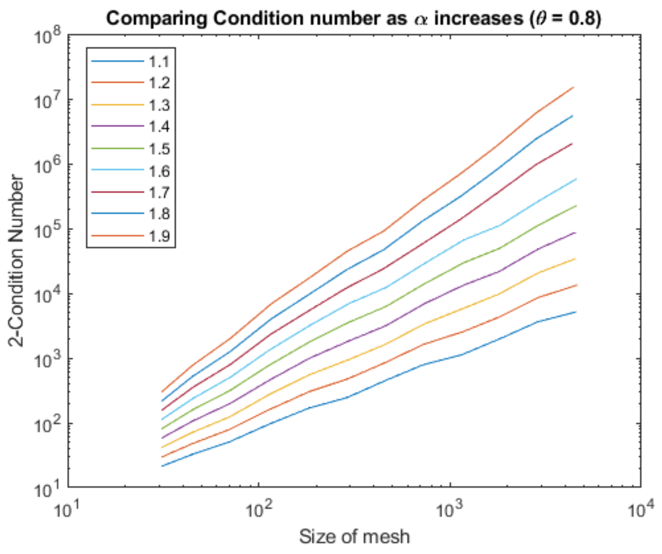

Figure 4.3: By altering the refinement index, θ from 0.613 to 0.8, the trends of the condition number of the full stiffness matrix increasing with n, α are still seen.

For example, with the tensor based low rank approximation that was discussed earlier in this chapter, we have that if $\widehat{\mathbf{A}}$ is left by itself, it has $\rho(\widehat{\mathbf{A}}) = 85.4990, \kappa(\widehat{\mathbf{A}}) = 4347.3$, which both can lead to methods that either do not converge or do not converge very fast in term of the number of iterations needed to get an approximate solution within a certain threshold. However, if we apply our tensor based low rank approximation, $\widehat{\mathbf{T}}$ as a left preconditioner we have $\rho(\widehat{\mathbf{T}}^{-1}\widehat{\mathbf{A}}) = 1.0004, \kappa(\widehat{\mathbf{T}}^{-1}\widehat{\mathbf{A}}) = 1.0442$, which is orders of magnitude better in both the spectral radius and condition number. This should not really be that suprising though. At the most basic, since we don't do anything with the strong diagonally dominant aspects and we make minor alterations with the low rank off-diagonal blocks, we could think of this as a modified Block Jacobi preconditioner [def.1.3.5]. Upon using a few steps of this block Jacobi schema as a preconditioner, naturally it will have those effects on an ill conditioned matrix. However, the novelty in this idea does not come from the fact that using the diagonally dominant part of a matrix to approximate the original, but rather how it can be implimented using the Kronecker structure of the blocks and the fact that these low rank off diagonal blocks could be thought of as modifications to the main diagonal blocks.

Since this method is based around a diagonally dominant block matrix, what we have been calling "blockdiag($\mathbf{A}$)" plus some low rank block, it seems to be of the form

$$\mathbf{A} = \text{blockdiag}(\mathbf{A}) + \sum_{\ell} \mathbf{A}_{ij}^{\ell} \vec{e}_i \vec{e}_j \tag{4.17}$$

where here the $\vec{e}_i$ are the unit vectors, placing the low-rank subblocks. It might be a small abuse of notation, but these $\mathbf{E}_i$ are also a sort of placement matrices that are expressed throughough this thesis. Since the inverse of the blockdiag($\mathbf{A}$) is well known and we are adding a low rank update to it, this implimentation is screaming a block analogue of the Sherman-Morrison-Woodbury which was presented in theorem 1.1.19. By the relationship that the outer product of two vectors is also the Kronecker product of those two vectors, we could consider that we write this system as

$$\mathbf{A} = \text{blockdiag}(\mathbf{A}) + \sum_{\ell} \sum_{i,j} \mathbf{E}_i^{\ell} \otimes \mathbf{A}_{ij}^{\ell} \tag{4.18}$$

Now forming the inverse would be

$$\mathbf{A}^{-1} = \left(\text{blockdiag}(\mathbf{A}) + \sum_{\ell} \sum_{i} \mathbf{E}_i^{\ell} \otimes \mathbf{A}_i^{\ell} \right)^{-1} \tag{4.19}$$

if we wanted to think of this as a Sherman-Morrison-Woodbury formula and solve this directly. We can also use the iterative process described above to find the inverse implicitly, which will also solve the equation.

For a preconditioner, since $\mathbf{A}$ is SPD, we would want $\widehat{\mathbf{T}}$ to also be SPD. Currently, the way that the tensor based approximation is implimented doesn't guarentee symmetry; however, this can easily be enforced, especially if we only put the non-redudant blocks into the tensor and force symmetry after the decomposition. The more important part is that $\widehat{\mathbf{T}}$ has all positive eigenvalues which means that it is strictly positive definite. While positive definiteness is seen in all numerical experiments, a proof of this fact still needs to be considered but is outside the scope of this thesis.

Chapter 5

Summary, Implications, and Conclusions

5.1 Summary

In this thesis, we apply new numerical linear algebra and multilinear algebra techniques to approximate a dense matrix coming from a discretized fPDE equation, namely construct tensors at many levels, where the blocks are coming from hierarchical matrix structure.

To be more explicit, fPDEs are used to to model physical phenomena but can suffer from singularities with even smooth initial data. This results in using adaptive meshes, which breaks known Toeplitz structures, motivating the need for a new robust method for storing and manipulating the matrices coming from these settings. While Toeplitz structure is broken, hierarchical structure is found.

While there are matrix approximation methods for specific classes of matrices using higher order structure, there was a need to expand these multilinear algebraic methods to another important class of matrices - hierarchical. The first thing that was considered is how to construct tensors at these different levels, when the sub-blocks are found to be low-rank. Then methods were generalized to be more inclusive of all sizes and strcture of these $\mathcal{H}$-matrices. After all, adaptive meshes from the original discretization can be any size and shape they want. These inclusive tensor methods are tested against the original and allow for a more robust method. Storage is one consideration. Because of the structure of the problem (i.e. diagonal dominance, symmetry, positive definiteness), as well as the fact that this system gets rather ill-conditioned as the mesh gets larger, this same framework is applied to preconditioning these unruly matrices.

Lastly, storage, approximation, and computational flop counts are all considered to demonstrate the possible novelty of this method compared to just locally approximating these hierarchical subblocks. Also, numerical experiments show that this method can be used to precondition those matrices. While significant work has been conducted, there are more considerations to better impliment these methods.

5.2 Future Work

Although the method here provided a large amount of flexibility when it comes to storage saving needs, most of the tensor rank were determined by truncating the core of the two tensors just so that it was able to meet the tolerance threshold that we defined. There is no guarentee that this method actually has found the minimal storage between the two tensors and their cores for the maximal approximation. While theoretically possible to just iterate through all the possible truncation ranks in the two tensors, that becomes a very large problem as that lives in 6 dimension; there are two 3-order tensors constructed. Because of that it would be interesting to explore how integer programming techniques, especially branch and bound algorithms could be used to find the optimal truncation rank so that the user defined threshold is still met. However, if we already have nice computational speed-up, the flip side of this would be is the extra computational work of finding the optimal truncation rank actual worth it. In fact, most of the truncation ranks in the thesis were just estimated and verified to be under the user defined tolerance threshold.

Expanding on this idea, there is no reason to assume that for a generic problem the resulting hierarchical matrix will have only two levels ℓ of tensors comeing from two different sizes of low rank subblocks. In fact, for $\theta = 0.6130, \alpha = 1.5,$`iter` $=$ 12, we get an adaptive mesh hierarchical matrix that is $n = 730$. Upon pruning the matrix to discover the hierarchical structure, the structure can be visualized in Figure 5.1.

This example shows that there are four different tensors to consider and if the user defined tolerance must be met, it has to be balanced artfully over these levels of tensors. For the sake of generality, we have this information in an d dimensional array,

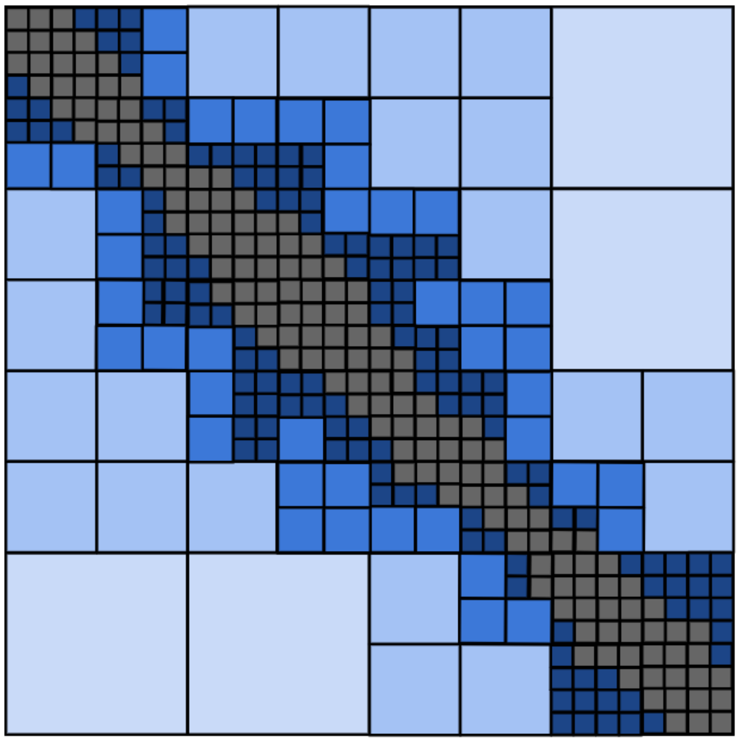

Figure 5.1: An adaptive mesh that hase four levels of low-rank hierarchical structure

and there are ℓ tensors constructed from the different sized blocks of the hierarchical matrix, Determining the optimal truncation rank is a minimization problem in $\mathbb{Z}^{d\ell}$, which is no easy feat.

Along the same vein, it would be interesting to explore if there are any trends about how the dimensions of the optimal tensors relate, meaning could it be observed or verified that containing more lateral slices (more block matrices) with less information about each slice more beneficial than containing more information about the block matrices but less information about the number of slices?

It was out of a lot of convenience that we decided to use the HOSVD algorithm on third order tensors. However, is this the best choice? Are there possible dimensions that are higher than three where we can exploit even more structure at higher levels?

Lastly, how does this method compare to different kernels that arise in scientific applications? Or even $\mathcal{H}$-matrices arising from different scientific applications. The methodology conducted here uses a lot of properties that arise in this discretized fPDE system, but how many of these traits can be generalized?

Appendix A

A.1 List of Notation

c	scalar
$\vec{v}$	vector
$\mathbf{A}$	matrix
$\widehat{\mathbf{A}}$	matrix approximation
$\mathbf{A}_{:,i}$	i^{th} column of $\mathbf{A}$
$\mathbf{A}_{j,:}$	j^{th} row of $\mathbf{A}$
$\mathcal{A}$	tensor
$\widehat{\mathcal{A}}$	tensor approximation
$\mathcal{A}_{i::}$	i^{th} horizontal slice of $\mathcal{A}$
$\mathcal{A}_{:j:}$	j^{th} lateral slice of $\mathcal{A}$
$\mathcal{A}_{::k}$	k^{th} frontal slice of $\mathcal{A}$
$\mathcal{M}_{\mathcal{E}}[\cdot]$	tensor-to-matrix mapping
$\mathcal{T}_{\mathcal{E}}[\cdot]$	matrix-to-tensor mapping
$\mathbf{E}$	Placement matrix
$\mathbf{U}, \mathbf{V}, \mathbf{W}$	Orthogonal matrix

A.2 Matlab Code

A.2.1 hmat2ten

```
function [lev2, lev3, full_ten, lrappx, idx2, idx3] = ...
        hmat2tens(hmat, fullmat, lev2, lev3, full_ten, lrappx, idx2, idx3)
% Forms tensors that are then decomposed using HOSVD to get a storage
% saving formulation. They take advantage of the hierarchical approximation
% of A and actually uses these to access the real A_full.

%

if hmat.flag == 0
[lev2, lev3, full_ten, lrappx, idx2, idx3] = ...
        hmat2tens(hmat.H_11, fullmat, lev2, lev3, full_ten, lrappx, idx2, idx3);
[lev2, lev3, full_ten, lrappx, idx2, idx3] = ...
        hmat2tens(hmat.H_12, fullmat, lev2, lev3, full_ten, lrappx, idx2, idx3);
[lev2, lev3, full_ten, lrappx, idx2, idx3] = ...
        hmat2tens(hmat.H_21, fullmat, lev2, lev3, full_ten, lrappx, idx2, idx3);
[lev2, lev3, full_ten, lrappx, idx2, idx3] = ...
        hmat2tens(hmat.H_22, fullmat, lev2, lev3, full_ten, lrappx, idx2, idx3);
elseif hmat.flag == 1 %Low Rank, can HOSVD
rows = hmat.row_start: hmat.row_start + hmat.n_row - 1;
cols = hmat.col_start: hmat.col_start + hmat.n_col - 1;
lrappx(rows, cols) = hmat.rk_matrix.A * hmat.rk_matrix.B';
if length(rows) == 32 %lev == 2 %size 32 x 32
lev3 = cat(3, lev3, fullmat(rows,cols));
idx3 = [idx3; {rows, cols}];
elseif length(rows) == 64 %lev == 3
lev2 = cat(3, lev2, fullmat(rows,cols));
idx2 = [idx2; {rows, cols}];
end
elseif hmat.flag == -1 %Full rank, no advantage
% construct full tensor?
rows = hmat.row_start: hmat.row_start + hmat.n_row - 1;
cols = hmat.col_start: hmat.col_start + hmat.n_col - 1;
lrappx(rows, cols) = hmat.full_matrix;
full_ten = cat(3, full_ten, hmat.full_matrix);
end

end
```

Bibliography

[1] *Chapter 8 further applications of fractional models*, in North-Holland Mathematics Studies, Anatoly A. Kilbas, Hari M. Srivastava, and Juan J. Trujillo, eds., vol. 204 of Theory and Applications of Fractional Differential Equations, North-Holland, pp. 449–468.

[2] *Low-Rank Matrices and Matrix Partitioning*, Lecture Notes in Computational Science and Engineering, Springer, 1 ed., pp. 9–47.

[3] Daniele. Bertaccini and Fabio Durastante, *Block structured preconditioners in tensor form for the all-at-once solution of a finite volume fractional diffusion equation*, 95, pp. 92–97.

[4] Daniele Bertaccini and Fabio Durastante, *Limited memory block preconditioners for fast solution of fractional partial differential equations*, 77, pp. 950–970.

[5] Steffen Börm, Lars Grasedyck, and Wolfgang Hackbusch, *Introduction to hierarchical matrices with applications*, 27, pp. 405–422.

[6] Raymond H. Chan, *The spectrum of a family of circulant preconditioned toeplitz systems*, 26, pp. 503–506. Publisher: Society for Industrial and Applied Mathematics.

[7] Raymond H. Chan and Michael K. Ng, *Conjugate gradient methods for toeplitz systems*, 38, pp. 427–482. Publisher: Society for Industrial and Applied Mathematics.

[8] Raymond H. Chan and Gilbert Strang, *Toeplitz equations by conjugate gradients with circulant preconditioner*, 10, pp. 104–119.

[9] Sheng Chen, Jie Shen, and Li-Lian Wang, *Generalized jacobi functions and their applications to fractional differential equations*.

[10] Pingfei Dai, Qingbiao Wu, Hong Wang, and Xiangcheng Zheng, *An efficient matrix splitting preconditioning technique for two-dimensional unsteady space-fractional diffusion equations*, 371, p. 112673.

[11] Lieven De Lathauwer, Bart De Moor, and Joos Vandewalle, *A multilinear singular value decomposition*, 21, pp. 1253–1278. Publisher: Society for Industrial and Applied Mathematics.

[12] Gene H. Golub and Charles F. Van Loan, *Matrix Computations*, JHU Press. Google-Books-ID: X5YfsuCWpxMC.

[13] Wolfgang Hackbusch, *Survey on the technique of hierarchical matrices*, 44, pp. 71–101.

[14] Christopher Hillar and Lek-Heng Lim, *Most tensor problems are NP-hard*.

[15] Frank L. Hitchcock, *The expression of a tensor or a polyadic as a sum of products*, 6, pp. 164–189. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/sapm192761164.

[16] Johan Håstad, *Tensor rank is NP-complete*, 11, pp. 644–654.

[17] Ilghiz Ibraghimov, *Application of the three-way decomposition for matrix compression*, 9, pp. 551–565. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/nla.297.

[18] Misha E. Kilmer and Arvind K. Saibaba, *Structured matrix approximations via tensor decompositions.*

[19] Tamara G. Kolda and Brett W. Bader, *Tensor decompositions and applications*, 51, pp. 455–500.

[20] James G. Nagy and Misha E. Kilmer, *Kronecker product approximation for preconditioning in three-dimensional imaging applications*, 15, pp. 604–613. Conference Name: IEEE Transactions on Image Processing.

[21] Ivan V. Oseledets, *Tensor-train decomposition*, 33, pp. 2295–2317. Publisher: Society for Industrial and Applied Mathematics.

[22] Will Pazner and Per-Olof Persson, *Approximate tensor-product preconditioners for very high order discontinuous galerkin methods*, 354, pp. 344–369.

[23] Britta Schmitt, Boris N. Khoromskij, Venera Khoromskaia, and Volker Schulz, *Tensor method for optimal control problems constrained by fractional 3d elliptic operator with variable coefficients.*

[24] Darya A. Sushnikova and Ivan V. Oseledets, *Preconditioners for hierarchical matrices based on their extended sparse form*, 31, pp. 29–40. Publisher: De Gruyter.

[25] Xuan Zhao, Xiaozhe Hu, Wei Cai, and George Em Karniadakis, *Adaptive finite element method for fractional differential equations using hierarchical matrices*, 325, pp. 56–76.