University of California
Santa Barbara

Advancements in Algorithms for Approximations of Rank Structured Matrices

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

Doctor of Philosophy
in
Electrical and Computer Engineering

by

Kristen Lessel

Committee in charge:

Professor Shivkumar Chandrasekaran, Chair
Professor Bassam Bamieh
Professor John Gilbert
Professor João Hespanha

March 2020

The Dissertation of Kristen Lessel is approved.

Professor Bassam Bamieh

Professor John Gilbert

Professor João Hespanha

Professor Shivkumar Chandrasekaran, Committee Chair

March 2020

Advancements in Algorithms for Approximations of Rank Structured Matrices

by

Kristen Lessel

To my Grandparents, Jim and Joan Lessel

Acknowledgements

First, and foremost I would like to thank my adviser, Professor Chandrasekaran. I could have never known how far I would come, and how much I would learn when I first met him. I feel that his teaching and guidance has given me the skill set that will allow me to tackle any real life problem I come across. I feel unable to express my gratitude appropriately with only these few words on paper, but I am extremely grateful, and feel so lucky to have had the chance to work with him.

I would also like to thank my Grandparents, Jim and Joan Lessel for their love and support during my years in grad school.

Lastly I would like to thank Barret Chrisman for his unwavering love and support, especially through all of the many stressful times over the past few years. All of which helped me to be where I am today, and who I am today.

Curriculum Vitæ

Kristen Lessel

Education

2020	Ph.D. in Computer Engineering (Expected), University of California, Santa Barbara.
2013	M.S. in Electrical and Computer Engineering, University of California, Santa Barbara.
2004	B.S. in Applied Mathematics, University of California, Santa Barbara.

Publications

Lessel, K., M. Hartman, and Shivkumar Chandrasekaran. "A fast memory efficient construction algorithm for hierarchically semi-separable representations." SIAM Journal on Matrix Analysis and Applications 37.1 (2016): 338-353.

Lessel, Kristen, and Stephen McClain. "Low uncertainty measurements of bidirectional reflectance factor on the NPOESS/VIIRS solar diffuser." Earth Observing Systems XII. Vol. 6677. International Society for Optics and Photonics, 2007.

Awards and Honors

Texas Instruments Scholarship Award for 2013 by the Society of Women Engineers

Abstract

Advancements in Algorithms for Approximations of Rank Structured Matrices

by

Kristen Lessel

Many problems in mathematical physics and engineering involve solving linear systems $Ax = b$ which are highly structured [1, 2]. These structured matrices, which typically arise from discretizations of partial differential or integral equations, can be represented compactly through specific algebraic representations. Two such algebraic representations which fall into this category are the fast multipole method (FMM), originally introduced by Greengard and Rokhlin [3], and Hierarchically Semi-Separable (HSS) ([4]) . These representations, which exploit the low-rank structure of off-diagonal blocks, have enabled fast solvers (linear time in certain scenarios) and are commonly used practice today.

In this thesis, we provide new advancements which further enhance the performance of these algorithms. The contributions of this thesis are twofold: (i) we present a new fast memory efficient construction algorithm for Hierarchically Semi-Separable (HSS) representations, and (ii) we present a fast matrix-matrix multiply for the fast multipole method (FMM).

The HSS representation takes advantage of the fact that off-diagonal blocks are known to have low rank in order to yield fast solvers. The memory consumption of the HSS representation itself is $O(n)$ if the rank of the off-diagonal blocks is small. If the user is not required to store the matrix A, but instead only provides a functional interface in order to access the elements of the matrix, it is worthwhile to ask for the algorithm which computes the HSS representation to be memory efficient as well. Previous algorithms,

[4, 5], have shown the HSS representation can be computed in $O(n^2)$ flops. Randomized algorithms also exist [6, 7, 8]. However, the memory requirements of these algorithms can be excessive, requiring as much as $O(n^2)$ peak workspace memory [4, 6, 7, 5]. We deal with this issue and present an algorithm that requires $O(n^{1.5})$ peak workspace memory in the worst case, while still requiring only $O(n^2)$ flops.

The HSS Representation assumes off-diagonal blocks which have low rank, but in practice there are many cases for which this criteria is not satisfied, and in fact can be as much as $O(\sqrt{n})$. For this reason, there is much interest in FMM, as it relaxes this requirement, only demanding that off-diagonal blocks corresponding to well-separated clusters have low rank. However, the structure of the FMM inverse is not known. To better understand this problem, we consider the problem of computing a 1D FMM representation of the matrix-matrix product of two 1D FMM matrices. We show that the product of two standard (3 pt) 1D FMM matrices possesses a slightly modified 5 pt 1D FMM structure, and we provide a linear time ($O(n)$ flop) algorithm for computing this product. Further, this work suggests that the inverse of an FMM matrix is not itself FMM.

Contents

Chapter 1

Introduction

The Fast Multipole Method (FMM) was originally introduced by Greengard and Rokhlin [3] and has proven to be useful in a variety of applicactions [1, 2]. With this contribution, accurate calculations of the motions of n particles interacting via gravitational or electrostatic forces can be carried out in only $O(n)$ flops as compared with standard techniques which require $O(n^2)$ flops. However, FMM representations may not be closed under inversion, and so Hierarchically Semi-Separable (HSS) representations were introduced[4]. Previous HSS construction algorithms take as much as $O(n^2)$ memory at a cost of $O(n^2)$ flops[4, 2]. HSS representations have the benefit that inversion and multiplication can be performed in $O(n)$ flops, however this comes with the constraint that all off-diagonal blocks must be low-rank.

In chapter 2 we will present a memory efficient HSS construction algorithm for $n \times n$ matrices with off-diagonal blocks that have low rank. Previous construction algorithms can use as much as $O(n^2)$ memory [4, 2]. Randomized algorithms also exist[6, 7, 8]. Here we give an algorithm which takes only $O(n^{1.5})$ peak workspace memory at most, while still requiring only $O(n^2)$ flops.

The HSS representation is often used in practice [7, 9], however, since the HSS rep-

resentation requires that all off-diagonal blocks have low rank, this might not be the best approach because this criteria is not always satisfied. We will give one such common example in chapter 3, in which off-diagonal blocks actually have rank $O(\sqrt{n})$. In these cases we might do better to consider using the FMM representation instead, which would take advantage of this structure. Unfortunately, FMM itself is not closed under multiplication or inversion, and further, the structure of the inverse is not known. The FMM inverse is at least as complicated as the FMM matrix-matrix multiply, and so we propose that to understand the structure of the FMM inverse we must first understand the structure that results from the FMM matrix-matrix multiply. Here we will examine the structure resulting from the 1D FMM matrix-matrix multiply in detail, which will then allow us to provide insights into the structure of the inverse itself. Further, we will also give on example where the matrix-matrix multiply will allow us to directly compute an approximation to the inverse. In this thesis, we will only be addressing the 1D FMM representation and therefore any reference to FMM is implied to be referring to 1D FMM unless otherwise stated.

1.1 Partition Trees

Define a **regular binary tree** to be an ordered rooted tree in which each parent node has two children. Denote the set of regular binary trees as $\mathcal{T}_{reg}$. Specifically each child of a parent node is either a **left child** or a **right child**. A subtree rooted at the left (right) child of a given node, r, is called r's **left (right) subtree** [10]. Each node in this regular binary tree is associated with an index $(k; i)$, where k denotes the node's depth from the root node, and i denotes left to right ordering in that level. Label the root node as $(0; 1)$. Then every parent node has a labeling $(k; i)$, with left child labeled as $(k + 1; 2i - 1)$ and right child labeled as $(k + 1; 2i)$.

Define a **partition tree** to be a regular binary tree that has an integer, $n_{k;i}$ at every node, $(k;i)$, which satisfy the property that $n_{k;i} = n_{k+1;2i-1} + n_{k+1;2i}$. A partition tree, T, of depth d, is **complete**, if all levels $k \in \{0, 1, ..., d-1\}$ of T have 2^k vertices [10].

We use partition trees to hierarchically partition a matrix $A \in \mathbb{R}^{m \times n}$. Let T_R and T_c be partition trees. Let $m_{k;i}$ and $n_{k;i}$ denote the integer at node $(k;i)$ in T_R and T_c, respectively. Let $A_{0;1,1} = A$, $m_{0;1} = m$ and $n_{0;1} = n$. Partitioning the rows and columns of the matrix A according to the integers at the first level of T_R and T_c,

$$A_{0;1,1} = \begin{array}{c} \\ m_{1;1} \\ m_{1;2} \end{array} \begin{array}{c} \begin{array}{cc} n_{1;1} & n_{1;2} \end{array} \\ \begin{pmatrix} A_{1;1,1} & A_{1;1,2} \\ A_{1;2,1} & A_{1;2,2} \end{pmatrix} \end{array}. \tag{1.1}$$

Recursively partitioning the block rows and block columns of A according to the integers stored at the corresponding nodes of T_R and T_C, respectively,

$$A_{k-1;i,k-1;i} = \begin{array}{c} \\ m_{k;2i-1} \\ m_{k;2i} \end{array} \begin{array}{c} \begin{array}{cc} n_{k;2i-1} & n_{k;2i} \end{array} \\ \begin{pmatrix} A_{k;2i-1,2i-1} & A_{k;2i-1,2i} \\ A_{k;2i,2i-1} & A_{k;2i,2i} \end{pmatrix} \end{array}. \tag{1.2}$$

Figure 1.1 shows an example of a matrix which has rows partitioned according to a partition tree T_R, and columns partitioned according to a partition tree T_C. In any level, when it is the case that the row partitions are the same as the column partitions, we use the more simplified notation, $A_{k;i,j} = A_{k;i,k;j}$ (Figure 1.1). In this case, it is the convention that the partition tree corresponding to the row partitions and the partition tree corresponding to the column partitions are merged. For the purpose of this thesis we will focus on this case.

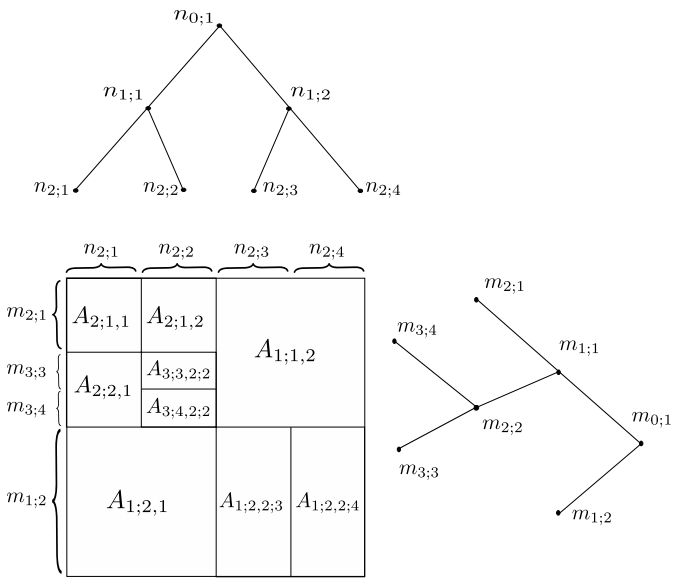


Figure 1.1: Example Block Partitioning of a Matrix, A, with Corresponding Partition Trees, T_R (bottom right) and T_C (top left)

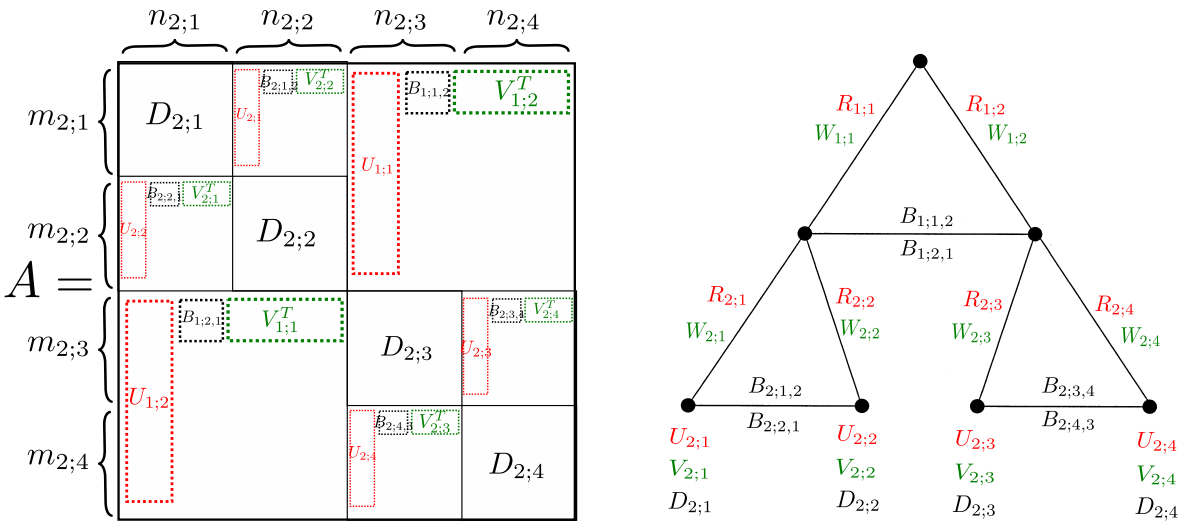
1.2 HSS Representation

Any matrix has an HSS representation, but if the matrix has off diagonal blocks with low numerical rank, this representation will consume less memory. The HSS representation is defined as follows. Let T be a partition tree for the matrix A. Partition the rows and columns of A according to T as defined in 1.1 above. Suppose, $D_{k;i} = A_{k;i}$ if $(k;i)$ is a leaf node and if $(k;i)$ is not a leaf node,

$$\begin{aligned} A_{k;2i-1,2i} &= U_{k;2i-1} B_{k;2i-1;2i} V_{k;2i}^T, \\ A_{k;2i,2i-1} &= U_{k;2i} B_{k;2i;2i-1} V_{k;2i-1}^T, \end{aligned} \tag{1.3}$$

such that equation (1.4) holds.

$$U_{k;i} = \begin{pmatrix} U_{k+1;2i-1} R_{k+1;2i-1} \\ U_{k+1;2i} R_{k+1;2i} \end{pmatrix}, \quad V_{k;i} = \begin{pmatrix} V_{k+1;2i-1} W_{k+1;2i-1} \\ V_{k+1;2i} W_{k+1;2i} \end{pmatrix}. \tag{1.4}$$

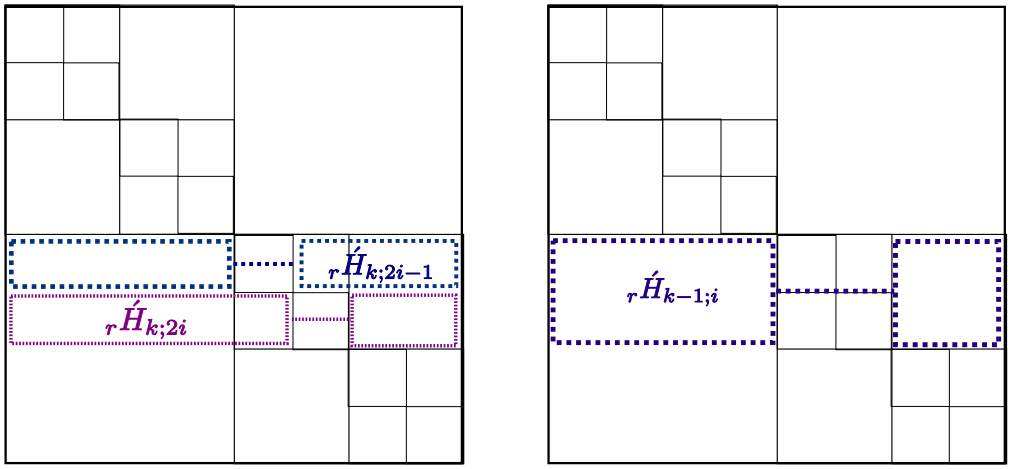


(a) 2 level HSS (b) 2 level HSS tree

Figure 1.2: 2 level HSS Matrix and Binary Tree

Then, the **HSS representation** consists of **basis matrices**, $U_{k;i}$, and $V_{k;i}$, as well as $D_{k;i}$ for all leaf nodes, $(k;i)$. Further, for all $(k;i)$, which are not leaf nodes, the representation consists of **translation operators**, $R_{k;i}$, $W_{k;i}$, and **expansion coefficients**, $B_{k;i-1,i}$, for i odd, and $B_{k;i;i-1}$ for i even. The existence of this factorization is established [4], and this will become more obvious as we proceed.

Define an **HSS tree** of the matrix A to be the corresponding partition tree of A decorated with the matrices $U_{k;i}$, $V_{k;i}$, $D_{k;i}$, $R_{k;i}$, $W_{k;i}$, $B_{k;i,j}$. We store $U_{k;i}$, $V_{k;i}$, $D_{k;i}$ at each leaf node $(k;i)$. $R_{k;i}$ and $W_{k;i}$ are stored at each edge connecting parent to child node, $(k;i)$. Further we add edges to the partition tree from node $(k;i)$ to node $(k;j)$ corresponding to $B_{k;i,j}$, where $j = i+1$ and $j = i-1$ (these edges corresponding to the expansion coefficients run between sibling nodes). Since $U_{k;i}$ ($V_{k;i}$) are only stored at leaf nodes, only the smaller $R_{k;i}$ ($W_{k;i}$) are stored at each subsequent level. Basis matrices at higher levels are obtained via matrix multiplication with corresponding translation operators (1.4). In figure 1.2 we give an example of a 2-level HSS representation of a matrix, along with its corresponding HSS tree. In order that the translation operators



(a) Hankel Blocks, $_r\acute{H}_{k;2i-1}$ and, $_r\acute{H}_{k;2i}$ with $k = 3$, $i = 3$

(b) Hankel block, $_r\acute{H}_{k-1;i}$, with $k = 2$, $i = 3$

Figure 1.3: Row Hankel Blocks for HSS

$R_{k;i}$ and $W_{k;i}$ be as defined above, we must have that each $U_{k;i}$ be a column basis for the corresponding row Hankel block, which is

$$_r\acute{H}_{k;i} = \begin{pmatrix} A_{k;i,1} & A_{k;i,2} & \dots & A_{k;i,i-1} & A_{k;i,i+1} & \dots & A_{k;i,end} \end{pmatrix}. \tag{1.5}$$

And likewise, we must have that each $V_{k;i}$ be a row basis for its corresponding column Hankel block, which is

$$_c\acute{H}_{k;i} = \begin{pmatrix} A^T_{k;1,i} & A^T_{k;2,i} & \dots & A^T_{k;i-1,i} & A^T_{k;i+1,i} & \dots & A^T_{k;end,i} \end{pmatrix}. \tag{1.6}$$

Notice that, $_r\acute{H}_{k;i}$ is the i^{th} block row of A in the k level HSS structure, excluding the $A_{k;i,i}$ block. Similarly, $_c\acute{H}_{k;i}$ is the i^{th} block column of A corresponding to the k^{th} level of the HSS representation, excluding the $A_{k;i,i}$ block as illustrated by Figures 1.3a, and 1.3b. The i^{th} block row of A in the $(k-1)^{st}$ level HSS structure, excluding the $A_{k-1;i,i}$

block, is ${}_r\acute{H}_{k-1;i}$ (Figure 1.3b) , and is defined as

$$
\begin{aligned}
{}_r\acute{H}_{k-1;i} &= \begin{pmatrix} A_{k;2i-1,1} & A_{k;2i-1,2} & \dots & A_{k;2i-1,2i-2} & A_{k;2i-1,2i+1} & \dots & A_{k;2i-1,end} \\ A_{k;2i,1} & A_{k;2i,2} & \dots & A_{k;2i,2i-2} & A_{k;2i,2i+1} & \dots & A_{k;2i,end} \end{pmatrix} \\
&= \begin{pmatrix} A_{k-1;i,1} & A_{k-1;i,2} & \dots & A_{k-1;i,i-1} & A_{k-1;i,i+1} & \dots & A_{k-1;i,end} \end{pmatrix}.
\end{aligned} \tag{1.7}
$$

The same definition is extended to the column Hankel blocks.

1.3 1D FMM Representation

Any matrix has an FMM representation, but if the matrix has the property that off diagonal blocks which do not touch the diagonal have low numerical rank, this representation will consume less memory.

1.3.1 1D FMM 3pt Representation

The FMM 3pt representation is defined as follows. Let T be a partition tree for the matrix A. Partition the rows and columns of A according to T as defined in 1.1 above. If $(k-1, i, j)$ is a leaf node with $i = j$ then

$$
\begin{aligned}
A_{k;2i-1,2i-1} &= D_{k;2i-1,2i-1}, \\
A_{k;2i-1,2i} &= D_{k;2i-1,2i}, \\
A_{k;2i,2i-1} &= D_{k;2i,2i-1}, \\
A_{k;2i,2i} &= D_{k;2i,2i}.
\end{aligned} \tag{1.8}
$$

If $(k-1,i,j)$ is a leaf node with $i = j-1$ then

$$\begin{aligned} A_{k;2i-1,2i+1} &= U_{k;2i-1} B_{k;2i-1;2i+1} V_{k;2i+1}^T, \\ A_{k;2i-1,2i+2} &= U_{k;2i-1} B_{k;2i-1;2i+2} V_{k;2i+2}^T, \\ A_{k;2i,2i+1} &= D_{k;2i,2i+1}, \\ A_{k;2i,2i+2} &= U_{k;2i} B_{k;2i;2i+2} V_{k;2i+2}^T. \end{aligned} \tag{1.9}$$

And if $(k-1,i,j)$ is a leaf node with $i = j+1$ then

$$\begin{aligned} A_{k;2i+1,2i-1} &= U_{k;2i+1} B_{k;2i+1;2i-1} V_{k;2i-1}^T, \\ A_{k;2i+1,2i} &= D_{k;2i+1;2i}, \\ A_{k;2i+2,2i-1} &= U_{k;2i+2} B_{k;2i+2;2i-1} V_{k;2i-1}^T, \\ A_{k;2i+2,2i} &= U_{k;2i+2} B_{k;2i+2;2i} V_{k;2i}^T. \end{aligned} \tag{1.10}$$

such that equation (1.4) holds.

Then, the **FMM representation** consists of **basis matrices**, $U_{k;i}$, and $V_{k;i}$, as well as $D_{k;i}$ for all leaf nodes, $(k;i)$. Further, for all $(k;i)$, which are not leaf nodes, the representation consists of **translation operators**, $R_{k;i}$, $W_{k;i}$, and **expansion coefficients**, $B_{k;i,i+2}$, $B_{k;i,i+3}$, $B_{k;i+1,i+3}$, $B_{k;i+2,i}$, $B_{k;i+3,i}$ and $B_{k;i+3,i+1}$.

Define a **3pt FMM tree** of the matrix A to be the corresponding partition tree of A decorated with the matrices $U_{k;i}$, $V_{k;i}$, $D_{k;i}$, $R_{k;i}$, $W_{k;i}$, $B_{k;i,j}$. We store $U_{k;i}$, $V_{k;i}$, $D_{k;i}$ at each leaf node $(k;i)$. $R_{k;i}$ and $W_{k;i}$ are stored at each edge connecting parent to child node, $(k;i)$. Further we add edges to the partition tree from node $(k;i)$ to node $(k;j)$ corresponding to $B_{k;i,j}$ (these edges corresponding to the expansion coefficients run between cousin nodes). Just as in the HSS representation, since $U_{k;i}$ ($V_{k;i}$) are only stored at leaf nodes, only the smaller $R_{k;i}$ ($W_{k;i}$) are stored at each subsequent level. Basis matrices at higher levels are obtained via matrix multiplication with corresponding

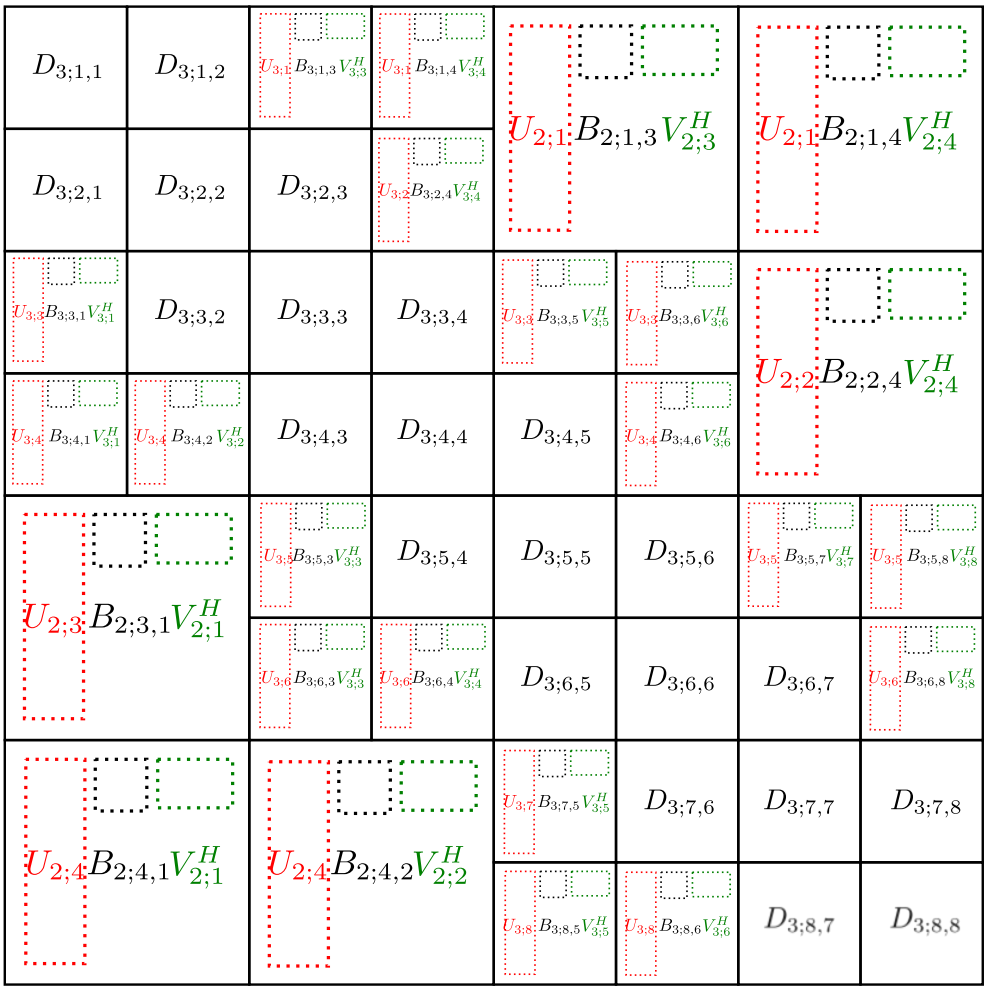


Figure 1.4: 3 Level 3pt FMM Matrix

translation operators (1.4). In figure 1.4 and 1.5 we give an example of a 3-level FMM representation of a matrix, along with its corresponding FMM tree. Note that in the FMM 3pt representation the edges between cousin nodes correspond to the expansion coefficients, whereas in HSS these edges run in between sibling nodes instead. Just as in the HSS representation, in order that the translation operators $R_{k;i}$ and $W_{k;i}$ be as defined in 1.4, we must have that each $U_{k;i}$ be a column basis for the corresponding row

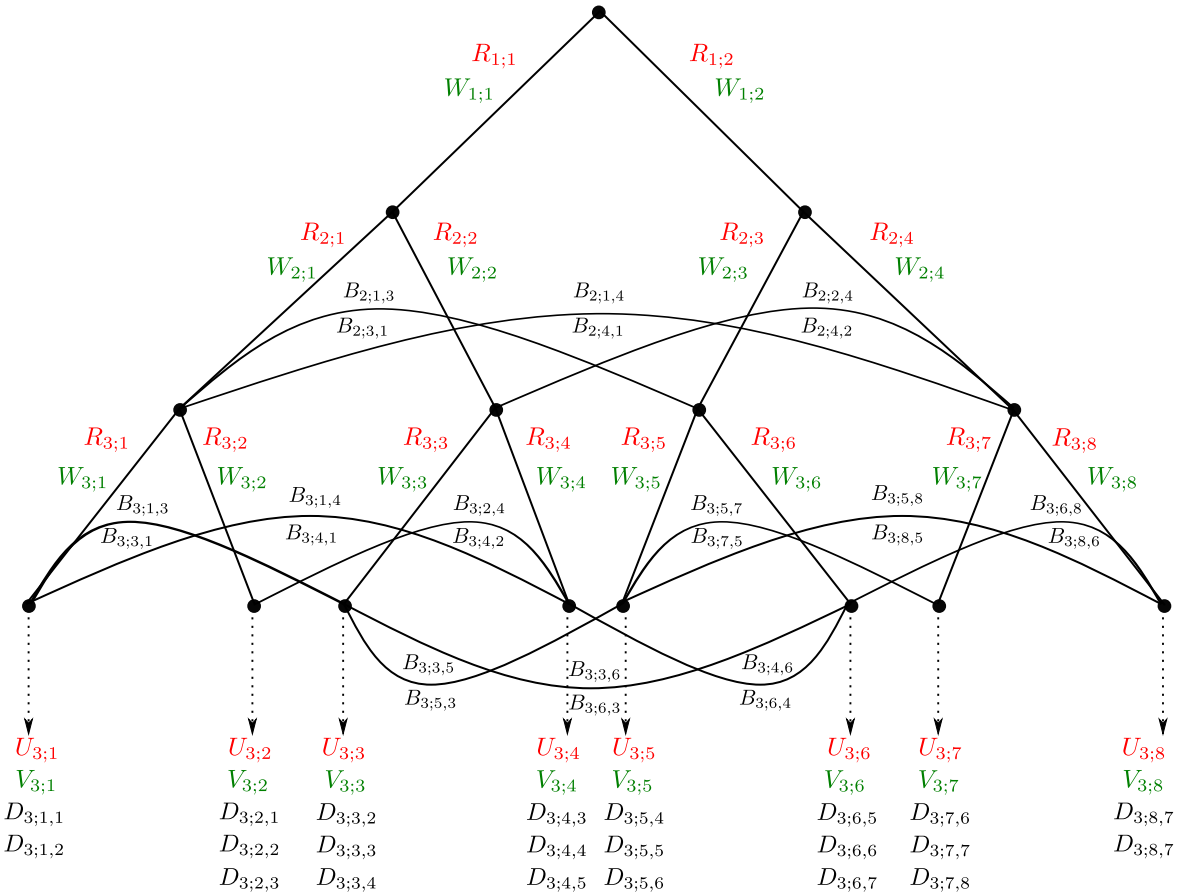
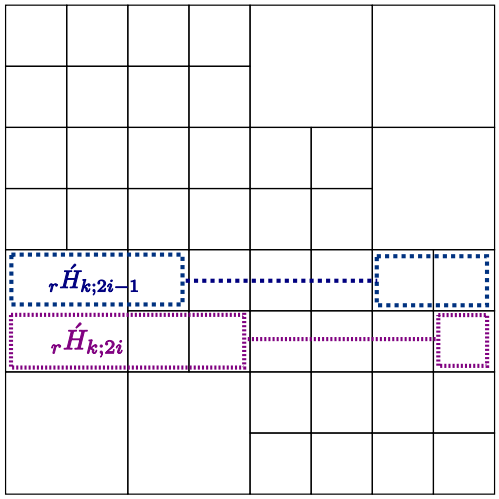
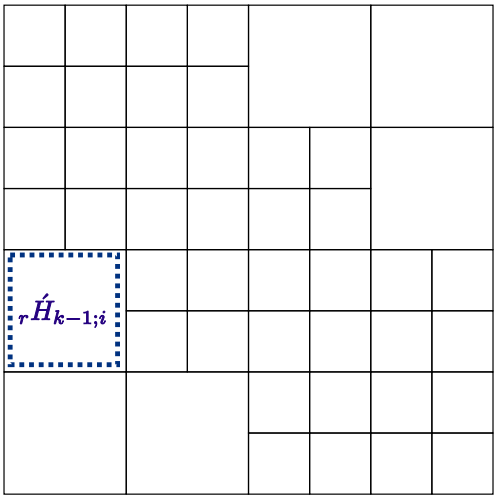


Figure 1.5: 3 Level 3pt FMM Tree



(a) Hankel Blocks, $_r\acute{H}_{k;2i-1}$ and, $_r\acute{H}_{k;2i}$ with $k = 3$, $i = 3$



(b) Hankel block, $_r\acute{H}_{k-1;i}$, with $k = 2$, $i = 3$

Figure 1.6: Row Hankel Blocks for FMM

Hankel block, which is

$$_r\acute{H}_{k;i} = \begin{pmatrix} A_{k;i,1} & A_{k;i,2} & \dots & A_{k;i,i-2} & A_{k;i,i+2} & \dots & A_{k;i,end} \end{pmatrix}. \tag{1.11}$$

And likewise, we must have that each $V_{k;i}$ be a row basis for its corresponding column Hankel block, which is

$$_c\acute{H}_{k;i} = \begin{pmatrix} A^T_{k;1,i} & A^T_{k;2,i} & \dots & A^T_{k;i-2,i} & A^T_{k;i+2,i} & \dots & A^T_{k;end,i} \end{pmatrix}. \tag{1.12}$$

Notice that, $_r\acute{H}_{k;i}$ is the i^{th} block row of A in the k level FMM structure, excluding the $A_{k;i,i-1}$, $A_{k;i,i}$ and $A_{k;i,i+1}$ blocks. Similarly, $_c\acute{H}_{k;i}$ is the i^{th} block column of A corresponding to the k^{th} level of the HSS representation, excluding the $A_{k;i,i}$, $A_{k;i-1,i}$ and $A_{k;i+1,i}$ blocks as illustrated by Figures 1.6a, and 1.6b. The i^{th} block row of A in the $(k-1)^{st}$ level HSS structure, excluding the $A_{k-1;i,i-1}$, $A_{k-1;i,i}$ and $A_{k-1;i,i+1}$ blocks, is $_r\acute{H}_{k-1;i}$ (Figure

1.6b) , and is defined as

$$
\begin{aligned}
{}_r\acute{H}_{k-1;i} &= \begin{pmatrix} A_{k;2i-1,1} & A_{k;2i-1,2} & \dots & A_{k;2i-1,2i-4} & A_{k;2i-1,2i+3} & \dots & A_{k;2i-1,end} \\ A_{k;2i,1} & A_{k;2i,2} & \dots & A_{k;2i,2i-4} & A_{k;2i,2i+3} & \dots & A_{k;2i,end} \end{pmatrix} \\
&= \begin{pmatrix} A_{k-1;i,1} & A_{k-1;i,2} & \dots & A_{k-1;i,i-2} & A_{k-1;i,i+2} & \dots & A_{k-1;i,end} \end{pmatrix}.
\end{aligned} \tag{1.13}
$$

The same definition is extended to the column Hankel blocks.

1.3.2 FMM 5pt Representation

The FMM 5pt representation is the representation we obtain when we multiply two FMM 3pt matrices and is defined as follows. Let T be a partition tree for the matrix A. Partition the rows and columns of A according to T as defined in 1.1 above. If $(k-1,i,j)$ is a leaf node with $i=j$ then

$$
\begin{aligned}
A_{k;2i-1,2i-1} &= D_{k;2i-1,2i-1}, \\
A_{k;2i-1,2i} &= D_{k;2i-1,2i}, \\
A_{k;2i,2i-1} &= D_{k;2i,2i-1}, \\
A_{k;2i,2i} &= D_{k;2i,2i}.
\end{aligned} \tag{1.14}
$$

If $(k-1,i,j)$ is a leaf node with $i=j-1$ then

$$
\begin{aligned}
A_{k;2i-1,2i+1} &= U_{k;2i-1} B_{k;2i-1;2i+1} V^T_{k;2i+1}, \\
A_{k;2i-1,2i+2} &= D_{k;2i-1;2i+2}, \\
A_{k;2i,2i+1} &= D_{k;2i,2i+1}, \\
A_{k;2i,2i+2} &= D_{k;2i;2i+2}.
\end{aligned} \tag{1.15}
$$

And if $(k-1,i,j)$ is a leaf node with $i = j+1$ then

$$\begin{aligned} A_{k;2i+1,2i-1} &= D_{k;2i+1;2i-1}, \\ A_{k;2i+1,2i} &= D_{k;2i+1;2i}, \\ A_{k;2i+2,2i-1} &= U_{k;2i+2} B_{k;2i+2;2i-1} V_{k;2i-1}^T, \\ A_{k;2i+2,2i} &= D_{k;2i+2;2i}. \end{aligned} \tag{1.16}$$

And if $(k-1,i,j)$ is a leaf node with $i = j-2$ then

$$\begin{aligned} A_{k;2i+3,2i-1} &= U_{k;2i+3} B_{k;2i+3;2i-1} V_{k;2i-1}^T, \\ A_{k;2i+3,2i} &= U_{k;2i+3} B_{k;2i+3;2i} V_{k;2i}^T, \\ A_{k;2i+4,2i-1} &= U_{k;2i+4} B_{k;2i+4;2i-1} V_{k;2i-1}^T, \\ A_{k;2i+4,2i} &= U_{k;2i+4} B_{k;2i+4;2i} V_{k;2i}^T. \end{aligned} \tag{1.17}$$

Finally, if $(k-1,i,j)$ is a leaf node with $i = j+2$ then

$$\begin{aligned} A_{k;2i-1,2i+3} &= U_{k;2i-1} B_{k;2i-1;2i+3} V_{k;2i+3}^T, \\ A_{k;2i-1,2i+4} &= U_{k;2i-1} B_{k;2i-1;2i+4} V_{k;2i+4}^T, \\ A_{k;2i,2i+3} &= U_{k;2i} B_{k;2i;2i+3} V_{k;2i+3}^T, \\ A_{k;2i,2i+4} &= U_{k;2i} B_{k;2i;2i+4} V_{k;2i+4}^T. \end{aligned} \tag{1.18}$$

such that equation (1.4) holds.

Then, the **FMM 5pt representation** consists of **basis matrices**, $U_{k;i}$, and $V_{k;i}$, as well as $D_{k;i}$ for all leaf nodes, $(k;i)$. Further, for all $(k;i)$, which are not leaf nodes, the representation consists of **translation operators**, $R_{k;i}$, $W_{k;i}$, and **expansion coefficients**, $B_{k;i,i+2}$, $B_{k;i,i+3}$, $B_{k;i+1,i+3}$, $B_{k;i+2,i}$, $B_{k;i+3,i}$ and $B_{k;i+3,i+1}$ for $(k;i,j)$ not leaf nodes and $B_{k;i,i+3}$ and $B_{k;i+3,i+1}$ for $(k;i,j)$ leaf nodes.

Define a **5pt FMM tree** of the matrix A to be the corresponding partition tree of

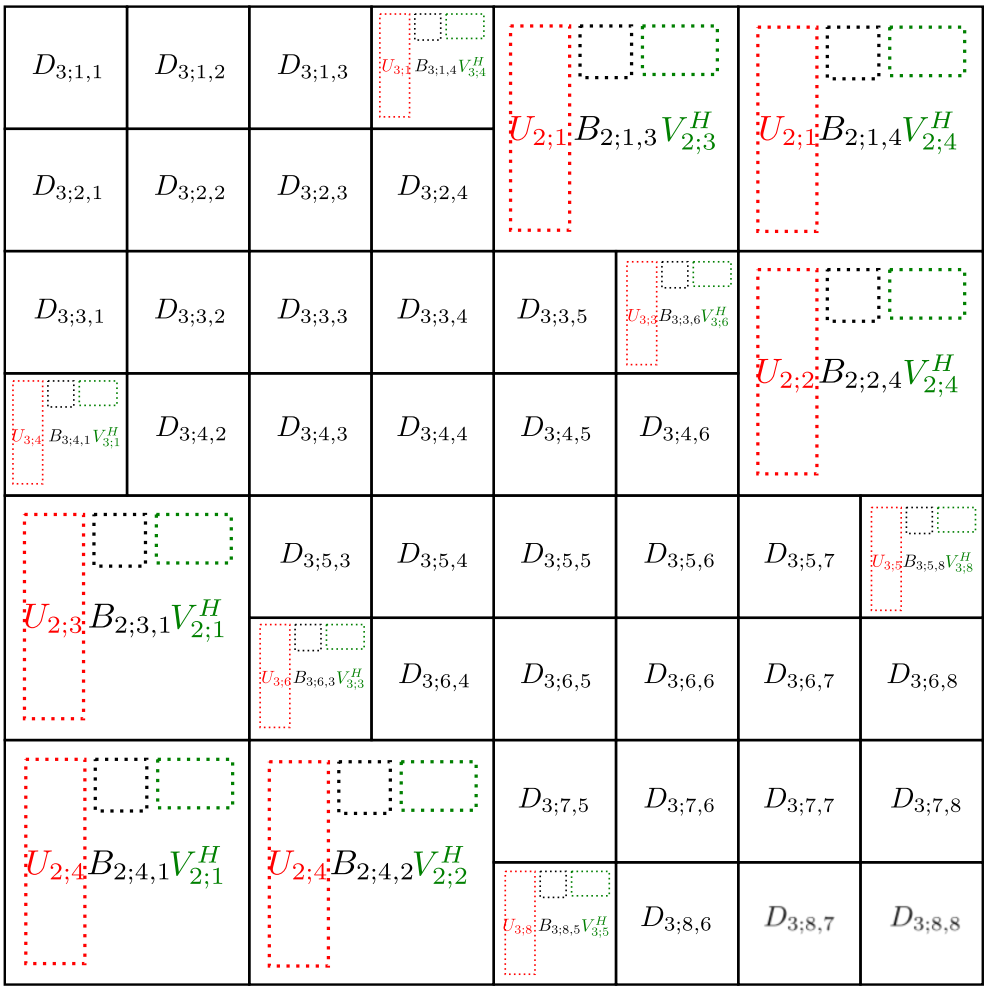


Figure 1.7: 3 Level 5pt FMM Matrix

A decorated with the matrices $U_{k;i}$, $V_{k;i}$, $D_{k;i}$, $R_{k;i}$, $W_{k;i}$, $B_{k;i,j}$. We store $U_{k;i}$, $V_{k;i}$, $D_{k;i}$ at each leaf node $(k;i)$. $R_{k;i}$ and $W_{k;i}$ are stored at each edge connecting parent to child node, $(k;i)$. Further we add edges to the partition tree from node $(k;i)$ to node $(k;j)$ corresponding to $B_{k;i,j}$ listed above. Again, since $U_{k;i}$ ($V_{k;i}$) are only stored at leaf nodes, only the smaller $R_{k;i}$ ($W_{k;i}$) are stored at each subsequent level. Basis matrices at higher levels are obtained via matrix multiplication with corresponding translation operators (1.4). In figure 1.7 and 1.8 we give an example of a 3-level 5pt FMM representation of a matrix, along with its corresponding FMM tree. Note that in the FMM 5pt representa-

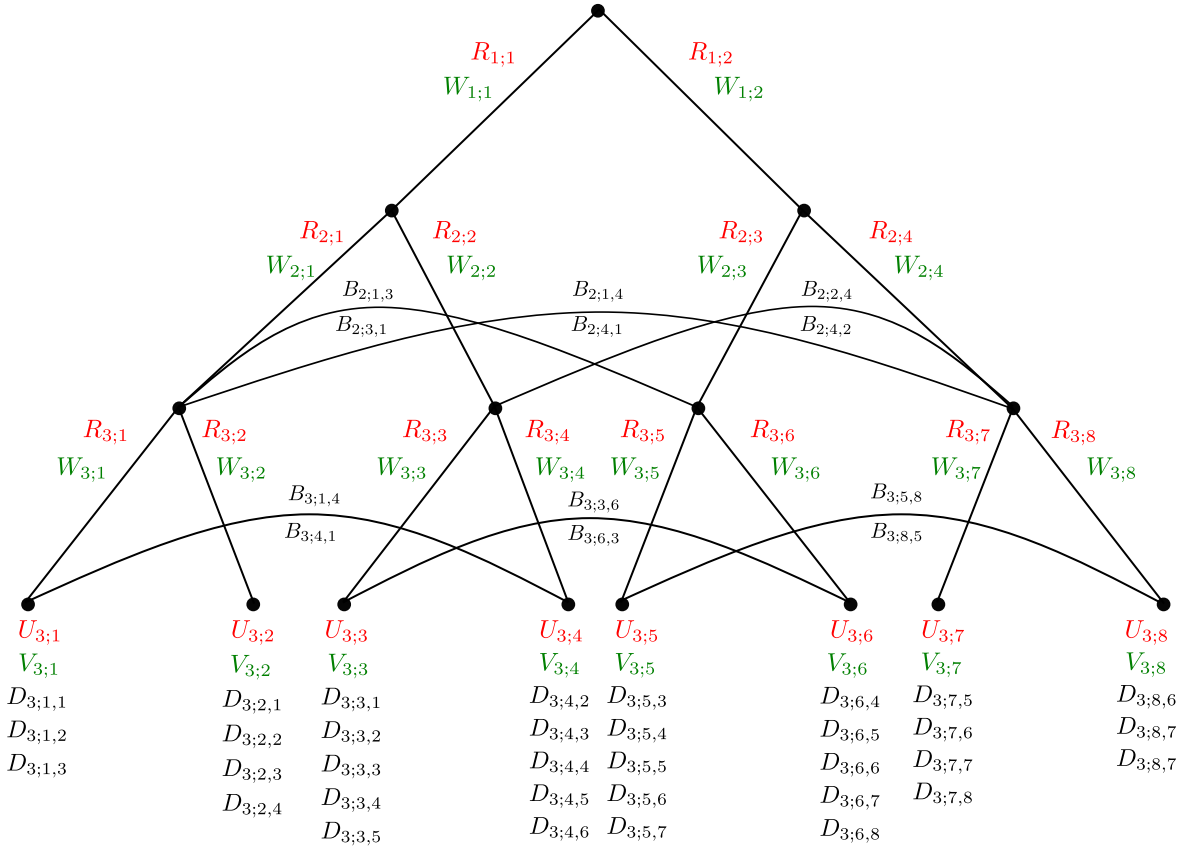


Figure 1.8: 3 Level 5pt FMM Tree

tion, at the leaf level, the edges representing the expansion coefficients skip yet one more cousin node as compared with the 3pt FMM tree. We will explain how this structure arises in detail as a result of the multiply of two 3pt FMM matrices in chapter 3

1.4 Iterative vs Direct Methods

Computing $A^{-1}b$ can be done using iterative methods such as GMRES, Conjugate-gradient, and BiCGSTAB, to name a few. Iterative methods such as these repeatedly apply the matrix A to a new right hand side, and each iteration must wait for the previous iteration to converge. In some cases, for example when the condition number,

$$\kappa_A = \|A\|\|A^{-1}\|, \tag{1.19}$$

is large, the convergence of these iterations may be slow and we will not end up with a fast algorithm. For GMRES for example, if the eigenvalues corresponding to the matrix are real, then residuals can be bounded by an upper bound that has a decay rate

$$\frac{\sqrt{\kappa_A} - 1}{\sqrt{\kappa_A} + 1}$$

[11, 12]. We propose that in cases such as these, we might do better if we compute the inverse directly.

1.5 HSS and FMM as Preconditioners

Preconditioning is the application of a transformation which lowers the condition number (1.19) of a given linear system. The system can then be more efficiently solved using iterative methods. There are many examples of preconditioners, such as Jacobi,

incomplete LU, and algebraic multigrid, to name a few. Both HSS and FMM are also widely used as preconditioners [13, 14]. If the linear system is derived from a "smooth" kernel (one that is not highly oscillatory), the off-diagonal blocks of the corresponding matrix will have low rank. In these cases, using HSS and FMM representations as preconditioners has the benefit that their cost scales well with the matrix size ($O(n)$) while also being relatively broadly applicable.

1.6 Permissions and Attributions

1. The content of chapter 2 and appendix A.1 is the result of a collaboration with Matt Hartman and Professor Shivkumar Chandrasekaran, and has previously appeared in the SIAM Journal on Matrix Analysis and Applications [15]. It is reproduced here with the permission of SIAM: `https://epubs.siam.org`.

Chapter 2

A Fast Memory Efficient Construction Algorithm for Hierarchically Semi-Separable Representations

2.1 HSS Representation

The storage of a matrix $A \in \mathbb{R}^{n \times n}$ in general will consume $O(n^2)$ memory, whereas the HSS representation can potentially require much less than $O(n^2)$ memory. Therefore, we are interested in construction algorithms which are efficient in both peak memory consumption as well as speed. Previous algorithms, [4] and [5], consume at most $O(n^2)$ memory, with a complexity of at most $O(n^2)$ flops. Other construction algorithms use randomized techniques [6], [7], [8], [2]. As no explicit statement is given in regard to peak memory consumption, we have inferred the memory complexity for these algorithms to be at most $O(n^2)$. $\mathcal{H}^2$ matrices are similar to HSS representations in hierarchical structure

and the reader is referred to [16] for more details on complexity counts.

Here we introduce a fast algorithm that computes the HSS representation of a matrix while using at most $O(n^{1.5})$ memory. The algorithm we present is very similar to [4] and [5], with key differences introduced to minimize peak memory usage. Specifically, the algorithm separates computations into 4 passes, the traversal of the HSS tree is modified and a new method of computation for the expansion coefficients is introduced. Note that merging the 4 passes, and using our newly introduced method for tree traversal [1], would yield a viable algorithm very similar to that given in [5], which would give a negligible savings in flops at the expense of the increase in peak memory by a factor of 4. It is possible to combine all 4 passes, while using our newly introduced method for tree traversal and retain $O(n^{1.5})$ memory, but since memory allocation is our focus here we did not chose to merge these passes. Also note that any rank revealing factorization can be substituted for the svd in our algorithm. Several such factorizations are rank revealing QR factorization [17], [18], [19], Interpolative Decomposition [20], [6] and rank revealing LU factorization [21], [22]. If a non-orthogonal decomposition is used then the formulas to compute the translation operators must be suitably adjusted as follows,

$$R_{k+1;2i-1} = U^{\dagger}_{k+1;2i-1}\,(U_{k;i})_1, \quad R_{k+1;2i} = U^{\dagger}_{k+1;2i}\,(U_{k;i})_2, \tag{2.1}$$

where,

$$U_{k;i} = \begin{pmatrix} (U_{k;i})_1 \\ (U_{k;i})_2 \end{pmatrix}. \tag{2.2}$$

The same adjustment is extended to the column translation operators.

[1] See Section 2.2.1, Non-Leaf Node Computations

2.2 HSS Construction Algorithm

In this section we present a recursive two phase construction algorithm. The first phase (Algorithm 1) is a deepest-first algorithm that computes the basis matrices at leaf nodes and translation operators at edges connecting parent to child nodes. In the second phase (Algorithm 2 and 3), the expansion coefficients are computed. In this algorithm, rather than asking the user to give the whole dense matrix as input, the user is asked only to provide a subroutine which generates subblocks of the matrix. Similar algorithm structure has been used by others [7] [5], and has been used since the first implementation of these methods [4]. In practice the algorithm needs a parameter in order to compress the Hankel blocks. One can provide a tolerance, ε, and/or an upper bound for the rank of the off-diagonal blocks, p. Notice if only an upper bound for the rank is provided, the error becomes uncontrollable. These issues are of tangential interest to our paper.

2.2.1 HSS Algorithm - Phase 1

Leaf Node Computations - Phase 1

In order to obtain the $U_{k;i}$'s at the leaf node we take an SVD of each row Hankel block, ${}_r\acute{H}_{k;i}$,

$$ {}_r\acute{H}_{k;i} = ({}_rU_{k;i})({}_r\Sigma_{k;i})({}_rQ^T_{k;i}) = \begin{pmatrix} U_{k;i} & * \end{pmatrix} \begin{pmatrix} \Sigma_{k;i} & 0 \\ 0 & * \end{pmatrix} \begin{pmatrix} Q^T_{k;i} \\ * \end{pmatrix}. \tag{2.3} $$

where, by definition, both ${}_rU_{k;i}$ and ${}_rQ_{k;i}$ are orthogonal matrices, and ${}_r\Sigma_{k;i}$ is a diagonal matrix. Likewise for $V_{k;i}$ at each leaf node $(k;i)$,

$$ {}_c\acute{H}_{k;i} = ({}_cP_{k;i})({}_c\Lambda_{k;i})({}_cV_{k;i}) = \begin{pmatrix} P_{k;i} & * \end{pmatrix} \begin{pmatrix} \Lambda_{k;i} & 0 \\ 0 & * \end{pmatrix} \begin{pmatrix} V_{k;i}^T \\ * \end{pmatrix}. \tag{2.4} $$

Only the columns of ${}_rU_{k;i}$ and ${}_cV_{k;i}$ which correspond to singular values above the chosen tolerance, ε, or rank, p are kept. The user can choose to give as input a tolerance, ε, such that all singular values smaller than ε are discarded. Alternately, the user can choose to give as input the rank, p, such that both $U_{k;i}$ and $V_{k;i}$ contain at most p columns. We denote TRUNC_SVD() as a function that returns an svd to this tolerance, ε, or alternatively this rank, p. $U_{k;i}$, and $V_{k;i}$ are stored at the leaf node $(k;i)$ in our HSS tree, and both $(\Sigma_{k;i})(Q_{k;i}^T)$ and $(P_{k;i})(\Lambda_{k;i})$ are returned to the parent function.

Non-Leaf Node Computations - Phase 1

In order to generate the translation operators $R_{k;2i-1}$, $R_{k;2i}$, $W_{k;2i-1}$ and $W_{k;2i}$, define

$$ {}_r\tilde{H}_{k;2i-1} = \Sigma_{k;2i-1}\, Q_{k;2i-1}^T, \tag{2.5} $$

$$ {}_c\tilde{H}_{k;2i-1} = P_{k;2i-1}\, \Lambda_{k;2i-1}, \tag{2.6} $$

$$ {}_r\tilde{H}_{k;2i} = \Sigma_{k;2i}\, Q_{k;2i}^T, \tag{2.7} $$

$$ {}_c\tilde{H}_{k;2i} = P_{k;2i}\, \Lambda_{k;2i}. \tag{2.8} $$

Vertically concatenate each pair of blocks ${}_r\tilde{H}_{k;2i-1}$ and ${}_r\tilde{H}_{k;2i}$, and remove the portion of the blocks which correspond to the columns that lie in the diagonal block $D_{k-1;i}$ as shown in Figure 2.1a. Likewise, horizontally concatenate each pair, ${}_c\tilde{H}_{k;2i-1}$ and ${}_c\tilde{H}_{k;2i}$ after removing the rows which correspond to the diagonal block $D_{k-1;i}$, as shown in Figure 2.1b. To remove these columns from ${}_r\tilde{H}_{k;2i-1}$ $({}_r\tilde{H}_{k;2i})$ we can instead remove them

from $Q_{k;2i-1}$ ($Q_{k;2i}$). Expand $Q_{k;i}$ to obtain

$$Q_{k;i}^T = \begin{pmatrix} Q_{k;i,1}^T & Q_{k;i,2}^T & \dots & Q_{k;i,2^k-1}^T \end{pmatrix}, \tag{2.9}$$

where each $Q_{k;i,j}^T$ is of size $m_{k;i} \times n_{k;j}$. Then define

$$\tilde{Q}_{k;i}^T = \begin{pmatrix} Q_{k;i,1}^T & \dots & Q_{k;i,i-1}^T & Q_{k;i,i+1}^T & \dots & Q_{k;i,2^k-1}^T \end{pmatrix}, \tag{2.10}$$

where we have excluded the block $Q_{k;i,i}^T$ from $Q_{k;i}^T$. This process can be repeated similarly for the column blocks, and the compressed row and column Hankel blocks at node $(k-1;i)$ are given by

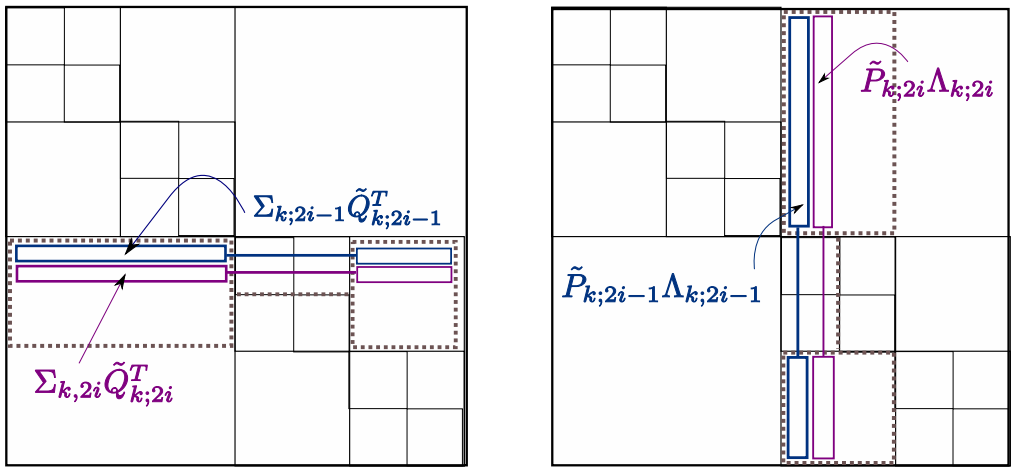
$${}_rH_{k-1;i} = \begin{pmatrix} \Sigma_{k;2i-1}\, \tilde{Q}_{k;2i-1}^T \\ \Sigma_{k,2i}\, \tilde{Q}_{k;2i}^T \end{pmatrix}, \tag{2.11}$$

$${}_cH_{k-1;i} = \begin{pmatrix} \tilde{P}_{k;2i-1}\, \Lambda_{k;2i-1} & \tilde{P}_{k;2i}\, \Lambda_{k;2i} \end{pmatrix}. \tag{2.12}$$

A visual representation of (2.11) and (2.12) are shown in Figures 2.1a and 2.1b. The corresponding part of the original Hankel blocks, ${}_r\acute{H}_{k-1;i}$, and ${}_c\acute{H}_{k-1;i}$, with definitions given in (1.7), are outlined in brown dashed lines.

We can then determine the translation operators, $R_{k;2i-1}$, and $R_{k;2i}$, from ${}_rH_{k-1;i}$, by performing a truncated SVD (Algorithm 1, line 16). Likewise, we can determine the translation operators, $W_{k;2i-1}$, and $W_{k;2i}$, from ${}_cH_{k-1;i}$. We then proceed iteratively in order to obtain all translation operators at each edge from parent to child in our HSS structure. At this stage we have stored $D_{k;i}$ and we have computed and stored every basis matrix, $U_{k;i}$ and $V_{k;i}$, at the leaf nodes. At each edge connecting child node $(k;i)$ to parent node we have computed and stored every translation operator, $R_{k;i}$ and $W_{k;i}$.

Algorithm 1 summarizes the computation of all $U_{k;i}$ at leaf nodes and $R_{k;i}$ at edges



(a) ${}_rH_{k-1;i}$, with $k = 3$, $i = 3$ (b) ${}_cH_{k-1;i}$, with $k = 3$, $i = 3$

Figure 2.1: Compressed Row and Column Hankel Blocks

connecting child node $(k; i)$ to parent node[2]. We can compute $V_{k;i}$ and $W_{k;i}$ similarly by applying Algorithm 1 to A^T. In this algorithm nodes are visited in a **deepest first order**, by which we mean that at each parent node the algorithm will visit the child node which is the root of the subtree with the greatest depth. If the depth of both subtrees are equal, then the left child is visited first.

2.2.2 HSS Algorithm - Phase 2

In the second phase of our algorithm we compute all expansion coefficients, $B_{k;i,j}$, via matrix multiplication. Memory consumption for this computation will maximally require $O(p^2 n)$ in the worst case, (as compared with $O(p^2 \log n)$ for a symmetric tree).

[2]In algorithms 1, 2 and 3 the translation operators and the expansion coefficients are computed at the parent node to which they belong and so it can be more convenient to associate them with these nodes as opposed to the edges at which they occur.

Algorithm 1 Pass 1U - Memory Efficient HSS Algorithm

Require: $tree = [root,\ treeL,\ treeR]$. ▷ $tree$ is a tuple, containing root node, and right and left subtree

1: **function** HSS_Basis($tree$)
2: **if** $tree$ is a leaf node **then**
3: $[U_{k;i}, \Sigma_{k;i}, Q_{k;i}^T] = \text{TRUNC\_SVD}([A_{k;i,1}\ \ A_{k;i,2}\ \ \dots\ \ A_{k;i,i-1}\ \ A_{k;i,i+1}\ \ \dots\ \ A_{k;i,end}])$
4: **return** $\Sigma_{k;i}\tilde{Q}_{k;i}^T$ ▷ $\tilde{Q}_{k;i}$ is defined as previously stated in equation (2.10)
5: **else** ▷ $tree$ is **not** a leaf node
6: $[\_,\ treeL,\ treeR] = tree$
7: **if** DEPTH($treeL$) $\geq$ DEPTH($treeR$) **then**
8: ${}_rH_{k+1;2i-1} = \text{HSS\_Basis}(treeL)$
9: ${}_rH_{k+1;2i} = \text{HSS\_Basis}(treeR)$
10: **else**
11: ${}_rH_{k+1;2i} = \text{HSS\_Basis}(treeR)$
12: ${}_rH_{k+1;2i-1} = \text{HSS\_Basis}(treeL)$
13: **end if**
14: ${}_rH_{k;i} = \begin{pmatrix} {}_rH_{k+1;2i-1} \\ {}_rH_{k+1;2i} \end{pmatrix}$
15: ${}_rH_{k+1;2i-1} = ();\qquad {}_rH_{k+1;2i} = ()$
16: $[\begin{pmatrix} R_{k+1;2i-1} \\ R_{k+1;2i} \end{pmatrix},\ \Sigma_{k;i},\ X_{k;i}] = \text{TRUNC\_SVD}({}_rH_{k;i})$
17: **return** $\Sigma_{k;i}\,\tilde{X}_{k;i}^T$ ▷ $\tilde{X}_{k;i}$ is defined as previously stated in equation (2.10)
18: **end if**
19: **end function**

Leaf Node Computations - Phase 2

For (k_1, i), (k_2, j) leaf nodes, let us define

$$B_{k_1;i,k_2,j} = U^T_{k_1,i} A_{k_1;i,k_2,j} V_{k_2;j} \tag{2.13}$$

Then, for (k_1, i), a leaf node, and (k_1, j) is not, define

$$\begin{aligned} B_{k_1;i,k_2;j} &= B_{k_1;i,k_2+1;2j-1} W_{k_2+1;2j-1} \\ &+ B_{k_1;i,k_2+1;2j} W_{k_2+1;2j}, \end{aligned} \tag{2.14}$$

Now for (k_1, i), not a leaf node, and (k_2, j) is a leaf node, define

$$\begin{aligned} B_{k_1;i,k_2;j} &= R^T_{k_1+1;2i-1} B_{k_1+1;2i-1,k_2,j} \\ &+ R^T_{k_1+1;2i} B_{k_1+1;2i,k_2,j}. \end{aligned} \tag{2.15}$$

Non-Leaf Node Computations - Phase 2

For both (k_1, i), and (k_2, j) not leaf nodes, we will have $k_1 = k_2$, and thus we can write

$$B_{k;i,j.} = B_{k_1;i,k_2;j}, \tag{2.16}$$

where $k = k_1 = k_2$. Let us then define

$$\begin{aligned} B_{k;i,j} &= R^T_{k+1;2i-1} B_{k+1;2i-1,2j-1} W_{k+1;2j-1} \\ &+ R^T_{k+1;2i-1} B_{k+1;2i-1,2j} W_{k+1;2j} \\ &+ R^T_{k+1;2i} B_{k+1;2i,2j-1} W_{k+1;2j-1} \\ &+ R^T_{k+1;2i} B_{k+1;2i,2j} W_{k+1;2j}, \end{aligned} \tag{2.17}$$

Thus we have defined a set of recursions with which we can compute any $B_{k;i,j}$ in our HSS structure.

To compute all $B_{k_1;i,k_2;j}$, we present Algorithm 2 and 3. This algorithm specifically computes the expansion coefficients, $B_{k;i-1,i}$. The expansion coefficient $B_{k;i,i-1}$ can be computed analogously by simply interchanging the pair of indices $(k_1; i)$ with $(k_2; j)$ in Algorithms 2 and 3. Note that in the following algorithm a node is a tuple which consists of data, and left and right subtree, though, the details of where we store the data are not relevant here since we access them by index notation. These are implementation dependent details and for more information the reader can refer to the Matlab code that is published on the Scientific Computing Group website [23]. Also, note that we use the notation $B_{k_1;i,k_2;j} = ()$ to mean that the variable $B_{k_1;i,k_2;j}$ is no longer needed and can be explicitly cleared from memory.

2.3 Memory Consumption

In this section we will compute the peak workspace consumption for Algorithms 1, 2, and 3. The HSS tree itself will consume at most $O(np^2)$ memory. In Phase 1, the main workspace consumption are the compressed Hankel blocks, (2.11), and (2.12). In Phase 2 it is the matrices $B_{k_1;i,k_2;j}$. Let $p_0 = \max_{(k;i) \text{ leaf node}} n_{k;i}$. The assumption here is that p is small compared to n. In particular we will establish,

Theorem 1 *Algorithms 1, 2 and 3 have a memory complexity of at most* $O\left(p^{0.5}n^{1.5}\right)$.

2.3.1 Memory Consumption - Phase 1

In Phase 1 (Algorithm 1), we visit nodes in a deepest first ordering, and notice that what further complicates our memory complexity calculation is the recursive nature of

Algorithm 2 Pass 2BU - Computation of Expansion Coefficients ($B_{k;i-1,i}$) Corresponding to Diagonal Blocks

Require: $tree$ is not a leaf node. $tree = [root, treeL, treeR]$. ▷ $tree$ is a tuple, containing root node, and right and left subtree

1: **function** B_DIAG(tree)
2: $[-, treeL, treeR] = tree$ ▷ $\exists\ (k1; i)$ s.t. it is the numbering for the root node of $treeL$.
3: ▷ $\exists\ (k2; j)$ s.t. it is the numbering for the root node of $treeR$.
4: **if** $treeL$ is a leaf node and $treeR$ is a leaf node **then**
5: $B_{k_1;i,k_2;j} = U^T_{k_1;i} A_{k_1;i,k_2;j} V_{k_2;j}$
6: **else if** $treeL$ is **not** a leaf node and $treeR$ is **not** a leaf node **then**
7: $B_{k_1;i,k_2;j} =$ B_OFFDIAG($treeL$, $treeR$)
8: B_DIAG($treeL$)
9: B_DIAG($treeR$)
10: **else if** $treeL$ is a leaf node and $treeR$ is **not** a leaf node **then**
11: $[-, treeRL\ \ treeRR] = treeR$
12: $B_{k_1;i,k_2+1;2j-1} =$ B_OFFDIAG($treeL$, $treeRL$)
13: $B_{k_1;i,k_2+1;2j} =$ B_OFFDIAG($treeL$, $treeRR$)
14: $B_{k_1;i,k_2;j} = B_{k_1;i,k_2+1;2j-1} W_{k_2+1;2j-1} + B_{k_1;i,k_2+1;2j} W_{k_2+1;2j}$
15: $B_{k_1;i,k_2+1;2j-1} = ();\qquad B_{k_1;i,k_2+1;2j} = ()$
16: B_DIAG($treeR$)
17: **else if** $treeL$ is **not** a leaf node and $treeR$ is a leaf node **then**
18: $[-, treeLL, treeLR] = treeL$
19: $B_{k_1+1;2i-1,k_2,j} =$ B_OFFDIAG($treeLL$, $treeR$)
20: $B_{k_1+1;2i,k_2,j} =$ B_OFFDIAG($treeLR$, $treeR$)
21: $B_{k_1;i,k_2;j} = R^T_{k_1+1;2i-1} B_{k_1+1;2i-1,k_2;j} + R^T_{k_1+1;2i} B_{k_1+1;2i,k_2;j}$
22: $B_{k_1+1;2i-1,k_2;j} = ();\qquad B_{k_1+1;2i,k_2;j} = ()$
23: B_DIAG($treeL$)
24: **end if**
25: **end function**

Algorithm 3 Pass 2BU - Computation of Expansion Coefficients ($B_{k;i-1,i}$) Corresponding to Off-Diagonal Blocks

Require: $tree = [root,\ treeL,\ treeR]$. ▷ $tree$ is a tuple, containing root node, and right and left subtree

function B_OffDiag(treeL,treeR)
  $[-,\ treeLL,\ treeLR] = treeL$
  $[-,\ treeRL,\ treeRR] = treeR$
  if $treeL$ is a leaf node and $treeR$ is a leaf node **then**
    $B_{k_1;i,k_2;j} = U^T_{k_1;i} A_{k_1;i,k_2;j} V_{k_2;j}$
    return $B_{k_1;i,k_2;j}$
  else if $treeL$ is **not** a leaf node and $treeR$ is **not** a leaf node **then**
    $B_{k_1+1;2i-1,k_2+1;2j-1}$ = B_Offdiag($treeLL$,$treeRL$)
    $B_{k_1+1;2i-1,k_2+1;2j-}$ = B_Offdiag($treeLL$,$treeRR$)
    $B_{k_1+1;2i,k_2+1;2j-1}$ = B_Offdiag($treeLR$,$treeRL$)
    $B_{k_1+1;2i,k_2+1;2j}$ = B_Offdiag($treeLR$,$treeRR$)
    $B_{k_1;i,k_2;j} = R^T_{k_1+1;2i-1} B_{k_1+1;2i-1,k_2+1;2j-1} W_{k_2+1;2j-1}$
      $+ R^T_{k_1+1;2i-1} B_{k_1+1;2i-1,k_2+1;2j} W_{k_2+1;2j}$
      $+ R^T_{k_1+1;2i} B_{k_1+1;2i,k_2+1;2j-1} W_{k_2+1;2j-1}$
      $+ R^T_{k_1+1;2i} B_{k_1+1;2i,k_2+1;2j} W_{k_2+1;2j}$
    $B_{k_1+1;2i-1,k_2+1;2j-1} = ();$ $B_{k_1+1;2i-1,k_2+1;2j} = ();$
    $B_{k_1+1;2i,k_2+1;2j-1} = ();$ $B_{k_1+1;2i,k_2+1;2j} = ()$
    return $B_{k_1;i,k_2;j}$
  else if $treeL$ is a leaf node and $treeR$ is **not** a leaf node **then**
    $B_{k_1;i,k_2+1;2j-1}$ = B_Offdiag($treeL$,$treeRL$)
    $B_{k_1;i,k_2+1;2j}$ = B_Offdiag($treeL$,$treeRR$)
    $B_{k_1;i,k_2;j} = B_{k_1;i,k_2+1;2j-1} W_{k_2+1;2j-1} + B_{k_1;i,k_2+1;2j} W_{k_2+1;2j}$
    $B_{k_1;i,k_2+1;2j-1} = ();$ $B_{k_1;i,k_2+1;2j} = ()$
    return $B_{k_1;i,k_2;j}$
  else if $treeL$ is **not** a leaf node and $treeR$ is a leaf node **then**
    $B_{k_1+1;2i-1,k_2;j}$ = B_Offdiag($treeLL$,$treeR$)
    $B_{k_1+1;2i,k_2;j}$ = B_Offdiag($treeLR$,$treeR$)
    $B_{k_1;i,k_2;j} = R^T_{k_1+1;2i-1} B_{k_1+1;2i-1,k_2;j} + R^T_{k_1+1;2i} B_{k_1+1;2i,k_2;j}$
    $B_{k_1+1;2i-1,k_2;j} = ();$ $B_{k_1+1;2i,k_2;j} = ()$
    return $B_{k_1;i,k_2;j}$
  end if
end function

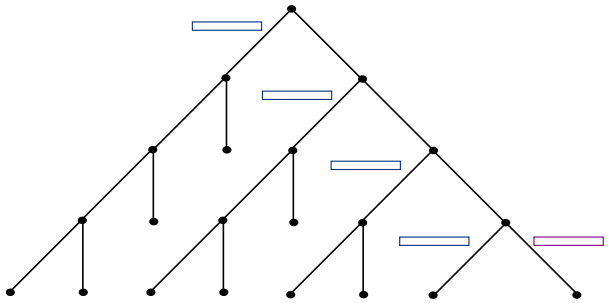


Figure 2.2: Example of a tree with worst-case memory block cardinality of 5

our algorithm. The workspace allocated in one call to Algorithm 1 is added to any further recursive calls to Algorithm 1. Therefore, we have to consider the depth of the stack of these recursive calls, which is determined by the HSS tree. Therefore, for a given number of nodes, we must find the HSS tree for which peak memory complexity will be maximum. Without loss of generality, any HSS tree can be re-ordered such that the depth of the left subtree is always larger or equal to the depth of the right subtree, and therefore, in this section we only consider trees which have this property. Notice that we make memory allocations at lines 8 and 9, and lines 11 and 12 of Algorithm 1. Since we are only considering trees which have the property that at each parent node, left subtrees have depth which is larger or equal to that of the right, we will focus on calls to line 8 and line 9 only. When a call is made to line 9 a block of size $n \times p$ is left in memory from line 8. After returning from a call to line 9, line 14 is executed the block is cleared from memory at line 15. Then, for a given number of nodes, we want to find the HSS tree which will result in the maximum number of nested calls to line 9. It is easier to visualize memory consumption by tracking the memory allocation as one moves along the HSS tree, as is shown in Figure 2.2 . Proposition 1 claims that for a given number of nodes, such a tree satisfies properties $(P1)$ and $(P2)$, which are defined in the following section. An example of trees which satisfy these properties is given in Figure 2.3.

Definitions

Denote $V(G)$ as the set of nodes for graph G.

For a set S, $|S|$ is the cardinality of S.

A regular binary tree $T \in \mathcal{T}_{reg}$ has the **P1** property if for each node in T, the depth of the left subtree is greater than or equal to the depth of the right subtree (Figure 2.3). We denote the set of trees with the P1 property as $\mathcal{T}_{P1}$.

For a regular binary tree $T \in \mathcal{T}_{reg}$, a **root-leaf path** is a sequence of nodes $(u_0, ..., u_k)$ such that u_0 is the root of T, each subsequent node is a child of the preceding node, and u_k is a leaf node. The set of root-leaf paths for T is denoted $\mathcal{p}_{rl}(T)$. Figure 2.3a shows a regular binary tree with circled nodes denoting one such root-leaf path.

For each $T \in \mathcal{T}_{P1}$ and root-leaf path $\rho \in \mathcal{p}_{rl}(T)$, the **memory block cardinality** $b_m : \mathcal{p}_{rl}(T) \to \mathbb{N}$ maps ρ to a natural number using the following rule: $b_m(\rho)$ is equal to the number nodes in ρ whose right children are also in ρ, plus one.

For a tree $T \in \mathcal{T}_{P1}$, the **worst-case memory block cardinality**, denoted $b_{wc}(T)$, is defined as $\max_{\rho \in p_{rl}(T)} b_m(p)$.

For each $T \in \mathcal{T}_{P1}$, the **primary right branch** is the subgraph of T consisting of the root node and subsequent right children. All primary right branches are line graphs. The path along the primary right branch is denoted as $p_{pri}(T)$ (Figure 2.3a).

A regular binary tree $T \in \mathcal{T}_{P1}$ has the property **P2** if $b_{wc}(T) = b_m(p_{pri}(T))$. We denote the set of trees with the P2 property as $\mathcal{T}_{P2}$.

Results and Proofs

Our strategy is to determine the fewest number of nodes that a P1 tree can have while generating a fixed number of memory blocks. This results in the following proposition:

Proposition 1 *For all* $d \in \mathbb{N}$,

$$\min_{T \in \mathcal{T}_{P1}, b_{wc}(T)=d+1} |V(T)| = d^2 + d + 1. \tag{2.18}$$

To show this, we use the following lemma, which tells us that the class of trees that we need to search in consists of trees with the P2 property:

Lemma 1 *For all* $T_1 \in \mathcal{T}_{P1} \setminus \mathcal{T}_{P2}$, *there exists* $T_2 \in \mathcal{T}_{P2}$ *such that* $b_{wc}(T_1) = b_{wc}(T_2)$ *and* $|V(T_2)| < |V(T_1)|$.

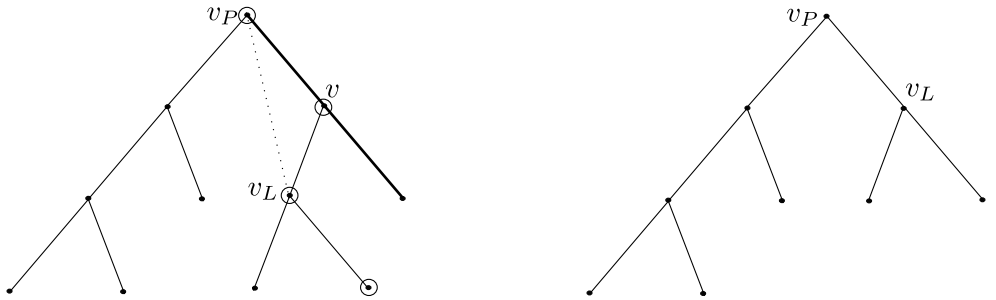
Proof: We begin with the claim that $T_1 \in \mathcal{T}_{P1} \setminus \mathcal{T}_{P2}$. Then there exists $\rho \in \mathcal{p}_{rl}(T_1)$ such that $b_m(\rho) > b_m(p_{pri}(T_1))$. Let $v \in \rho$ be the earliest node in ρ such that the subsequent node is a left child. We create a new tree $T_2^{(1)}$ by removing v and its right subtree from T_1, and creating an edge between the parent of v and the left child of v (if v is root, then we skip the latter). Now the path $\hat{\rho} := \rho \setminus \{v\}$ is a root-leaf path for $T_2^{(1)}$ and satisfies $b_m(\hat{\rho}) = b_m(\rho)$. Given that T_1 has a finite number of nodes, and the above procedure which obtains $T_2^{(1)}$ from T_1 removes at least two nodes, we can proceed inductively by repetition of this procedure on $T_2^{(1)}$. And by induction, for some $k > 0$ we obtain $T_2^{(k)}$, that satisfies $T_2^{(k)} \in \mathcal{T}_{P2}$, such that $b_{wc}(T_1) = b_{wc}(T_2^{(k)})$ and $|V(T_2^{(k)})| < |V(T_1)|$. Finally, define, $T_2 := T_2^{(k)}$ Figures 2.3a and 2.3b give examples of such trees T_1 and T_2, respectively. ■

We are now ready to prove Proposition 1.

Proof: We begin with the claim that for all $d \in \mathbb{N}$,

$$\min_{T \in \mathcal{T}_{P1}, b_{wc}(T)=d+1} |V(T)| \geq d^2 + d + 1. \tag{2.19}$$

Suppose otherwise. Then there exists $d \in \mathbb{N}$ and $T \in \underset{T \in \mathcal{T}_{P1}, b_{wc}(T)=d+1}{\arg\min} |V(T)|$ such that $|V(T)| < d^2 + d + 1$. By Lemma 1, $T \in \mathcal{T}_{P2}$, therefore $b_{wc}(T) = b_m(p_{pri}(T)) = d + 1$.



(a) T_1 decorated with dotted line denoting the newly created root-leaf path which gives rise to T_2. Here the primary right branch is the subgraph shown in bold, v_p is the parent node of v, and v_L is the left child of node v. Circled nodes denote a single root-leaf path

(b) T_2

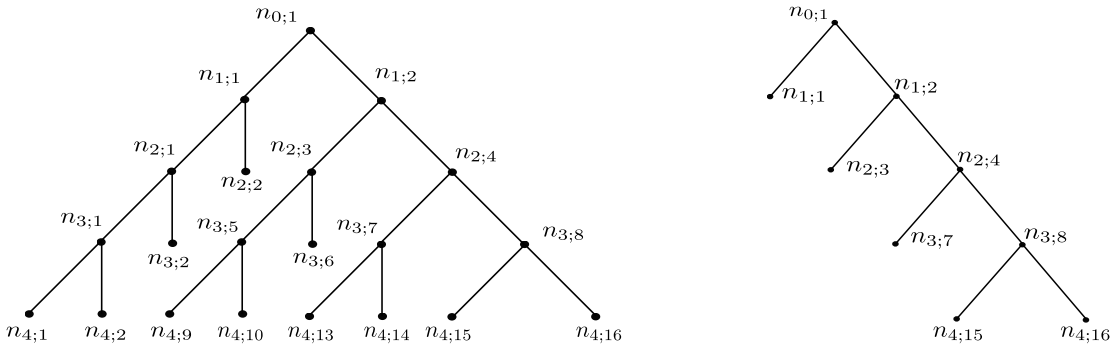
Figure 2.3: Example T_1 and T_2 for Lemma 1, with $b_{wc}(T_1) = b_{wc}(T_2) = 3$

Then, using the P1 property and the definition of b_m, we have that the primary branch of T has depth d.

Due to the P1 property, the left subtree of the root node must have have a depth of at least d. Such a subtree must contain at least $2d-1$ nodes. Applying the same logic to the right child of the root node, the next left subtree must contain at least $2(d-1)-1$ nodes. We repeat this for the ith right offspring of the root node, obtaining left subtrees consisting of at least $2(d-i)-1$ nodes for $i \in \{0, ..., d-1\}$. Including the primary right branch, which consists of $d+1$ nodes, we have

$$\begin{aligned} |V(T)| &\geq d+1+\sum_{k=1}^{d}(2(d-k)+1) \\ &= 2d^2+2d+1-2\sum_{k=1}^{d}k \\ &= d^2+d+1, \end{aligned}$$

which contradicts $|V(T)| < d^2+d+1$. Thus (2.19) is satisfied for all $d \in \mathbb{N}$. Furthermore, by choosing the fewest number of nodes at each step of the construction of T, we obtain a tree that satisfies $T \in \mathcal{T}_{P1}$, $b_{wc}(T) = d+1$, and $|V(T)| = d^2+d+1$. Thus we have the



(a) Example Worst Case HSS Tree for Memory Consumption

(b) Maximal Depth Partition Tree

Figure 2.4: Example Partition Trees

result. ■

Note that for any binary tree, the number of non-leaf nodes, N_N, will be one less than the number of leaf nodes, N_L, i.e. $N_N = N_L - 1$. Then, since $N_L = n/p_0$, we have

$$|V(T)| = N = \frac{2n}{p_0} - 1. \tag{2.20}$$

Proposition 1 implies that the worst case number of memory blocks is $O\left(\sqrt{N}\right)$. Since a single memory block consumes approximately $O(pn)$ memory, using (2.20), we have that the worst case memory usage is $O\left(p^{0.5}n^{1.5}\right)$. An example of a tree that generates the worst case number of memory blocks is given in Figure 2.4a. In the case of such a worst case memory tree, using a more accurate count we can show the maximum memory consumption is actually $n^{1.5}p^{0.5} - n$. However, for a complete partition tree, Phase 1 will take at most $O(pn \log n)$ memory. Further, for a regular binary tree of maximum depth (see for example Figure 2.4b) memory consumption is only $O(np)$.

2.3.2 Memory Consumption - Phase 2

In Phase 2 of our algorithm, at most 3 $p \times p$ blocks are stored in memory during each recursive call, and this number can easily be cut down to one $p \times p$ block, though we present the algorithm as is for clarity. Notice that each of the $p \times p$ blocks which are returned on lines 8 through 11 are cleared from memory on lines 13 and 14. This equivalent statement is true for each of the if statement cases in Algorithms 2 and 3. This can clearly be seen by looking at lines 20 and 26 in Algorithm 3, as well as line 15 and 22 in Algorithm 2. Therefore, given a tree of maximal depth, $\frac{n}{p}$ (Figure 2.4b), peak memory consumption for Phase 2 is np.

2.3.3 Total Memory Consumption

If a deepest first algorithm is not used the memory complexity can become $O(n^2)$, whereas the worst case deepest first search path memory complexity is $O\left(p^{0.5}n^{1.5}\right)$. For a complete partition tree however, the memory complexity is $O(pn \log n)$. As shown in [4], basis matrices and translation operators can be generated for smooth matrices using Chebyshev polynomials. If the basis matrices are computed a priori, it is then only necessary to execute Algorithms 2 and 3 to compute the expansion coefficients at a cost of at most $O(np)$. In the case of the complete partition tree this cost is reduced to $O(p^2 \log n)$.

2.4 Flop Count

Let us compute the total number of flops for our algorithm. To begin, we see that for any binary tree with a fixed number of nodes, the number of non-leaf nodes, N_N, will be one less than the number of leaf nodes, N_L. In other words, $N_N = N_L - 1$. We also

have that $N_L = n/p$. For Phase 1, we can see that at a leaf node, we generate 2 Hankel blocks, and take 2 SVD's at a cost of approximately $2n^2p + 4n^2$. For a non-leaf node in Phase 1, we take 2 SVD's and perform 2 matrix multiplies at a cost of $(2np^2 + 2np)$. So the total flop count for Phase 1 will be $4n^2p + 6n^2$.

For Phase 2, in which we compute all $B_{k;i,j}$, the number of flops will be maximum for an even tree. The computation of each $B_{k;i,j}$ in the HSS tree requires the multiplication of matrices of size $p \times p$ (Algorithm 3 line 5 and 12), and costs at most $8p^3$. The computation of $B_{1;1,2}$ ($B_{1;2,1}$) begins at the bottom level of the recursion, Algorithm 3 line 5. This line will be executed on $(\frac{n}{2p} \times \frac{n}{2p})$ blocks, at a cost of $2p^3$ flops each. The calculation of $B_{1;1,2}$ ($B_{1;2,1}$) continues at line 12 of Algorithm 3, which will be executed on $\frac{1}{4}(\frac{n}{2p} \times \frac{n}{2p})$ blocks, at a cost of $8p^3$ each. The next stage of calculation for $B_{1;1,2}$ ($B_{1;2,1}$), again on line 12 of Algorithm 2, will be executed on $\frac{1}{16}(\frac{n}{2p} \times \frac{n}{2p})$ blocks, and so on. Following this recursion, the calculation of $B_{1;1,2}$ (or $B_{1;2,1}$) has a cost of n^2p flops. At level 1 in the HSS tree, only $B_{1;1,2}$, and $B_{1;2,1}$ must be computed, and so the flop count for both is $2n^2p$. In a similar fashion, we compute the number of flops required to compute $B_{k;i,j}$ at level 2 in our HSS tree. At this level there are 4 $B_{2;i,j}$, at a cost of n^2p flops. At level 3, we will compute 8 $B_{3;i,j}$ at a cost of $\frac{1}{2}n^2p$. Summing all of these computations the cost of all $B_{k;i,j}$, will be at most $4n^2p$ flops.

Thus for Phase 1 and Phase 2 combined, our algorithm has a complexity of $O(n^2p)$.

2.5 Numerical Experiments

In this section we will describe some experiments which demonstrate the speed and peak memory consumption of our algorithm. Experiments were carried out in Julia on a machine with a 2.5GHz Intel Core i5-3210M processor running Ubuntu with 8GB of RAM.

For both the CPU run-time and the peak memory consumption simulations, the $n \times n$ matrix A was chosen according to the formula $A_{i,j} = \sqrt{|x_i^n - x_j^n|}$, where $x_i^n = i/n$, for $i = 0, 1, \ldots, n$. The rank, p, was fixed across all tests. In the case of the HSS tree which generates the worst case peak memory for a given number of nodes, the structure of the tree was determined by calculating the number of nodes, N, in the HSS tree for a given matrix with dimension, n, via (2.20). We construct the tree for the given dimension, n, ranging from 256 to 1048576, according to the proof of Proposition 1. In tables 2.1 and 2.2 each leaf node has a partition size of no more than $2(3p) - 1$, and no less than $3p$. The measurement of peak memory given in table 2.2 was obtained from a counter in the code which simply kept track of memory allocations and de-allocations. The numbers listed are the maximum number of floating point allocations in memory during run-time. Note that we only keep track of floating point array allocations, not lower order integer allocations. CPU run-times and peak memory measurements are reported in Table 2.1 and 2.2.

n	time(s)	
	$p = 3$	$p = 6$
256	0.89	0.91
512	0.98	1.00
1024	1.26	1.22
2048	2.24	2.22
4096	6.11	6.35
8192	21.09	19.56
16384	71.18	78.24
32768	314.07	380.84
65536	1553	1559
131072	5768	5745
262144	23190	23523
524288	91485	89974
1048576	365345	381836

Table 2.1: CPU run-times in seconds for the worst case memory HSS tree with $p = 3$ (where p is the rank of the off diagonal blocks), with minimum partition at leaf nodes of $3p$.

n	Peak Floating Point Allocations	
	$p = 3$	$p = 6$
256	17376	28332
512	42240	69492
1024	106626	168948
2048	272502	426492
4096	712518	1089996
8192	1893606	2850060
16384	5067546	7574412
32768	13836228	20270172
65536	38172840	55344900
131072	105846120	152691348
262144	294842598	423384468
524288	825534894	1179370380
1048576	2319016548	3302139564

Table 2.2: Peak memory consumption in units of floating point allocations for the worst case memory HSS tree with $p = 3$ (where p is the rank of the off diagonal blocks), with minimum partition at leaf nodes of $3p$.

Chapter 3

Fast Matrix-Matrix Multiply for the Fast Multipole Method

3.1 Motivation and Examples

In chapter 2 we have covered a memory efficient algorithm for HSS. As mentioned previously, HSS assumes off-diagonal blocks have low rank, however, in practice there are examples where this is not actually the case. The 2D Laplacian is one such example. As illustrated in Figure 3.1, the off-diagonal blocks of this matrix have rank $O(\sqrt{n})$. Using the HSS representation in this case would require one to compress these off-diagonal blocks or store this extra structure. Compressing these off-diagonal blocks would result in a loss of accuracy since the blocks would be compressed from rank $O(\sqrt{n})$ to $O(p)$, while storing this extra structure would lead to dramatically increased memory and CPU usage. Clearly by using the 2D FMM representation we could instead preserve some of this structure. In practice we are interested in inversion, however, since the structure of the FMM inverse remains mysterious, we propose that we should first understand the FMM matrix-matrix multiply. As motivation we present several examples where the need

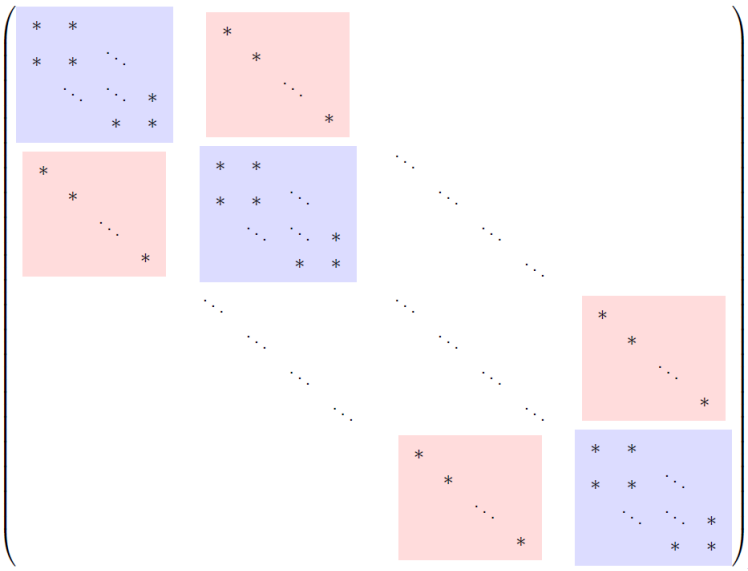


Figure 3.1: Structure of the 2D Laplacian. Off-diagonal blocks shown in red have a rank of $\sqrt{n}$

for the matrix-matrix multiply arises, as well as one special case where we can use the matrix-matrix multiply to compute the inverse itself.

3.1.1 Schur Compliment

To compute the inverse we will have to compute the Schur Complement (shown in red in Equation 3.1). The Schur Complement arises as the result of performing Gaussian elimination, and so either explicitly or implicitly it will be computed. And so of course you see that here we have need of matrix-matrix multiply.

$$\begin{pmatrix} A & B \\ C & D \end{pmatrix} = \begin{pmatrix} \mathbf{I} & 0 \\ CA^{-1} & \mathbf{I} \end{pmatrix} \begin{pmatrix} A & B \\ 0 & (D - CA^{-1}B) \end{pmatrix} \tag{3.1}$$

3.1.2 Matrix Functions

Matrix functions can often be well approximated by polynomials. For example in Equation 3.2 we give the formula for the approximation of the exponential of a matrix A.

$$f(A) = \exp(A) = \sum_{k=0}^{\infty} \frac{A^k}{k!} \tag{3.2}$$

This example makes clear that if we have access to an algorithm which computes the matrix-matrix multiply, we would then directly be able to compute an approximation to $f(A)$. Further, if we can write $f(A)$ as a short polynomial then it will have good FMM structure. We will qualify this statement further in section 3.2. As an example, there has been some recent interest in this in regard to exponential integrators[24].

3.1.3 Computation of Eigenvalues and Eigenvectors

In order to compute the eigenvalues and eigenvectors of a matrix A, in practice, matrix-matrix multiplies must be carried out. The QR algorithm for computing approximations to eigenvalues and eigenvectors, for example, requires that we iteratively perform matrix-matrix multiplies[25].

3.1.4 Special Case Where FMM Matrix-Matrix Multiply Can Be Used to Compute the Inverse

If the eigenvalues of the matrix A have some special properties, in other words, if $\lambda(A) < 1$, then we can write

$$f(A) = (I - A)^{-1} = \sum_{k=0}^{\infty} A^k. \tag{3.3}$$

Here we can again see that if we have access to an algorithm which computes the matrix-matrix multiply, we can directly compute an approximation to this inverse. If we can write $(I - A)^{-1}$ as a short polynomial, we will have good FMM structure for this inverse as well (see section 3.2).

3.2 FMM 3pt and 5pt Structure

In this section we provide some illustrations of the HSS, FMM 3pt and FMM 5pt structures. We begin by first looking at the low rank structure of the HSS representation. In Figure 3.2 we show an example of a 2 level representation in which the off-diagonal blocks are low rank. The image on the left of Figure 3.2 shows an illustration of the low rank structure, while the image on the right shows the low rank structure explicitly.

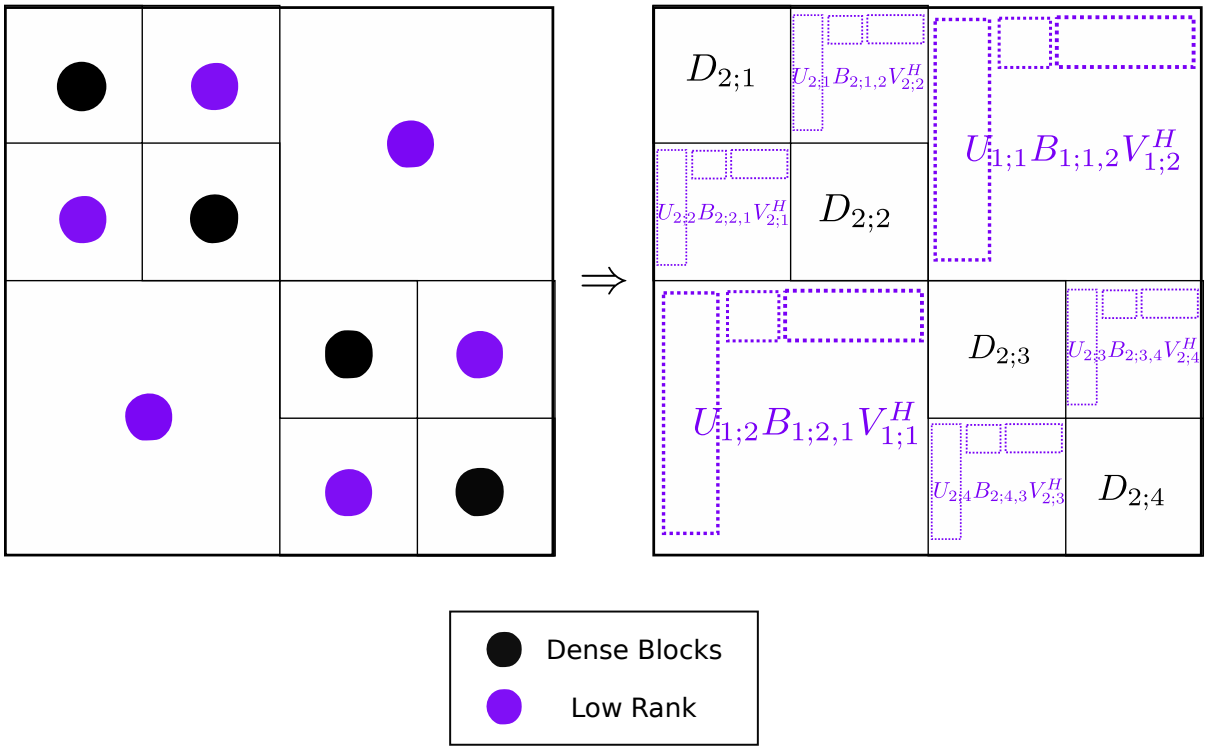


Figure 3.2: Example 2 level HSS Representation

In contrast, in Figure 3.3 we show an example of a 2 level FMM representation. Any block that touches the diagonal is recursively partitioned, except at a leaf node where it would be considered a dense block (Figure 3.4). Once a block is compressed it will not be revisited.

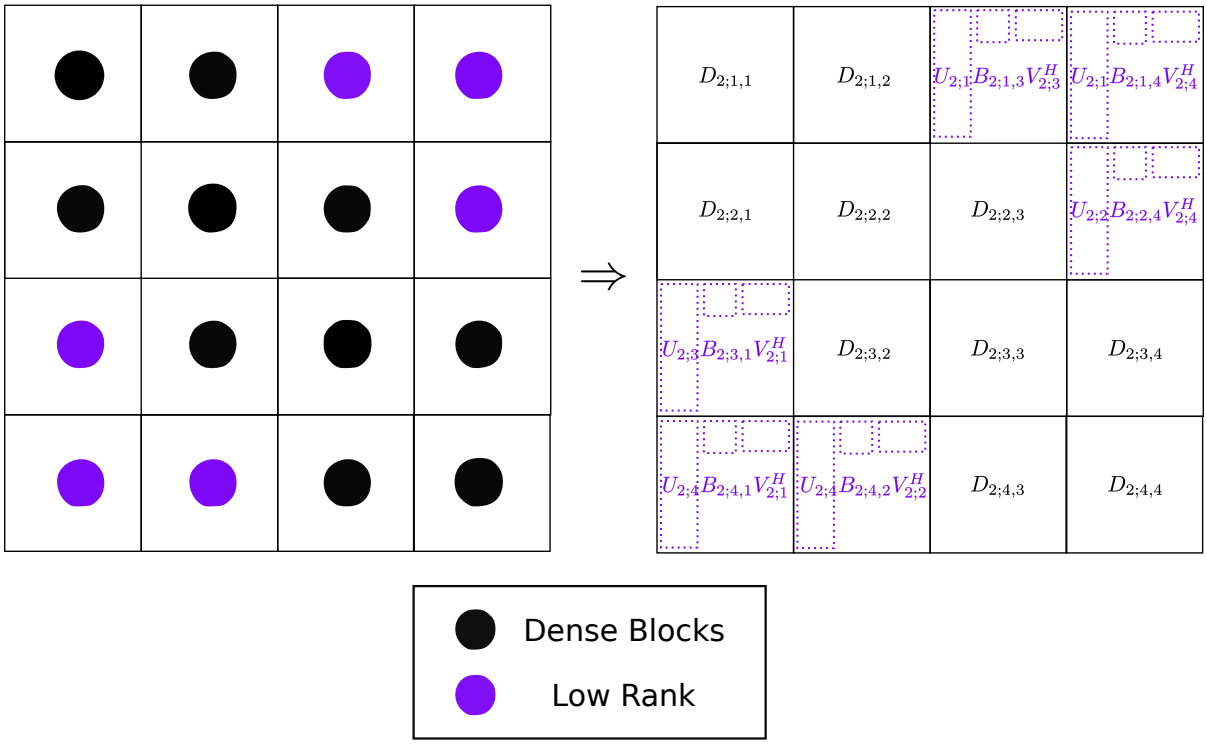


Figure 3.3: Example 2 level FMM Representation

In this thesis, we will show that we obtain an FMM 5pt representation when we multiply two FMM 3pt representations. We will go through these derivations in more detail in subsection 3.3.4. In Figure 3.5 we give a visual example which shows both the FMM 3pt and the FMM 5pt structure. The FMM 3pt structure resembles that of a tridiagonal matrix. When multiplying two tridiagonal matrices, the result is a pentadiagonal matrix. Therefore, it should make sense intuitively, that when we multiply two 3pt FMM representations, we obtain a 5pt FMM representation. This gives an illustrative example of why the derivations for the matrix-matrix multiply are somewhat

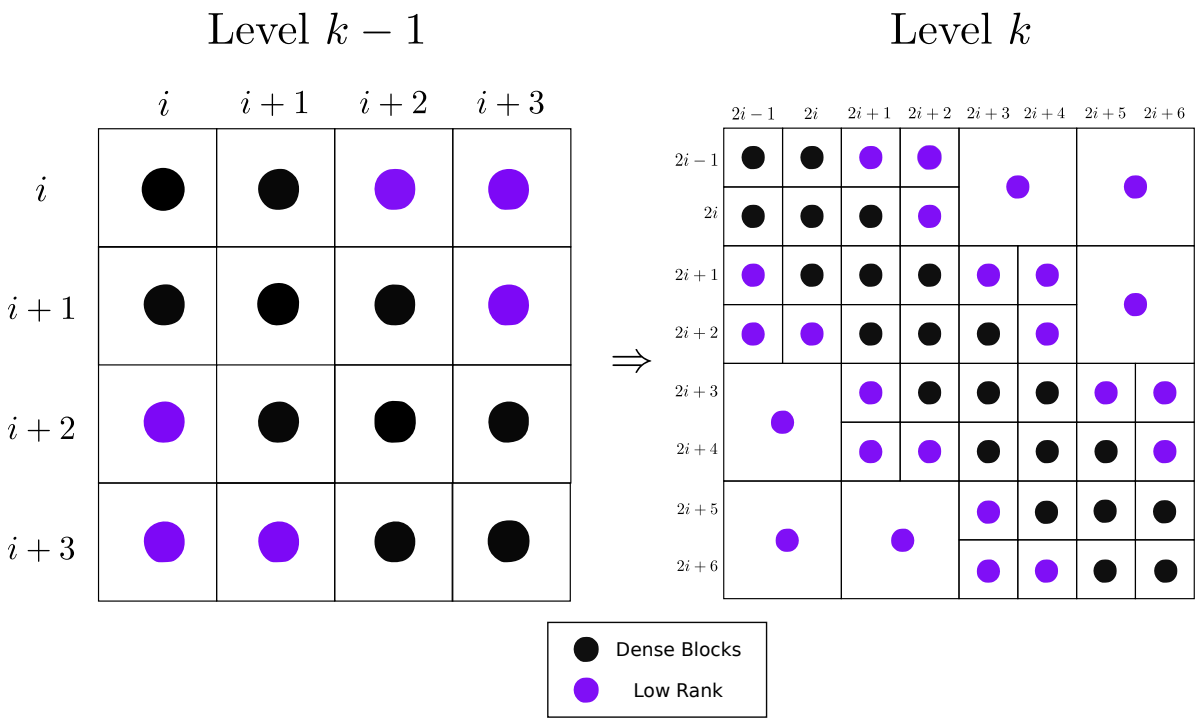


Figure 3.4: Recursive FMM Structure

laborious. Additionally, if we consider the case where we can write the inverse as the sum of the powers of our matrix (as shown in subsection 3.1.4), it also gives an illustrative example of why we make the conjecture that the inverse of a (3pt) FMM matrix is likely not FMM.

Notice that the difference in the low rank structure of 3pt FMM versus 5pt FMM is that in 3pt FMM representation blocks on the diagonal and blocks that touch the diagonal are considered dense, whereas in the 5pt FMM representation, neighbors of blocks that touch the diagonal are considered dense as well (Figure 3.5). In practice when a 5pt FMM representation is computed, it is converted back to a 3pt FMM representation.

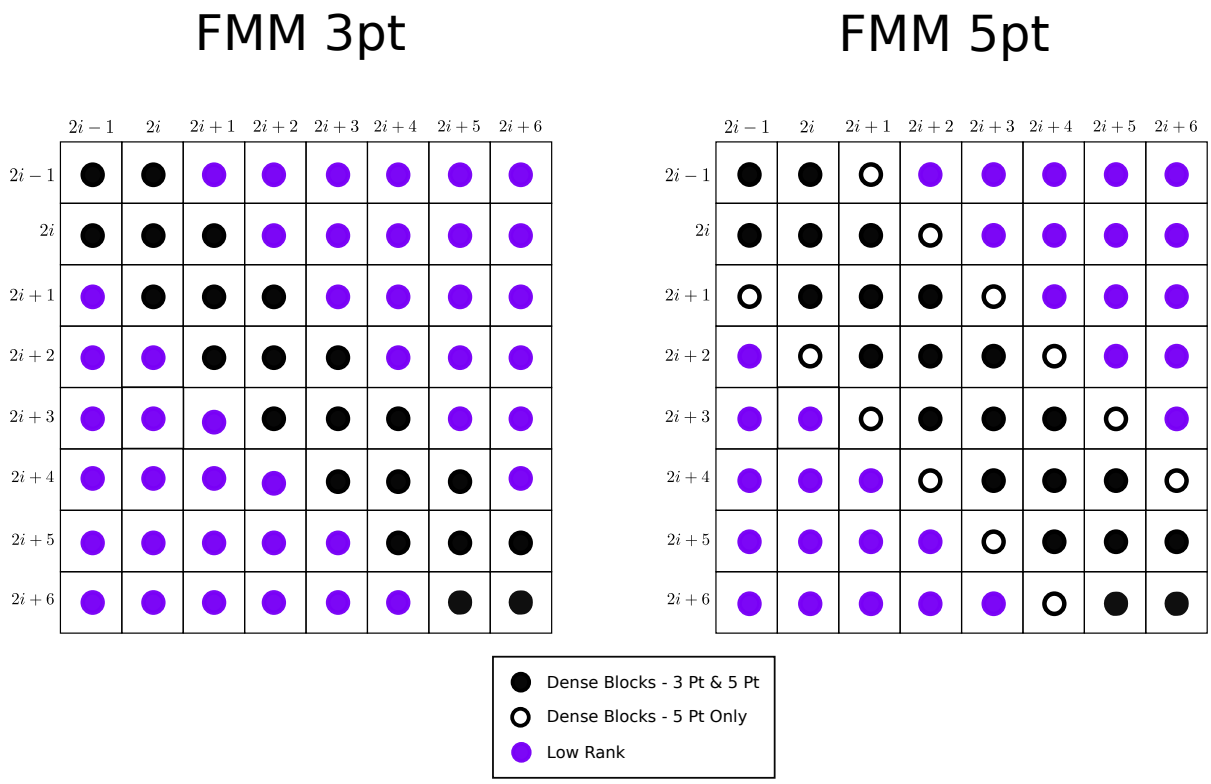


Figure 3.5: Recursive FMM Structure

With this compression comes a loss in precision of

$$\begin{aligned}\sum_{k=1}^{\log(n)} (kp + n_0) &= p\log^2(n) + n_0\log(n) \\ &\approx O(p\log^2(n)),\end{aligned}$$

where n_0 is the smallest partition of the matrix, and is determined by the underlying problem itself. (Anything smaller than $n_0 = 3p$ will yield no gain in compression). In practice only $O(pc)$ is kept, where c is some smaller constant. If this term becomes too large, a compression step should be performed to compute a new low rank expression at each level. We expect the computation cost for this to be roughly poly-logarithmic.

3.3 FMM Matrix-Matrix Multiply Recursions

In order to consider recursion equations for the FMM matrix-matrix multiply we must cover formulas for the dense blocks ($D_{k;i,j}$), up-sweep recursions ($G_{k;i}$), column and row basis operators ($U_{k;i}, V_{k;i}$), column and row translation operators ($R_{k;i}, W_{k;i}$), down-sweep recursions ($F_{k;i,j}$) and expansion coefficients ($B_{k;i,j}$). All of these formulas together will allow us to define the FMM 5pt representation.

3.3.1 3pt FMM Dense Block ($D_{k;i,j}$) Recursions

We begin with a linear system which is characterized by a matrix A. We begin by defining,

$$A = D_{0;1,1} = \begin{pmatrix} D_{1;1,1} & D_{1;1,2} \\ D_{1;2,1} & D_{1;2,2} \end{pmatrix} \tag{3.4}$$

Recursive definitions for the dense blocks for 3pt FMM are as follows

$$\begin{aligned}
D_{k-1;i,i} &= \begin{pmatrix} D_{k;2i-1,2i-1} & D_{k;2i-1,2i} \\ D_{k;2i,2i-1} & D_{k;2i,2i} \end{pmatrix} \\
D_{k-1;i,i+1} &= \begin{pmatrix} D_{k;2i-1,2i+1} & D_{k;2i-1,2i+2} \\ D_{k;2i,2i+1} & D_{k;2i,2i+2} \end{pmatrix} \\
&= \begin{pmatrix} U_{k;2i-1}B_{k;2i-1,2i+1}V_{k;2i+1} & U_{k;2i-1}B_{k;2i-1,2i+2}V^T_{k;2i+2} \\ D_{k;2i,2i+1} & U_{k;2i}B_{k;2i,2i+2}V_{k;2i+2} \end{pmatrix} \\
D_{k-1;i+1,i} &= \begin{pmatrix} D_{k;2i+1,2i-1} & D_{k;2i+1,2i} \\ D_{k;2i+2,2i-1} & D_{k;2i+2,2i} \end{pmatrix} \\
&= \begin{pmatrix} U_{k;2i+1}B_{k;2i+1,2i-1}V_{k;2i-1} & D_{k;2i+1,2i} \\ U_{k;2i+2}B_{k;2i+2,2i-1}V_{k;2i-1} & U_{k;2i}B_{k;2i+2,2i}V^T_{k;2i} \end{pmatrix}.
\end{aligned} \tag{3.5}$$

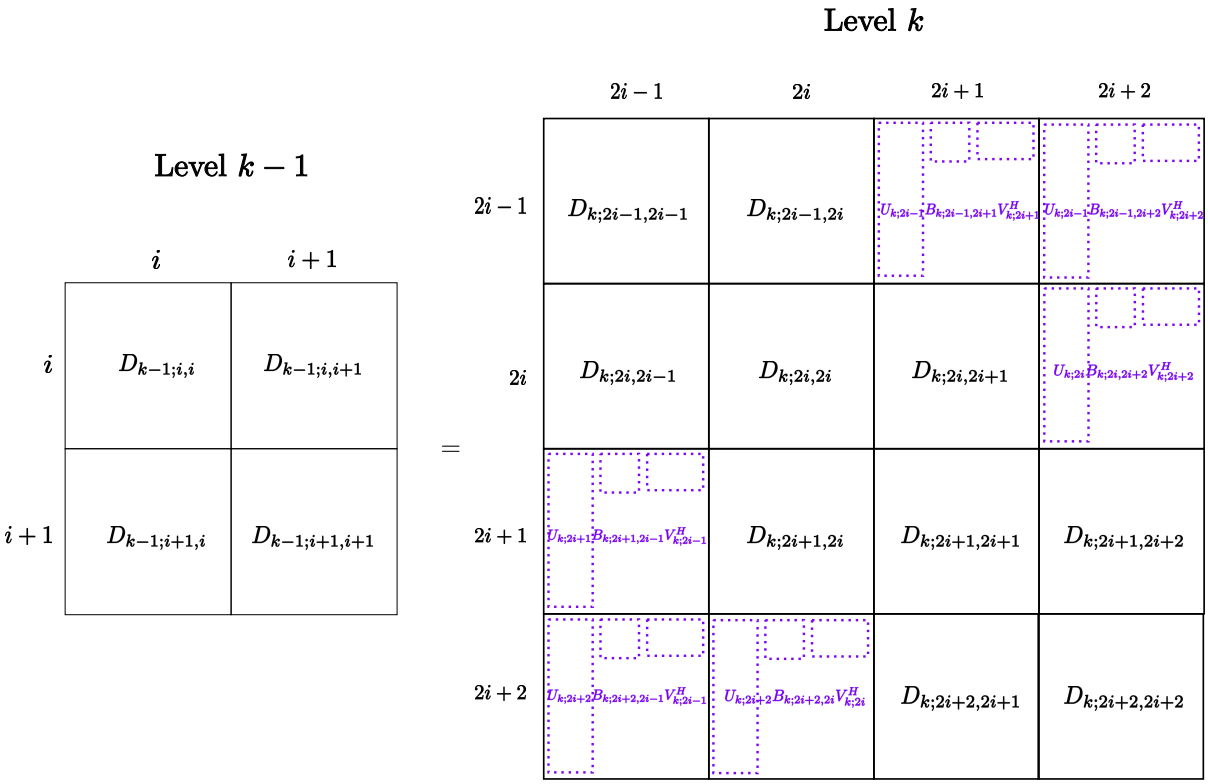


Figure 3.6: Recursive 3pt FMM Structure

In 3pt FMM any block which touches the diagonal will be recursively partitioned, except at the leaf level where it will be considered a dense block. More specifically, if the row and column indices of any block are separated by 2 or more, the block is low rank. If the row and column indices are separated by less than 2 they must be recursively divided (except at the leaf level where it will be considered a dense block). An illustrative example is given in Figure 3.6 for clarity.

3.3.2 5pt FMM Dense Block ($D_{k;i,j}$) Recursions

The 5pt FMM representation is the representation that results from the multiply of two 3pt FMM representations. The recursive definitions for the dense blocks in the 5pt

FMM representation are as follows

$$
\begin{aligned}
D_{k-1;i,i} &= \begin{pmatrix} D_{k;2i-1,2i-1} & D_{k;2i-1,2i} \\ D_{k;2i,2i-1} & D_{k;2i,2i} \end{pmatrix} \\
D_{k-1;i,i+1} &= \begin{pmatrix} D_{k;2i-1,2i+1} & D_{k;2i-1,2i+2} \\ D_{k;2i,2i+1} & D_{k;2i,2i+3} \end{pmatrix} \\
&= \begin{pmatrix} D_{k;2i-1,2i+1} & U_{k;2i-1}B_{k;2i-1,2i+2}V_{k;2i+2} \\ D_{k;2i,2i+1} & D_{k;2i,2i+2} \end{pmatrix} \\
D_{k-1;i,i+2} &= \begin{pmatrix} D_{k;2i-1,2i+3} & D_{k;2i-1,2i+4} \\ D_{k;2i,2i+3} & D_{k;2i,2i+4} \end{pmatrix} \\
&= \begin{pmatrix} U_{k;2i-1}B_{k;2i-1,2i+3}V_{k;2i+3}^T & U_{k;2i-1}B_{k;2i-1,2i+4}V_{k;2i+4}^T \\ U_{k;2i}B_{k;2i,2i+3}V_{k;2i+3}^T & U_{k;2i}B_{k;2i,2i+4}V_{k;2i+4}^T \end{pmatrix} \\
D_{k-1;i+1,i} &= \begin{pmatrix} D_{k;2i+1,2i-1} & D_{k;2i+1,2i} \\ D_{k;2i+2,2i-1} & D_{k;2i+2,2i} \end{pmatrix} \\
&= \begin{pmatrix} D_{k;2i+1,2i-1} & D_{k;2i+1,2i} \\ U_{k;2i+2}B_{k;2i+2,2i-1}V_{k;2i-1}^T & D_{k;2i+2,2i} \end{pmatrix} \\
D_{k-1;i+2,i} &= \begin{pmatrix} D_{k;2i+3,2i-1} & D_{k;2i+3,2i} \\ D_{k;2i+4,2i-1} & D_{k;2i+4,2i} \end{pmatrix} \\
&= \begin{pmatrix} U_{k;2i+3}B_{k;2i+3,2i-1}V_{k;2i-1}^T & U_{k;2i+3}B_{k;2i+3,2i}V_{k;2i}^T \\ U_{k;2i+4}B_{k;2i+4,2i-1}V_{k;2i-1}^T & U_{k;2i+4}B_{k;2i+4,2i}V_{k;2i}^T \end{pmatrix}.
\end{aligned}
\tag{3.6}
$$

In this representation, neighbors of blocks that touch the diagonal are now considered dense blocks as well. More specifically if the row and column indices of a given block are separated by 1 or 0, then that block is recursively divided (except at the leaf level where it is considered a dense block). If the row and column indices are separated by 2 it is divided into four low-rank blocks. Finally, if the row and column indices are separated

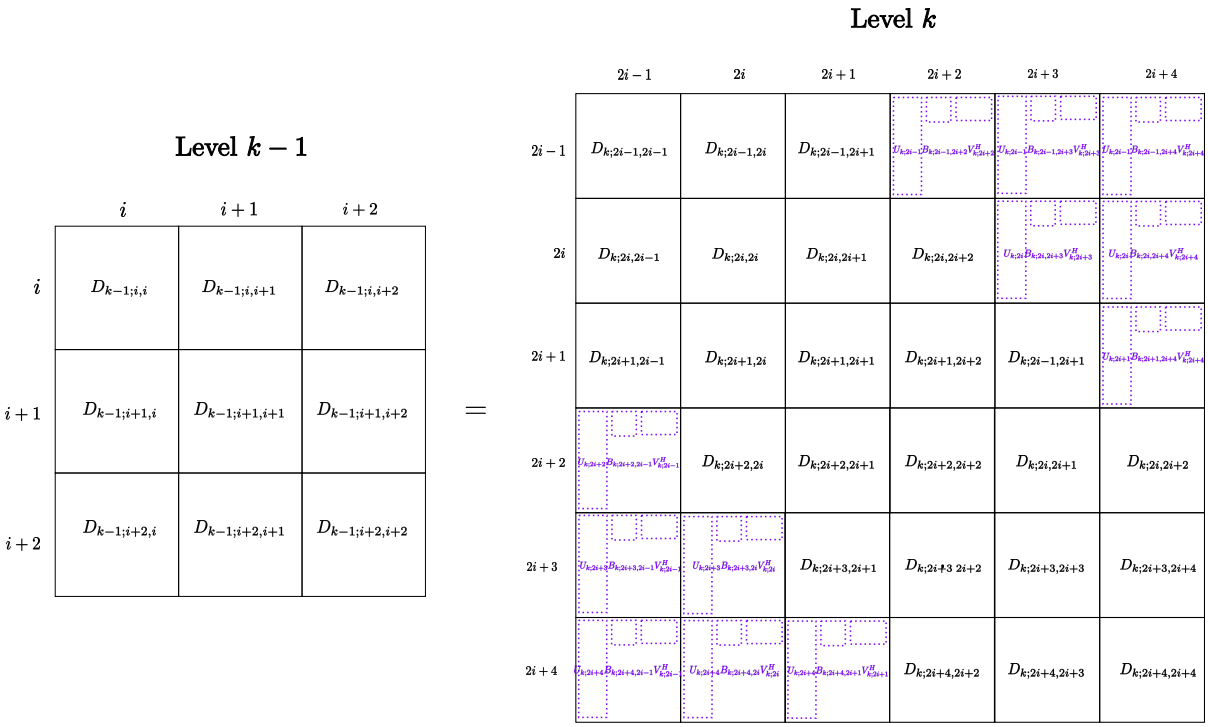


Figure 3.7: Recursive 5pt FMM Structure

by 3 or more the block is already low rank. An illustrative example is given in Figure 3.7 for clarity.

3.3.3 Notation for the FMM Matrix-Matrix Multiply

We want to carry out a matrix-matrix multiply,

$$A\tilde{A} = C$$

, where A and $\tilde{A}$ are 3pt FMM and C is 5pt FMM. We would like to know the exact structure of C and how we can compute it. We will follow the same notation for the operators in each matrix itself, namely that each operator in $\tilde{A}$ is decorated with a tilde to denote that it is associated with this matrix. In Figure 3.8 we give an example of a block

8x8 matrix-matrix matrix multiply to illustrate the notation. The choice in notation was simply so that formulas would not be any longer than necessary.

$$
\begin{pmatrix}
D_{3;1,1} & D_{3;1,2} & U_{3;1}B_{3;1,3}V_{3;3}^H & U_{3;1}B_{3;1,4}V_{3;4}^H & U_{3;1}R_{3;1}B_{2;1,3}W_{3;5}^HV_{3;5}^H & U_{3;1}R_{3;1}B_{2;1,3}W_{3;6}^HV_{3;6}^H & U_{3;1}R_{3;1}B_{2;1,4}W_{3;7}^HV_{3;7}^H & U_{3;1}R_{3;1}B_{2;1,4}W_{3;8}^HV_{3;8}^H \\
D_{3;2,1} & D_{3;2,2} & D_{3;2,3} & U_{3;2}B_{3;2,4}V_{3;4}^H & U_{3;2}R_{3;2}B_{2;1,3}W_{3;5}^HV_{3;5}^H & U_{3;2}R_{3;2}B_{2;1,3}W_{3;6}^HV_{3;6}^H & U_{3;2}R_{3;2}B_{2;1,4}W_{3;7}^HV_{3;7}^H & U_{3;2}R_{3;2}B_{2;1,4}W_{3;8}^HV_{3;8}^H \\
U_{3;3}B_{3;3,1}V_{3;1}^H & D_{3;3,2} & D_{3;3,3} & D_{3;3,4} & U_{3;3}B_{3;3,5}V_{3;5}^H & U_{3;3}B_{3;3,6}V_{3;6}^H & U_{3;3}R_{3;3}B_{2;2,4}W_{3;7}^HV_{3;7}^H & U_{3;3}R_{3;3}B_{2;2,4}W_{3;8}^HV_{3;8}^H \\
U_{3;4}B_{3;4,1}V_{3;1}^H & U_{3;4}B_{3;4,2}V_{3;2}^H & D_{3;4,3} & D_{3;4,4} & D_{3;4,5} & U_{3;4}B_{3;4,6}V_{3;6}^H & U_{3;4}R_{3;4}B_{2;2,4}W_{3;7}^HV_{3;7}^H & U_{3;4}R_{3;4}B_{2;2,4}W_{3;8}^HV_{3;8}^H \\
U_{3;5}R_{3;5}B_{2;3,1}W_{3;1}^HV_{3;1}^H & U_{3;5}R_{3;5}B_{2;3,1}W_{3;2}^HV_{3;2}^H & U_{3;5}B_{3;5,3}V_{3;3}^H & U_{3;5}B_{3;5,4}V_{3;4}^H & D_{3;5,5} & D_{3;5,6} & U_{3;5}B_{3;5,7}V_{3;7}^H & U_{3;5}B_{3;5,8}V_{3;8}^H \\
U_{3;6}R_{3;6}B_{2;3,1}W_{3;1}^HV_{3;1}^H & U_{3;6}R_{3;6}B_{2;3,1}W_{3;2}^HV_{3;2}^H & U_{3;6}B_{3;6,3}V_{3;3}^H & U_{3;6}B_{3;6,4}V_{3;4}^H & D_{3;6,5} & D_{3;6,6} & D_{3;6,7} & U_{3;6}B_{3;5,8}V_{3;8}^H \\
U_{3;7}R_{3;7}B_{2;4,1}W_{3;1}^HV_{3;1}^H & U_{3;7}R_{3;7}B_{2;4,1}W_{3;2}^HV_{3;2}^H & U_{3;7}R_{3;7}B_{2;4,2}W_{3;3}^HV_{3;3}^H & U_{3;7}R_{3;7}B_{2;4,2}W_{3;4}^HV_{3;4}^H & U_{3;7}B_{3;7,5}V_{3;5}^H & D_{3;7,6} & D_{3;7,7} & D_{3;7,8} \\
U_{3;8}R_{3;8}B_{2;4,1}W_{3;1}^HV_{3;1}^H & U_{3;8}R_{3;8}B_{2;4,1}W_{3;2}^HV_{3;2}^H & U_{3;8}R_{3;8}B_{2;4,2}W_{3;3}^HV_{3;3}^H & U_{3;8}R_{3;8}B_{2;4,2}W_{3;4}^HV_{3;4}^H & U_{3;8}B_{3;8,5}V_{3;5}^H & U_{3;8}B_{3;8,6}V_{3;6}^H & D_{3;8,7} & D_{3;8,8}
\end{pmatrix}
$$

$$
\begin{pmatrix}
\tilde{D}_{3;1,1} & \tilde{D}_{3;1,2} & \tilde{U}_{3;1}\tilde{B}_{3;1,3}\tilde{V}_{3;3}^H & \tilde{U}_{3;1}\tilde{B}_{3;1,4}\tilde{V}_{3;4}^H & \tilde{U}_{3;1}\tilde{R}_{3;1}\tilde{B}_{2;1,3}\tilde{W}_{3;5}^H\tilde{V}_{3;5}^H & \tilde{U}_{3;1}\tilde{R}_{3;1}\tilde{B}_{2;1,3}\tilde{W}_{3;6}^H\tilde{V}_{3;6}^H & \tilde{U}_{3;1}\tilde{R}_{3;1}\tilde{B}_{2;1,4}\tilde{W}_{3;7}^H\tilde{V}_{3;7}^H & \tilde{U}_{3;1}\tilde{R}_{3;1}\tilde{B}_{2;1,4}\tilde{W}_{3;8}^H\tilde{V}_{3;8}^H \\
\tilde{D}_{3;2,1} & \tilde{D}_{3;2,2} & \tilde{D}_{3;2,3} & \tilde{U}_{3;2}\tilde{B}_{3;2,4}\tilde{V}_{3;4}^H & \tilde{U}_{3;2}\tilde{R}_{3;2}\tilde{B}_{2;1,3}\tilde{W}_{3;5}^H\tilde{V}_{3;5}^H & \tilde{U}_{3;2}\tilde{R}_{3;2}\tilde{B}_{2;1,3}\tilde{W}_{3;6}^H\tilde{V}_{3;6}^H & \tilde{U}_{3;2}\tilde{R}_{3;2}\tilde{B}_{2;1,4}\tilde{W}_{3;7}^H\tilde{V}_{3;7}^H & \tilde{U}_{3;2}\tilde{R}_{3;2}\tilde{B}_{2;1,4}\tilde{W}_{3;8}^H\tilde{V}_{3;8}^H \\
\tilde{U}_{3;3}\tilde{B}_{3;3,1}\tilde{V}_{3;1}^H & \tilde{D}_{3;3,2} & \tilde{D}_{3;3,3} & \tilde{D}_{3;3,4} & \tilde{U}_{3;3}\tilde{B}_{3;3,5}\tilde{V}_{3;5}^H & \tilde{U}_{3;3}\tilde{B}_{3;3,6}\tilde{V}_{3;6}^H & \tilde{U}_{3;3}\tilde{R}_{3;3}\tilde{B}_{2;2,4}\tilde{W}_{3;7}^H\tilde{V}_{3;7}^H & \tilde{U}_{3;3}\tilde{R}_{3;3}\tilde{B}_{2;2,4}\tilde{W}_{3;8}^H\tilde{V}_{3;8}^H \\
\tilde{U}_{3;4}\tilde{B}_{3;4,1}\tilde{V}_{2;1}^H & \tilde{U}_{3;4}\tilde{B}_{3;4,2}\tilde{V}_{2;2}^H & \tilde{D}_{3;4,3} & \tilde{D}_{3;4,4} & \tilde{D}_{3;4,5} & \tilde{U}_{3;4}\tilde{B}_{3;4,6}^H\tilde{V}_{3;6}^H & \tilde{U}_{3;4}\tilde{R}_{3;4}\tilde{B}_{2;2,4}\tilde{W}_{3;7}^H\tilde{V}_{3;7}^H & \tilde{U}_{3;4}\tilde{R}_{3;4}\tilde{B}_{2;2,4}\tilde{W}_{3;8}^H\tilde{V}_{3;8}^H \\
\tilde{U}_{3;5}\tilde{R}_{3;5}\tilde{B}_{2;3,1}\tilde{W}_{3;1}^H\tilde{V}_{3;1}^H & \tilde{U}_{3;5}\tilde{R}_{3;5}\tilde{B}_{2;3,1}\tilde{W}_{3;2}^H\tilde{V}_{3;2}^H & \tilde{U}_{3;5}\tilde{B}_{3;5,3}\tilde{V}_{3;3}^H & \tilde{D}_{3;5,4} & \tilde{D}_{3;5,5} & \tilde{D}_{3;5,6} & \tilde{U}_{3;5}\tilde{B}_{3;5,7}\tilde{V}_{3;7}^H & \tilde{U}_{3;5}\tilde{B}_{3;5,8}\tilde{V}_{3;8}^H \\
\tilde{U}_{3;6}\tilde{R}_{3;6}\tilde{B}_{2;3,1}\tilde{W}_{3;1}^H\tilde{V}_{3;1}^H & \tilde{U}_{3;6}\tilde{R}_{3;6}\tilde{B}_{2;3,1}\tilde{W}_{3;2}^H\tilde{V}_{3;2}^H & \tilde{U}_{3;6}\tilde{B}_{3;6,3}\tilde{V}_{3;3}^H & \tilde{U}_{3;6}\tilde{B}_{3;6,4}\tilde{V}_{3;4}^H & \tilde{D}_{3;6,5} & \tilde{D}_{3;6,6} & \tilde{D}_{3;6,7} & \tilde{U}_{3;6}\tilde{B}_{3;5,8}\tilde{V}_{3;8}^H \\
\tilde{U}_{3;7}\tilde{R}_{3;7}\tilde{B}_{2;4,1}\tilde{W}_{3;1}^H\tilde{V}_{3;1}^H & \tilde{U}_{3;7}\tilde{R}_{3;7}\tilde{B}_{2;4,1}\tilde{W}_{3;2}^H\tilde{V}_{3;2}^H & \tilde{U}_{3;7}\tilde{R}_{3;7}\tilde{B}_{2;4,2}\tilde{W}_{3;3}^H\tilde{V}_{3;3}^H & \tilde{U}_{3;7}\tilde{R}_{3;7}\tilde{B}_{2;4,2}\tilde{W}_{3;4}^H\tilde{V}_{3;4}^H & \tilde{U}_{3;7}\tilde{B}_{3;7,5}\tilde{V}_{3;5}^H & \tilde{D}_{3;7,6} & \tilde{D}_{3;7,7} & \tilde{D}_{3;7,8} \\
\tilde{U}_{3;8}\tilde{R}_{3;8}\tilde{B}_{2;4,1}\tilde{W}_{3;1}^H\tilde{V}_{3;1}^H & \tilde{U}_{3;8}\tilde{R}_{3;8}\tilde{B}_{2;4,1}\tilde{W}_{3;2}^H\tilde{V}_{3;2}^H & \tilde{U}_{3;8}\tilde{R}_{3;8}\tilde{B}_{2;4,2}\tilde{W}_{3;3}^H\tilde{V}_{3;3}^H & \tilde{U}_{3;8}\tilde{R}_{3;8}\tilde{B}_{2;4,2}\tilde{W}_{3;4}^H\tilde{V}_{3;4}^H & \tilde{U}_{3;8}\tilde{B}_{3;8,5}\tilde{V}_{3;5}^H & \tilde{U}_{3;8}\tilde{B}_{3;8,6}\tilde{V}_{3;6}^H & \tilde{D}_{3;8,7} & \tilde{D}_{3;8,8}
\end{pmatrix}
$$

$$
=
\begin{pmatrix}
C_{3;1,1} & C_{3;1,2} & C_{3;1,3} & C_{3;1,6} & C_{3;1,5} & C_{3;1,6} & C_{3;1,7} & C_{3;1,8} \\
C_{3;2,1} & C_{3;2,2} & C_{3;2,3} & C_{3;2,6} & C_{3;2,5} & C_{3;2,6} & C_{3;2,7} & C_{3;2,8} \\
C_{3;3,1} & C_{3;3,2} & C_{3;3,3} & C_{3;3,6} & C_{3;3,5} & C_{3;3,6} & C_{3;3,7} & C_{3;3,8} \\
C_{3;4,1} & C_{3;4,2} & C_{3;4,3} & C_{3;4,6} & C_{3;4,5} & C_{3;4,6} & C_{3;4,7} & C_{3;4,8} \\
C_{3;5,1} & C_{3;5,2} & C_{3;5,3} & C_{3;5,6} & C_{3;5,5} & C_{3;5,6} & C_{3;5,7} & C_{3;5,8} \\
C_{3;6,1} & C_{3;6,2} & C_{3;6,3} & C_{3;6,6} & C_{3;6,5} & C_{3;6,6} & C_{3;6,7} & C_{3;6,8} \\
C_{3;7,1} & C_{3;7,2} & C_{3;7,3} & C_{3;7,6} & C_{3;7,5} & C_{3;7,6} & C_{3;7,7} & C_{3;7,8} \\
C_{3;8,1} & C_{3;8,2} & C_{3;8,3} & C_{3;8,6} & C_{3;8,5} & C_{3;8,6} & C_{3;8,7} & C_{3;8,8}
\end{pmatrix}
$$

Figure 3.8: Example Block 8x8 Multiply

3.3.4 FMM Matrix-Matrix Multiply Formula Derivations

Up-Sweep Recursions

Let us make the standard up-sweep definition,

$$G_{k;i} = V_{k;i}^H \tilde{U}_{k;i}. \tag{3.7}$$

The recursion can then be obtained as follows

$$\begin{aligned} G_{k-1;i} &= \begin{pmatrix} W_{k;2i-1}^H V_{k;2i-1}^H & W_{k;2i}^H V_{k;2i}^H \end{pmatrix} \begin{pmatrix} \tilde{U}_{k;2i-1}\tilde{R}_{k;2i-1} \\ \tilde{U}_{k;2i}\tilde{R}_{k;2i} \end{pmatrix} \\ &= W_{k;2i-1}^H V_{k;2i-1}^H \tilde{U}_{k;2i-1}\tilde{R}_{k;2i-1} + W_{k;2i}^H V_{k;2i}^H \tilde{U}_{k;2i}\tilde{R}_{k;2i} \\ &= W_{k;2i-1}^H G_{k;2i-1}\tilde{R}_{k;2i-1} + W_{k;2i}^H G_{k;2i}\tilde{R}_{k;2i}. \end{aligned}$$

Where we have simply plugged in Equation 1.4 for $U_{k;j}$ and $V_{k;j}$.

Column and Row Basis and Translation Operators

Before we can consider the matrix-matrix multiply recursions, we must first obtain a recursion for the column and row basis operators, $U_{k;i}(C)$ and $V_{k;i}(C)$, as well as discover what the row and column translation operators $R_{k;i}(C)$ and $W_{k;i}(C)$ are. We claim

$$U_{k;i}(C) = \begin{pmatrix} U_{k;i} & D_{k;i,i-1}\tilde{U}_{k;i-1} & D_{k;i,i}\tilde{U}_{k;i} & D_{k;i,i+1}\tilde{U}_{k;i+1} \end{pmatrix} \tag{3.8}$$

Recall that we are looking for operators that satisfy an equation of the form

$$U_{k-1;i}(C) = \begin{pmatrix} U_{k;2i-1}(C)R_{k;2i-1}(C) \\ U_{k;2i}(C)R_{k+1;2i}(C) \end{pmatrix}. \tag{3.9}$$

As we show in Figure 3.9, let us begin with our claim, Equation 3.8. We then simply plug in the recursive definitions given earlier for our basis operators (equations 1.4) and the 3pt dense blocks (equations 3.5). We then carry out the multiply and notice we can factor the matrix as shown.

$$
\begin{aligned}
U_{k-1;i}(C) &= \begin{pmatrix} U_{k-1;i} & D_{k-1;i,i-1}\tilde{U}_{k-1;i-1} & D_{k-1;i,i}\tilde{U}_{k-1;i} & D_{k-1;i,i+1}\tilde{U}_{k-1;i+1} \end{pmatrix} \\
&= \begin{pmatrix} \begin{pmatrix} U_{k;2i-1}R_{k;2i-1} \\ U_{k;2i}R_{k;2i} \end{pmatrix} & \begin{pmatrix} U_{k;2i-1}B_{2i-1,2i-3}V_{k;2i-3}^H & D_{k;2i-1,2i-2} \\ U_{k;2i}B_{k;2i,2i-3}V_{2i-3}^H & U_{k;2i}B_{k;2i,2i-2}V_{k;2i-2}^H \end{pmatrix} \begin{pmatrix} \tilde{U}_{k;2i-3}\tilde{R}_{k;2i-3} \\ \tilde{U}_{k;2i-2}\tilde{R}_{k;2i-2} \end{pmatrix} \cdots \\
\begin{pmatrix} D_{k;2i-1,2i-1} & D_{k;2i-1,2i} \\ D_{k;2i,2i-1} & D_{k;2i,2i} \end{pmatrix} \begin{pmatrix} \tilde{U}_{k;2i-1}\tilde{R}_{k;2i-1} \\ \tilde{U}_{k;2i}\tilde{R}_{k;2i} \end{pmatrix} & \begin{pmatrix} U_{k;2i-1}B_{k;2i-1,2i+1}V_{k;2i+1}^H & U_{k;2i-1}B_{k;2i-1,2i+2}V_{k;2i+2}^H \\ D_{k;2i,2i+21} & U_{k;2i}B_{k;2i,2i+2}V_{k;2i+2}^H \end{pmatrix} \begin{pmatrix} \tilde{U}_{k;2i+1}\tilde{R}_{k;2i+1} \\ \tilde{U}_{k;2i+2}\tilde{R}_{k;2i+2} \end{pmatrix} \end{pmatrix} \\
&= \begin{pmatrix} \begin{pmatrix} U_{k;2i-1}R_{k;2i-1} \\ U_{k;2i}R_{k;2i} \end{pmatrix} & \begin{pmatrix} U_{k;2i-1}B_{2i-1,2i-3}G_{k;2i-3}\tilde{R}_{k;2i-3} + D_{k;2i-1,2i-2}\tilde{U}_{k;2i-2}\tilde{R}_{k;2i-2} \\ U_{k;2i}B_{k;2i,2i-3}G_{k;2i-3}\tilde{R}_{k;2i-3} + U_{k;2i}B_{k;2i,2i-2}G_{k;2i-2}\tilde{R}_{k;2i-2} \end{pmatrix} \cdots \\
\begin{pmatrix} D_{k;2i-1,2i-1}\tilde{U}_{k;2i-1}\tilde{R}_{k;2i-1} + D_{k;2i-1,2i}\tilde{U}_{k;2i}\tilde{R}_{k;2i} \\ D_{k;2i,2i-1}\tilde{U}_{k;2i-1}\tilde{R}_{k;2i-1} + D_{k;2i,2i}\tilde{U}_{k;2i}\tilde{R}_{k;2i} \end{pmatrix} & \begin{pmatrix} U_{k;2i-1}B_{k;2i-1,2i+1}G_{k;2i+1}\tilde{R}_{k;2i+1} + U_{k;2i-1}B_{k;2i-1,2i+2}G_{k;2i+2}\tilde{R}_{k;2i+2} \\ D_{k;2i,2i+21}\tilde{U}_{k;2i+1}\tilde{R}_{k;2i+1} + U_{k;2i}B_{k;2i,2i+2}G_{k;2i+2}\tilde{R}_{k;2i+2} \end{pmatrix} \end{pmatrix} \\
&= \begin{pmatrix} \begin{pmatrix} U_{k;2i-1} & D_{k;2i-1,2i-1} & D_{k;2i-1,2i-2}\tilde{U}_{k;2i-2} & D_{k;2i-1,2i-1}\tilde{U}_{k;2i-1} & D_{k;2i-1,2i}\tilde{U}_{k;2i} \end{pmatrix} \begin{pmatrix} R_{k;2i-1} & B_{2i-1,2i-3}G_{k;2i-3}\tilde{R}_{k;2i-3} & 0 & B_{k;2i-1,2i+1}G_{k;2i+1}\tilde{R}_{k;2i+1} + B_{k;2i-1,2i+2}G_{k;2i+2}\tilde{R}_{k;2i+2} \\ 0 & \tilde{R}_{k;2i-2} & 0 & 0 \\ 0 & 0 & \tilde{R}_{k;2i-1} & 0 \\ 0 & 0 & \tilde{R}_{k;2i} & 0 \end{pmatrix} \\
\begin{pmatrix} U_{k;2i} & D_{k;2i,2i-1}\tilde{U}_{k;2i-1} & D_{k;2i,2i}\tilde{U}_{k;2i} & D_{k;2i,2i+1}\tilde{U}_{k;2i+1} \end{pmatrix} \begin{pmatrix} R_{k;2i} & B_{k;2i,2i-3}G_{k;2i-3}\tilde{R}_{k;2i-3} + B_{k;2i,2i-2}G_{k;2i-2}\tilde{R}_{k;2i-2} & 0 & B_{k;2i,2i+2}G_{k;2i+2}\tilde{R}_{k;2i+2} \\ 0 & 0 & \tilde{R}_{k;2i-1} & 0 \\ 0 & 0 & \tilde{R}_{k;2i} & 0 \\ 0 & 0 & 0 & \tilde{R}_{k;2i+1} \end{pmatrix} \end{pmatrix} \\
&= \begin{pmatrix} U_{k;2i-1}(C)R_{k;2i-1}(C) \\ U_{k;2i}(C)R_{k;2i}(C) \end{pmatrix}
\end{aligned}
$$

Figure 3.9: Derivation of Column Basis Translation Operator Recursions

So Equation 3.8 indeed holds. Further, we have that

$$R_{k;2i-1}(C) = \begin{pmatrix} R_{k;2i-1} & B_{k;2i-1,2i-3}G_{k;2i-3}\tilde{R}_{k;2i-3} & 0 & B_{k;2i-1,2i+1}G_{k;2i+1}\tilde{R}_{k;2i+1} + B_{k;2i-1,2i+2}G_{k;2i+2}\tilde{R}_{k;2i+2} \\ 0 & \tilde{R}_{k;2i-2} & 0 & 0 \\ 0 & 0 & \tilde{R}_{k;2i-1} & 0 \\ 0 & 0 & \tilde{R}_{k;2i} & 0 \end{pmatrix} \tag{3.10}$$

$$R_{k;2i}(C) = \begin{pmatrix} R_{k;2i} & B_{k;2i,2i-3}G_{k;2i-3}\tilde{R}_{k;2i-3} + B_{k;2i,2i-2}G_{k;2i-2}\tilde{R}_{k;2i-2} & 0 & B_{k;2i,2i+2}G_{k;2i+2}\tilde{R}_{k;2i+2} \\ 0 & 0 & \tilde{R}_{k;2i-1} & 0 \\ 0 & 0 & \tilde{R}_{k;2i} & 0 \\ 0 & 0 & 0 & \tilde{R}_{k;2i+1} \end{pmatrix}. \tag{3.11}$$

Likewise, the recursion for the row bases of C ($V_{k;i}(C)$) can be found by carrying out these same computations on the transpose. For brevity we will not repeat these derivations here. Doing so we find that

$$V_{k;i}(C) \;=\; \left(\begin{array}{cccc} \tilde{V}_{k;i} & \tilde{D}^{H}_{k;i-1,i} V_{k;i-1} & \tilde{D}^{H}_{k;i,i} V_{k;i} & \widetilde{D}^{H}_{k;i+1,i} V_{k;i+1} \end{array} \right),$$

and likewise the row translation operators are

$$W_{k;2i-1}(C) = \begin{pmatrix} \tilde{W}_{k;2i-1} & \tilde{B}^H_{k;2i-3,2i-1} G^H_{k;2i-3} W_{k;2i-3} & 0 & \tilde{B}^H_{k;2i+1,2i-1} G^H_{k;2i+1} W_{k;2i+1} + \tilde{B}^H_{k;2i+2,2i-1} G^H_{k;2i+2} W_{k;2i+2} \\ 0 & W_{k;2i-2} & 0 & 0 \\ 0 & 0 & W_{k;2i-1} & 0 \\ 0 & 0 & W_{k;2i} & 0 \end{pmatrix} \tag{3.12}$$

$$W_{k;2i}(C) = \begin{pmatrix} \tilde{W}_{k;2i} & \tilde{B}^H_{k;2i-3,2i} G^H_{k;2i-3} W_{k;2i-3} + \tilde{B}^H_{k;2i-2,2i} G^H_{k;2i-2} W_{k;2i-2} & 0 & \tilde{B}^H_{k;2i+2,2i} G^H_{k;2i+2} W_{k;2i+2} \\ 0 & 0 & W_{k;2i-1} & 0 \\ 0 & 0 & W_{k;2i} & 0 \\ 0 & 0 & 0 & W_{k;2i+1} \end{pmatrix}. \tag{3.13}$$

Matrix-Matrix Multiply

We are now ready to consider the multiplication $A\tilde{A} = C$, which we can write in terms of dense blocks at the root level,

$$D_{0;1,1}\tilde{D}_{0;1,1} = D_{0;1,1}(C), \tag{3.14}$$

where $D_{0;1,1}$ and $\tilde{D}_{0;1,1}$ are given in 3pt FMM form and $D_{0;1,1}(C)$ will be computed in 5pt FMM form. Let us generalize the problem and look at a formula of the form

$$D_{0;1,1}\tilde{D}_{0;1,1} + U_{0;1,1}(C)F_{0;1,1}V^H_{0;1,1}(C) = D_{0;1,1}(C), \tag{3.15}$$

Where $F_{k;i,j}$ are our down-sweep Recursions. Notice that if we simply set $F_{0;1,1} = \{\}_{0\times 0}$ (a 0×0 empty matrix) we get back our original equation, 3.14. Therefore, beginning with Equation 3.15 we will compute the down-sweep recursions $F_{k;i,j}$. The set of indices that $F_{k;i,j}$ needs to be computed over are shown outlined in red in Figure 3.10. As can be seen in the figure, there are 10 unique such down-sweep recursion equations. We will compute one such derivation here, and along the way we will get the representation for the matrix C as well.

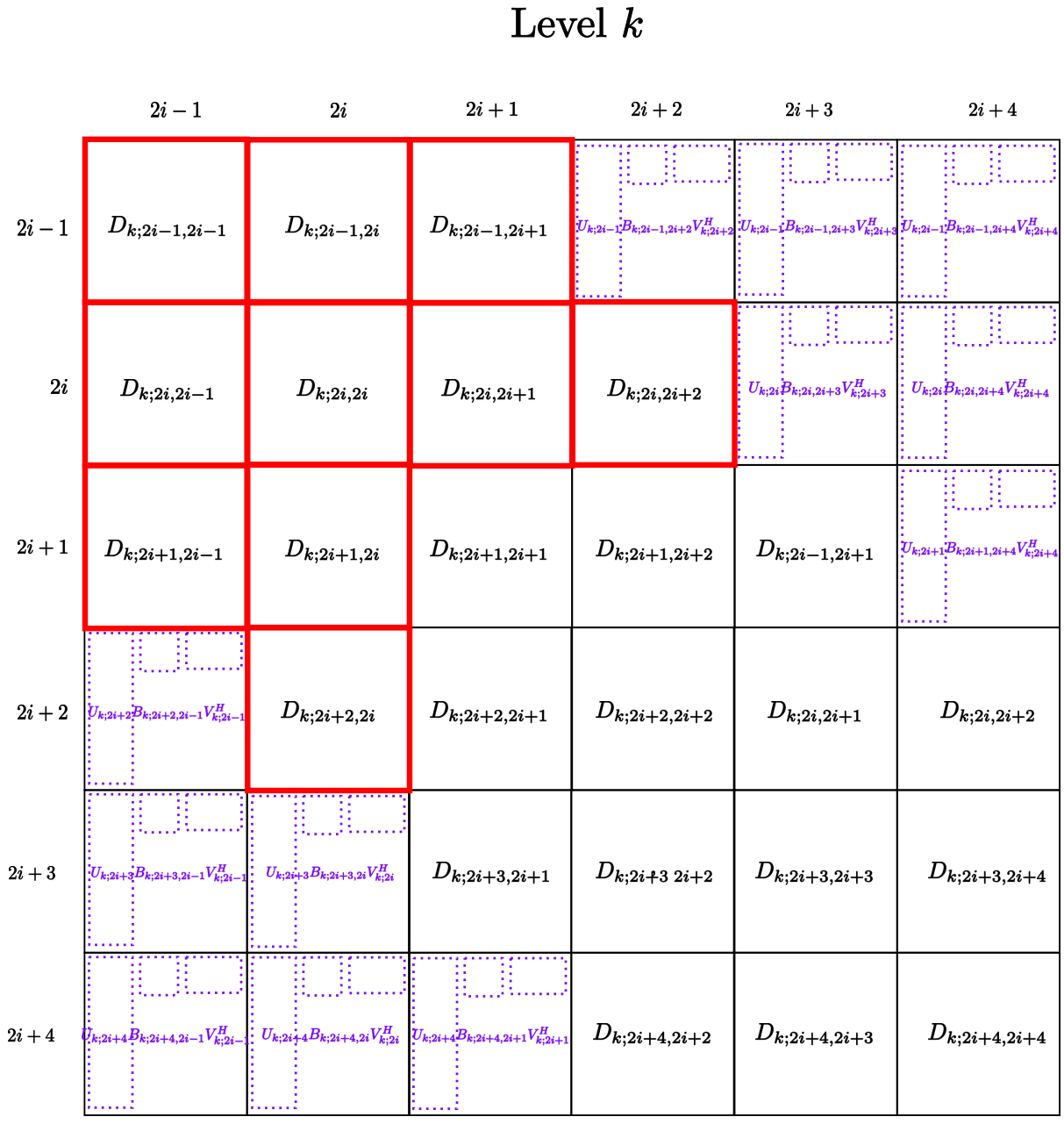


Figure 3.10: Set of indices (shown outlined in red) for which $F_{k;i,j}$ must be computed

In order to begin the derivation for the up-sweep recursions we must consider the recursions for the matrix C itself. Namely, we must compute recursions for $C_{k-1;i,i}$,

$C_{k-1;i,i+1}$, $C_{k-1;i,i+2}$, $C_{k-1;i+1,i}$ and $C_{k-1;i+2,i}$ as shown in Figure 3.11. We will only demonstrate the derivations for one such block, as the derivations for the others follow the same steps. We choose $C_{k-1;i,i+1}$ and give an example of how to recurse down this term.

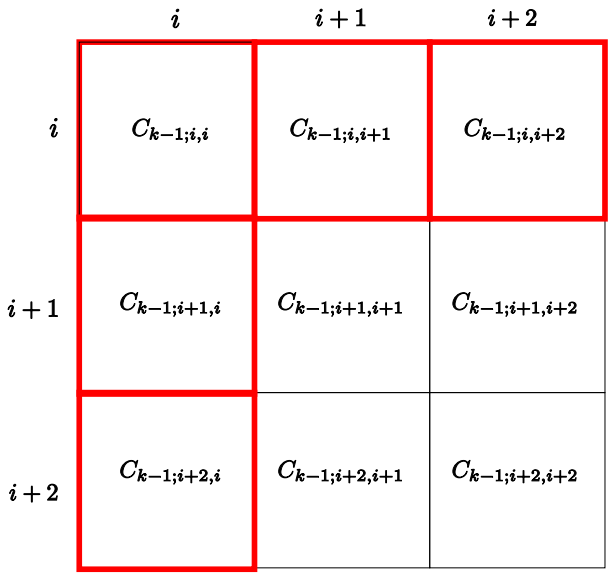


Figure 3.11: Blocks in C for which recursions must be computed

We claim

$$C_{k-1;i,i+1} = D_{k-1;i,i}\tilde{D}_{k-1;i,i+1} + D_{k-1;i,i+1}\tilde{D}_{k-1;i+1,i+1} + U_{k-1;i}(C)F_{k-1;i,i+1}V^{H}_{k-1;i+1}(C). \quad (3.16)$$

Notice that this equation is trivially satisfied when $k = 1$. If $(k-1; i, i+1)$ is a leaf node then we can simply compute $C_{k;i,i+1} = D_{k;i,i+1}(C)$ directly from this expression. If $(k-1; i, i+1)$ is not a leaf node, then we split this block and recurse down for each of the 4 sub-blocks. Let us assume we are at a leaf node and recurse down this block by plugging in definitions given in Equations 1.4 and 3.5:

$$
\begin{aligned}
C_{k-1;i,i+1} &= D_{k-1;i,i}\tilde{D}_{k-1;i,i+1} + D_{k-1;i,i+1}\tilde{D}_{k-1;i+1,i+1} + U_{k-1;i}(C)F_{k-1;i,i+1}V^H_{k-1;i+1}(C) \\
&= \begin{pmatrix} D_{k;2i-1,2i-1} & D_{k;2i-1,2i} \\ D_{k;2i,2i-1} & D_{k;2i,2i} \end{pmatrix} \begin{pmatrix} \tilde{U}_{k;2i-1}\tilde{B}_{k;2i-1,2i+1}\tilde{V}^H_{2i+1} & \tilde{U}_{k;2i-1}\tilde{B}_{k;2i-1,2i+2}\tilde{V}^H_{k;2i+2} \\ \tilde{D}_{k;2i,2i+1} & \tilde{U}_{k;2i}\tilde{B}_{k;2i,2i+2}\tilde{V}^H_{k;2i+2} \end{pmatrix} \\
&+ \begin{pmatrix} U_{k;2i-1}B_{k;2i-1,2i+1}V^H_{2i+1} & U_{k;2i-1}B_{k;2i-1,2i+2}V^H_{k;2i+2} \\ D_{k;2i,2i+1} & U_{k;2i}B_{k;2i,2i+2}V^H_{k;2i+2} \end{pmatrix} \begin{pmatrix} \tilde{D}_{k;2i+1,2i+1} & \tilde{D}_{k;2i+1,2i+2} \\ \tilde{D}_{k;2i+2,2i+1} & \tilde{D}_{k;2i+2,2i+2} \end{pmatrix} \\
&+ \begin{pmatrix} U_{k;2i-1}(C)R_{k;2i-1}(C) \\ U_{k;2i}(C)R_{k;2i}(C) \end{pmatrix} F_{k-1;i,i+1} \begin{pmatrix} W^H_{k;2i+1}(C)V^H_{k;2i+1}(C) & W^H_{k;2i+2}(C)V^H_{k;2i+2}(C) \end{pmatrix}
\end{aligned}
$$

We then simply carry out the multiply as shown in Figure 3.12.

$$
\begin{aligned}
C_{k-1;i,i+1} &= \begin{pmatrix} D_{k;2i-1,2i-1}\tilde{U}_{k;2i-1}\tilde{B}_{k;2i-1,2i+1}\tilde{V}^H_{2i+1} + D_{k;2i-1,2i}\tilde{D}_{k;2i,2i+1} & D_{k;2i-1,2i-1}\tilde{U}_{k;2i-1}\tilde{B}_{k;2i-1,2i+2}\tilde{V}^H_{k;2i+2} + D_{k;2i-1,2i}\tilde{U}_{k;2i}\tilde{B}_{k;2i,2i+2}\tilde{V}^H_{k;2i+2} \\ D_{k;2i,2i-1}\tilde{U}_{k;2i-1}\tilde{B}_{k;2i-1,2i+1}\tilde{V}^H_{2i+1} + D_{k;2i,2i}\tilde{D}_{k;2i,2i+1} & D_{k;2i,2i-1}\tilde{U}_{k;2i-1}\tilde{B}_{k;2i-1,2i+2}\tilde{V}^H_{k;2i+2} + D_{k;2i,2i}\tilde{U}_{k;2i}\tilde{B}_{k;2i,2i+2}\tilde{V}^H_{k;2i+2} \end{pmatrix} \\
&+ \begin{pmatrix} U_{k;2i-1}B_{k;2i-1,2i+1}V^H_{2i+1}\tilde{D}_{k;2i+1,2i+1} + U_{k;2i-1}B_{k;2i-1,2i+2}V^H_{k;2i+2}\tilde{D}_{k;2i+2,2i+1} & U_{k;2i-1}B_{k;2i-1,2i+1}V^H_{2i+1}\tilde{D}_{k;2i+1,2i+2} + U_{k;2i-1}B_{k;2i-1,2i+2}V^H_{k;2i+2}\tilde{D}_{k;2i+2,2i+2} \\ D_{k;2i,2i+1}\tilde{D}_{k;2i+1,2i+1} + U_{k;2i}B_{k;2i,2i+2}V^H_{k;2i+2}\tilde{D}_{k;2i+2,2i+1} & D_{k;2i,2i+1}\tilde{D}_{k;2i+1,2i+2} + U_{k;2i}B_{k;2i,2i+2}V^H_{k;2i+2}\tilde{D}_{k;2i+2,2i+2} \end{pmatrix} \\
&+ \begin{pmatrix} U_{k;2i-1}(C)R_{k;2i-1}(C)F_{k-1;i,i+1}W^H_{k;2i+1}(C) & U_{k;2i-1}(C)R_{k;2i-1}(C)F_{k-1;i,i+1}W^H_{k;2i+2}(C)V^H_{k;2i+2}(C) \\ U_{k;2i}(C)R_{k;2i}(C)F_{k-1;i,i+1}W^H_{k;2i+1}(C)V^H_{k;2i+1}(C) & U_{k;2i}(C)R_{k;2i}(C)F_{k-1;i,i+1}W^H_{k;2i+2}(C)V^H_{k;2i+2}(C) \end{pmatrix} \\
&= \begin{pmatrix} C_{k;2i-1,2i+1} & C_{k;2i-1,2i+2} \\ C_{k;2i,2i+1} & C_{k;2i,2i+2} \end{pmatrix}
\end{aligned}
$$

Figure 3.12: Derivation of recursions for $C_{k-1;i,i+1}$

Let us look at each one of these four blocks individually, beginning with the upper left block, $C_{k;2i-1,2i+1}$. We have

$$
\begin{aligned}
& C_{k-1;2i-1,2i+1} \\
= \; & D_{k;2i-1,2i}\tilde{D}_{k;2i,2i+1} + \\
& \begin{pmatrix} U_{k;2i-1} & D_{k;2i-1,2i-2}\tilde{U}_{k;2i-2} & D_{k;2i-1,2i-1}\tilde{U}_{k;2i-1} & D_{k;2i-1,2i}\tilde{U}_{k;2i} \end{pmatrix} \cdot \\
& \left(\begin{pmatrix} 0 & 0 & B_{k;2i-1,2i+1} & B_{k;2i-1,2i+2} \\ 0 & 0 & 0 & 0 \\ \tilde{B}_{k;2i-1,2i+1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + R_{k;2i-1}(C)F_{k-1;i,i+1}W^H_{k;2i+1} \right) \begin{pmatrix} \tilde{V}^H_{k;2i+1} \\ V^H_{k;2i}\tilde{D}_{k;2i,2i+1} \\ V^H_{k;2i+1}\tilde{D}_{k;2i+1,2i+1} \\ V^H_{k;2i+2}\tilde{D}_{k;2i+2,2i+1} \end{pmatrix}
\end{aligned}
$$

Then if we define

$$C_{k;i,i+2} = D_{k;i,i+1}\tilde{D}_{k;i+1,i+2} + U_{k;i}(C)F_{k;i,i+2}V^H_{k;i+2}(C) \tag{3.17}$$

The correct down-sweep recursion is

$$F_{k;2i-1,2i+1} = \begin{pmatrix} 0 & 0 & B_{k;2i-1,2i+1} & B_{k;2i-1,2i+2} \\ 0 & 0 & 0 & 0 \\ \tilde{B}_{k;2i-1,2i+1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + R_{k;2i-1}(C)F_{k-1;i,i+1}W^H_{k;2i+1}. \tag{3.18}$$

We then follow these same steps to compute all remaining $F_{k;i,j}$ shown in Figure 3.10. We must also compute the expansion coefficients $B_{k;i,j}(C)$. The set of indices that $B_{k;i,j}(C)$ need to be computed over is shown in Figure 3.13. Recall that we are recursing down $C_{k-1,i,i+1}$ (Figure 3.12). Let us now look at the upper right block of this term, $C_{k;2i-1,2i+2}$. We have

$$
\begin{aligned}
& C_{k;2i-1,2i+2} \\
= \; & \begin{pmatrix} U_{k;2i-1} & D_{k;2i-1,2i-2}\tilde{U}_{k;2i-2} & D_{k;2i-1,2i-1}\tilde{U}_{k;2i-1} & D_{k;2i-1,2i}\tilde{U}_{k;2i} \end{pmatrix} \cdot \\
& \left(\begin{pmatrix} 0 & B_{k;2i-1,2i+1} & B_{k;2i-1,2i+2} \\ 0 & 0 & 0 \\ \tilde{B}_{k;2i-1,2i+2} & 0 & 0 \\ \tilde{B}_{k;2i,2i+2} & 0 & 0 \end{pmatrix} + R_{k;2i-1}(C)F_{k-1;i,i+1}W^H_{k;2i+2} \right) \begin{pmatrix} \tilde{V}^H_{k;2i+2} \\ V^H_{k;2i+1}\tilde{D}_{k;2i+1,2i+2} \\ V^H_{k;2i+2}\tilde{D}_{k;2i+2,2i+2} \\ V^H_{k;2i+3}\tilde{D}_{k;2i+3,2i+2} \end{pmatrix}
\end{aligned}
$$

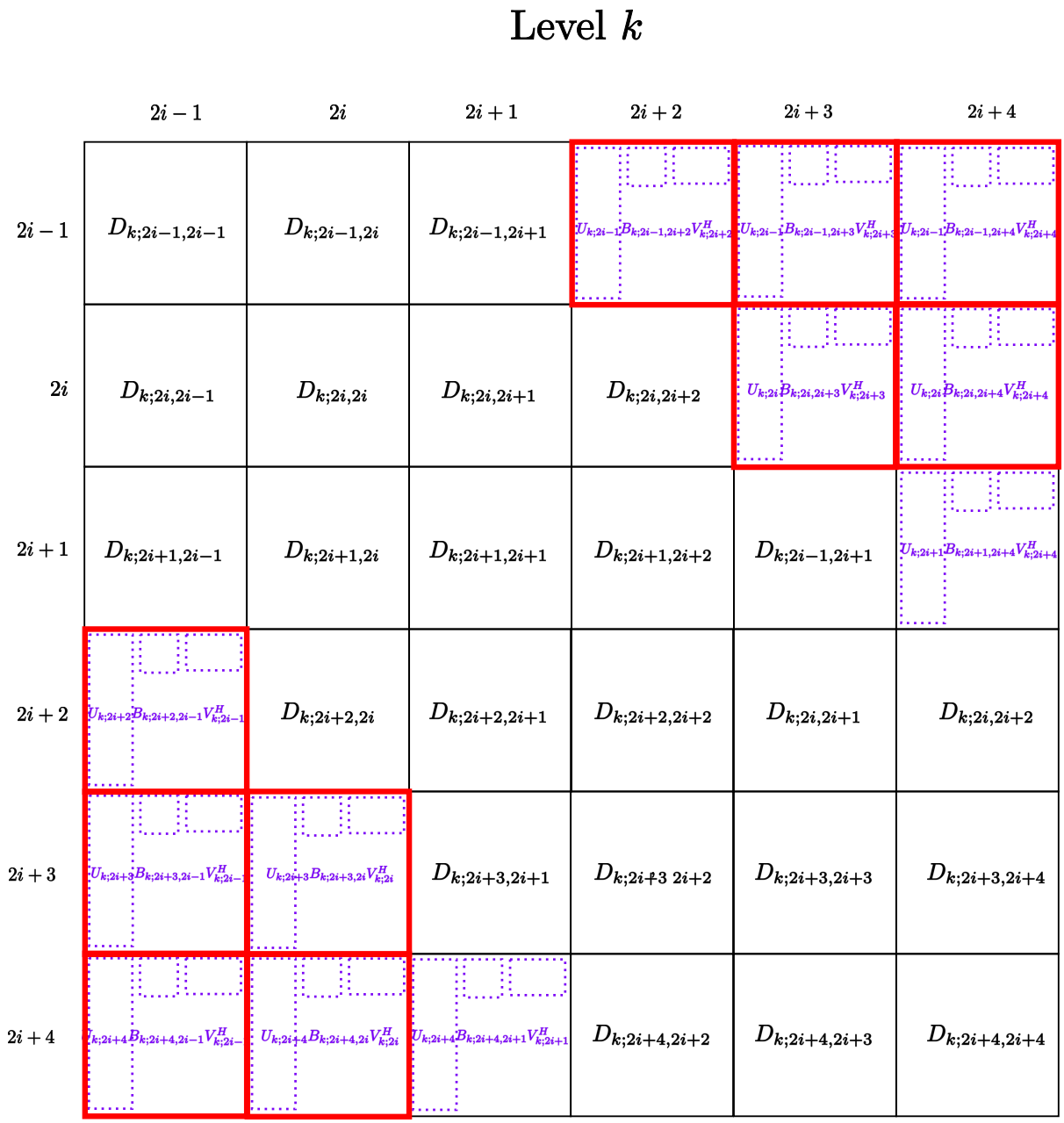


Figure 3.13: Set of indices (shown outlined in red) for which $B_{k;i,j}(C)$ must be computed

Then if we define

$$C_{k;i,i+3} \qquad U_{k;i}(C)B_{k;i,i+3}(C)V^H_{k;i+3}(C), \tag{3.19}$$

we can conclude

$$B_{k;2i-1,2i+2}(C) = \begin{pmatrix} 0 & B_{k;2i-1,2i+1} & B_{k;2i-1,2i+2} \\ 0 & 0 & 0 \\ \tilde{B}_{k;2i-1,2i+2} & 0 & 0 \\ \tilde{B}_{k;2i,2i+2} & 0 & 0 \end{pmatrix} + R_{k;2i-1}(C)F_{k-1;i,i+1}W^H_{k;2i+2}. \tag{3.20}$$

We can then follow this same procedure to compute all $B_{k;i,j}(C)$. We can similarly compute the recursions for blocks $C_{k;2i,2i+1}$ and $C_{k;2i,2i+2}$. The entire recursion for the $C_{k-1;i,i+1}$ block is shown in Figure 3.14.

$$
\begin{aligned}
C_{k-1;i,i+1} = &\begin{pmatrix} D_{k;2i-1,2i}\tilde{D}_{k;2i,2i+1} & 0 \\ D_{k;2i,2i}\tilde{D}_{k;2i,2i+1} + D_{k;2i,2i+1}\tilde{D}_{k;2i+1,2i+1} & D_{k;2i,2i+1}\tilde{D}_{k;2i+1,2i+2} \end{pmatrix} + \\
&\begin{pmatrix} \begin{pmatrix} U_{k;2i-1} & D_{k;2i-1,2i-2}\tilde{U}_{k;2i-2} & D_{k;2i-1,2i-1}\tilde{U}_{k;2i-1} & D_{k;2i-1,2i}\tilde{U}_{k;2i} \end{pmatrix} & 0 \\ 0 & \begin{pmatrix} U_{k;2i} & D_{k;2i,2i-1}\tilde{U}_{k;2i-1} & D_{k;2i,2i}\tilde{U}_{k;2i} & D_{k;2i,2i+1}\tilde{U}_{k;2i+1} \end{pmatrix} \end{pmatrix} \cdot \\
&\begin{pmatrix} \begin{pmatrix} 0 & 0 & B_{k;2i-1,2i+1} & B_{k;2i-1,2i+2} \\ 0 & 0 & 0 & 0 \\ \tilde{B}_{k;2i-1,2i+1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + R_{k;2i-1}(C)F_{k-1;i,i+1}W^H_{k;2i+1} & \begin{pmatrix} 0 & B_{k;2i-1,2i+1} & B_{k;2i-1,2i+2} \\ 0 & 0 & 0 \\ \tilde{B}_{k;2i-1,2i+2} & 0 & 0 \\ \tilde{B}_{k;2i,2i+2} & 0 & 0 \end{pmatrix} + R_{k;2i-1}(C)F_{k-1;i,i+1}W^H_{k;2i+2} \\ \begin{pmatrix} 0 & 0 & 0 & B_{k;2i,2i+2} \\ \tilde{B}_{k;2i-1,2i+1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + R_{k;2i}(C)F_{k-1;i,i+1}W^H_{k;2i+1} & \begin{pmatrix} 0 & 0 & 0 & B_{k;2i,2i+2} \\ \tilde{B}_{k;2i-1,2i+2} & 0 & 0 & 0 \\ \tilde{B}_{k;2i,2i+2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + R_{k;2i}(C)F_{k-1;i,i+1}W^H_{k;2i+2}(C) \end{pmatrix} \cdot \\
&\begin{pmatrix} \begin{pmatrix} \tilde{V}^H_{k;2i+1} \\ V^H_{k;2i}\tilde{D}_{k;2i,2i+1} \\ V^H_{k;2i+1}\tilde{D}_{k;2i+1,2i+1} \\ V^H_{k;2i+2}\tilde{D}_{k;2i+2,2i+1} \end{pmatrix} & 0 \\ 0 & \begin{pmatrix} \tilde{V}^H_{k;2i+2} \\ V^H_{k;2i+1}\tilde{D}_{k;2i+1,2i+2} \\ V^H_{k;2i+2}\tilde{D}_{k;2i+2,2i+2} \\ V^H_{k;2i+3}\tilde{D}_{k;2i+3,2i+2} \end{pmatrix} \end{pmatrix} \\
&= \begin{pmatrix} C_{k;2i-1,2i+1} & C_{k;2i-1,2i+2} \\ C_{k;2i,2i+1} & C_{k;2i,2i+2} \end{pmatrix}
\end{aligned}
$$

Figure 3.14: Derivation of recursions for $C_{k-1;i,i+1}$

3.3.5 FMM Matrix-Matrix Multiply Formula Summary

Here we will simply list all of the formulas necessary for computing the 5pt FMM representation that results from the multiplication of two 3pt FMM matrices.

Up-sweep Formulas

$$\begin{aligned} G_{k;i} &= V_{k;i}^H \tilde{U}_{k;i} \\ G_{k-1;i} &= W_{k;2i-1}^H G_{k;2i-1} \tilde{R}_{k;2i-1} + W_{k;2i}^H G_{k;2i} \tilde{R}_{k;2i} \end{aligned}$$

Column and Row Bases

$$U_{k;i}(C) = \left(\begin{array}{cccc} U_{k;i} & D_{k;i,i-1}\tilde{U}_{k;i-1} & D_{k;i,i}\tilde{U}_{k;i} & D_{k;i,i+1}\tilde{U}_{k;i+1} \end{array} \right)$$

$$V_{k;i}(C) = \left(\begin{array}{cccc} \tilde{V}_{k;i} & \tilde{D}_{k;i-1,i}^H V_{k;i-1} & \tilde{D}_{k;i,i}^H V_{k;i} & \widetilde{D}_{k;i+1,i}^H V_{k;i+1} \end{array} \right)$$

Column and Row Translation Operators

See Figure 3.15.

$$
R_{k;2i-1}(C) = \begin{pmatrix} R_{k;2i-1} & B_{k;2i-1,2i-3}G_{k;2i-3}\tilde{R}_{k;2i-3} & 0 & B_{k;2i-1,2i+1}G_{k;2i+1}\tilde{R}_{k;2i+1} + B_{k;2i-1,2i+2}G_{k;2i+2}\tilde{R}_{k;2i+2} \\ 0 & \tilde{R}_{k;2i-2} & 0 & 0 \\ 0 & 0 & \tilde{R}_{k;2i-1} & 0 \\ 0 & 0 & \tilde{R}_{k;2i} & 0 \end{pmatrix}
$$

$$
R_{k;2i}(C) = \begin{pmatrix} R_{k;2i} & B_{k;2i,2i-3}G_{k;2i-3}\tilde{R}_{k;2i-3} + B_{k;2i,2i-2}G_{k;2i-2}\tilde{R}_{k;2i-2} & 0 & B_{k;2i,2i+2}G_{k;2i+2}\tilde{R}_{k;2i+2} \\ 0 & 0 & \tilde{R}_{k;2i-1} & 0 \\ 0 & 0 & \tilde{R}_{k;2i} & 0 \\ 0 & 0 & 0 & \tilde{R}_{k;2i+1} \end{pmatrix}
$$

$$
W_{k;2i-1}(C) = \begin{pmatrix} \tilde{W}_{k;2i-1} & \tilde{B}^H_{k;2i-3,2i-1}G^H_{k;2i-3}W_{k;2i-3} & 0 & \tilde{B}^H_{k;2i+1,2i-1}G^H_{k;2i+1}W_{k;2i+1} + \tilde{B}^H_{k;2i+2,2i-1}G^H_{k;2i+2}W_{k;2i+2} \\ 0 & W_{k;2i-2} & 0 & 0 \\ 0 & 0 & W_{k;2i-1} & 0 \\ 0 & 0 & W_{k;2i} & 0 \end{pmatrix}
$$

$$
W_{k;2i}(C) = \begin{pmatrix} \tilde{W}_{k;2i} & \tilde{B}^H_{k;2i-3,2i}G^H_{k;2i-3}W_{k;2i-3} + \tilde{B}^H_{k;2i-2,2i}G^H_{k;2i-2}W_{k;2i-2} & 0 & \tilde{B}^H_{k;2i+2,2i}G^H_{k;2i+2}W_{k;2i+2} \\ 0 & 0 & W_{k;2i-1} & 0 \\ 0 & 0 & W_{k;2i} & 0 \\ 0 & 0 & 0 & W_{k;2i+1} \end{pmatrix}
$$

Figure 3.15: Column and row translation operator formulas

Down-sweep Formulas

See Figure 3.16 and Figure 3.17.

$$
\begin{aligned}
F_{1,:,:} &= (\ )_{0\times 0} \\
F_{k,2i-1,2i-1} &= \begin{pmatrix} B_{k;2i-1,2i-3}G_{k;2i-3}\tilde{B}_{k;2i-3,2i-1} + B_{k;2i-1,2i+1}G_{k;2i+1}\tilde{B}_{k;2i+1,2i-1} + B_{k;2i-1,2i+2}G_{k;2i+2}\tilde{B}_{k;2i+2,2i-1} & 0\ 0\ 0 \\ 0 & 0\ 0\ 0 \\ 0 & 0\ 0\ 0 \end{pmatrix} + R_{k;2i-1}(C)F_{k-1;i,i}W^H_{k;2i-1}(C) \\
F_{k;2i,2i} &= \begin{pmatrix} B_{k;2i,2i-3}G_{k;2i-3}\tilde{B}_{k;2i-3,2i} + B_{k;2i,2i-2}G_{k;2i-2}\tilde{B}_{k;2i-2,2i} + B_{k;2i,2i+2}G_{k;2i+2}\tilde{B}_{k;2i+2,2i} & 0\ 0\ 0 \\ 0 & 0\ 0\ 0 \\ 0 & 0\ 0\ 0 \end{pmatrix} + R_{k;2i}(C)F_{k-1;i,i}W^H_{k;2i}(C) \\
F_{k;2i-1,2i} &= \begin{pmatrix} B_{k;2i-1,2i-3}G_{k;2i-3}\tilde{B}_{k;2i-3,2i} + B_{k;2i-1,2i+2}G_{k;2i+2}\tilde{B}_{k;2i+2,2i} & 0\ 0\ B_{k;2i-1,2i+1} \\ \tilde{B}_{k;2i-2,2i} & 0\ 0\ 0 \\ 0 & 0\ 0\ 0 \end{pmatrix} + R_{k;2i-1}(C)F_{k-1;i,i}W^H_{k;2i}(C) \\
F_{k;2i,2i-1} &= \begin{pmatrix} B_{k;2i,2i-3}G_{k;2i-3}\tilde{B}_{k;2i-3,2i-1} + B_{k;2i,2i+2}G_{k;2i+2}\tilde{B}_{k;2i+2,2i-1} & B_{k;2i,2i-2}\ 0\ 0 \\ 0 & 0\ 0\ 0 \\ 0 & 0\ 0\ 0 \end{pmatrix} + R_{k;2i}(C)F_{k-1;i,i}W^H_{k;2i-1}(C) \\
F_{k;2i-1,2i+1} &= \begin{pmatrix} 0\ 0\ 0 & B_{k;2i-1,2i+1}\ B_{k;2i-1,2i+2} \\ 0\ 0\ 0 & 0\ 0 \\ \tilde{B}_{k;2i-1,2i+1}\ 0\ 0 & 0\ 0 \\ 0\ 0\ 0 & 0\ 0 \end{pmatrix} + R_{k;2i-1}(C)F_{k-1;i,i+1}W^H_{k;2i+1}
\end{aligned}
$$

Figure 3.16: Down-sweep formulas

$$
\begin{aligned}
F_{k;2i,2i+2} &= \begin{pmatrix} 0 & 0 & B_{k;2i,2i+2} & 0 \\ \tilde{B}_{k;2i-1,2i+2} & 0 & 0 & 0 \\ \tilde{B}_{k;2i,2i+2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + R_{k;2i}(C) F_{k-1;i,i+1} W_{k;2i+2}^H(C) \\
F_{k;2i,2i+1} &= \begin{pmatrix} 0 & 0 & 0 & B_{k;2i,2i+2} \\ \tilde{B}_{k;2i-1,2i+1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + R_{k;2i}(C) F_{k-1;i,i+1} W_{k;2i+1}^H(C) \\
F_{k;2i+1,2i-1} &= \begin{pmatrix} 0 & 0 & B_{k;2i+1,2i-1} & 0 \\ 0 & 0 & 0 & 0 \\ \tilde{B}_{k;2i+1,2i-1} & 0 & 0 & 0 \\ \tilde{B}_{k;2i+2,2i-1} & 0 & 0 & 0 \end{pmatrix} + R_{k;2i+1}(C) F_{k-1;i+1,i} W_{k;2i-1}^H(C) \\
F_{k;2i+1,2i} &= \begin{pmatrix} 0 & B_{k;2i+1,2i-1} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \tilde{B}_{k;2i+2;2i} & 0 & 0 & 0 \end{pmatrix} + R_{k;2i+1}(C) F_{k-1;i+1,i} W_{k;2i}^H(C) \\
F_{k;2i+2,2i} &= \begin{pmatrix} 0 & B_{k;2i+2,2i-1} & B_{k;2i+2;2i} & 0 \\ 0 & 0 & 0 & 0 \\ \tilde{B}_{k;2i+2;2i} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + R_{k;2i+2}(C) F_{k-1;i+1,i} W_{k;2i}^H(C)
\end{aligned}
$$

Figure 3.17: Down-sweep formulas

Expansion Coefficients

See Figure 3.18 and Figure 3.19.

$$
\begin{aligned}
B_{k;2i-1,2i+2}(C) &= \begin{pmatrix} 0 & B_{k;2i-1,2i+1} & B_{k;2i-1,2i+2} & 0 \\ 0 & 0 & 0 & 0 \\ \tilde{B}_{k;2i-1,2i+2} & 0 & 0 & 0 \\ \tilde{B}_{k;2i,2i+2} & 0 & 0 & 0 \end{pmatrix} + R_{k;2i-1}(C) F_{k-1;i,i+1} W_{k;2i+2}^H(C) \\
B_{k;2i+2,2i-1}(C) &= \begin{pmatrix} 0 & 0 & B_{k;2i+2,2i-1} & B_{k;2i+2,2i} \\ \tilde{B}_{k;2i+1,2i-1} & 0 & 0 & 0 \\ \tilde{B}_{k;2i+2,2i-1} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + R_{k;2i+2}(C) F_{k-1;i+1,i} W_{k;2i-1}^H(C) \\
B_{k;2i-1,2i+3}(C) &= \begin{pmatrix} B_{k;2i-1,2i+1} \tilde{G}_{k;2i+1} \tilde{B}_{k;2i+1,2i+3} & B_{k;2i-1,2i+2} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + R_{k;2i-1}(C) F_{k-1;i,i+2} W_{k-1;2i+3}^H(C) \\
B_{k;2i-1,2i+4}(C) &= \begin{pmatrix} B_{k;2i-1,2i+1} \tilde{G}_{k;2i+1} \tilde{B}_{k;2i+1,2i+4} + B_{k;2i-1,2i+2} G_{k;2i+2} \tilde{B}_{k;2i+1,2i+4} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} + R_{k;2i-1}(C) F_{k-1;i,i+2} W_{k;2i+4}^H(C) \\
B_{k;2i,2i+3}(C) &= \begin{pmatrix} 0 & B_{k;2i,2i+2} & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ \tilde{B}_{k;2i+1,2i+3} & 0 & 0 & 0 \end{pmatrix} + R_{k;2i}(C) F_{k-1;i,i+2} W_{k;2i+3}^H(C)
\end{aligned}
$$

Figure 3.18: Expansion Coefficients

$$
\begin{aligned}
B_{k;2i,2i+4}(C) &= \begin{pmatrix} B_{k;2i,2i+2}G_{k;2i+2}\tilde{B}_{k;2i+2,2i+4} & 0\ 0\ 0 \\ 0 & 0\ 0\ 0 \\ 0 & 0\ 0\ 0 \\ \tilde{B}_{k;2i+1,2i+4} & 0\ 0\ 0 \end{pmatrix} + R_{k;2i}(C)F_{k-1;i,i+2}W^H_{k;2i+4}(C) \\
B_{k;2i+3,2i-1}(C) &= \begin{pmatrix} B_{k;2i+3,2i+1}G_{k;2i+1}\tilde{B}_{k;2i+1,2i-1} & 0\ 0\ 0 \\ \tilde{B}_{k;2i+2,2i-1} & 0\ 0\ 0 \\ 0 & 0\ 0\ 0 \\ 0 & 0\ 0\ 0 \end{pmatrix} + R_{k;2i+3}(C)F_{k-1;i+2,i}W^H_{k;2i-1}(C) \\
B_{k;2i+3,2i}(C) &= \begin{pmatrix} 0 & 0\ 0\ B_{k;2i+3,2i+1} \\ \tilde{B}_{k;2i+2;2i} & 0\ 0\ 0 \\ 0 & 0\ 0\ 0 \\ 0 & 0\ 0\ 0 \end{pmatrix} + R_{k;2i+3}(C)F_{k;i+2,i}W^H_{k;2i}(C) \\
B_{k,2i+4,2i-1}(C) &= \begin{pmatrix} B_{k;2i+4,2i+1}G_{k;2i+1}\tilde{B}_{k;2i+1,2i-1} + B_{k;2i+4,2i+2}G_{k;2i+2}\tilde{B}_{k;2i+2,2i-1} & 0\ 0\ 0 \\ 0 & 0\ 0\ 0 \\ 0 & 0\ 0\ 0 \\ 0 & 0\ 0\ 0 \end{pmatrix} + R_{k;2i+4}(C)F_{k-1;i+2,i}W^H_{k;2i-1}(C) \\
B_{k;2i+4,2i}(C) &= \begin{pmatrix} B_{k;2i+4,2i+2}G_{k;2i+2}\tilde{B}_{k;2i+2;2i} & 0\ 0\ B_{k;2i+4,2i+1} \\ 0 & 0\ 0\ 0 \\ 0 & 0\ 0\ 0 \\ 0 & 0\ 0\ 0 \end{pmatrix} + R_{k;2i+4}(C)F_{k-1;i+2,i}W^H_{k;2i}(C)
\end{aligned}
$$

Figure 3.19: Expansion Coefficients

Dense Blocks

$$
\begin{aligned}
D_{k;i,i} &= D_{k;i,i-1}\tilde{D}_{k;i-1,i} + D_{k;i,i}\tilde{D}_{k;i,i} + D_{k;i,i+1}\tilde{D}_{k;i+1,i} + U_{k;i}(C)F_{k;i,i}V^H_{k;i}(C) \\
D_{k;i,i+1} &= D_{k;i,i}\tilde{D}_{k;i,i+1} + D_{k;i,i+1}\tilde{D}_{k;i+1,i+1} + U_{k;i}(C)F_{k;i,i+1}V^H_{k;i+1}(C) \\
D_{k;i+1,i} &= D_{k;i+1,i}\tilde{D}_{k;i,i} + D_{k;i+1,i+1}\tilde{D}_{k;i+1,i} + U_{k;i+1}(C)F_{k;i+1,i}V^H_{k;i}(C) \\
D_{k;i,i+2} &= D_{k;i,i+1}\tilde{D}_{k;i+1,i+2} + U_{k;i}(C)F_{k;i,i+2}V^H_{k;i+2}(C) \\
D_{k;i+2,i} &= D_{k;i+2,i+1}\tilde{D}_{k;i+1,i} + U_{k;i+2}(C)F_{k;i+2,i}V^H_{k;i}(C)
\end{aligned}
$$

3.4 Experimental Results

Experiments were performed in Julia[1] on a machine with a 2.8 GHz Intel i7-7700 processor with 32 GB of RAM. We chose Julia because of its superior memory handling capabilities (garbage-collection) as well as its high performance (when compared with programs such as Matlab), while still having the ease of use of a scripting language[26]. As can be seen in Table 3.1 the time for the matrix-matrix multiply scales linearly with n as compared with the standard multiply[2]. We show a relative error[3] for the FMM multiply alone on the order of 1e−15 with cpu run-times as shown in Table 3.1.

[1] Code is given in appendix A.1

[2] When we refer to the standard multiply, we simply store each matrix A and $\tilde{A}$ in memory and call $A * \tilde{A}$.

[3] We isolated the relative error for the matrix-matrix multiply by removing the error from the FMM construction algorithm.

n	time(s)	
	FMM Multiply	Standard Multiply
256	.0042	.0004
512	0.007	0.014
1024	0.013	0.035
2048	0.116	0.129
4096	0.231	1.03
8192	0.348	7.29
16384	0.618	51.5
32768	1.23	Out of Memory
65536	1.60	Out of Memory
131072	2.87	Out of Memory

Table 3.1: CPU run-times for the FMM matrix-matrix multiply and standard multiply

Chapter 4

Conclusions

In Chapter 2 we focused on the case where the partition tree corresponding to the rows of a matrix is the same as the tree which corresponds to its columns, though we can generalize the algorithm presented here to more complicated partition trees, including FMM representations. The open question remains as to whether a linear memory algorithm exists which does not give up the $O(n^2)$ flop constraint.

In Chapter 3 we have covered a linear time ($O(n)$ flop) algorithm for the 1D FMM matrix-matrix multiply. Based on our results, we conjecture that the inverse of FMM is not itself FMM. There is a focus in the current literature on the 3pt FMM in regard to approximating the inverse. However, the work presented here brings focus to the existence of $k$pt FMM representations. The natural conjecture arises that some classes of matrices might be better represented by a $k$pt representation, rather than a 3pt representation. Several further questions arise from the results of this work, which we list here:

- For what class of matrices would a 3pt representation be good enough to approximate the inverse? Clearly the special case of computing the inverse using the matrix-matrix multiply that we illustrated in chapter 3 gives us one example for which this would be true.

- Conversely, does there exist some family of FMM representations where no $k$pt representation yields a good approximation to the inverse? We conjecture that this would seem to be true if we consider that we lose a factor of $\log^2 n$ in rank for each multiply when we convert from 5pt to 3pt. Once k approaches $\sqrt{n}$ it would become very inefficient. In this case perhaps we should consider Graph Induced Rank Structure (GIRS)[27] as a way to think about the inverse of FMM instead.

- Are there conditions under which a $k$pt FMM representation could be good enough to approximate an inverse?

- Are there families of matrices where a $k$pt FMM representation gives a good approximation to the inverse, but a $(k-2)$pt representation does not? What is lost in terms of accuracy and what is the tradeoff in extra memory storage when choosing between a $k$pt and $(k-2)$pt FMM representation in general?

We propose that future work should consist of the derivation of these matrix-matrix multiply formulas for both 2D and 3D FMM as well as derivation of master formulas for the matrix-matrix multiply for a 3pt times an $k$pt FMM representation. Such master formulas, if they can be derived, will dictate what the formulas for the matrix-matrix multiply will look like for any choice of k (odd), and would give further insight into the structure of the FMM inverse itself.

Appendix A

FMM/HSS Julia Codes

A.1 HSS Julia Codes

A.1.1 HSS Construction

```
function Gen_HSS_Memory_Efficient(tree,r)
#This is a memory efficient, two pass algorithm that takes in a Tree
#which is a partition tree for a given matrix which is described by the
#function, f, below and returns its corresponding Hss representation.
#Computation of U,R,V,W is done via deepest first post-ordering.  B
#matrices are computed by in descending order (starting at root and finishing
#at the child).  Relavant matrices are then multiplied and added
#from child to root in order to compute each B matrix.  (Bottom Up Routine
# to compute B).
#INPUT:             r        (Int64) largest allowable rank of hankel blocks -
#                            this determines the amount of compression, and
#                            should be compatible with your input tree.
#                            (if the maximum allowable rank is p, then the
#                            tree partitions should be no smaller than 3p)
#                            Corresponding singular values below this rank
#                            will be dropped.
#                   tree     (Tree) contains partition dimensions for
#                            each subdivision of the input matrix, for each
#                            of which there are a left and right child
#                            structure
#OUTPUT:            hss     (Hss) contains matrices (Us, Vs, Bs Rs and
#                           Ws) that compose the hss representation of the
#                            input matrix, in addition to the partition
#                            dimensions
#                   peakMem (Array{Int64}(1)) first entry is a memory counter
#                            and second entry is the maximum memory
# Author: Kristen Lessel - June 2015
```

```
    N = tree.m;
    pm_count = 0;
    pm_max = 0;
    peakMem = [pm_count,pm_max];
    hss, dummy1, dummy2, peakMem = HSS_Basis_TranslationOp(tree,N,r,peakMem);
    row_idx = 1;
    col_idx = 1;
    hss, peakMem = HSS_Expansion_Coeffs_Diag(hss,row_idx,col_idx,N,peakMem);
    hss, peakMem
end

function HSS_Basis_TranslationOp(tree::Tree,N::Int64,r::Int64,peakMem::Array{Int64,1})
#Computes U's, V's, R's, W's and D's of HSS structure, and stores these
#heirarchically
#INPUT:             N       (Int64) grid size
#                   rank    (Int64) largest allowable rank of hankel blocks -
#                           this determines the amount of compression, and
#                           should be compatible with your input tree.
#                           (if the maximum allowable rank is p, then the
#                           tree partitions should be no smaller than 3p)
#                           Corresponding singular values below this rank
#                           will be dropped.
#                   tree    (Union(Node_Spine,Leaf_Spine))
#                           contains partition dimensions for each
#                           subdivision of the input matrix, for each
#                           of which there are a left and right child
#                           structure
#OUTPUT:            hss     (Union(Node,Leaf)) contains matrices
#                           (U, V, R and W) that compose the
#                           hss representation of the input matrix, in
#                           addition to the partition dimensions
#                   rH      (Array{Float64,2}) row hankel block with 'current' U
#                           removed
#                   cH      (Array{Float64,2}) column hankel block with 'current' V'
#                           removed
  if isa(tree,Leaf_Spine)
      m = tree.m;
      d = tree.d;

      #generate indices for currrent diagonal block
      m_idx = collect(d:(d+m)-1);
      #generate indices for current off diagonal hankel block
      butm_idx = [1:m_idx[1]-1; m_idx[end]+1:N];

      #function call to generate lowest level row and column hankel blocks
      rH2 = f(m_idx,butm_idx,N);
      cH2= f(butm_idx,m_idx,N);
      #keep track of peak memory
      count = 2*length(m_idx)*length(butm_idx);
      peakMem[1] = peakMem[1]+count;
      peakMem[2] = max(peakMem[1],peakMem[2]);
```

```
    #take svds of upper row/left column and lower row/right column hankel blocks
    Ur, Rr, Vr = svd_ranktol(rH2,r);
    Uc, Rc, Vc = svd_ranktol(cH2,r);
    #keep track of peak memory
    count = length(Ur)+ length(Vc)+length(Rr)+length(Vr)+length(Uc)+length(Rc);
    peakMem[1] = peakMem[1]+count;
    peakMem[2] = max(peakMem[1],peakMem[2]);

    #keep track of peak memory
    count = length(rH2)+length(cH2);
    peakMem[1] = peakMem[1]-count;
    peakMem[2] = max(peakMem[1],peakMem[2]);

    rH2 = Array{Float64}(undef,0);
    cH2 = Array{Float64}(undef,0);

    #D = f(m_idx,m_idx,N)
    leaf = Leaf(m,m,d,tree.depth,Ur,Vc,f(m_idx,m_idx,N));

    rH = diagm(Rr)*Vr';
    cH = Uc*diagm(Rc);

    count = length(Rr)+length(Rc);
    peakMem[1] = peakMem[1]-count;
    peakMem[2] = max(peakMem[1],peakMem[2]);

    leaf, rH, cH, peakMem

else #If not a Leaf

    #Descend into the tree, depth first
    if tree.treeL.depth >= tree.treeR.depth #left node deeper than right
        #left recursive call
        hssL, upperRow, leftCol, peakMem = HSS_Basis_TranslationOp(tree.treeL,N,r,peakMem);

        #right recursive call
        hssR, lowerRow, rightCol, peakMem = HSS_Basis_TranslationOp(tree.treeR,N,r,peakMem);
    else #right node deeper than left
        #right recursive call
        hssR, lowerRow, rightCol, peakMem = HSS_Basis_TranslationOp(tree.treeR,N,r,peakMem);

        #left recursive call
        hssL, upperRow, leftCol, peakMem = HSS_Basis_TranslationOp(tree.treeL,N,r,peakMem);
    end

    #starting index (row/col value) of its current diagonal block.
    d = tree.d;

    #indexes of rol/col hankel blocks excluding corresponding part of diagonal block
    uR_idx = [1:(d-1); (d+hssR.m):(N-hssL.m)];
    lR_idx = [1:(d-1); (d+hssL.m):(N-hssR.m)];
```

```
rH_top = upperRow[:, uR_idx];
rH_bottom = lowerRow[:, lR_idx];
cH_left = leftCol[uR_idx,:];
cH_right = rightCol[lR_idx,:];
#keep track of peak memory
count = length(rH_top)+ length(rH_bottom)+length(cH_left)+length(cH_right);
peakMem[1] = peakMem[1]+count;
peakMem[2] = max(peakMem[1],peakMem[2]);

#merge remainder of Hankel blocks from children
rH2 = [rH_top; rH_bottom];
cH2 = [cH_left cH_right];

count = length(rH_top)+ length(rH_bottom)+length(cH_left)+length(cH_right);
peakMem[1] = peakMem[1]-count;
peakMem[2] = max(peakMem[1],peakMem[2]);

#take svd of remaining portions of blocks to determine Rs and Ws
Ur, Sr, Vr = svd_ranktol(rH2,r);
Uc, Sc, Vc = svd_ranktol(cH2,r);
#keep track of peak memory
count = length(Ur) +length(Sr) +length(Vr) + length(Uc) +length(Sc) +length(Vc)
peakMem[1] = peakMem[1]+count;
peakMem[2] = max(peakMem[1],peakMem[2]);

#keep track of peak memory %just added 10/8/15
count = length(rH2)+length(cH2);
peakMem[1] = peakMem[1]-count;
peakMem[2] = max(peakMem[1],peakMem[2]);

rH2 = Array{Float64}(undef,0);
cH2 = Array{Float64}(undef,0);

#partition Us to get R's, partition V's to get Ws
Rl = Ur[1:size(upperRow,1),:];
Rr = Ur[size(upperRow,1)+1:end,:];
Wl = Vc[1:size(leftCol,2),:];
Wr = Vc[size(leftCol,2)+1:end,:];
#keep track of peak memory
count = length(Rl)+length(Rr)+length(Wl)+length(Wr);
peakMem[1] = peakMem[1]+count;
peakMem[2] = max(peakMem[1],peakMem[2]);

count = length(Ur) +length(Vc);
peakMem[1] = peakMem[1]-count;
peakMem[2] = max(peakMem[1],peakMem[2]);


hss = Node(hssL,hssR,tree.m,tree.m,tree.depth,d,Array{Float64,2}(undef,0,0),Array{Float64,2}(
    undef,0,0),Rl,Rr,Wl,Wr);
#return relavant portions of hankel blocks
rH = diagm(Sr)*Vr';
```

```
      cH = Uc*diagm(Sc);

      #keep track of peak memory
      count = 2*r;
      peakMem[1] = peakMem[1]-count;
      peakMem[2] = max(peakMem[1],peakMem[2]);


      hss, rH, cH, peakMem
  end
end


function HSS_Expansion_Coeffs_Diag(hss::Hss,row_idx::Int64,col_idx::Int64,N::Int64,peakMem::Array{
    Int64,1})
#Recursively computes Expansion Coefficients (B's) of HSS structure for the
#upper left and lower right block of the current node
#   -------
#  \ x \   \
#  \--------\
#  \   \ x \
#   -------
#INPUT:              hss     (Union(Node,Leaf)) branch of current node,
#                    row_idx (Int64) row index of current node
#                    col_idx (Int64) col index of current node
#                    N       (Int64) grid size
#OUTPUT:             hss     (Union(Node,Leaf)) contains matrices (U, V, B R and
#                            W) that compose the HSS representation of the
#                            input tree for a corresponding matrix, as well
#                            as corresponding partition dimensions and
#                            depth for each node

    if isa(hss.treeL,Leaf) && isa(hss.treeR,Leaf) #both nodes are leaves
        Au = Array{Float64}(undef,2);
        Bu = Array{Float64}(undef,2);
        Al = Array{Float64}(undef,2);
        Bl = Array{Float64}(undef,2);

        col_idx = col_idx + hss.treeL.m;
        m_idx = collect(row_idx:(row_idx+hss.treeL.m-1));
        butm_idx = collect(col_idx:(col_idx+hss.treeR.m-1));

        Au = f(m_idx,butm_idx,N);
        Bu = hss.treeL.U'*Au*hss.treeR.V;
        Au =  Array{Float64}(undef,0);
        #keep track of peak memory
        count = length(m_idx)*length(butm_idx) +  length(Bu);
        peakMem[1] = peakMem[1]+count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        count = length(m_idx)*length(butm_idx);
        peakMem[1] = peakMem[1]-count;
```

```
        peakMem[2] = max(peakMem[1],peakMem[2]);


        Al = f(butm_idx,m_idx,N);
        Bl = hss.treeR.U'*Al*hss.treeL.V;
        Al =  Array{Float64}(undef,0);
        #keep track of peak memory
        count = length(m_idx)*length(butm_idx) +  length(Bl);
        peakMem[1] = peakMem[1]+count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        count = length(m_idx)*length(butm_idx);
        peakMem[1] = peakMem[1]-count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        hss.Bu = Bu;
        hss.Bl = Bl;
        hss, peakMem

    elseif !isa(hss.treeL,Leaf) && !isa(hss.treeR, Leaf) #neither node is a leaf

        #Calculate B's for off-diagonal blocks
        col_idx = col_idx + hss.treeL.m;
        Bu, Bl, peakMem = HSS_Expansion_Coeffs_OffDiag(hss.treeL,hss.treeR,row_idx,col_idx,N,peakMem);
        col_idx = col_idx - hss.treeL.m;

        hss.Bl = Bl;
        hss.Bu = Bu;

        #Calculate B's for upper left diagonal block
        hssL, peakMem = HSS_Expansion_Coeffs_Diag(hss.treeL,row_idx,col_idx,N,peakMem);

        #Calculate B's for lower right diagonal block
        row_idx = row_idx + hss.treeL.m;
        col_idx = col_idx + hss.treeL.m;
        hssR, peakMem = HSS_Expansion_Coeffs_Diag(hss.treeR,row_idx,col_idx,N,peakMem);

        hss.treeL = hssL;
        hss.treeR = hssR;
        hss, peakMem

    elseif isa(hss.treeL,Leaf) && !isa(hss.treeR, Leaf) #left node is a leaf, right node is not
        bu_l = Array{Float64}(undef,2);
        bu_r = Array{Float64}(undef,2);
        bl_l = Array{Float64}(undef,2);
        bl_r = Array{Float64}(undef,2);

        #Calculate B's for off-diagonal blocks

        #left block of upper right corner and upper block of lower left corner
        #(these are computed in the same pass)
        #    ---------------
```

```
#   \       \   \   \
#   \       \ x \   \
#   \       \   \   \
#   \---------------\
#   \   x   \   \   \
#   \-------\---\---\
#   \       \   \   \
#    ---------------
#           (9)

col_idx = col_idx + hss.treeL.m;
Bu_l, Bl_l, peakMem = HSS_Expansion_Coeffs_OffDiag(hss.treeL,hss.treeR.treeL,row_idx,col_idx,N
    ,peakMem);
bu_l = Bu_l*hss.treeR.Wl;
bl_l = hss.treeR.Rl'*Bl_l;
#keep track of peak memory
count = length(bu_l)+length(bl_l);
peakMem[1] = peakMem[1]+count;
peakMem[2] = max(peakMem[1],peakMem[2]);

count = length(Bu_l) + length(Bl_l); #gives number of elements
peakMem[1] = peakMem[1]-count;
peakMem[2] = max(peakMem[1],peakMem[2]);

Bu_l =  Array{Float64}(undef,0);
Bl_l =  Array{Float64}(undef,0);

#right block of upper right corner and lower block of lower left corner
#(these are computed in the same pass)
#    ---------------
#   \       \   \   \
#   \       \   \ x \
#   \       \   \   \
#   \---------------\
#   \       \   \   \
#   \-------\---\---\
#   \   x   \   \   \
#    ---------------
#           (10)

col_idx = col_idx + hss.treeR.treeL.m;
Bu_r, Bl_r, peakMem = HSS_Expansion_Coeffs_OffDiag(hss.treeL,hss.treeR.treeR,row_idx,col_idx,N
    ,peakMem);
col_idx = col_idx - hss.treeL.m - hss.treeR.treeL.m;
bu_r = Bu_r*hss.treeR.Wr;
bl_r = hss.treeR.Rr'*Bl_r;
#keep track of peak memory
count = length(bu_r)+length(bl_r);
peakMem[1] = peakMem[1]+count;
peakMem[2] = max(peakMem[1],peakMem[2]);

count = length(Bu_r)+length(Bl_r);
```

```
        peakMem[1] = peakMem[1]-count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        Bu_r = Array{Float64}(undef,0);
        Bl_r = Array{Float64}(undef,0);

        hss.Bl = bl_l + bl_r;
        hss.Bu = bu_l + bu_r;

        count = length(bl_l)+length(bl_r)+length(bu_l)+length(bu_r);
        peakMem[1] = peakMem[1]-count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        bl_l = Array{Float64}(undef,0);
        bl_r = Array{Float64}(undef,0);
        bu_l = Array{Float64}(undef,0);
        bu_r = Array{Float64}(undef,0);

        #Calculate B's for lower right diagonal block upper right
        #block of lower right corner and lower left block of lower
        #right corner
        #    ---------------
        #   \       \   \   \
        #   \       \   \   \
        #   \       \   \   \
        #   \---------------\
        #   \       \   \ x \
        #   \-------\---\---\
        #   \       \ x \   \
        #    ---------------
        #           (11)

        row_idx = row_idx + hss.treeL.m;
        col_idx = col_idx + hss.treeL.m;
        hssR, peakMem = HSS_Expansion_Coeffs_Diag(hss.treeR,row_idx,col_idx,N,peakMem);

        hss.treeR = hssR;
        hss, peakMem

    elseif !isa(hss.treeL,Leaf) && isa(hss.treeR, Leaf) #right node is a leaf, left node is not
        bu_l = Array{Float64}(undef,2);
        bu_r = Array{Float64}(undef,2);
        bl_l = Array{Float64}(undef,2);
        bl_r = Array{Float64}(undef,2);

        #Calculate B's for off-diagonal blocks

        #upper block of upper right corner and left block of lower left corner
        #(these are computed in the same pass)
        #    ---------------
        #   \   \   \   x   \
        #   \---\---\-------\
```

```
#   \   \   \       \
#   \---------------\
#   \   \   \       \
#   \ x \   \       \
#   \   \   \       \
#    ---------------
#          (12)

col_idx = col_idx + hss.treeL.m;
Bu_l, Bl_l, peakMem = HSS_Expansion_Coeffs_OffDiag(hss.treeL.treeL,hss.treeR,row_idx,col_idx,N
    ,peakMem);
bu_l = hss.treeL.Rl'*Bu_l;
bl_l = Bl_l*hss.treeL.Wl;
#keep track of peak memory
count = length(bu_l)+length(bl_l);
peakMem[1] = peakMem[1]+count;
peakMem[2] = max(peakMem[1],peakMem[2]);

count = length(Bu_l)+length(Bl_l);
peakMem[1] = peakMem[1]-count;
peakMem[2] = max(peakMem[1],peakMem[2]);

Bu_l = Array{Float64}(undef,0);
Bl_l = Array{Float64}(undef,0);

#lower block of upper right corner and right block of lower left corner
#(these are computed in the same pass)
#    ---------------
#   \   \   \       \
#   \---\---\-------\
#   \   \   \   x   \
#   \---------------\
#   \   \   \       \
#   \   \ x \       \
#   \   \   \       \
#    ---------------
#          (13)

row_idx  = row_idx + hss.treeL.treeL.m;
Bu_r, Bl_r, peakMem = HSS_Expansion_Coeffs_OffDiag(hss.treeL.treeR,hss.treeR,row_idx,col_idx,N
    ,peakMem);
row_idx = row_idx - hss.treeL.treeL.m;
col_idx = col_idx - hss.treeL.m;
bu_r = hss.treeL.Rr'*Bu_r;
bl_r = Bl_r*hss.treeL.Wr;
#keep track of peak memory
count = length(bu_r)+length(bl_r);
peakMem[1] = peakMem[1]+count;
peakMem[2] = max(peakMem[1],peakMem[2]);

count = length(Bu_r)+length(Bl_r);
peakMem[1] = peakMem[1]-count;
```

```
        peakMem[2] = max(peakMem[1],peakMem[2]);

        Bu_r = Array{Float64}(undef,0);
        Bl_r = Array{Float64}(undef,0);

        hss.Bl = bl_l + bl_r;
        hss.Bu = bu_l + bu_r;

        #keep track of peak memory
        count = length(bl_l)+length(bl_r)+length(bu_l)+length(bu_r);
        peakMem[1] = peakMem[1]-count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        bl_l = Array{Float64}(undef,0);
        bl_r = Array{Float64}(undef,0);
        bu_l = Array{Float64}(undef,0);
        bu_r = Array{Float64}(undef,0);

        #Calculate B's for upper right diagonal block

        #lower left block of upper left corner and upper right block of upper
        # left corner
        #    ---------------
        #   \   \ x \       \
        #   \---\---\-------\
        #   \ x \   \       \
        #   \---------------\
        #   \   \   \       \
        #   \   \   \       \
        #   \   \   \       \
        #    ---------------
        #           (14)

        hssL, peakMem = HSS_Expansion_Coeffs_Diag(hss.treeL,row_idx,col_idx,N,peakMem);

        hss.treeL = hssL;
        hss, peakMem
    end
end

function HSS_Expansion_Coeffs_OffDiag(treeL::Hss,treeR::Hss,row_idx::Int64,col_idx::Int64,N::Int64,
    peakMem::Array{Int64,1})
#Computes Expansion Coefficients (B's) of HSS structure for the upper right
#and lower left block of the current node
#   -------
#  \   \ x \
#  \-------\
#  \ x \   \
#   -------
#INPUT:             treeL   (Union(Node,Leaf)) left branch of current node
#                   treeR   (Union(Node,Leaf)) right branch of current node
#                   row_idx (Int64) row index of current node
```

```
#                   col_idx (Int64) col index of current node
#                   N       (Int64) grid size
#OUTPUT:            Bu      (Array{Float62,2}) Expansion coefficient matrix B at the
#                           current level which corresponds to the upper
#                           right block
#                   Bl      (Array{Float62,2}) Expansion coefficient matrix B at the
#                           current level which corresponds to the lower
#                           left block

    Au = Array{Float64,2}(undef);
    Bu = Array{Float64,2}(undef);
    Al = Array{Float64,2}(undef);
    Bl = Array{Float64,2}(undef);

    bu_l = Array{Float64,2}(undef);
    bu_r = Array{Float64,2}(undef);
    bl_l = Array{Float64,2}(undef);
    bl_r = Array{Float64,2}(undef);
    if isa(treeL, Leaf) && isa(treeR, Leaf) #If node is a leaf
        m_idx = collect((row_idx):(row_idx+treeL.m-1));
        butm_idx = collect(col_idx:(col_idx+treeR.m-1));

        Au = f(m_idx,butm_idx,N);
        Bu = treeL.U'*Au*treeR.V;
        #keep track of peak memory
        count = length(Au)+length(Bu);
        peakMem[1] = peakMem[1]+count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        count = length(Au);
        peakMem[1] = peakMem[1]-count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        Au = Array{Float64}(undef,0);

        Al = f(butm_idx,m_idx,N);
        Bl = treeR.U'*Al*treeL.V;
        #keep track of peak memory
        count = length(Al)+length(Bl);
        peakMem[1] = peakMem[1]+count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        count = length(Al);
        peakMem[1] = peakMem[1]-count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        Al = Array{Float64}(undef,0);

        Bu, Bl, peakMem

    elseif !isa(treeL, Leaf) && !isa(treeR,Leaf)  #If neither node is a leaf
        #COMPUTE EXPANSION COEFFICIENTS
```

```
#upper left corner of both upper right and lower left blocks
#(these are computed at the same time. One block will be of
#dimension MxN, the other will always be of dimension NxM)
#
#    ----------------
#   \        \ x \<--\-- MxN
#   \         --------\
#   \        \    \    \
#   \----------------\
#   \ x \<--\--------\-- NxM
#   \-------\        \
#   \    \   \        \
#    ----------------
#           (1)

Bu_ll, Bl_ll, peakMem = HSS_Expansion_Coeffs_OffDiag(treeL.treeL,treeR.treeL,row_idx,col_idx,N
    ,peakMem);
bu_ll = treeL.Rl'*Bu_ll*treeR.Wl;
bl_ll = treeR.Rl'*Bl_ll*treeL.Wl;
#keep track of peak memory
count = length(bu_ll)+length(bl_ll);
peakMem[1] = peakMem[1]+count;
peakMem[2] = max(peakMem[1],peakMem[2]);

count = length(Bu_ll)+length(Bl_ll);
peakMem[1] = peakMem[1]-count;
peakMem[2] = max(peakMem[1],peakMem[2]);

Bu_ll = Array{Float64}(undef,0);
Bl_ll = Array{Float64}(undef,0);

#Lower left corner lower left block, and also the upper right corner of
#the upper right block
#    ----------------
#   \        \   \ x \
#   \        \---\---\
#   \        \   \   \
#   \----------------\
#   \   \    \       \
#   \-------\        \
#   \ x \   \        \
#    ----------------
#           (2)

col_idx = col_idx +treeR.treeL.m;
Bu_lr, Bl_lr, peakMem = HSS_Expansion_Coeffs_OffDiag(treeL.treeL,treeR.treeR,row_idx,col_idx,N
    ,peakMem);
bu_lr = treeL.Rl'*Bu_lr*treeR.Wr;
bl_lr = treeR.Rr'*Bl_lr*treeL.Wl;
#keep track of peak memory
count = length(bu_lr)+length(bl_lr);
```

```
peakMem[1] = peakMem[1]+count;
peakMem[2] = max(peakMem[1],peakMem[2]);

count = length(Bu_lr)+length(Bl_lr);
peakMem[1] = peakMem[1]-count;
peakMem[2] = max(peakMem[1],peakMem[2]);

Bu_lr = Array{Float64}(undef,0);
Bl_lr = Array{Float64}(undef,0);

#upper right corner of lower block, lower left corner of upper block
#    ---------------
#   \        \   \   \
#   \        \---\---\
#   \        \ x \   \
#   \---------------\
#   \   \ x \        \
#   \-------\        \
#   \   \   \        \
#    ---------------
#           (3)

col_idx = col_idx -treeR.treeL.m;
row_idx = row_idx +treeL.treeL.m;
Bu_rl, Bl_rl, peakMem = HSS_Expansion_Coeffs_OffDiag(treeL.treeR,treeR.treeL,row_idx,col_idx,N
    ,peakMem);
bu_rl = treeL.Rr'*Bu_rl*treeR.Wl;
bl_rl = treeR.Rl'*Bl_rl*treeL.Wr;
#keep track of peak memory
count = length(bu_rl)+length(bl_rl);
peakMem[1] = peakMem[1]+count;
peakMem[2] = max(peakMem[1],peakMem[2]);

count = length(Bu_rl)+length(Bl_rl);
peakMem[1] = peakMem[1]-count;
peakMem[2] = max(peakMem[1],peakMem[2]);

Bu_rl = Array{Float64}(undef,0);
Bl_rl = Array{Float64}(undef,0);

#lower right corner of both lower left and upper left blocks
#    ---------------
#   \        \   \   \
#   \        \---\---\
#   \        \   \ x \
#   \---------------\
#   \   \   \        \
#   \-------\        \
#   \   \ x \        \
#    ---------------
#           (4)
```

```
        col_idx = col_idx +treeR.treeL.m;
        Bu_rr, Bl_rr, peakMem = HSS_Expansion_Coeffs_OffDiag(treeL.treeR,treeR.treeR,row_idx,col_idx,N
            ,peakMem);
        bu_rr = treeL.Rr'*Bu_rr*treeR.Wr;
        bl_rr = treeR.Rr'*Bl_rr*treeL.Wr;
        #keep track of peak memory
        count = length(bu_rr)+length(bl_rr);
        peakMem[1] = peakMem[1]+count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        count = length(Bu_rr)+length(Bl_rr);
        peakMem[1] = peakMem[1]-count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        Bu_rr = Array{Float64}(undef,0);
        Bl_rr = Array{Float64}(undef,0);

        #Add b's to form the B's
        Bu = bu_ll+bu_rl+bu_lr+bu_rr;
        Bl = bl_ll+bl_rl+bl_lr+bl_rr;
        #keep track of peak memory
        count = length(Bu)+length(Bl);
        peakMem[1] = peakMem[1]+count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        count = length(bu_ll)+length(bu_rl)+length(bu_lr)+length(bu_rr)+length(bl_ll)+length(bl_rl)+
            length(bl_lr)+length(bl_rr);
        peakMem[1] = peakMem[1]-count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        bu_ll = Array{Float64}(undef,0);
        bu_rl = Array{Float64}(undef,0);
        bu_lr = Array{Float64}(undef,0);
        bu_rr = Array{Float64}(undef,0);
        bl_ll = Array{Float64}(undef,0);
        bl_rl = Array{Float64}(undef,0);
        bl_lr = Array{Float64}(undef,0);
        bl_rr = Array{Float64}(undef,0);

        Bu, Bl, peakMem

    elseif isa(treeL, Leaf) && !isa(treeR, Leaf) #If left node is a leaf, and right node is not
        #COMPUTE EXPANSION COEFFICIENTS

        #left block of upper right corner and upper block of lower left corner
        #(these are computed in the same pass)
        #     ----------------
        #    \         \    \    \
        #    \         \ x \<--\-- MxN
        #    \         \    \    \
        #    \----------------\
        #    \    x<--\--------\-- NxM
```

```
#   \-------\       \
#   \       \       \
#    ---------------
#           (5)

Bu_l, Bl_l, peakMem = HSS_Expansion_Coeffs_OffDiag(treeL,treeR.treeL,row_idx,col_idx,N,peakMem
    );
bu_l = Bu_l*treeR.Wl;
bl_l = treeR.Rl'*Bl_l;
#keep track of peak memory
count = length(bu_l)+length(bl_l);
peakMem[1] = peakMem[1]+count;
peakMem[2] = max(peakMem[1],peakMem[2]);

count = length(Bu_l)+length(Bl_l);
peakMem[1] = peakMem[1]-count;
peakMem[2] = max(peakMem[1],peakMem[2]);

Bu_l = Array{Float64}(undef,0);
Bl_l = Array{Float64}(undef,0);

#right block of upper right corner, upper left block of lower left corner.
#    ---------------
#   \       \   \   \
#   \       \   \ x \
#   \       \   \   \
#   \---------------\
#   \       \       \
#   \-------\       \
#   \   x   \       \
#    ---------------
#           (6)

col_idx = col_idx + treeR.treeL.m;
Bu_r, Bl_r, peakMem = HSS_Expansion_Coeffs_OffDiag(treeL,treeR.treeR,row_idx,col_idx,N,peakMem
    );
col_idx = col_idx - treeR.treeL.m;
bu_r = Bu_r*treeR.Wr;
bl_r = treeR.Rr'*Bl_r;
#keep track of peak memory
count = length(bu_r)+length(bl_r);
peakMem[1] = peakMem[1]+count;
peakMem[2] = max(peakMem[1],peakMem[2]);

count = length(Bu_r)+length(Bl_r);
peakMem[1] = peakMem[1]-count;
peakMem[2] = max(peakMem[1],peakMem[2]);

Bu_r = Array{Float64}(undef,0);
Bl_r = Array{Float64}(undef,0);

Bu = bu_l + bu_r;
```

```
        Bl = bl_l + bl_r;
        #keep track of peak memory
        count = length(Bu)+length(Bl);
        peakMem[1] = peakMem[1]+count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        count = length(bu_l)+length(bu_r)+length(bl_l)+length(bl_r);
        peakMem[1] = peakMem[1]-count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        bu_l = Array{Float64}(undef,0);
        bu_r = Array{Float64}(undef,0);
        bl_l = Array{Float64}(undef,0);
        bl_r = Array{Float64}(undef,0);

        Bu, Bl, peakMem

    elseif !isa(treeL, Leaf) && isa(treeR, Leaf) #If right node is a leaf, and left node is not
        #COMPUTE EXPANSION COEFFICIENTS

        #upper block of upper right corner and left block of lower left corner
        #(these are computed in the same pass)
        #    ----------------
        #   \        \    x <-\-- MxN
        #   \        \--------\
        #   \        \        \
        #   \----------------\
        #   \    \    \        \
        #   \ x \<--\--------\-- NxM
        #   \    \    \        \
        #    ----------------
        #            (7)

        Bu_l, Bl_l, peakMem = HSS_Expansion_Coeffs_OffDiag(treeL.treeL,treeR,row_idx,col_idx,N,peakMem
            );
        bu_l = treeL.Rl'*Bu_l;
        bl_l = Bl_l*treeL.Wl;
        #keep track of peak memory
        count = length(bu_l)+length(bl_l);
        peakMem[1] = peakMem[1]+count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        count = length(Bu_l)+length(Bl_l);
        peakMem[1] = peakMem[1]-count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        Bu_l = Array{Float64}(undef,0);
        Bl_l = Array{Float64}(undef,0);

        #upper block of upper right corner and left block of lower left corner
        #(these are computed in the same pass)
        #    ----------------
```

```
        #   \        \        \
        #   \        \-------\
        #   \        \   x   \
        #   \---------------\
        #   \   \    \       \
        #   \   \ x \       \
        #   \   \    \       \
        #    ---------------
        #           (8)

        row_idx = row_idx + treeL.treeL.m;
        Bu_r, Bl_r, peakMem = HSS_Expansion_Coeffs_OffDiag(treeL.treeR,treeR,row_idx,col_idx,N,peakMem
            );
        row_idx = row_idx - treeL.treeL.m;
        bu_r = treeL.Rr'*Bu_r;
        bl_r = Bl_r*treeL.Wr;
        #keep track of peak memory
        count = length(bu_r)+length(bl_r);
        peakMem[1] = peakMem[1]+count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        count = length(Bu_r)+length(Bl_r);
        peakMem[1] = peakMem[1]-count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        Bu_r = Array{Float64}(undef,0);
        Bl_r = Array{Float64}(undef,0);

        Bu = bu_l + bu_r;
        Bl = bl_l + bl_r;
        #keep track of peak memory
        count = length(Bu)+length(Bl);
        peakMem[1] = peakMem[1]+count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        count = length(bu_l)+length(bu_r)+length(bl_l)+length(bl_r);
        peakMem[1] = peakMem[1]-count;
        peakMem[2] = max(peakMem[1],peakMem[2]);

        bu_l = Array{Float64}(undef,0);
        bu_r = Array{Float64}(undef,0);
        bl_l = Array{Float64}(undef,0);
        bl_r = Array{Float64}(undef,0);

        Bu, Bl, peakMem
    end
end
```

A.1.2 HSS Script

This section contains code which gives examples of how to call the functions in appendix A.1.

```
cd("C:\\Users\\klessel\\Dropbox\\Julia_home")
```

```
push!(LOAD_PATH, pwd())
# push!(LOAD_PATH,"/home/klessel/Dropbox/Julia_home/Polynomials/src")
using HSS_Types
using Polynomials
using LinearAlgebra
# include("/home/klessel/Dropbox/Julia_home/Gen_HSS_Memory_Efficient.jl")
# include("/home/klessel/Dropbox/Julia_home/Build_Matrix_From_HSS.jl")
include("C:\\Users\\klessel\\Dropbox\\Julia_home\\Gen_HSS_Memory_Efficient.jl")
include("C:\\Users\\klessel\\Dropbox\\Julia_home\\Build_Matrix_From_HSS.jl")
include("C:\\Users\\klessel\\Dropbox\\Julia_home\\FMM_Functions.jl")
include("C:\\Users\\klessel\\Dropbox\\Julia_home\\svd_ranktol.jl")
N = Int64;
r = Int64;
d = Int64;
minPart = Int64;
N = 256;
r=3; #hankel block rank
minPart = 3*r;

# d = 1;
# tree, dummy = Gen_Tree_Complete(minPart,N,d);

#tree = Gen_Tree_Mtree(minPart,N);
tree, dummy = label_tree(tree,1);

##
# #complete test tree
# N = 160;
# leaf1 = Leaf_Spine(4,4,-1,-1);
# leaf2 = Leaf_Spine(8,8,-1,-1);
# node1 = Node_Spine(leaf1,leaf2,12,12,-1,-1);
#
# leaf3 = Leaf_Spine(7,7,-1,-1);
# leaf4 = Leaf_Spine(21,21,-1,-1);
# node2 = Node_Spine(leaf3,leaf4,28,28,-1,-1);
#
# leaf5 = Leaf_Spine(14,14,-1,-1);
# leaf6 = Leaf_Spine(16,16,-1,-1);
# node3 = Node_Spine(leaf5,leaf6,30,30,-1,-1);
#
# leaf7 = Leaf_Spine(25,25,-1,-1);
# leaf8 = Leaf_Spine(65,65,-1,-1);
# node4 = Node_Spine(leaf7,leaf8,90,90,-1,-1);
#
# node5 = Node_Spine(node1,node2,40,40,-1,-1);
# node6 = Node_Spine(node3,node4,120,120,-1,-1);
#
# tree = Node_Spine(node5,node6,160,160,-1,-1);
# tree, dummy = label_tree(tree,1);
##
println("N = ", N, ", rank = ", r, ", Minimum Partition = ", minPart);
hss, peakMem = @time Gen_HSS_Memory_Efficient(tree,r);
```

```
println("Peak memory count is ", peakMem[2], " Float64 assignments");

#test
A = Array{Float64,2}(undef,N,N);
A = f(1:N,1:N,N);

B, dummy1, dummy2 = Build_Matrix_From_HSS(hss);
rel_err =  norm(A-B)/norm(A);
println("||A-B||/||A|| = ", rel_err);
```

A.1.3 HSS Functions

```
function Gen_Tree_Complete(minPart::Int64,N::Int64,d::Int64)
#function(minPart,N)
#This code is for testing purposes only.  Generates a tree
#structure that is not associated with any function.  Function takes in the
#size of the desired matrix and generates a tree by splitting this
#repeatedly in half on the left and right for a 'complete' tree.
#INPUT:              minPart     (Int64) minimum number of partitions
#                                type of tree that will be generated
#                    N           (Uint) number of points in desired
#                                function
#OUTPUT:             tree        (Union(Node_Spine,Leaf_Spine) contains split dimensions and depth at
#                                each level

    if N/2 < minPart # leaf
        depth = 0;
        leafL = Leaf_Spine(N,N,depth,d);

        d = d + N;
        leafL, depth, d
    else # node
        treeL, depth_l, d_l = Gen_Tree_Complete(minPart,convert(Int64,N/2),d);
        treeR, depth_r, d_r = Gen_Tree_Complete(minPart,convert(Int64,N/2),d_l);

        depth = 1 + max(depth_l,depth_r);
        tree = Node_Spine(treeL,treeR,N,N,depth,d);
        tree, depth, d_r
    end
end

function Gen_Tree_Mtree(minPart::Int64,N::Int64)
#function [tree] = Gen_TestTree(tree,N)
#This code is for testing purposes only. Artificually generates a tree
#structure that is not associated with any function.  Function takes in the
#number of desired nodes and generates a Worst case memory tree
# must have the line 'using Polynomials' in the main routine
#
#INPUT:
#                    N           (Int64) number of points in desired
#                                function
```

```
#                   minPart     (Int64) minimum number of partitions
#OUTPUT:            tree        (Tree) contains split dimensions at
#                               each level, as well as whether or not the
#                               node is a leaf


    # number of nodes
    n = 2*floor(N/minPart) -1; # n = N_N = N_L -1, and N_L = N/minPart.

    #depth of tree
    nodeRoots = roots(Poly([1-n,1,1]));
    d_exact  = nodeRoots[nodeRoots.>0]; #choose the positive root.

    #find maximum depth, d_max, of the tree we will generate
    d_max = convert(Int64,ceil(d_exact[1]));

    #Call the function initially on the root node of the tree.
    #   .<-
    #  / \
    # /\ /\
    #/\ /\/\
    tree = Gen_MTree_Right(N,minPart,d_max);
    tree
end

function Gen_MTree_Right(m,minPart,depth)
#Descend into right branch of the root of the current subtree (pictured below)
#    .
#  / \<-
# /\ /\
#/\ /\/\
#INPUT:
#                   m           (Int64) partition dimension of current
#                               node
#                   minPart     (Int64) minimum number of partitions
#                   depth       (Int64) depth of current node
#                   pB          (Bool) a flag that denotes whether the
#                               current node is on the prime branch
#OUTPUT:            tree        (Tree) contains split dimensions at
#                               each level, as well as whether or not the
#                               node is a leaf
    if !(m < 2*minPart)
    #Case 1: If we do have enough rows to split into 2 blocks of minimum
    #partition size split and recurse on both children.  If we do not, then do
    #not partition further; Return two leaf nodes.

        #max number of blocks of minimum partition size we can have on this left subtree
        numBlocks = convert(Int64,floor(m/minPart));
        #remaining depth of the left subtree.
        r_depth = numBlocks-1;

        #if we cannot generate a full left subtree - only generate children
```

```
#corresponding to the the number of partitions we have left.
if m <(depth +1)*minPart
    mL = r_depth*minPart;
else
    mL = depth*minPart;
end

#Left Subtree Call
treeL = Gen_MTree_Left(mL, minPart, depth);

#Right Subtree Call
#do we have enough rows to partition into two blocks of minumum partition size?
if  m >= 2*minPart && m < 3*minPart
    #Case A: do we only have enough to split into two blocks and no
    #more?(blocks must be of at least minimum partition size and no
    #more than 2 times the minimum partition size)
    #Ex: minPart = 60
    #       /\170
    #   110  60
    #

    mL = m- minPart; #extra rows tacked onto left child
    leafL = Leaf_Spine(mL,mL,-1,-1);

    mR = minPart;
    leafR = Leaf_Spine(mR,mR,-1,-1);

    tree  = Node_Spine(leafL,leafR,m,m,-1,-1);
    tree
elseif m<(depth+1)*minPart
    #Case B: do we have enough to split into more than two blocks, but cannot
    #generate a full left going subtree?
    #Ex:(N = 4096) Depth of full tree = 12. MinPart = 60.
    #(Though here I am only showing partition  dimensions for 3 levels.
    #Dots indicate a part of the tree not shown. )
    #    . . /\
    #  .      /\204
    #  . 144/\ 60
    #  .  60  84
    #  /\

    mR = m - (numBlocks-1)*minPart;

    treeR = Leaf_Spine(mR,mR,-1,-1);
    tree = Node_Spine(treeL,treeR,m,m,-1,-1);
    tree
else #(depth+1)*minPart < m
    #Case C: We have enough rows to generate a full leftgoing subtree of depth d_max

    mR = m- depth*minPart;
    treeR = Gen_MTree_Right(mR, minPart, depth);
```

```
            tree = Node_Spine(treeL,treeR,m,m,-1,-1);
            tree
        end

    else #Case 2: we did not have enough rows to partition - label node as a leaf and return
        leaf = Leaf_Spine(m,m,-1,-1);
        leaf

    end

end

function Gen_MTree_Left(m,minPart,depth)
#Generate left branch of the root of the current subtree (pictured below)
#      .
#     / \
# ->/\ /\
#  /\ /\/\
#INPUT:
#                    m           (Int64) partition dimension of current
#                                node
#                    minPart     (Int64) minimum number of partitions
#                    depth       (Int64) depth of current node
#                    pB          (Bool) a flag that denotes whether the
#                                current node is on the prime branch
#OUTPUT:             tree        (Tree) contains split dimensions at
#                                each level, as well as whether or not the
#                                node is a leaf

    if m < 2*minPart  # if Leaf

        #pB = false;
        leafL = Leaf_Spine(m,m,-1,-1);

        leafL
    else #node
        depth -= 1;
        mL = m - minPart;
        treeL = Gen_MTree_Left(mL,minPart,depth);

        treeR = Leaf_Spine(minPart,minPart,-1,-1);

        tree = Node_Spine(treeL,treeR,m,m,-1,-1);
        tree
    end

end

function label_tree(tree,d)
#This function labels each node with the starting index (row/col value)
# of its corresponding diagonal block. Can also label the depth if the
# commented lines are uncommented
```

```
#
#INPUT:             tree    (Tree) contains partition dimensions for
#                           each subdivision of the input matrix, for each
#                           of which there are a left and right child
#                           structure
#                   d       (Int64) row and col index which corresponds to
#                           the first element in the current diagonal block
#                           at every node.
#OUTPUT:            tree    (Tree) same as input,  that contains valid
#                           diagonal (row/col) index at each node
#                           (and depth if commented lines are uncommented.
#
# Author: Kristen Lessel - Sept 2014

    if !isa(tree,Leaf_Spine) # if not a leaf node

        tree.d = d;
        tree.treeL,d = label_tree(tree.treeL,d);
        depth_l = tree.treeL.depth;

        tree.treeR,d = label_tree(tree.treeR,d);
        depth_r = tree.treeR.depth;

        tree.depth = 1 + max(depth_l,depth_r);
        tree, d
    else
        tree.depth = 0;
        tree.d = d;
        d = d + tree.m;
        tree, d
    end
end
function Build_Matrix_From_HSS(hss)
#Takes in an HSS matrix (structure) and returns the dense matrix A
#(nxn double). This code should be use for error checking/debugging only,
#in practice you should not construct the full matrix A since the amount of
#memory required to do so will be large. Uses feild labled Bu instead of Br
#
#INPUT:             hss    (structure) contains matrices (Us, Vs, Bs Rs and
#                           Ws) that compose the hss representation of the
#                           input matrix, as well as partition dimensions
#                           and if branch is a leaf.
#
#OUTPUT:            A       dense matrix A

    if isa(hss, Leaf) #if leaf node
        U = hss.U;
        V = hss.V;
        D = hss.D;

        D, U, V
    else
```

```
        #left recursive call
        Dl, Ul, Vl = Build_Matrix_From_HSS(hss.treeL);

        #right recursive call
        Dr, Ur, Vr = Build_Matrix_From_HSS(hss.treeR);

        #construct upper right and lower left block of current level
        urb = Ul*hss.Bu*Vr';
        llb = Ur*hss.Bl*Vl';

        #create D, U and V to pass to parent
        D = [Dl urb; llb Dr];
        U = [Ul*hss.Rl; Ur*hss.Rr];
        V = [Vl*hss.Wl; Vr*hss.Wr];

        D, U, V
    end
end

## HSS_Types.jl

module HSS_Types

export Leaf, Node, Hss, Leaf_Spine, Node_Spine, Tree

mutable struct Leaf{T1 <: Integer, T2 <: Number}
    m::T1
    n::T1
    depth::T1
    d::T1 #diagonal index
    U::Array{T2,2}
    V::Array{T2,2}
    D::Array{T2,2}
end

#Define node
mutable struct Node{T1 <: Integer, T2 <: Number}
    treeL::Union{Node,Leaf} #Union(Node{T1,T2},Leaf{T1,T2}) doesn't work for some reason
    treeR::Union{Node,Leaf}
    m::T1
    n::T1
    depth::T1
    d::T1
    Bu::Array{T2,2}
    Bl::Array{T2,2}
    Rl::Array{T2,2}
    Rr::Array{T2,2}
    Wl::Array{T2,2}
    Wr::Array{T2,2}
end

#Define HSS structure
```

```
const Hss = Union{Leaf,Node}

# #Define partition(spine) tree
mutable struct Leaf_Spine{T <: Integer}
    m::T
    n::T
    depth::T
    d::T
end

mutable struct Node_Spine{T <: Integer}
    treeL::Union{Node_Spine{T},Leaf_Spine{T}}
    treeR::Union{Node_Spine{T},Leaf_Spine{T}}
    m::T
    n::T
    depth::T
    d::T
end

Tree = Union{Leaf_Spine,Node_Spine}

end
```

A.2 FMM Julia Codes

A.2.1 FMM Construction

```
function FMM_Construction2(tree::Tree,r,f::Function)
#This is a memory efficient, two pass algorithm that takes in a Tree
#which is a partition tree for a given matrix which is described by the
#function, f, below and returns its corresponding FMM representation.
#Computation of U,R,V,W is done via deepest first post-ordering. B
#matrices are computed by in descending order (starting at root and finishing
#at the child). Relavant matrices are then multiplied and added
#from child to root in order to compute each B matrix. (Bottom Up Routine
# to compute B).
#INPUT:             r       (Int64) largest allowable rank of hankel blocks -
#                           this determines the amount of compression, and
#                           should be compatible with your input tree.
#                           (if the maximum allowable rank is p, then the
#                           tree partitions should be no smaller than 3p)
#                           Corresponding singular values below this rank
#                           will be dropped.
#                   tree    (Tree) contains partition dimensions for
#                           each subdivision of the input matrix, for each
#                           of which there are a left and right child
#                           structure
#OUTPUT:            fmm     (FMM) contains matrices (Us, Vs, Bs Rs and
#                           Ws) that compose the fmm representation of the
#                           input matrix, in addition to the partition
```

```
#                           dimensions
#                   peakMem (Array{Int64,1}) first entry is a memory counter
#                           and second entry is the maximum memory
# Author: Kristen Lessel - November 2015
    N = tree.m;

    empty_tree = create_empty_tree(tree.depth);
    #pm_count = 0;
    #pm_max = 0;
    #peakMem = [pm_count,pm_max];
    fmm, dummy1, dummy2 = FMM_Basis_TranslationOp(empty_tree,tree,empty_tree,N,r,f);

    row_idx = 1;
    col_idx = 1;
    fmm  = FMM_Expansion_Coeffs_Diag(fmm,row_idx,col_idx,N,f);

    fmm
end


function FMM_Basis_TranslationOp(treeL::Tree,tree::Tree,treeR::Tree,N::Int64,r::Int64,f::Function)
#Computes U's, V's, R's, W's and D's of FMM structure, and stores these
#heirarchically
#INPUT:             N       (Int64) grid size
#                   rank    (Int64) largest allowable rank of hankel blocks -
#                           this determines the amount of compression, and
#                           should be compatible with your input tree.
#                           (if the maximum allowable rank is p, then the
#                           tree partitions should be no smaller than 3p)
#                           Corresponding singular values below this rank
#                           will be dropped.
#                   tree    (Union(Node_Spine,Leaf_Spine))
#                           contains partition dimensions for each
#                           subdivision of the input matrix, for each
#                           of which there are a left and right child
#                           structure
#OUTPUT:            fmm     (Union(Node,Leaf)) contains matrices
#                           (U, V, R and W) that compose the
#                           hss representation of the input matrix, in
#                           addition to the partition dimensions
#                   rH      (Array{Float64}(2)) row hankel block with 'current' U
#                           removed
#                   cH      (Array{Float64}(2)) column hankel block with 'current' V'
#                           removed
    if isa(treeL,Leaf_Spine) && isa(tree,Leaf_Spine) && isa(treeR,Leaf_Spine)

        d1_idx = collect(treeL.col_idx:treeL.col_idx+treeL.m-1);

        d2_idx = collect(tree.col_idx:tree.col_idx+tree.m-1);


        d3_idx = collect(treeR.col_idx:treeR.col_idx+treeR.m-1);
```

```
        m_idx = collect(treeL.col_idx:(treeL.col_idx+treeL.m+tree.m+treeR.m)-1);
        #generate indices for current off diagonal hankel block
        butm_idx = [1:m_idx[1]-1; m_idx[end]+1:N];

        diag_idx = collect(tree.col_idx:(tree.col_idx+tree.m)-1);

        #function call to generate lowest level row and column hankel blocks
        rH2 = f(diag_idx,butm_idx,N);
        cH2= f(butm_idx,diag_idx,N);

        #take svds of upper row/left column and lower row/right column hankel blocks
        Ur, Rr, Vr = svd_ranktol(rH2,r);
        Uc, Rc, Vc = svd_ranktol(cH2,r);

        rH2 = Array{Float64}(undef,0);
        cH2 = Array{Float64}(undef,0);

        leaf = Leaf(tree.m,tree.col_idx,tree.depth,Ur,Vc,f(d2_idx,d1_idx,N),
                    f(d2_idx,d2_idx,N),f(d2_idx,d3_idx,N));

        rH = diagm(Rr)*Vr';
        cH = Uc*diagm(Rc);

        leaf, rH, cH

    elseif isa(tree, Node_Spine)
        #If the center tree is a node

        #Descend into the tree, depth first
        if tree.treeL.depth >= tree.treeR.depth #left node deeper than right

            if isa(treeL,Node_Spine) && isa(tree,Node_Spine) && isa(treeR,Node_Spine)
                # 2) All nodes

                #left recursive call
                fmmL, upperRow, leftCol = FMM_Basis_TranslationOp(treeL.treeR,tree.treeL,tree.treeR,N,
                    r,f);

                #right recursive call
                fmmR, lowerRow, rightCol = FMM_Basis_TranslationOp(tree.treeL,tree.treeR,treeR.treeL,N
                    ,r,f);

                #these are used to indexes of rol/col hankel blocks excluding corresponding part of
                    diagonal block
                diag_width_l = treeL.treeL.m + treeR.treeL.m + treeR.treeR.m;
                diag_width_r = treeL.treeL.m + treeL.treeR.m + treeR.treeR.m;

                #indexes used to obtain compressed hankel blocks for the next level
                uR_idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width_l):(N-(treeL.treeR.m + tree
                    .treeL.m + tree.treeR.m))];
                lR_idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width_r):(N-(tree.treeL.m + tree.
```

```
        treeR.m + treeR.treeL.m))];

elseif isa(treeL,Node_Spine) &&isa(tree,Node_Spine) && isa(treeR,Leaf_Spine)
    # 3) Left and center are nodes, right is leaf
    #left recursive call
    fmmL, upperRow, leftCol = FMM_Basis_TranslationOp(treeL.treeR,tree.treeL,tree.treeR,N,
        r,f);

    #right recursive call
    fmmR, lowerRow, rightCol = FMM_Basis_TranslationOp(tree.treeL,tree.treeR,treeR,N,r,f);

    #these are used to indexes of rol/col hankel blocks excluding corresponding part of
        diagonal block
    diag_width_l = treeL.treeL.m + treeR.m;
    diag_width_r = treeL.treeL.m + treeL.treeR.m;

    #indexes used to obtain compressed hankel blocks for the next level
    uR_idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width_l):(N-(treeL.treeR.m + tree
        .treeL.m + tree.treeR.m))];
    lR_idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width_r):(N-(tree.treeL.m + tree.
        treeR.m + treeR.m))];

elseif isa(treeL,Leaf_Spine) && isa(tree,Node_Spine) &&  isa(treeR,Leaf_Spine)
    # 8) Left and Right are leaves, center is node

    #left recursive call
    fmmL, upperRow, leftCol = FMM_Basis_TranslationOp(treeL,tree.treeL,tree.treeR,N,r,f);

    #right recursive call
    fmmR, lowerRow, rightCol = FMM_Basis_TranslationOp(tree.treeL,tree.treeR,treeR,N,r,f);

    #these are used to indexes of rol/col hankel blocks excluding corresponding part of
        diagonal block
    diag_width_l = treeR.m
    diag_width_r = treeL.m

    #indexes used to obtain compressed hankel blocks for the next level
    uR_idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width_l):(N-(treeL.m + tree.treeL
        .m + tree.treeR.m))];
    lR_idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width_r):(N-(tree.treeL.m + tree.
        treeR.m + treeR.m))];

elseif isa(treeL,Leaf_Spine) && isa(tree,Node_Spine) && isa(treeR,Node_Spine)
    # 5) Left is a leaf, center and right are nodes

    #left recursive call
    fmmL, upperRow, leftCol = FMM_Basis_TranslationOp(treeL,tree.treeL,tree.treeR,N,r,f);

    #right recursive call
    fmmR, lowerRow, rightCol = FMM_Basis_TranslationOp(tree.treeL,tree.treeR,treeR.treeL,N
        ,r,f);
```

```
            #these are used to indexes of rol/col hankel blocks excluding corresponding part of
                diagonal block
            diag_width_l = treeR.treeL.m + treeR.treeR.m;
            diag_width_r = treeL.m + treeR.treeR.m;

            #indexes used to obtain compressed hankel blocks for the next level
            uR_idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width_l):(N-(treeL.m + tree.treeL
                .m + tree.treeR.m))];
            lR_idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width_r):(N-(tree.treeL.m + tree.
                treeR.m + treeR.treeL.m))];

        end

    else #right node deeper than left
        if isa(treeL,Node_Spine) && isa(tree,Node_Spine) && isa(treeR,Node_Spine)
            # 2) All nodes

            #right recursive call
            fmmR, lowerRow, rightCol = FMM_Basis_TranslationOp(tree.treeL,tree.treeR,treeR.treeL,N
                ,r,f);

            #left recursive call
            fmmL, upperRow, leftCol = FMM_Basis_TranslationOp(treeL.treeR,tree.treeL,tree.treeR,N,
                r,f);

            #these are used to indexes of rol/col hankel blocks excluding corresponding part of
                diagonal block
            diag_width_l = treeL.treeL.m + treeR.treeL.m + treeR.treeR.m;
            diag_width_r = treeL.treeL.m + treeL.treeR.m + treeR.treeR.m;

            #indexes used to obtain compressed hankel blocks for the next level
            uR_idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width_l):(N-(treeL.treeR.m + tree
                .treeL.m + tree.treeR.m))];
            lR_idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width_r):(N-(tree.treeL.m + tree.
                treeR.m + treeR.treeL.m))];

        elseif isa(treeL,Node_Spine) && isa(tree,Node_Spine) && isa(treeR,Leaf_Spine)
            # 3) Left and center are nodes, right is leaf

            #right recursive call
            fmmR, lowerRow, rightCol = FMM_Basis_TranslationOp(tree.treeL,tree.treeR,treeR,N,r,f);

            #left recursive call
            fmmL, upperRow, leftCol = FMM_Basis_TranslationOp(treeL.treeR,tree.treeL,tree.treeR,N,
                r,f);

            #these are used to indexes of rol/col hankel blocks excluding corresponding part of
                diagonal block
            diag_width_l = treeL.treeL.m + treeR.m;
            diag_width_r = treeL.treeL.m + treeL.treeR.m;

            #indexes used to obtain compressed hankel blocks for the next level
```

```
            uR_idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width_l):(N-(treeL.treeR.m + tree
                .treeL.m + tree.treeR.m))];
            lR_idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width_r):(N-(tree.treeL.m + tree.
                treeR.m + treeR.m))];

        elseif isa(treeL,Leaf_Spine) && isa(tree,Node_Spine) && isa(treeR,Leaf_Spine)
            # 8) Left and Right are leaves, center is node

            #right recursive call
            fmmR, lowerRow, rightCol = FMM_Basis_TranslationOp(tree.treeL,tree.treeR,treeR,N,r,f);

            #left recursive call
            fmmL, upperRow, leftCol = FMM_Basis_TranslationOp(treeL,tree.treeL,tree.treeR,N,r,f);

            #these are used to indexes of rol/col hankel blocks excluding corresponding part of
                diagonal block
            diag_width_l = treeR.m
            diag_width_r = treeL.m

            #indexes used to obtain compressed hankel blocks for the next level
            uR_idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width_l):(N-(treeL.m + tree.treeL
                .m + tree.treeR.m))];
            lR_idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width_r):(N-(tree.treeL.m + tree.
                treeR.m + treeR.m))];

        elseif isa(treeL,Leaf_Spine) && isa(tree, Node) && isa(treeR,Node_Spine)
            # 5) Left is a leaf, center and right are nodes

            #right recursive call
            fmmR, lowerRow, rightCol = FMM_Basis_TranslationOp(tree.treeL,tree.treeR,treeR.treeL,N
                ,r,f);

            #left recursive call
            fmmL, upperRow, leftCol = FMM_Basis_TranslationOp(treeL,tree.treeL,tree.treeR,N,r,f);

            #these are used to indexes of rol/col hankel blocks excluding corresponding part of
                diagonal block
            diag_width_l = treeR.treeL.m + treeR.treeR.m;
            diag_width_r = treeL.m + treeR.treeR.m;

            #indexes used to obtain compressed hankel blocks for the next level
            uR_idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width_l):(N-(treeL.m + tree.treeL
                .m + tree.treeR.m))];
            lR_idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width_r):(N-(tree.treeL.m + tree.
                treeR.m + treeR.treeL.m))];

        end
    end

    rH_top = upperRow[:, uR_idx];
    rH_bottom = lowerRow[:, lR_idx];
    cH_left = leftCol[uR_idx,:];
```

```
        cH_right = rightCol[lR_idx,:];

        #merge remainder of Hankel blocks from children
        rH2 = [rH_top; rH_bottom];
        cH2 = [cH_left cH_right];

        #take svd of remaining portions of blocks to determine Rs and Ws
        Ur, Sr, Vr = svd_ranktol(rH2,r);
        Uc, Sc, Vc = svd_ranktol(cH2,r);

        rH2 = Array{Float64}(undef,0);
        cH2 = Array{Float64}(undef,0);

        #partition Us to get R's, partition V's to get Ws
        Rl = Ur[1:size(upperRow,1),:];
        Rr = Ur[size(upperRow,1)+1:end,:];
        Wl = Vc[1:size(leftCol,2),:];
        Wr = Vc[size(leftCol,2)+1:end,:];

        emptyleaf = Leaf(-1,-1,-1,Array{Float64,2}(undef,0,0),Array{Float64,2}(undef,0,0),Array{
            Float64,2}(undef,0,0),Array{Float64,2}(undef,0,0),Array{Float64,2}(undef,0,0));
        fmm = Node(fmmL,fmmR,emptyleaf,emptyleaf,tree.m,tree.m,tree.col_idx,tree.col_idx,tree.depth,
            Array{Float64,2}(undef,0,0),Array{Float64,2}(undef,0,0),Array{Float64,2}(undef,0,0),Array{
            Float64,2}(undef,0,0),Array{Float64,2}(undef,0,0),Array{Float64,2}(undef,0,0),Rl,Rr,Wl,Wr)
            ;

        #return relavant portions of hankel blocks
        rH = diagm(Sr)*Vr';
        cH = Uc*diagm(Sc);

        fmm, rH, cH

    elseif isa(tree, Leaf_Spine)
        #if center tree is a leaf, but left and/or right tree are/is node(s)

        if isa(treeL,Node_Spine) &&isa(tree,Leaf_Spine) && isa(treeR,Node_Spine)
            # 4) Left and Right trees are nodes, center tree is a leaf

            #recursive call
            fmm, row, col = FMM_Basis_TranslationOp(treeL.treeR,tree,treeR.treeL,N,r,f);

            # #these are used to indexes of rol/col hankel blocks excluding corresponding part of
                diagonal block
            diag_width = treeL.treeL.m + treeR.treeR.m;

            # #indexes used to obtain compressed hankel blocks for the next level
            idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width):(N-(treeL.treeR.m + tree.m +
                treeR.treeL.m))];

        elseif isa(treeL,Leaf_Spine) &&isa(tree,Leaf_Spine) && isa(treeR,Node_Spine)
            #6) if left and center trees are leaves and right tree is a node
```

```
            #recursive call
            fmm, row, col = FMM_Basis_TranslationOp(treeL,tree,treeR.treeL,N,r,f);

            #these are used to indexes of rol/col hankel blocks excluding corresponding part of
                diagonal block
            diag_width = treeR.treeR.m;

            #indexes used to obtain compressed hankel blocks for the next level
            idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width):(N-(treeL.m + tree.m + treeR.
                treeL.m))];

        elseif isa(treeL,Node_Spine) &&isa(tree,Leaf_Spine) && isa(treeR,Leaf_Spine)
            #7) if left tree is a node and center and right trees are leaves

            #recursive call
            fmm, row, col = FMM_Basis_TranslationOp(treeL.treeR,tree,treeR,N,r,f);

            #these are used to indexes of rol/col hankel blocks excluding corresponding part of
                diagonal block
            diag_width = treeL.treeL.m;

            #indexes used to obtain compressed hankel blocks for the next level
            idx = [1:(treeL.col_idx-1); (treeL.col_idx + diag_width):(N-(treeL.treeR.m +tree.m + treeR
                .m))];

        end

        rH = row[:,idx];
        cH = col[idx,:];

        fmm, rH, cH
end
end

#function FMM_Expansion_Coeffs_Diag(fmm::Fss,row_idx::Int64,col_idx::Int64,N::Int64,peakMem::Array{
    Int64,1})
function FMM_Expansion_Coeffs_Diag(fmm::FMM,row_idx::Int64,col_idx::Int64,N::Int64,f::Function)
#Recursively computes Expansion Coefficients (B's) of FMM structure for the
#upper left and lower right block of the current node
#
#   ---------------
#  \    \    \B13\B14\
#  \--------\--------\
#  \    \    \    \B24\
#  \-----------------\
#  \B31\    \    \    \
#  \--------\--------\
#  \B41\B42\    \    \
#   ---------------
#INPUT:              fmm     (Union(Node,Leaf)) branch of current node,
#                    row_idx (Int64) row index of current node
#                    col_idx (Int64) col index of current node
```

```
#                   N       (Int64) grid size
#OUTPUT:            fmm     (Union(Node,Leaf)) contains matrices (U, V, B R and
#                           W) that compose the HSS representation of the
#                           input tree for a corresponding matrix, as well
#                           as corresponding partition dimensions and
#                           depth for each node

    if isa(fmm.fmmUL,Leaf) && isa(fmm.fmmLR,Leaf)
        #do nothing.  Return
        #   -------
        #  \ D \ D \
        #  \-------\
        #  \ D \ D \
        #   -------

        fmm

    elseif !isa(fmm.fmmUL,Leaf) && !isa(fmm.fmmLR,Leaf)

        #Off diagonal block calls.  Recursively compute B13,B14,B24,B31,B41,B42
        #   -------
        #  \   \ x \
        #  \-------\
        #  \ x \   \
        #   -------
        col_idx = col_idx + fmm.fmmUL.m;
        fmmUR, fmmLL = FMM_Expansion_Coeffs_OffDiag(fmm.fmmUL,fmm.fmmLR,row_idx,col_idx,N,f);
        col_idx = col_idx - fmm.fmmUL.m;

        #Upper Left Diagonal Block FMM Call
        #   -------
        #  \ x \   \
        #  \-------\
        #  \   \   \
        #   -------
        fmmUL = FMM_Expansion_Coeffs_Diag(fmm.fmmUL,row_idx,col_idx,N,f);

        #Lower Right Diagonal Block FMM Call
        #   -------
        #  \   \   \
        #  \-------\
        #  \   \ x \
        #   -------
        col_idx = col_idx + fmm.fmmUL.m;
        row_idx = row_idx + fmm.fmmUL.m;
        fmmLR = FMM_Expansion_Coeffs_Diag(fmm.fmmLR,row_idx,col_idx,N,f) ;

        fmm.fmmUR = fmmUR;
        fmm.fmmLL = fmmLL;
        fmm.fmmUL = fmmUL;
        fmm.fmmLR = fmmLR;
        fmm
```

```
    elseif isa(fmm.fmmUL, Leaf) && !isa(fmm.fmmLR, Leaf)

        #compute both upper b's and lower b's here (2 of them)
        col_idx = col_idx + fmm.fmmUL.m;
        fmmUR, fmmLL = FMM_Expansion_Coeffs_OffDiag(fmm.fmmUL,fmm.fmmLR, row_idx, col_idx, N, f);
        col_idx = col_idx - fmm.fmmUL.m;

        #Upper Left Diagonal BLock is a Leaf, so no call
        #fmmUL  = LeafOffDiag(fmm.fmmUL.m,fmm.fmmLR.m,fmm.fmmUL.col_idx,fmm.fmmLR.col_idx,0,Array{
            Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(
            undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(undef,0,0));

        #Lower Right Diagonal Block FMM Call
        col_idx = col_idx + fmm.fmmUL.m;
        row_idx = row_idx + fmm.fmmUL.m;
        fmmLR = FMM_Expansion_Coeffs_Diag(fmm.fmmLR, row_idx, col_idx, N, f);

        fmm.fmmUR = fmmUR;
        fmm.fmmLL = fmmLL;

        fmm.fmmLR = fmmLR
        fmm

    elseif !isa(fmm.fmmUL, Leaf) && isa(fmm.fmmLR, Leaf)

        #compute both upper b's and lower b's here (2 of them)
        col_idx = col_idx + fmm.fmmUL.m;
        fmmUR, fmmLL = FMM_Expansion_Coeffs_OffDiag(fmm.fmmUL,fmm.fmmLR, row_idx, col_idx, N, f);
        col_idx = col_idx - fmm.fmmUL.m;

        #Upper Left Diagonal Block FMM Call
        fmmUL = FMM_Expansion_Coeffs_Diag(fmm.fmmUL, row_idx, col_idx, N, f);

        #Lower Right Diagonal Block is a Leaf, so no call
        #fmmLR  = LeafOffDiag(fmm.fmmLR.m,fmm.fmmUL.m,fmm.fmmLR.col_idx,fmm.fmmUL.col_idx,0,Array{
            Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(
            undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(undef,0,0));


        fmm.fmmUR = fmmUR;
        fmm.fmmLL = fmmLL;
        fmm.fmmUL = fmmUL;

        fmm

    end

end

function FMM_Expansion_Coeffs_OffDiag(fmmUL::FMM, fmmLR::FMM, row_idx::Int64, col_idx::Int64, N::Int64, f::
    Function)
```

```
if isa(fmmUL,Leaf) && isa(fmmLR,Leaf)
    ## These are zero arrays becuase B's at the leaf will be computed by Off_Diag_Outter.

    leafUR = LeafOffDiag(fmmUL.m,fmmLR.m,fmmUL.col_idx,fmmLR.col_idx,0,Array{Float64}(undef,0,0),
        Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{
        Float64}(undef,0,0),Array{Float64}(undef,0,0));

    leafLR = LeafOffDiag(fmmLR.m,fmmUL.m,fmmLR.col_idx,fmmUL.col_idx,0,Array{Float64}(undef,0,0),
        Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{
        Float64}(undef,0,0),Array{Float64}(undef,0,0));

    leafUR, leafLR

elseif !isa(fmmUL,Leaf) && !isa(fmmLR,Leaf)
    #upper left corner of both upper right and lower left blocks
    #(these are computed at the same time. One block will be of
    #dimension MxN, the other will always be of dimension NxM)
    #
    #    ----------------
    #   \        \ x \<--\-- MxN
    #   \         --------\
    #   \        \    \    \
    #   \----------------\
    #   \ x \<--\--------\-- NxM
    #   \-------\        \
    #   \   \   \        \
    #    ----------------
    #           (1)


    B13, B31 = FMM_Expansion_Coeffs_OffDiag_Outter(fmmUL.fmmUL,fmmLR.fmmUL,row_idx,col_idx,N,f);

    #Lower left corner lower left block, and also the upper right corner of
    #the upper right block
    #    ----------------
    #   \        \    \ x \
    #   \        \---\---\
    #   \        \   \    \
    #   \----------------\
    #   \   \    \        \
    #   \-------\        \
    #   \ x \   \        \
    #    ----------------
    #           (2)

    col_idx = col_idx + fmmLR.fmmUL.m;
    B14, B41 = FMM_Expansion_Coeffs_OffDiag_Outter(fmmUL.fmmUL,fmmLR.fmmLR,row_idx,col_idx,N,f);

    #lower right corner of both lower left and upper left blocks
    #    ----------------
    #   \        \   \    \
```

```
    #   \        \---\---\
    #   \        \   \ x \
    #   \---------------\
    #   \   \    \       \
    #   \-------\       \
    #   \   \ x \       \
    #    ---------------
    #           (3)

    row_idx = row_idx +fmmUL.fmmUL.m;
    B24, B42 = FMM_Expansion_Coeffs_OffDiag_Outter(fmmUL.fmmLR,fmmLR.fmmLR,row_idx,col_idx,N,f);

    #upper right corner of lower block, lower left corner of upper block
    #    ---------------
    #   \        \   \   \
    #   \        \---\---\
    #   \        \ x \   \
    #   \---------------\
    #   \   \ o \       \
    #   \-------\       \
    #   \   \    \       \
    #    ---------------
    #           (4)

    col_idx = col_idx - fmmLR.fmmUL.m #changed on 4/18/16
    fmmUR_LL, fmmLL_UR = FMM_Expansion_Coeffs_OffDiag(fmmUL.fmmLR,fmmLR.fmmUL,row_idx,col_idx,N,f)
        ; # x, o =  FMM_Expansion_Coeffs_OffDiag()

    offDiag_depth = max(fmmLR.depth,fmmUL.depth);
    emptyleaf = LeafOffDiag(-1,-1,-1,-1,-1,Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),
                Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),
                    Array{Float64}(undef,0,0));
    fmmUR = Node(emptyleaf, emptyleaf, emptyleaf,fmmUR_LL,fmmUL.m,fmmLR.m,fmmLR.col_idx,
                fmmUL.col_idx,offDiag_depth ,B13,B14,B24,Array{Float64}(undef,0,0),Array{Float64
                    }(undef,0,0),
                Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),
                    Array{Float64}(undef,0,0),
                Array{Float64}(undef,0,0));
    fmmLL = Node(emptyleaf,emptyleaf,fmmLL_UR,emptyleaf,fmmLR.m,fmmUL.m,fmmUL.col_idx,
                fmmLR.col_idx,offDiag_depth, Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),
                    Array{Float64}(undef,0,0),
                B31,B41,B42,Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(
                    undef,0,0),
                Array{Float64}(undef,0,0));

    fmmUR, fmmLL

elseif isa(fmmUL,Leaf) && !isa(fmmLR,Leaf)
    #right block of upper right corner and bottom block of lower left corner
    #(these are computed at the same time. One block will be of
    #dimension MxN, the other will always be of dimension NxM)
    #
```

```
#    ----------------
#   \        \    \    \
#   \    D   \    \B13\<-- MxN
#   \        \    \    \
#   \----------------\
#   \        \    \    \
#   \-------\---\---\
#   \  B31  \    \    \
#    ----------------
#        ^     (5)
#        \
#       NxM

col_idx = col_idx +fmmLR.fmmUL.m;
B13, B31 = FMM_Expansion_Coeffs_OffDiag_Outter(fmmUL,fmmLR.fmmLR,row_idx,col_idx,N,f);

#left block of upper right corner and top block of lower left corner
#(these are computed at the same time. One block will be of
#dimension MxN, the other will always be of dimension NxM)
#
#    ----------------
#   \        \    \    \
#   \    D   \ x \    \<-- MxN
#   \        \    \    \
#   \----------------\
#   \    x   \    \    \
#   \-------\---\---\
#   \        \    \    \
#    ----------------
#        ^     (6)
#        \
#       NxM


col_idx = col_idx - fmmLR.fmmUL.m;

fmmUR_L, fmmLL_L = FMM_Expansion_Coeffs_OffDiag(fmmUL,fmmLR.fmmUL,row_idx,col_idx,N,f);
emptyleaf = LeafOffDiag(-1,-1,-1,-1,-1,Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),
            Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),
                Array{Float64}(undef,0,0));
fmmUR = Node(emptyleaf, emptyleaf, emptyleaf,fmmUR_L,fmmUL.m,fmmLR.m,fmmLR.col_idx,
            fmmUL.col_idx,fmmLR.depth ,B13,Array{Float64}(undef,0,0),Array{Float64}(undef
                ,0,0),Array{Float64}(undef,0,0),
            Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),
                Array{Float64}(undef,0,0),
            Array{Float64}(undef,0,0),Array{Float64}(undef,0,0));
fmmLL = Node(emptyleaf,emptyleaf,fmmLL_L,emptyleaf,fmmLR.m,fmmUL.m,fmmUL.col_idx,
            fmmLR.col_idx,fmmLR.depth, Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),
                Array{Float64}(undef,0,0),
            B31,Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(undef,0,0)
                ,
            Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(undef,0,0));
```

```
    fmmUR, fmmLL

elseif !isa(fmmUL,Leaf) && isa(fmmLR,Leaf)
    #upper block of upper right corner, and left block of lower left corner
    #(these are computed at the same time. One block will be of
    #dimension MxN, the other will always be of dimension NxM)
    #
    #    ---------------
    #   \   \   \  B13  \
    #   \---\---\-------\
    #   \   \   \       \
    #   \---------------\
    #   \   \   \       \
    #   \B31\   \       \
    #   \   \   \       \
    #    ---------------
    #           (7)

    B13, B31 = FMM_Expansion_Coeffs_OffDiag_Outter(fmmUL.fmmUL,fmmLR,row_idx,col_idx,N,f);

    #left block of upper right corner and top block of lower left corner
    #(these are computed at the same time. One block will be of
    #dimension MxN, the other will always be of dimension NxM)
    #
    #    ---------------
    #   \   \   \       \
    #   \---\---\-------\
    #   \   \   \   x   \
    #   \---------------\
    #   \   \   \       \
    #   \   \ x \       \
    #   \   \   \       \
    #    ---------------
    #          (8)


    row_idx = row_idx + fmmUL.fmmUL.m;
    fmmUR_R, fmmLL_R = FMM_Expansion_Coeffs_OffDiag(fmmUL.fmmLR,fmmLR,row_idx,col_idx,N,f);

    emptyleaf = LeafOffDiag(-1,-1,-1,-1,Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),
                Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),
                    Array{Float64}(undef,0,0));
    fmmUR = Node(emptyleaf, emptyleaf, emptyleaf,fmmUR_R,fmmUL.m,fmmLR.m,fmmLR.col_idx,
                fmmUL.col_idx,fmmUL.depth ,B13,Array{Float64}(undef,0,0),Array{Float64}(undef
                    ,0,0),Array{Float64}(undef,0,0),
                Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),
                    Array{Float64}(undef,0,0),
                Array{Float64}(undef,0,0),Array{Float64}(undef,0,0));
    fmmLL = Node(emptyleaf,emptyleaf,fmmLL_R,emptyleaf,fmmLR.m,fmmUL.m,fmmUL.col_idx,
                fmmLR.col_idx,fmmUL.depth, Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),
                    Array{Float64}(undef,0,0),
```

```
                        B31, Array{Float64}(undef,0,0), Array{Float64}(undef,0,0), Array{Float64}(undef,0,0)
                            ,
                        Array{Float64}(undef,0,0), Array{Float64}(undef,0,0), Array{Float64}(undef,0,0));

        fmmUR, fmmLL

    end
end

function FMM_Expansion_Coeffs_OffDiag_Outter(fmmL::FMM,fmmR::FMM, row_idx::Int64, col_idx::Int64,N::
    Int64, f::Function)
#function FMM_Expansion_Coeffs_OffDiag(treeL::Hss, treeR::Hss, row_idx::Int64, col_idx::Int64,N::Int64)
#Computes Expansion Coefficients (B's) of HSS structure for the upper right
#and lower left block of the current node
#   -------
#  \    \ x \
#  \--------\
#  \ x \    \
#   -------
#INPUT:              fmmL   (Union(Node,Leaf)) left branch of current node
#                    fmmR   (Union(Node,Leaf)) right branch of current node
#                    row_idx (Int64) row index of current node
#                    col_idx (Int64) col index of current node
#                    N       (Int64) grid size
#OUTPUT:             Bu      (Array{Float62,2}) Expansion coefficient matrix B at the
#                            current level which corresponds to the upper
#                            right block
#                    Bl      (Array{Float62,2}) Expansion coefficient matrix B at the
#                            current level which corresponds to the lower
#                            left block

    Au = Array{Float64,2};
    Bu = Array{Float64,2};
    Al = Array{Float64,2};
    Bl = Array{Float64,2};

    bu_l = Array{Float64,2};
    bu_r = Array{Float64,2};
    bl_l = Array{Float64,2};
    bl_r = Array{Float64,2};
    if isa(fmmL, Leaf) && isa(fmmR, Leaf) #If node is a leaf
        m_idx = collect((row_idx):(row_idx+fmmL.m-1));
        butm_idx = collect(col_idx:(col_idx+fmmR.m-1));

        Au = f(m_idx,butm_idx,N);
        Bu = fmmL.U'*Au*fmmR.V;

        Au = Array{Float64}(undef,0);

        Al = f(butm_idx,m_idx,N);
        Bl = fmmR.U'*Al*fmmL.V;
```

```
    Al = Array{Float64}(undef,0);

    Bu, Bl

elseif !isa(fmmL, Leaf) && !isa(fmmR,Leaf)  #If neither node is a leaf
    #COMPUTE EXPANSION COEFFICIENTS

    #upper left corner of both upper right and lower left blocks
    #(these are computed at the same time. One block will be of
    #dimension MxN, the other will always be of dimension NxM)
    #
    #    ----------------
    #   \        \ x \<--\-- MxN
    #   \         -------\
    #   \        \   \   \
    #   \----------------\
    #   \ x \<--\-------\-- NxM
    #   \-------\        \
    #   \   \   \        \
    #    ----------------
    #           (1)

    Bu_ll, Bl_ll = FMM_Expansion_Coeffs_OffDiag_Outter(fmmL.fmmUL,fmmR.fmmUL,row_idx,col_idx,N,f);
    bu_ll = fmmL.Rl'*Bu_ll*fmmR.Wl;
    bl_ll = fmmR.Rl'*Bl_ll*fmmL.Wl;

    Bu_ll = Array{Float64}(undef,0);
    Bl_ll = Array{Float64}(undef,0);

    #Lower left corner lower left block, and also the upper right corner of
    #the upper right block
    #    ----------------
    #   \        \   \ x \
    #   \        \---\---\
    #   \        \   \   \
    #   \----------------\
    #   \   \   \        \
    #   \-------\        \
    #   \ x \   \        \
    #    ----------------
    #           (2)

    col_idx = col_idx +fmmR.fmmUL.m;
    Bu_lr, Bl_lr = FMM_Expansion_Coeffs_OffDiag_Outter(fmmL.fmmUL,fmmR.fmmLR,row_idx,col_idx,N,f);
    bu_lr = fmmL.Rl'*Bu_lr*fmmR.Wr;
    bl_lr = fmmR.Rr'*Bl_lr*fmmL.Wl;

    Bu_lr = Array{Float64}(undef,0);
    Bl_lr = Array{Float64}(undef,0);

    #upper right corner of lower block, lower left corner of upper block
    #    ----------------
```

```
#   \        \   \   \
#   \        \---\---\
#   \        \ x \   \
#   \----------------\
#   \   \ x \        \
#   \-------\        \
#   \   \   \        \
#    ---------------
#           (3)

col_idx = col_idx -fmmR.fmmUL.m;
row_idx = row_idx +fmmL.fmmUL.m;
Bu_rl, Bl_rl = FMM_Expansion_Coeffs_OffDiag_Outter(fmmL.fmmLR,fmmR.fmmUL,row_idx,col_idx,N,f);
bu_rl = fmmL.Rr'*Bu_rl*fmmR.Wl;
bl_rl = fmmR.Rl'*Bl_rl*fmmL.Wr;

Bu_rl = Array{Float64}(undef,0);
Bl_rl = Array{Float64}(undef,0);

#lower right corner of both lower left and upper left blocks
#    ---------------
#   \        \   \   \
#   \        \---\---\
#   \        \   \ x \
#   \----------------\
#   \   \   \        \
#   \-------\        \
#   \   \ x \        \
#    ---------------
#           (4)

col_idx = col_idx +fmmR.fmmUL.m;
Bu_rr, Bl_rr = FMM_Expansion_Coeffs_OffDiag_Outter(fmmL.fmmLR,fmmR.fmmLR,row_idx,col_idx,N,f);
bu_rr = fmmL.Rr'*Bu_rr*fmmR.Wr;
bl_rr = fmmR.Rr'*Bl_rr*fmmL.Wr;

Bu_rr = Array{Float64}(undef,0);
Bl_rr = Array{Float64}(undef,0);

#Add b's to form the B's
Bu = bu_ll+bu_rl+bu_lr+bu_rr;
Bl = bl_ll+bl_rl+bl_lr+bl_rr;

bu_ll = Array{Float64}(undef,0);
bu_rl = Array{Float64}(undef,0);
bu_lr = Array{Float64}(undef,0);
bu_rr = Array{Float64}(undef,0);
bl_ll = Array{Float64}(undef,0);
bl_rl = Array{Float64}(undef,0);
bl_lr = Array{Float64}(undef,0);
bl_rr = Array{Float64}(undef,0);
```

```
    Bu, Bl

elseif isa(fmmL, Leaf) && !isa(fmmR, Leaf) #If left node is a leaf, and right node is not
    #COMPUTE EXPANSION COEFFICIENTS

    #left block of upper right corner and upper block of lower left corner
    #(these are computed in the same pass)
    #    ----------------
    #   \        \   \   \
    #   \        \ x \<--\-- MxN
    #   \        \   \   \
    #   \----------------\
    #   \   x<--\--------\-- NxM
    #   \-------\        \
    #   \        \        \
    #    ----------------
    #           (5)

    Bu_l, Bl_l = FMM_Expansion_Coeffs_OffDiag_Outter(fmmL,fmmR.fmmUL,row_idx,col_idx,N,f);
    bu_l = Bu_l*fmmR.Wl;
    bl_l = fmmR.Rl'*Bl_l;

    Bu_l = Array{Float64}(undef,0);
    Bl_l = Array{Float64}(undef,0);

    #right block of upper right corner, upper left block of lower left corner.
    #    ----------------
    #   \        \   \   \
    #   \        \   \ x \
    #   \        \   \   \
    #   \----------------\
    #   \        \        \
    #   \-------\        \
    #   \   x   \        \
    #    ----------------
    #           (6)

    col_idx = col_idx + fmmR.fmmUL.m;
    Bu_r, Bl_r = FMM_Expansion_Coeffs_OffDiag_Outter(fmmL,fmmR.fmmLR,row_idx,col_idx,N,f);
    col_idx = col_idx - fmmR.fmmUL.m;
    bu_r = Bu_r*fmmR.Wr;
    bl_r = fmmR.Rr'*Bl_r;

    Bu_r = Array{Float64}(undef,0);
    Bl_r = Array{Float64}(undef,0);

    Bu = bu_l + bu_r;
    Bl = bl_l + bl_r;

    bu_l = Array{Float64}(undef,0);
    bu_r = Array{Float64}(undef,0);
    bl_l = Array{Float64}(undef,0);
```

```
    bl_r = Array{Float64}(undef,0);

    Bu, Bl

elseif !isa(fmmL, Leaf) && isa(fmmR, Leaf) #If right node is a leaf, and left node is not
    #COMPUTE EXPANSION COEFFICIENTS

    #upper block of upper right corner and left block of lower left corner
    #(these are computed in the same pass)
    #    ----------------
    #   \        \    x <-\-- MxN
    #   \        \--------\
    #   \        \         \
    #   \----------------\
    #   \    \    \         \
    #   \ x \<--\--------\-- NxM
    #   \    \    \         \
    #    ----------------
    #            (7)

    Bu_l, Bl_l = FMM_Expansion_Coeffs_OffDiag_Outter(fmmL.fmmUL,fmmR,row_idx,col_idx,N,f);
    bu_l = fmmL.Rl'*Bu_l;
    bl_l = Bl_l*fmmL.Wl;

    Bu_l = Array{Float64}(undef,0);
    Bl_l = Array{Float64}(undef,0);

    #upper block of upper right corner and left block of lower left corner
    #(these are computed in the same pass)
    #    ----------------
    #   \        \        \
    #   \        \--------\
    #   \        \    x   \
    #   \----------------\
    #   \    \    \         \
    #   \    \ x \         \
    #   \    \    \         \
    #    ----------------
    #            (8)

    row_idx = row_idx + fmmL.fmmUL.m;
    Bu_r, Bl_r = FMM_Expansion_Coeffs_OffDiag_Outter(fmmL.fmmLR,fmmR,row_idx,col_idx,N,f);
    row_idx = row_idx - fmmL.fmmUL.m;
    bu_r = fmmL.Rr'*Bu_r;
    bl_r = Bl_r*fmmL.Wr;

    Bu_r = Array{Float64}(undef,0);
    Bl_r = Array{Float64}(undef,0);

    Bu = bu_l + bu_r;
    Bl = bl_l + bl_r;
```

```
        bu_l = Array{Float64}(undef,0);
        bu_r = Array{Float64}(undef,0);
        bl_l = Array{Float64}(undef,0);
        bl_r = Array{Float64}(undef,0);

        Bu, Bl
    end
end
```

A.2.2 FMM Matrix-Vector Multiply

```
function FMM_Multiply(fmm::FMM,x::Array{Float64,2}) #(fmm::FMM,x::Array{Float64}(2))
#function [b] = HSS_Multiply(hss,x)
#Takes in an a matrix in fmm form, A_fmm, and multiplies this with an input
#array, x.  Gives an array b as output.  A_fmm*x = b.
#
#INPUT:              fmm     (FMM) a matrix in fmm form as given by the
#                            function FMM_Construction(). Contains partition
#                            dimensions and matrices (U, V, R,W and B), which are
#                            heirachically stored, and compose the fmm
#                            representation of the input matrix.
#                    x       (Array{Float64,2}) input array to multiply with A.
#
#OUTPUT:             b       (Array{Float64,2}) output array from the resulting multiplication
#
# Author: Kristen Lessel, klessel@engineering.ucsb.edu
#
# Copyright (C) 2016 Kristen Lessel, (klessel@engineering.ucsb.edu)
#
# This program is free software: you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation, either version 3 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful,
# but WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
# GNU General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program.  If not, see <http://www.gnu.org/licenses/>.

    # Notes regarding notation:
    # fmm0 contains 4 substructures, fmmUL, fmmUR, fmmLL and fmmLR, shown here:
    #      fmm0
    #    ---------
    #   \ UL \ UR \
    #   \----\----\
    #   \ LL \ LR \
    #    ---------
    #
    # Visual interpretation of a left call:
```

```
#     fmmL                fmm0                fmmR
#    ________           ________           ________
#   \    \ x \         \ x \ x \         \    \    \
#   \---\---\         \---\---\         \---\---\
#   \    \    \         \    \    \         \    \    \
#    ________           ________           ________
# Next level:
#                fmmL                fmm0    fmmR
#    ________________           ________________
#   \    \    \    \ x \         \ x \ x \    \    \
#   \________\________\         \________\________\
#   \    \    \    \    \         \    \    \    \    \
#   \________________\         \________________\
#   \    \    \    \    \         \    \    \    \    \
#   \________________\         \________\________\
#   \    \    \    \    \         \    \    \    \    \
#    ________________           ________________
# So when going down one level in the fmm tree we set our new fmm0 = fmm0.fmmUL,
# set the new fmmL =  fmmL.fmmUR, and set the new fmmR = fmm0.fmmUR
# (right calls work similarly)
# Visual interpretation of a right call:
#     fmmL                fmm0                fmmR
#    ________           ________           ________
#   \    \    \         \    \    \         \    \    \
#   \---\---\         \---\---\         \---\---\
#   \    \    \         \ x \ x \         \ x \    \
#    ________           ________           ________
# Next level:
#       fmmL    fmmO                fmmR
#    ________________           ________________
#   \    \    \    \    \         \    \    \    \    \
#   \________\________\         \________\________\
#   \    \    \    \    \         \    \    \    \    \
#   \________________\         \________________\
#   \    \    \    \    \         \    \    \    \    \
#   \________\________\         \________\________\
#   \    \    \ x \ x \         \ x \    \    \    \
#    ________________           ________________
```

```
f = Array{Float64}(undef,0,1); # Array(Float64, size(fmm.Rl,2),1);

gtree = FMM_Upsweep(fmm,x);

depth = 0;
empty_leafoffdiag = LeafOffDiag(-1,-1,-1,-1,depth,Array{Float64}(undef,0,0),Array{Float64}(undef
    ,0,0),
                        Array{Float64}(undef,0,0),Array{Float64}(undef,0,0),Array{Float64}(undef
                            ,0,0),
                        Array{Float64}(undef,0,0));

g = Array{Float64}(undef,0,1);
```

```
    empty_gleaf = Leaf_Gtree(0,1,depth,g);

    b = FMM_Downsweep(empty_leafoffdiag,fmm,empty_leafoffdiag,x,f,empty_gleaf,gtree,empty_gleaf);

    b

end

function FMM_Upsweep(fmm,x)
#upsweep to compute g's

if isa(fmm,Leaf)

    g = fmm.V'*x;
    gtree = Leaf_Gtree(fmm.m,fmm.col_idx,fmm.depth,g);

    gtree
else
    xl = x[1:fmm.fmmUL.m,:];
    gtreeL = FMM_Upsweep(fmm.fmmUL,xl);

    xr = x[fmm.fmmUL.m+1:end,:];
    gtreeR = FMM_Upsweep(fmm.fmmLR,xr);

    #new g
    g = fmm.Wl'*gtreeL.g+fmm.Wr'*gtreeR.g;

    gtree = Node_Gtree(gtreeL,gtreeR,fmm.m,fmm.col_idx,fmm.depth,g)

    gtree
end

end

function FMM_Downsweep(fmmL,fmm0,fmmR,x,f,gtreeL,gtree0,gtreeR)
#downsweep to compute f's

if isa(gtreeL, Leaf_Gtree) && isa(gtree0, Leaf_Gtree) && isa(gtreeR, Leaf_Gtree)
# Case 1) if all are leaves

    x_l = x[gtreeL.col_idx:(gtreeL.col_idx+gtreeL.m-1),:];
    x_0 = x[gtree0.col_idx:(gtree0.col_idx+gtree0.m-1),:];
    x_r = x[gtreeR.col_idx:(gtreeR.col_idx+gtreeR.m-1),:];

    b = fmm0.U*f + fmm0.D_l*x_l + fmm0.D_0*x_0 + fmm0.D_r*x_r;
    b

#elseif isa(fmm0, Node)
elseif isa(gtreeL, Node_Gtree) && isa(gtree0,Node_Gtree) && isa(gtreeR,Node_Gtree)
    # Case 2) if all are nodes
    #       fmmL     fmm0     fmmR
    #     ---------------------------------
```

```
    #     \ . \ . \   \   \       \       \
    #     \-------\-------\       \       \
    #     \ . \ . \ . \   \       \       \
    #      ---------------\---------------
    #     \B31\ . \ . \ . \B13\B14\       \
    # 2) \-------\-------\-------\       \
    #     \B41\B42\ . \ . \ . \B24\       \
    #      -------------------------------
    #     \       \   \ . \ . \ . \   \   \
    #     \       \---\---\---\---\---\---\
    #     \       \   \   \ . \ . \ . \   \
    #      ---------------\---------------
    #     \       \       \   \ . \ . \ . \
    #     \       \       \-------\-------\
    #     \       \       \   \   \ . \ . \
    #      -------------------------------

    fl =  fmm0.Rl*f + fmmL.B31*gtreeL.gtreeL.g + fmmR.B13*gtreeR.gtreeL.g + fmmR.B14*gtreeR.gtreeR.g;
    bl = FMM_Downsweep(fmmL.fmmUR,fmm0.fmmUL,fmm0.fmmUR,x,fl,gtreeL.gtreeR,gtree0.gtreeL,gtree0.gtreeR
        );

    fr = fmm0.Rr*f + fmmL.B41*gtreeL.gtreeL.g + fmmL.B42*gtreeL.gtreeR.g + fmmR.B24*gtreeR.gtreeR.g;
    br = FMM_Downsweep(fmm0.fmmLL,fmm0.fmmLR,fmmR.fmmLL,x,fr,gtree0.gtreeL,gtree0.gtreeR,gtreeR.gtreeL
        )
    b = [bl; br];

    b

elseif isa(gtreeL, Node_Gtree) && isa(gtree0, Node_Gtree) && isa(gtreeR, Leaf_Gtree)
    #Case 3) if Left and Center Trees are nodes, Right is Leaf
    #       fmmL     fmm0     fmmR
    #      -------------------------------
    #     \ . \ . \   \   \       \       \
    #     \-------\-------\       \       \
    #     \ . \ . \ . \   \       \       \
    #      ---------------\---------------
    #     \   \ . \ . \ . \  B13  \       \
    # 3) \-------\-------\-------\       \
    #     \   \   \ . \ . \   .   \       \
    #      -------------------------------
    #     \       \   \   \       \   \   \
    #     \       \   \ . \   .   \ . \   \
    #     \       \   \   \       \   \   \
    #      ---------------\---------------
    #     \       \       \   \ . \ . \ . \
    #     \       \       \-------\-------\
    #     \       \       \   \   \ . \ . \
    #      -------------------------------

    #fix addition of 0x0 (empty) arrays to px1 arrays (for the second call of the routine each B*g
        needs to be of size px1 for the addition to go through)
    if isempty(fmmR.B13)
```

```
        #If we are at the right most branch of the tree
        fl = fmm0.Rl*f + fmmL.B31*gtreeL.gtreeL.g;
    else
        fl = fmm0.Rl*f + fmmL.B31*gtreeL.gtreeL.g + fmmR.B13*gtreeR.g;
    end
    bl = FMM_Downsweep(fmmL.fmmUR,fmm0.fmmUL, fmm0.fmmUR,x,fl,gtreeL.gtreeR,gtree0.gtreeL,gtree0.
        gtreeR);

    fr = fmm0.Rr*f + fmmL.B41*gtreeL.gtreeL.g + fmmL.B42*gtreeL.gtreeR.g;
    br = FMM_Downsweep(fmm0.fmmLL,fmm0.fmmLR,fmmR,x,fr,gtree0.gtreeL,gtree0.gtreeR,gtreeR);

    b = [bl;br];

elseif isa(gtreeL, Node_Gtree) && isa(gtree0, Leaf_Gtree) && isa(gtreeR, Node_Gtree)
    #Case 4) if Left and Right Trees are nodes, Center is Leaf
    #                fmmL      fmm0    fmmR
    #     ---------------------------------
    #    \ . \ . \   \   \        \        \
    #    \-------\-------\        \        \
    #    \ . \ . \ . \   \        \        \
    #     ---------------\----------------
    #    \   \ . \ . \ . \        \        \
    #    \-------\-------\-------\        \
    #    \   \   \ . \ . \   .   \        \
    #     ---------------------------------
    #    \        \   \   \        \   \   \
    # 4) \        \B31\ . \   .    \ . \B13\
    #    \        \   \   \        \   \   \
    #     ---------------\----------------
    #    \        \        \   .   \ . \ . \
    #    \        \        \-------\-------\
    #    \        \        \        \ . \ . \
    #     ---------------------------------

    # Here R = I
    f_new = f + fmmL.B31*gtreeL.gtreeL.g + fmmR.B13*gtreeR.gtreeR.g;
    b = FMM_Downsweep(fmmL.fmmUR,fmm0,fmmR.fmmLL,x,f_new,gtreeL.gtreeR,gtree0,gtreeR.gtreeL);

    b

elseif isa(gtreeL, Leaf_Gtree) && isa(gtree0,Node_Gtree) && isa(gtreeR,Node_Gtree)
    #Case 5) if left tree is a leaf, and center and right trees are nodes
    #                fmmL      fmm0    fmmR
    #     ---------------------------------
    #    \        \        \        \        \
    #    \   .    \   .    \        \        \
    #    \        \        \        \        \
    #     ---------------\----------------
    #    \        \        \   \   \        \
    #    \   .    \   .    \ . \   \        \
    #    \        \        \   \   \        \
    #     ---------------------------------
```

```
    #    \        \    .    \ . \ . \    \    \
    # 5) \        \-------\-------\-------\
    #    \        \  B31   \ . \ . \ . \    \
    #     ----------------\----------------
    #    \        \        \    \ . \ . \ . \
    #    \        \        \-------\-------\
    #    \        \        \    \    \ . \ . \
    #     ---------------------------------

    fl = fmm0.Rl*f + fmmR.B13*gtreeR.gtreeL.g + fmmR.B14*gtreeR.gtreeR.g;
    bl = FMM_Downsweep(fmmL,fmm0.fmmUL,fmm0.fmmUR,x,fl,gtreeL,gtree0.gtreeL,gtree0.gtreeR);

    #fix addition of 0x0 arrays to px1 arrays (for the second call of the routine each B*g needs to be
        of size px1 for the addition to go through)
    if isempty(fmmL.B31)
        #If we are at the leftmost branch of the tree
        fr = fmm0.Rr*f + fmmR.B24*gtreeR.gtreeR.g;
    else
        fr = fmm0.Rr*f + fmmL.B31*gtreeL.g + fmmR.B24*gtreeR.gtreeR.g;
    end
    br = FMM_Downsweep(fmm0.fmmLL,fmm0.fmmLR,fmmR.fmmLL,x,fr,gtree0.gtreeL,gtree0.gtreeR,gtreeR.gtreeL
        );

    b = [bl;br];

    b

elseif isa(gtreeL, Leaf_Gtree) && isa(gtree0,Leaf_Gtree) && isa(gtreeR,Node_Gtree)
    #Case 6) if left and center trees are leaves, and right tree is a node
    #       fmmL    fmm0     fmmR
    #     ---------------------------------
    #    \        \        \        \        \
    #    \   .    \   .    \        \        \
    #    \        \        \        \        \
    #     ----------------\----------------
    #    \        \        \    \    \        \
    # 6) \   .    \   .    \ . \B13\        \
    #    \        \        \    \    \        \
    #     ---------------------------------
    #    \        \    .   \ . \ . \    \    \
    #    \        \-------\-------\-------\
    #    \        \        \ . \ . \ . \    \
    #     ----------------\----------------
    #    \        \        \    \ . \ . \ . \
    #    \        \        \-------\-------\
    #    \        \        \    \    \ . \ . \
    #     ---------------------------------

    # Here R = I
    f_new = f + fmmR.B13*gtreeR.gtreeR.g;
    b = FMM_Downsweep(fmmL,fmm0,fmmR.fmmLL,x,f_new,gtreeL,gtree0,gtreeR.gtreeL);
```

```
    b

elseif isa(gtreeL, Node_Gtree) && isa(gtree0,Leaf_Gtree) && isa(gtreeR,Leaf_Gtree)
    #Case 7) if left tree is a Node, and center and right trees are leaves
    #        fmmL    fmm0     fmmR
    #     ------------------------------
    #    \ . \ . \        \        \        \
    #    \-------\-------\        \        \
    #    \ . \ . \   .    \        \        \
    #     ---------------\----------------
    #    \   \   \        \        \        \
    # 7) \B31\ . \   .    \   .    \        \
    #    \   \   \        \        \        \
    #     ------------------------------
    #    \        \        \        \   \   \
    #    \        \   .    \   .    \ . \   \
    #    \        \        \        \   \   \
    #     ---------------\----------------
    #    \        \        \   .    \ . \ . \
    #    \        \        \-------\-------\
    #    \        \        \        \ . \ . \
    #     ------------------------------

    f_new = f + fmmL.B31*gtreeL.gtreeL.g;
    b = FMM_Downsweep(fmmL.fmmUR,fmm0,fmmR,x,f_new,gtreeL.gtreeR,gtree0,gtreeR);

    b

elseif isa(gtreeL, Leaf_Gtree) && isa(gtree0,Node_Gtree) && isa(gtreeR,Leaf_Gtree)
    #Case 8) if left and right trees are Nodes, and center tree is a leaf
    # At the root both the left and the right leaves will be emtpy
    #        fmmL    fmm0     fmmR
    #     ------------------------------
    #    \        \   \   \        \        \
    #    \   .    \ . \   \        \        \
    #    \        \   \   \        \        \
    #     ---------------\----------------
    #    \   .    \ . \ . \  B13   \        \
    # 8) \-------\-------\-------\        \
    #    \  B31   \ . \ . \   .    \        \
    #     ------------------------------
    #    \        \   \   \        \        \
    #    \        \   \ . \   .    \   .    \
    #    \        \   \   \        \        \
    #     ---------------\----------------
    #    \        \        \        \        \
    #    \        \        \   .    \   .    \
    #    \        \        \        \        \
    #     ------------------------------

    if isempty(fmmR.B13)
        #We are at the right most branch of the tree ( at the root of the tree
```

```
        #these dimensions match, but with an uneven tree we have to modify the equation
        #for empty LeafOffDiags)
        fl = fmm0.Rl*f;
    else
        fl = fmm0.Rl*f + fmmR.B13*gtreeR.g;
    end
    bl = FMM_Downsweep(fmmL,fmm0.fmmUL,fmm0.fmmUR,x,fl,gtreeL,gtree0.gtreeL,gtree0.gtreeR);

    if isempty(fmmL.B31)
       #if we are at the leftmost branch of the tree (at the root of the tree
        #these dimensions match, but with an uneven tree we have to modify the equation
        #for empty LeafOffDiags)
        fr = fmm0.Rr*f;
    else
        fr = fmm0.Rr*f + fmmL.B31*gtreeL.g;
    end
    br = FMM_Downsweep(fmm0.fmmLL,fmm0.fmmLR,fmmR,x,fr,gtree0.gtreeL,gtree0.gtreeR,gtreeR);

    b = [bl;br];
    b
end

end
```

A.2.3 FMM Matrix-Matrix Multiply

```
#
# Author: Kristen Lessel, kristenlessel@gmail.com
# June 15 2018
# FMM x FMM Multiply and utilities
# This version allows for passage of Arrays of Arrays or OffsetArrays of Arrays
# This version stores F, B anc B_C as Offset Arrays instead of using functions

function FMM_FMM_Multiply(D::OffsetArray{OffsetArray{Array{Float64}}},U::OffsetArray{Array{Float64}},R
    ::Array{OffsetArray{Array{Float64}}},B::Array{OffsetArray{OffsetArray{Array{Float64}}}},W::Array{
    OffsetArray{Array{Float64}}},V::OffsetArray{Array{Float64}},D̃::OffsetArray{OffsetArray{Array{
    Float64}}},Ũ::OffsetArray{Array{Float64}},R̃::Array{OffsetArray{Array{Float64}}},B̃::Array{
    OffsetArray{OffsetArray{Array{Float64}}}},W̃::Array{OffsetArray{Array{Float64}}},Ṽ::OffsetArray{
    Array{Float64}},depth::Int64)
    #Multiplies 2 fmm matrices, with U,R,B,W,V being from the first 3 point fmm structure,
    #and Ũ,R̃,B̃,W̃,Ṽ being from the second 3 point fmm structure. Output is a 5 point fmm
    #structure stored in arrays as listed here:
    #Input:              D:: OffsetArray{OffsetArray{Array{Float64}}}
    #                    U,V:: OffsetArray{Array{Float64}}
    #                    R,W:: Array{OffsetArray{Array{Float64}}}
    #                    B:: Array{OffsetArray{Array{Array{Float64}}}}
    #                    depth:: Int64  (depth of the two input fmm structures)
    #Output:             D_C:: OffsetArray{OffsetArray{Array{Float64}}}
    #                    U_C,V_C:: Array{Array{Float64}}
    #                    R_C,W_C:: Array{Array{Array{Float64}}}
    #                    B_C:: Array{Array{Array{Float64}}}
```

```
#############################################################################
#Compute U_C, V_C
U_C = Array{Array{Float64}}(undef,2^(depth)+1);
V_C = Array{Array{Float64}}(undef,2^(depth)+1);
for i = 1:2 .^depth
    U_C[i] = [U[i] D[i][i-1]*Ũ[i-1] D[i][i]*Ũ[i] D[i][i+1]*Ũ[i+1]];
    V_C[i] = [Ṽ[i] D̃[i-1][i]'*V[i-1] D̃[i][i]'*V[i] D̃[i+1][i]'V[i+1]];
end

#############################################################################
# Upsweep to Compute G
G = Array{OffsetArray{Array{Float64}}}(undef,depth);
for k = 1:depth
    G[k] = OffsetArray{Array{Float64}}(undef,-1:2^(k)+4);
end

#Compute G at the leaves
for i = 1:2^depth
    G[depth][i] = V[i]'*Ũ[i];
end

#Compute G at the nodes
for k = depth:-1:2
    for i = 1:2^(k-1)
        G[k-1][i] = W[k][2i-1]'*G[k][2i-1]*R̃[k][2i-1]+W[k][2i]'*G[k][2i]R̃[k][2i];
    end
end

#assign empty G matrices to the right and left of the edge of each tree
#(matrices outside the indices 1:2^k are 0x0 empty matrices)
for k = 1:depth
    G[k][-1] = zeros(0,0);
    G[k][0] = zeros(0,0);
    G[k][2^k+1] = zeros(0,0);
    G[k][2^k+2] = zeros(0,0);
    G[k][2^k+3] = zeros(0,0);
    G[k][2^k+4] = zeros(0,0);
end

#############################################################################
#Compute R_C, W_C
R_C = Array{Array{Array{Float64}}}(undef,depth);
W_C = Array{Array{Array{Float64}}}(undef,depth);
for k = 1:depth
    R_C[k] = Array{Array{Float64}}(undef,2^(k)+2);
    W_C[k] = Array{Array{Float64}}(undef,2^(k)+2);
end

R_C[1][1] = R[1][1]; #if i wanted to add this into the loop I would just have to store the 4 empty
    columns of matrices of B1;*,*
R_C[1][2] = R[1][2];
W_C[1][1] = W[1][1];
```

```
W_C[1][2] = W[1][2];

for k = 2:depth

    for i = 1:2^(k-1)

        #create zero blocks of the appropriate dimensions
        ml_Rl, nl_Rl = size(R[k][2i-3]); #m_Rl = m_Rr
        ml_Rr, nl_Rr = size(R[k][2i-2]);
        m0_Rl, n0_Rl = size(R[k][2i-1]);
        m0_Rr, n0_Rr = size(R[k][2i]);
        mr_Rl, nr_Rl = size(R[k][2i+1]);
        mr_Rr, nr_Rr = size(R[k][2i+2]);

        ml_R̃l, nl_R̃l = size(R̃[k][2i-3]);
        ml_R̃r, nl_R̃r = size(R̃[k][2i-2]);
        m0_R̃l, n0_R̃l = size(R̃[k][2i-1]);
        m0_R̃r, n0_R̃r = size(R̃[k][2i]);
        mr_R̃l, nr_R̃l = size(R̃[k][2i+1]);
        mr_R̃r, nr_R̃r = size(R̃[k][2i+2]);

        ml_Wl, nl_Wl = size(W[k][2i-3]); #m_Wl = m_Wr
        ml_Wr, nl_Wr = size(W[k][2i-2]);
        m0_Wl, n0_Wl = size(W[k][2i-1]);
        m0_Wr, n0_Wr = size(W[k][2i]);
        mr_Wl, nr_Wl = size(W[k][2i+1]);
        mr_Wr, nr_Wr = size(W[k][2i+2]);

        ml_W̃l, nl_W̃l = size(W̃[k][2i-3]);
        ml_W̃r, nl_W̃r = size(W̃[k][2i-2]);
        m0_W̃l, n0_W̃l = size(W̃[k][2i-1]);
        m0_W̃r, n0_W̃r = size(W̃[k][2i]);
        mr_W̃l, nr_W̃l = size(W̃[k][2i+1]);
        mr_W̃r, nr_W̃r = size(W̃[k][2i+2]);

        mrr_R̃l, nrr_R̃l =size(R̃[k][2i+3]);
        mrr_Wl, nrr_Wl =size(W[k][2i+3]);

        R_C[k][2i-1] = [R[k][2i-1] B[k][2i-1][2i-3]*G[k][2i-3]*R̃[k][2i-3] zeros(m0_Rl,n0_R̃l) B[k
            ][2i-1][2i+1]*G[k][2i+1]*R̃[k][2i+1]+B[k][2i-1][2i+2]*G[k][2i+2]*R̃[k][2i+2];
                       zeros(ml_R̃r,n0_Rl) R̃[k][2i-2] zeros(ml_R̃r,n0_R̃l) zeros(ml_R̃r,nr_R̃l);
                       zeros(m0_R̃l,n0_Rl) zeros(m0_R̃l,nl_R̃r) R̃[k][2i-1] zeros(m0_R̃l,nr_R̃l);
                       zeros(m0_R̃r,n0_Rl) zeros(m0_R̃r,nl_R̃r) R̃[k][2i] zeros(m0_R̃r,nr_R̃l)];
        W_C[k][2i-1] = [W̃[k][2i-1] B̃[k][2i-3][2i-1]'*G[k][2i-3]'*W[k][2i-3] zeros(m0_W̃l,n0_Wl) B̃[k
            ][2i+1][2i-1]'*G[k][2i+1]'*W[k][2i+1] + B̃[k][2i+2][2i-1]'*G[k][2i+2]'*W[k][2i+2];
                       zeros(ml_Wr,n0_W̃l) W[k][2i-2] zeros(ml_Wr,n0_Wl) zeros(ml_Wr,nr_Wl);
                       zeros(m0_Wl,n0_W̃l) zeros(m0_Wl,nl_Wr) W[k][2i-1] zeros(m0_Wl,nr_Wl);
                       zeros(m0_Wr,n0_W̃l) zeros(m0_Wr,nl_Wr) W[k][2i] zeros(m0_Wr,nr_Wl)];

        R_C[k][2i] = [R[k][2i] B[k][2i][2i-3]*G[k][2i-3]*R̃[k][2i-3] + B[k][2i][2i-2]*G[k][2i-2]*R̃[
            k][2i-2] zeros(m0_Rr,n0_R̃l) B[k][2i][2i+2]*G[k][2i+2]*R̃[k][2i+2];
                      zeros(m0_R̃l,n0_Rr) zeros(m0_R̃l,nl_R̃l) R̃[k][2i-1] zeros(m0_R̃l,nr_Rr);
```

```
                zeros(m0_R̃r,n0_Rr) zeros(m0_R̃r,nl_R̃l) R̃[k][2i] zeros(m0_R̃r,nr_Rr);
                zeros(mr_R̃l,n0_Rr) zeros(mr_R̃l,nl_R̃l) zeros(mr_R̃l,n0_R̃l) R̃[k][2i+1]];
        W_C[k][2i] = [W̃[k][2i] B̃[k][2i-3][2i]'*G[k][2i-3]'*W[k][2i-3]+B̃[k][2i-2][2i]'*G[k][2i-2]'*
            W[k][2i-2] zeros(m0_W̃r,n0_Wl) B̃[k][2i+2][2i]'*G[k][2i+2]'*W[k][2i+2];
                zeros(m0_Wl,n0_W̃r) zeros(m0_Wl,nl_Wl) W[k][2i-1] zeros(m0_Wl,nr_Wl);
                zeros(m0_Wr,n0_W̃r) zeros(m0_Wr,nl_Wl) W[k][2i] zeros(m0_Wr,nr_Wl);
                zeros(mr_Wl,n0_W̃r) zeros(mr_Wl,nl_Wl) zeros(mr_Wl,n0_Wl) W[k][2i+1]];

        if i == 2^(k-1)
            R_C[k][2i+1] = [R[k][2i+1] B[k][2i+1][2i-1]*G[k][2i-1]*R̃[k][2i-1] zeros(mr_Rr,nr_R̃l) B
                [k][2i+1][2i+3]*G[k][2i+3]*R̃[k][2i+3] + B[k][2i+1][2i+4]*G[k][2i+4]*R̃[k][2i+4];
                        zeros(m0_R̃r,nr_Rl) R̃[k][2i] zeros(m0_R̃r,nr_R̃l) zeros(m0_R̃r,nrr_R̃l);
                        zeros(mr_R̃l,nr_Rl) zeros(mr_R̃l,n0_R̃r) R̃[k][2i+1] zeros(mr_R̃l,nrr_R̃l);
                        zeros(mr_R̃r,nr_Rl) zeros(mr_R̃r,n0_R̃r) R̃[k][2i+2] zeros(mr_R̃r,nrr_R̃l)];

            R_C[k][2i+2] = [R[k][2i+2] B[k][2i+2][2i-1]*G[k][2i-1]*R̃[k][2i-1]+ B[k][2i+2][2i]*G[k
                ][2i]*R̃[k][2i] zeros(mr_Rr,nr_R̃l) B[k][2i+2][2i+4]*G[k][2i+4]*R̃[k][2i+4];
                        zeros(mr_R̃l,nr_Rr) zeros(mr_R̃l,n0_R̃l) R̃[k][2i+1] zeros(mr_R̃l,nrr_R̃l);
                        zeros(mr_R̃r,nr_Rr) zeros(mr_R̃r,n0_R̃l) R̃[k][2i+2] zeros(mr_R̃r,nrr_R̃l);
                        zeros(mrr_R̃l,nr_Rr) zeros(mrr_R̃l,n0_R̃l) zeros(mrr_R̃l,nr_R̃l) R̃[k][2i
                            +3]];
            W_C[k][2i+1] = [W̃[k][2i+1] B̃[k][2i-1][2i+1]'*G[k][2i-1]'*W[k][2i-1] zeros(mr_W̃l,nr_Wl)
                B̃[k][2i+3][2i+1]'*G[k][2i+3]'*W[k][2i+3] + B̃[k][2i+4][2i+1]'*G[k][2i+4]'*W[k][2i
                +4];
                        zeros(m0_Wr,nr_W̃l) W[k][2i] zeros(m0_Wr,nr_Wl) zeros(m0_Wr,nrr_Wl);
                        zeros(mr_Wl,nr_W̃l) zeros(mr_Wl,n0_Wr) W[k][2i+1] zeros(mr_Wl,nrr_Wl);
                        zeros(mr_Wr,nr_W̃l) zeros(mr_Wr,n0_Wr) W[k][2i+2] zeros(mr_Wr,nrr_Wl)];
            W_C[k][2i+2] = [W̃[k][2i+2] B̃[k][2i-1][2i+2]'*G[k][2i-1]'*W[k][2i-1] zeros(mr_W̃r,nr_Wl)
                B̃[k][2i+4][2i+2]'*G[k][2i+4]'*W[k][2i+4];
                        zeros(mr_Wl,nr_Wr) zeros(mr_Wl,n0_Wl) W[k][2i+1] zeros(mr_Wl,nrr_Wl);
                        zeros(mr_Wr,nr_Wr) zeros(mr_Wr,n0_Wl) W[k][2i+2] zeros(mr_Wr,nrr_Wl);
                        zeros(mrr_Wl,nr_Wr) zeros(mrr_Wl,n0_Wl) zeros(mrr_Wl,nr_Wl) W[k][2i
                            +3]];
        end

end

end

#############################################################################
#Down-sweep recursions (F matrices)

F = Array{OffsetArray{OffsetArray{Array{Float64}}}}(undef,depth);
for k = 1:depth
    F[k] = OffsetArray{OffsetArray{Array{Float64}}}(undef,1:2^k+2);
    for i = 1:2^k+2
        F[k][i] = OffsetArray{Array{Float64}}(undef,i-2:i+2);
    end
end

F[1][1][1] = zeros(0,0);
F[1][1][2] = zeros(0,0);
```

```
F[1][1][3] = zeros(0,0);
F[1][2][1] = zeros(0,0);
F[1][3][1] = zeros(0,0);
F[1][2][2] = zeros(0,0);
F[1][2][3] = zeros(0,0);
F[1][3][2] = zeros(0,0);

for k = 2:depth
    for i = 1:2^(k-1)

        #create zero blocks of the appropriate dimensions
        ml_Rl, nl_Rl = size(R[k][2i-3]); #n_Rl = n_Rr
        ml_Rr, nl_Rr = size(R[k][2i-2]);
        m0_Rl, n0_Rl = size(R[k][2i-1]);
        m0_Rr, n0_Rr = size(R[k][2i]);
        mr_Rl, nr_Rl = size(R[k][2i+1]);
        mr_Rr, nr_Rr = size(R[k][2i+2]);

        ml_R̃l, nl_R̃l = size(R̃[k][2i-3]);
        ml_R̃r, nl_R̃r = size(R̃[k][2i-2]);
        m0_R̃l, n0_R̃l = size(R̃[k][2i-1]);
        m0_R̃r, n0_R̃r = size(R̃[k][2i]);
        mr_R̃l, nr_R̃l = size(R̃[k][2i+1]);
        mr_R̃r, nr_R̃r = size(R̃[k][2i+2]);

        ml_Wl, nl_Wl = size(W[k][2i-3]); #m_Wl = m_Wr
        ml_Wr, nl_Wr = size(W[k][2i-2]);
        m0_Wl, n0_Wl = size(W[k][2i-1]);
        m0_Wr, n0_Wr = size(W[k][2i]);
        mr_Wl, nr_Wl =size(W[k][2i+1]);
        mr_Wr, nr_Wr =size(W[k][2i+2]);

        ml_W̃l, nl_W̃l = size(W̃[k][2i-3]);
        ml_W̃r, nl_W̃r = size(W̃[k][2i-2]);
        m0_W̃l, n0_W̃l = size(W̃[k][2i-1]);
        m0_W̃r, n0_W̃r = size(W̃[k][2i]);
        mr_W̃l, nr_W̃l = size(W̃[k][2i+1]);
        mr_W̃r, nr_W̃r = size(W̃[k][2i+2]);

        mrr_R̃l, nrr_R̃l =size(R̃[k][2i+3]);
        mrr_Wl, nrr_Wl =size(W[k][2i+3]);

        #The size of the zero blocks in F correspond to the number of rows in each
        #block of R_C, and the number of columns in each block of W_C^H.

        #F[k][2i-1][2i-1]
        F[k][2i-1][2i-1] = [B[k][2i-1][2i-3]*G[k][2i-3]*B̃[k][2i-3][2i-1] + B[k][2i-1][2i+1]*G[k][2i
            +1]*B̃[k][2i+1][2i-1] + B[k][2i-1][2i+2]*G[k][2i+2]*B̃[k][2i+2][2i-1] zeros(m0_Rl,ml_Wr)
            zeros(m0_Rl,m0_Wl) zeros(m0_Rl,m0_Wr);
                                zeros(ml_R̃r,m0_W̃l) zeros(ml_R̃r,ml_Wr) zeros(ml_R̃r,m0_Wl) zeros(ml_R̃r,m0_Wr
                                    );
                                zeros(m0_R̃l,m0_W̃l) zeros(m0_R̃l,ml_Wr) zeros(m0_R̃l,m0_Wl) zeros(m0_R̃l,m0_Wr
```

```
                    );
                zeros(m0_R̃r,m0_W̃l) zeros(m0_R̃r,ml_Wr) zeros(m0_R̃r,m0_Wl) zeros(m0_R̃r,m0_Wr
                    )] +
                R_C[k][2i-1]*F[k-1][i][i]*W_C[k][2i-1]';

#F[k][2i,2i]
F[k][2i][2i] = [B[k][2i][2i-3]*G[k][2i-3]*B̃[k][2i-3][2i] + B[k][2i][2i-2]*G[k][2i-2]*B̃[k][2i
    -2][2i] + B[k][2i][2i+2]*G[k][2i+2]*B̃[k][2i+2][2i] zeros(m0_Rr,m0_Wl) zeros(m0_Rr,m0_Wr)
    zeros(m0_Rr,mr_Wl);
                zeros(m0_R̃l,m0_W̃r) zeros(m0_R̃l,m0_Wl) zeros(m0_R̃l,m0_Wr) zeros(m0_R̃l,mr_Wl);
                zeros(m0_R̃r,m0_W̃r) zeros(m0_R̃r,m0_Wl) zeros(m0_R̃r,m0_Wr) zeros(m0_R̃r,mr_Wl);
                zeros(mr_R̃l,m0_W̃r) zeros(mr_R̃l,m0_Wl) zeros(mr_R̃l,m0_Wr) zeros(mr_R̃l,mr_Wl)] +
                R_C[k][2i]*F[k-1][i][i]*W_C[k][2i]';

#F[k][2i-1,2i]
F[k][2i-1][2i] = [B[k][2i-1][2i-3]*G[k][2i-3]*B̃[k][2i-3][2i] + B[k][2i-1][2i+2]*G[k][2i+2]*B̃[k
    ][2i+2][2i] zeros(m0_Rl,m0_Wl) zeros(m0_Rl,m0_Wr) B[k][2i-1][2i+1];
                B̃[k][2i-2][2i] zeros(ml_R̃r,m0_Wl) zeros(ml_R̃r,m0_Wr) zeros(ml_R̃r,mr_Wl);
                zeros(m0_R̃l,m0_W̃r) zeros(m0_R̃l,m0_Wl) zeros(m0_R̃l,m0_Wr) zeros(m0_R̃l,mr_Wl);
                zeros(m0_R̃r,m0_W̃r) zeros(m0_R̃r,m0_Wl) zeros(m0_R̃r,m0_Wr) zeros(m0_R̃r,mr_Wl)]
                    +
                R_C[k][2i-1]*F[k-1][i][i]*W_C[k][2i]';

#F[k][2i,2i-1]
F[k][2i][2i-1] = [B[k][2i][2i-3]*G[k][2i-3]*B̃[k][2i-3][2i-1] + B[k][2i][2i+2]*G[k][2i+2]*B̃[k
    ][2i+2][2i-1] B[k][2i][2i-2] zeros(m0_Rr,m0_Wl) zeros(m0_Rr,m0_Wr);
                zeros(m0_R̃l,m0_W̃l) zeros(m0_R̃l,ml_Wr) zeros(m0_R̃l,m0_Wl) zeros(m0_R̃l,m0_Wr);
                zeros(m0_R̃r,m0_W̃l) zeros(m0_R̃r,ml_Wr) zeros(m0_R̃r,m0_Wl) zeros(m0_R̃r,m0_Wr);
                B̃[k][2i+1][2i-1] zeros(mr_R̃l,ml_Wr) zeros(mr_R̃l,m0_Wl) zeros(mr_R̃l,m0_Wr)] +
                R_C[k][2i]*F[k-1][i][i]*W_C[k][2i-1]';

#F[k][2i-1,2i+1]
F[k][2i-1][2i+1] = [zeros(m0_Rl,mr_W̃l) zeros(m0_Rl,m0_Wr) B[k][2i-1][2i+1] B[k][2i-1][2i+2];
                zeros(ml_R̃r,mr_W̃l) zeros(ml_R̃r,m0_Wr) zeros(ml_R̃r,mr_Wl) zeros(ml_R̃r,mr_Wr
                    );
                B̃[k][2i-1][2i+1] zeros(m0_R̃l,m0_Wr) zeros(m0_R̃l,mr_Wl) zeros(m0_R̃l,mr_Wr);
                zeros(m0_R̃r,mr_W̃l) zeros(m0_R̃r,m0_Wr) zeros(m0_R̃r,mr_Wl) zeros(m0_R̃r,mr_Wr
                    )] +
                R_C[k][2i-1]*F[k-1][i][i+1]*W_C[k][2i+1]';

#F[k][2i,2i+2]
F[k][2i][2i+2] = [zeros(m0_Rr,mr_W̃r) zeros(m0_Rr,mr_Wl) B[k][2i][2i+2] zeros(m0_Rr,mrr_Wl);
                B̃[k][2i-1][2i+2] zeros(m0_R̃l,mr_Wl) zeros(m0_R̃l,mr_Wr) zeros(m0_R̃l,mrr_Wl);
                B̃[k][2i][2i+2] zeros(m0_R̃r,mr_Wl) zeros(m0_R̃r,mr_Wr) zeros(m0_R̃r,mrr_Wl);
                zeros(mr_R̃l,mr_W̃r) zeros(mr_R̃l,mr_Wl) zeros(mr_R̃l,mr_Wr) zeros(mr_R̃l,mrr_Wl)
                    ] +
                R_C[k][2i]*F[k-1][i][i+1]*W_C[k][2i+2]';

#F[k][2i,2i+1]
F[k][2i][2i+1] = [zeros(m0_Rr,mr_W̃l) zeros(m0_Rr,m0_Wr) zeros(m0_Rr,mr_Wl) B[k][2i][2i+2];
                B̃[k][2i-1][2i+1] zeros(m0_R̃l,m0_Wr) zeros(m0_R̃l,mr_Wl) zeros(m0_R̃l,mr_Wr);
                zeros(m0_R̃r,mr_W̃l) zeros(m0_R̃r,m0_Wr) zeros(m0_R̃r,mr_Wl) zeros(m0_R̃r,mr_Wr);
```

```
                                zeros(mr_R̃l,mr_W̃l) zeros(mr_R̃l,m0_Wr) zeros(mr_R̃l,mr_Wl) zeros(mr_R̃l,mr_Wr)]
                                    +
R_C[k][2i]*F[k-1][i][i+1]*W_C[k][2i+1]';

#F[k][2i+1,2i-1]
F[k][2i+1][2i-1] = [zeros(mr_Rl,m0_W̃l) zeros(mr_Rl,ml_Wr) B[k][2i+1][2i-1] zeros(mr_Rl,m0_Wr);
                    zeros(m0_R̃r,m0_W̃l) zeros(m0_R̃r,ml_Wr) zeros(m0_R̃r,m0_Wl) zeros(m0_R̃r,m0_Wr);
                    B̃[k][2i+1][2i-1] zeros(mr_R̃l,ml_Wr) zeros(mr_R̃l,m0_Wl) zeros(mr_R̃l,m0_Wr);
                    B̃[k][2i+2][2i-1] zeros(mr_R̃r,ml_Wr) zeros(mr_R̃r,m0_Wl) zeros(mr_R̃r,m0_Wr)] +
                        R_C[k][2i+1]*F[k-1][i+1][i]*W_C[k][2i-1]';

#F[k][2i+1,2i]
F[k][2i+1][2i] = [zeros(mr_Rl,m0_W̃r) B[k][2i+1][2i-1] zeros(mr_Rl,m0_Wr) zeros(mr_Rl,mr_Wl);
                  zeros(m0_R̃r,m0_W̃r) zeros(m0_R̃r,m0_Wl) zeros(m0_R̃r,m0_Wr) zeros(m0_R̃r,mr_Wl);
                  zeros(mr_R̃l,m0_W̃r) zeros(mr_R̃l,m0_Wl) zeros(mr_R̃l,m0_Wr) zeros(mr_R̃l,mr_Wl);
                  B̃[k][2i+2][2i] zeros(mr_R̃r,m0_Wl) zeros(mr_R̃r,m0_Wr) zeros(mr_R̃r,mr_Wl)] +
                      R_C[k][2i+1]*F[k-1][i+1][i]*W_C[k][2i]';


#F[k][2i+2,2i]
F[k][2i+2][2i] = [zeros(mr_Rr,m0_W̃r) B[k][2i+2][2i-1] B[k][2i+2][2i] zeros(mr_Rr,mr_Wl);
                  zeros(mr_R̃l,m0_W̃r) zeros(mr_R̃l,m0_Wl) zeros(mr_R̃l,m0_Wr) zeros(mr_R̃l,mr_Wl);
                  B̃[k][2i+2][2i] zeros(mr_R̃r,m0_Wl) zeros(mr_R̃r,m0_Wr) zeros(mr_R̃r,mr_Wl);
                  zeros(mrr_R̃l,m0_W̃r) zeros(mrr_R̃l,m0_Wl) zeros(mrr_R̃l,m0_Wr) zeros(mrr_R̃l,mr_Wl)] +
                      R_C[k][2i+2]*F[k-1][i+1][i]*W_C[k][2i]';


end
end

##############################################################################
#Calculation of B's (Expansion Coefficients)

B_C = Array{OffsetArray{OffsetArray{Array{Float64}}}}(undef,depth);
for k = 1:depth
    B_C[k] = OffsetArray{OffsetArray{Array{Float64}}}(undef,1:2^k+5);
    for i = 1:2^k+5
        B_C[k][i] = OffsetArray{Array{Float64}}(undef,i-5:i+5);
    end
end

B_C[1][1][4] = zeros(0,0);
B_C[1][1][5] = zeros(0,0);
B_C[1][1][6] = zeros(0,0);
B_C[1][2][5] = zeros(0,0);
B_C[1][2][6] = zeros(0,0);
B_C[1][4][1] = zeros(0,0);
B_C[1][5][1] = zeros(0,0);
B_C[1][6][1] = zeros(0,0);
B_C[1][5][2] = zeros(0,0);
B_C[1][6][2] = zeros(0,0);
```

```
for k = 2:depth
    for i = 1:2^(k-1)-1

        #create zero blocks of the appropriate dimensions
        ml_Rl, nl_Rl = size(R[k][2i-3]); #n_Rl = n_Rr
        ml_Rr, nl_Rr = size(R[k][2i-2]);
        m0_Rl, n0_Rl = size(R[k][2i-1]);
        m0_Rr, n0_Rr = size(R[k][2i]);
        mr_Rl, nr_Rl = size(R[k][2i+1]);
        mr_Rr, nr_Rr = size(R[k][2i+2]);

        ml_R̃l, nl_R̃l = size(R̃[k][2i-3]);
        ml_R̃r, nl_R̃r = size(R̃[k][2i-2]);
        m0_R̃l, n0_R̃l = size(R̃[k][2i-1]);
        m0_R̃r, n0_R̃r = size(R̃[k][2i]);
        mr_R̃l, nr_R̃l = size(R̃[k][2i+1]);
        mr_R̃r, nr_R̃r = size(R̃[k][2i+2]);

        ml_Wl, nl_Wl = size(W[k][2i-3]); #m_Wl = m_Wr
        ml_Wr, nl_Wr = size(W[k][2i-2]);
        m0_Wl, n0_Wl = size(W[k][2i-1]);
        m0_Wr, n0_Wr = size(W[k][2i]);
        mr_Wl, nr_Wl =size(W[k][2i+1]);
        mr_Wr, nr_Wr =size(W[k][2i+2]);

        ml_W̃l, nl_W̃l = size(W̃[k][2i-3]);
        ml_W̃r, nl_W̃r = size(W̃[k][2i-2]);
        m0_W̃l, n0_W̃l = size(W̃[k][2i-1]);
        m0_W̃r, n0_W̃r = size(W̃[k][2i]);
        mr_W̃l, nr_W̃l = size(W̃[k][2i+1]);
        mr_W̃r, nr_W̃r = size(W̃[k][2i+2]);

        mrr_Rl, nrr_Rl =size(R[k][2i+3]);
        mrr_Rr, nrr_Rr =size(R[k][2i+4]);
        mrr_R̃l, nrr_R̃l =size(R̃[k][2i+3]);
        mrr_R̃r, nrr_R̃r =size(R̃[k][2i+4]);
        mrr_Wl, nrr_Wl =size(W[k][2i+3]);
        mrr_Wr, nrr_Wr =size(W[k][2i+4]);
        mrr_W̃l, nrr_W̃l =size(W̃[k][2i+3]);
        mrr_W̃r, nrr_W̃r =size(W̃[k][2i+4]);

        mrrr_Wl, mrrr_Wr = size(W[k][2i+5])
        mrrr_R̃l, mrrr_R̃r = size(R̃[k][2i+5])

        #The size of the zero blocks in B correspond to the number of rows in each
        #coresponding block of R_C, and the number of columns in each block of W_C^H.

        #B_C[k][2i-1,2i+2]
        B_C[k][2i-1][2i+2] = [zeros(m0_Rl,mr_W̃r) B[k][2i-1][2i+1] B[k][2i-1][2i+2] zeros(m0_Rl,mrr_Wl)
            ;
                              zeros(ml_R̃r,mr_W̃r) zeros(ml_R̃r,mr_Wl) zeros(ml_R̃r,mr_Wr) zeros(ml_R̃r,
                                  mrr_Wl);
```

```
                    B̃[k][2i-1][2i+2] zeros(m0_R̃l,mr_Wl) zeros(m0_R̃l,mr_Wr) zeros(m0_R̃l,
                        mrr_Wl);
                    B̃[k][2i][2i+2] zeros(m0_R̃r,mr_Wl) zeros(m0_R̃r,mr_Wr) zeros(m0_R̃r,mrr_Wl)
                        ] +
                        R_C[k][2i-1]*F[k-1][i][i+1]*W_C[k][2i+2]'


#B_C[k][2i+2,2i-1]
B_C[k][2i+2][2i-1] = [zeros(mr_Rr,m0_W̃l) zeros(mr_Rr,ml_Wr) B[k][2i+2][2i-1] B[k][2i+2][2i];
                    B̃[k][2i+1][2i-1] zeros(mr_R̃l,ml_Wr) zeros(mr_R̃l,m0_Wl) zeros(mr_R̃l,m0_Wr
                        );
                    B̃[k][2i+2][2i-1] zeros(mr_R̃r,ml_Wr) zeros(mr_R̃r,m0_Wl) zeros(mr_R̃r,m0_Wr
                        );
                    zeros(mrr_R̃l,m0_W̃l) zeros(mrr_R̃l,ml_Wr) zeros(mrr_R̃l,m0_Wl) zeros(mrr_R̃l
                        ,m0_Wr)] +
                        R_C[k][2i+2]*F[k-1][i+1][i]*W_C[k][2i-1]';

#B_C[k][2i-1,2i+3]
B_C[k][2i-1][2i+3] = [B[k][2i-1][2i+1]*G[k][2i+1]*B̃[k][2i+1][2i+3] B[k][2i-1][2i+2] zeros(
    m0_Rl,mrr_Wl) zeros(m0_Rl,mrr_Wr);
                    zeros(ml_R̃r,mrr_W̃l) zeros(ml_R̃r,mr_Wr) zeros(ml_R̃r,mrr_Wl) zeros(ml_R̃r,
                        mrr_Wr);
                    zeros(m0_R̃l,mrr_W̃l) zeros(m0_R̃l,mr_Wr) zeros(m0_R̃l,mrr_Wl) zeros(m0_R̃l,
                        mrr_Wr);
                    zeros(m0_R̃r,mrr_W̃l) zeros(m0_R̃r,mr_Wr) zeros(m0_R̃r,mrr_Wl) zeros(m0_R̃r,
                        mrr_Wr)] +
                        R_C[k][2i-1]*F[k-1][i][i+2]*W_C[k][2i+3]';

#B_C[k][2i-1,2i+4]
B_C[k][2i-1][2i+4] = [B[k][2i-1][2i+1]*G[k][2i+1]*B̃[k][2i+1][2i+4] + B[k][2i-1][2i+2]*G[k][2i
    +2]*B̃[k][2i+2][2i+4] zeros(m0_Rl,mrr_Wl) zeros(m0_Rl,mrr_Wr) zeros(m0_Rl,mrrr_Wl);
                    zeros(ml_R̃r,mrr_W̃r) zeros(ml_R̃r,mrr_Wl) zeros(ml_R̃r,mrr_Wr) zeros(ml_R̃r,
                        mrrr_Wl);
                    zeros(m0_R̃l,mrr_W̃r) zeros(m0_R̃l,mrr_Wl) zeros(m0_R̃l,mrr_Wr) zeros(m0_R̃l,
                        mrrr_Wl);
                    zeros(m0_R̃r,mrr_W̃r) zeros(m0_R̃l,mrr_Wl) zeros(m0_R̃l,mrr_Wr) zeros(m0_R̃l,
                        mrrr_Wl)] +
                        R_C[k][2i-1]*F[k-1][i][i+2]*W_C[k][2i+4]';

#B_C[k][2i,2i+3]
B_C[k][2i][2i+3] = [zeros(m0_Rr,mrr_W̃l) B[k][2i][2i+2] zeros(m0_Rr,mrr_Wl) zeros(m0_Rr,mrr_Wr)
    ;
                    zeros(m0_R̃l,mrr_W̃l) zeros(m0_R̃l,mr_Wr) zeros(m0_R̃l,mrr_Wl) zeros(m0_R̃l,
                        mrr_Wr);
                    zeros(m0_R̃r,mrr_W̃l) zeros(m0_R̃r,mr_Wr) zeros(m0_R̃r,mrr_Wl) zeros(m0_R̃r,
                        mrr_Wr);
                    B̃[k][2i+1][2i+3] zeros(mr_R̃l,mr_Wr) zeros(mr_R̃l,mrr_Wl) zeros(mr_R̃l,mrr_Wr
                        )] +
                        R_C[k][2i]*F[k-1][i][i+2]*W_C[k][2i+3]';

#B_C[k][2i,2i+4]
B_C[k][2i][2i+4] = [B[k][2i][2i+2]*G[k][2i+2]*B̃[k][2i+2][2i+4] zeros(m0_Rr,mrr_Wl) zeros(m0_Rr
```

```
,mrr_Wr) zeros(m0_Rr,mrrr_Wl);
                    zeros(m0_R̃l,mrr_W̃r) zeros(m0_R̃l,mrr_Wl) zeros(m0_R̃l,mrr_Wr) zeros(m0_R̃l,
                        mrrr_Wl);
                    zeros(m0_R̃r,mrr_W̃r) zeros(m0_R̃r,mrr_Wl) zeros(m0_R̃r,mrr_Wr) zeros(m0_R̃r,
                        mrrr_Wl);
                    B̃[k][2i+1][2i+4] zeros(mr_R̃l,mrr_Wl) zeros(mr_R̃l,mrr_Wr) zeros(mr_R̃l,
                        mrrr_Wl)] +
                        R_C[k][2i]*F[k-1][i][i+2]*W_C[k][2i+4]';

#B_C[k][2i+3,2i-1]
B_C[k][2i+3][2i-1] = [B[k][2i+3][2i+1]*G[k][2i+1]*B̃[k][2i+1][2i-1] zeros(mrr_Rl,ml_Wr) zeros(mrr_Rl,
    m0_Wl) zeros(mrr_Rl,mr_Wl);
                    B̃[k][2i+2][2i-1] zeros(mr_R̃r,ml_Wr) zeros(mr_R̃r,m0_Wl) zeros(mr_R̃r,mr_Wl);
                    zeros(mrr_R̃l,m0_W̃l) zeros(mrr_R̃l,ml_Wr) zeros(mrr_R̃l,m0_Wl) zeros(mrr_R̃l,mr_Wl);
                    zeros(mrr_R̃r,m0_W̃l) zeros(mrr_R̃r,ml_Wr) zeros(mrr_R̃r,m0_Wl) zeros(mrr_R̃r,mr_Wl)]
                        +
                        R_C[k][2i+3]*F[k-1][i+2][i]*W_C[k][2i-1]';

#B_C[k][2i+3,2i]
B_C[k][2i+3][2i] = [zeros(mrr_Rl,m0_W̃r) zeros(mrr_Rl,m0_Wl) zeros(mrr_Rl,m0_Wr) B[k][2i+3][2i+1];
                    B̃[k][2i+2][2i] zeros(mr_R̃r,m0_Wl) zeros(mr_R̃r,m0_Wr) zeros(mr_R̃r,mr_Wl);
                    zeros(mrr_R̃l,m0_W̃r) zeros(mrr_R̃l,m0_Wl) zeros(mrr_R̃l,m0_Wr) zeros(mrr_R̃l,mr_Wl);
                    zeros(mrr_R̃r,m0_W̃r) zeros(mrr_R̃r,m0_Wl) zeros(mrr_R̃r,m0_Wr) zeros(mrr_R̃r,mr_Wl)] +
                        R_C[k][2i+3]*F[k-1][i+2][i]*W_C[k][2i]';

#B_C[k][2i+4,2i-1]
B_C[k][2i+4][2i-1] = [B[k][2i+4][2i+1]*G[k][2i+1]*B̃[k][2i+1][2i-1] + B[k][2i+4][2i+2]*G[k][2i+2]*B̃[k
    ][2i+2][2i-1] zeros(mrr_Rr,ml_Wr) zeros(mrr_Rr,m0_Wl) zeros(mrr_Rr,m0_Wr);
                    zeros(mrr_R̃l,m0_W̃l) zeros(mrr_R̃l,ml_Wr) zeros(mrr_R̃l,m0_Wl) zeros(mrr_R̃l,m0_Wr);
                    zeros(mrr_R̃r,m0_W̃l) zeros(mrr_R̃r,ml_Wr) zeros(mrr_R̃r,m0_Wl) zeros(mrr_R̃r,m0_Wr);
                    zeros(mrrr_R̃l,m0_W̃l) zeros(mrrr_R̃l,ml_Wr) zeros(mrrr_R̃l,m0_Wl) zeros(mrrr_R̃l,
                        m0_Wr)] +
                        R_C[k][2i+4]*F[k-1][i+2][i]*W_C[k][2i-1]';

#B_C[k][2i+4,2i]
B_C[k][2i+4][2i] = [B[k][2i+4][2i+2]*G[k][2i+2]*B̃[k][2i+2][2i] zeros(mrr_Rr,m0_Wl) zeros(mrr_Rr,m0_Wr)
    B[k][2i+4][2i+1]
                    zeros(mrr_R̃l,m0_W̃r) zeros(mrr_R̃l,m0_Wl) zeros(mrr_R̃l,m0_Wr) zeros(mrr_R̃l,mr_Wl);
                    zeros(mrr_R̃r,m0_W̃r) zeros(mrr_R̃r,m0_Wl) zeros(mrr_R̃r,m0_Wr) zeros(mrr_R̃r,mr_Wl);
                    zeros(mrrr_R̃l,m0_W̃r) zeros(mrrr_R̃l,m0_Wl) zeros(mrrr_R̃l,m0_Wr) zeros(mrrr_R̃l,mr_Wl
                        )] +
                        R_C[k][2i+4]*F[k-1][i+2][i]*W_C[k][2i]';

end
end

#############################################################################
#Dense Blocks (D)

D_C = OffsetArray{OffsetArray{Array{Float64}}}(undef,1:2^depth+2);
for k = 1:2^depth+2
    D_C[k] = OffsetArray{Array{Float64}}(undef,k-2:k+2);
```

```
end

#Get sizes of F's to create empty U_C[depth^2+1] and V_C[depth^2+1] so the multiply will go through
k = depth

m_V1, n_V1 = size(F[k][2^depth-1][2^depth+1]);
m_U1, n_U1 = size(F[k][2^depth+1][2^depth-1]);
m_D̃1, n_D̃1 = size(D̃[2^depth][2^depth+1]);#size(D̃[k^2][k^2+1]);  these are wrong somehow -1?
m_D1, n_D1 = size(D[2^depth+1][2^depth]); #size(D[k^2+1][k^2]);
V_C[2^depth+1] = zeros(n_D̃1,n_V1); #U_C[k^2+1]
U_C[2^depth+1] = zeros(m_D1,m_U1); #U_C[k^2+1]

for i =1:2^depth-1 #k^2-1
    D_C[i][i] = D[i][i-1]*D̃[i-1][i] + D[i][i]*D̃[i][i] + D[i][i+1]D̃[i+1][i] + U_C[i]*F[k][i][i]*V_C[i
        ]';
    D_C[i][i+1] = D[i][i]*D̃[i][i+1] + D[i][i+1]*D̃[i+1][i+1] + U_C[i]*F[k][i][i+1]*V_C[i+1]';
    D_C[i+1][i] = D[i+1][i]*D̃[i][i] + D[i+1][i+1]*D̃[i+1][i] + U_C[i+1]*F[k][i+1][i]*V_C[i]';
    D_C[i][i+2] = D[i][i+1]*D̃[i+1][i+2] + U_C[i]*F[k][i][i+2]*V_C[i+2]';
    D_C[i+2][i] = D[i+2][i+1]*D̃[i+1][i] + U_C[i+2]*F[k][i+2][i]*V_C[i]';
end

i = 2^depth; #k^2;
D_C[i][i] = D[i][i-1]*D̃[i-1][i] + D[i][i]*D̃[i][i] + D[i][i+1]D̃[i+1][i] + U_C[i]*F[k][i][i]*V_C[i]';

D_C,U_C,R_C,B_C,W_C,V_C

end


function Copy_FMM(fmm :: FMM)
    #Copy all matrices from the fmm structure into arrays
    #Input:              fmm: FMM
    #Output:             D:: OffsetArray{OffsetArray{Array{Float64}}}
    #                    U,V:: OffsetArray{Array{Float64}}
    #                    R,W:: Array{OffsetArray{Array{Float64}}}
    #                    B:: Array{OffsetArray{OffsetArray{Array{Float64}}}}

    ##########################################################################
    #Copy U's V's
    U = OffsetArray{Array{Float64}}(undef,0:2^(fmm.depth)+2);
    V = OffsetArray{Array{Float64}}(undef,0:2^(fmm.depth)+2);

    i = 1;
    U,V = Copy_UV(fmm,U,V,i);

    #assign empty matrices to out of bounds U's, V's
    U[0] = zeros(0,0)
    U[2^(fmm.depth)+1] = zeros(0,0)
    V[0] = zeros(0,0)
    V[2^(fmm.depth)+1] = zeros(0,0)
    ##########################################################################
    #Copy D's
```

```
D = OffsetArray{OffsetArray{Array{Float64}}}(undef,0:2^fmm.depth+1);
for k = 0:2^fmm.depth+1
    D[k] = OffsetArray{Array{Float64}}(undef,k-1:k+1); #(0:k+1)
end
i = 1;
D = Copy_D(fmm,D,i,fmm.depth)

#assign empty matrices to out of bounds D's
D[1][0] = zeros(size(U[1],1),0);
D[2^fmm.depth][2^fmm.depth+1] = zeros(size(U[2^fmm.depth],1),0);
D[0][1] = zeros(0,size(V[1]',2));
D[2^fmm.depth+1][2^fmm.depth] = zeros(0,size(V[2^fmm.depth]',2));

#########################################################################################
#Copy R's, W's
R = Array{OffsetArray{Array{Float64}}}(undef,fmm.depth);
W = Array{OffsetArray{Array{Float64}}}(undef,fmm.depth);
for k = 1:fmm.depth
    R[k] = OffsetArray{Array{Float64}}(undef,-1:2^(k)+4);
    W[k] = OffsetArray{Array{Float64}}(undef,-1:2^(k)+4);
end
i = 1;
l = 1;
R,W = Copy_RW(fmm,R,W,l,i)

#assign empty R, W matrices to the right and left of the edge of each tree
#(matrices outside the indices 1:2^k are 0x0 empty matrices)
for k = 1:fmm.depth
    R[k][-1] = zeros(0,0);
    R[k][0] = zeros(0,0);
    R[k][2^k+1] = zeros(0,0);
    R[k][2^k+2] = zeros(0,0);
    R[k][2^k+3] = zeros(0,0);
    R[k][2^k+4] = zeros(0,0);
    W[k][-1] = zeros(0,0);
    W[k][0] = zeros(0,0);
    W[k][2^k+1] = zeros(0,0);
    W[k][2^k+2] = zeros(0,0);
    W[k][2^k+3] = zeros(0,0);
    W[k][2^k+4] = zeros(0,0);
end
#########################################################################################
#Copy B's

B = Array{OffsetArray{OffsetArray{Array{Float64}}}}(undef,fmm.depth);
for k = 1:fmm.depth
    B[k] = OffsetArray{OffsetArray{Array{Float64}}}(undef,-1:2^k+4);
    for i = -1:2^k+4
        if mod(i,2) == 1
            B[k][i] = OffsetArray{Array{Float64}}(undef,i-2:i+3); #(0:k+1)
        else
            B[k][i] = OffsetArray{Array{Float64}}(undef,i-3:i+2);
```

```
            end
        end
    end

    i = 1;
    k = 0;
    B = Copy_B(fmm,B,k,i)

    ##########################################################################################
    #assign empty B matrices to the right and left of the edge of each tree
    #(matrices outside the indices 1:2^k are ?x0 empty matrices)
    for k = 1:fmm.depth #1:fmm.depth-1
        k_1r, _ = size(R[k][1]); #size(R[k+1][1]);
        l_1r, _ = size(R[k][2]);
        k_1w, _ = size(W[k][1]); #~
        l_1w, _ = size(W[k][2]); #~

        k_3r, _ = size(R[k][2^(k)-1])
        l_3r, _ = size(R[k][2^(k)])
        k_3w, _ = size(W[k][2^(k)-1]) #~
        l_3w, _ = size(W[k][2^(k)]) #~


        B[k][-1][1] = zeros(0,k_1w);        B[k][1][-1] = zeros(k_1r,0);
        B[k][-1][2] = zeros(0,l_1w);        B[k][2][-1] = zeros(k_1r,0);
        B[k][0][2] = zeros(0,l_1w);         B[k][2][0] = zeros(l_1r,0);

        #print("k = ", k, "\n")
        B[k][2^(k)-1][2^(k)+1] = zeros(k_3r,0);  B[k][2^(k)+1][2^(k)-1] = zeros(0,k_3w);
        B[k][2^(k)-1][2^(k)+2] = zeros(k_3r,0);  B[k][2^(k)+2][2^(k)-1] = zeros(0,k_3w);
        B[k][2^(k)][2^(k)+2] = zeros(l_3r,0);    B[k][2^(k)+2][2^(k)] = zeros(0,l_3w);

        B[k][2^(k)+1][2^(k)+3] = zeros(0,0);     B[k][2^(k)+3][2^(k)+1] = zeros(0,0);
        B[k][2^(k)+1][2^(k)+4] = zeros(0,0);     B[k][2^(k)+4][2^(k)+1] = zeros(0,0);
        B[k][2^(k)+2][2^(k)+4] = zeros(0,0);     B[k][2^(k)+4][2^(k)+2] = zeros(0,0);
    end

    D,U,R,B,W,V

end

function FivePointFMMtoMatrix(D_C::OffsetArray{OffsetArray{Array{Float64}}},U_C::Array{Array{Float64
    }},R_C::Array{Array{Array{Float64}}},B_C::Array{OffsetArray{OffsetArray{Array{Float64}}}},W_C::
    Array{Array{Array{Float64}}},V_C::Array{Array{Float64}},fmm_idx::Array{Array{Int64}},depth::Int64)
    # Reconstruct the matrix C to test
    #Input:              D_C:: OffsetArray{OffsetArray{Array{Float64}}}
    #                    U_C,V_C:: Array{Array{Float64}}
    #                    R_C,W_C:: Array{Array{Array{Float64}}}
    #                    B_C:: Array{Array{Array{Float64}}}
    #                    fmm_idx:: Array{Array{Float64}} (stores index of each diagonal block at every
        level of the fmm structure)
    #                    depth:: Int64 (depth of the FMM structure)
```

```
#Output:              C̃:: Array{Float64}


BigU_C = Array{Array{Array{Float64}}}(undef,depth);
BigV_C = Array{Array{Array{Float64}}}(undef,depth);


for k = 1:depth
    BigU_C[k] = Array{Array{Float64}}(undef,2^k);
    BigV_C[k] = Array{Array{Float64}}(undef,2^k);
end

#Compute Big U's and V's
for i = 1:2^depth
    BigU_C[depth][i] = U_C[i];
    BigV_C[depth][i] = V_C[i];
end


for k = depth:-1:2
    for i = 1:2^(k-1)
        BigU_C[k-1][i] = [BigU_C[k][2i-1]*R_C[k][2i-1];
                          BigU_C[k][2i]*R_C[k][2i]];
        BigV_C[k-1][i] = [BigV_C[k][2i-1]*W_C[k][2i-1];
                          BigV_C[k][2i]*W_C[k][2i]];
    end
end

#Multiply all the Blocks and insert them into their proper positions in C̃
C̃ = zeros(N,N);

# Add the last out of bounds index so the UBV' for loop below is easier to read
for k = 1:depth+1
    fmm_idx[k] = [fmm_idx[k]; N+1];
end

#Insert the D blocks
idx = fmm_idx[depth+1];

for i = 1:2^depth

    if i == 1
        #first row has only 3 D blocks
        C̃[idx[1]:idx[2]-1,idx[1]:idx[2]-1] = D_C[1][1];
        C̃[idx[1]:idx[2]-1,idx[2]:idx[3]-1] = D_C[1][2];
        C̃[idx[1]:idx[2]-1,idx[3]:idx[4]-1] = D_C[1][3];

    elseif i == 2
        #second row has only 4 D blocks
        C̃[idx[2]:idx[3]-1,idx[1]:idx[2]-1] = D_C[2][1];
        C̃[idx[2]:idx[3]-1,idx[2]:idx[3]-1] = D_C[2][2];
        C̃[idx[2]:idx[3]-1,idx[3]:idx[4]-1] = D_C[2][3];
```

```
            C̃[idx[2]:idx[3]-1,idx[4]:idx[5]-1] = D_C[2][4];

        elseif i == 2^depth-1
            #second to last row has only 4 D blocks
            C̃[idx[i]:idx[i+1]-1,idx[i-2]:idx[i-1]-1] = D_C[i][i-2];
            C̃[idx[i]:idx[i+1]-1,idx[i-1]:idx[i]-1] = D_C[i][i-1];
            C̃[idx[i]:idx[i+1]-1,idx[i]:idx[i+1]-1] = D_C[i][i];
            C̃[idx[i]:idx[i+1]-1,idx[i+1]:idx[i+2]-1] = D_C[i][i+1];

        elseif i == 2^depth
            #last row has only 3 D blocks
            C̃[idx[i]:idx[i+1]-1,idx[i-2]:idx[i-1]-1] = D_C[i][i-2];
            C̃[idx[i]:idx[i+1]-1,idx[i-1]:idx[i]-1] = D_C[i][i-1];
            C̃[idx[i]:idx[i+1]-1,idx[i]:idx[i+1]-1] = D_C[i][i];

        else
            #all other rows have 5 D blocks
            C̃[idx[i]:idx[i+1]-1,idx[i-2]:idx[i-1]-1] = D_C[i][i-2];
            C̃[idx[i]:idx[i+1]-1,idx[i-1]:idx[i]-1] = D_C[i][i-1];
            C̃[idx[i]:idx[i+1]-1,idx[i]:idx[i+1]-1] = D_C[i][i];
            C̃[idx[i]:idx[i+1]-1,idx[i+1]:idx[i+2]-1] = D_C[i][i+1];
            C̃[idx[i]:idx[i+1]-1,idx[i+2]:idx[i+3]-1] = D_C[i][i+2];

        end

    end

    #Multiply and insert all UBV' blocks

    #for each level
    for k = 2:size(B_C,1)
        #step through each L shaped block
        for i = 1:2^(k-1)-1 #-1 because nothing to do at the last block

            if i == 2^(k-1)-1
                #only 2 UVB blocks at the last L shaped blocks for every level
                C̃[fmm_idx[k+1][2i-1]:fmm_idx[k+1][2i]-1,fmm_idx[k+1][2i+2]:fmm_idx[k+1][2i+3]-1] =
                    BigU_C[k][2i-1]*B_C[k][2i-1][2i+2]*BigV_C[k][2i+2]';

                C̃[fmm_idx[k+1][2i+2]:fmm_idx[k+1][2i+3]-1,fmm_idx[k+1][2i-1]:fmm_idx[k+1][2i]-1] =
                    BigU_C[k][2i+2]*B_C[k][2i+2][2i-1]*BigV_C[k][2i-1]';

            else
                #10 UVB blocks at every other level
                C̃[fmm_idx[k+1][2i-1]:fmm_idx[k+1][2i]-1,fmm_idx[k+1][2i+2]:fmm_idx[k+1][2i+3]-1] =
                    BigU_C[k][2i-1]*B_C[k][2i-1][2i+2]*BigV_C[k][2i+2]';
                C̃[fmm_idx[k+1][2i-1]:fmm_idx[k+1][2i]-1,fmm_idx[k+1][2i+3]:fmm_idx[k+1][2i+4]-1] =
                    BigU_C[k][2i-1]*B_C[k][2i-1][2i+3]*BigV_C[k][2i+3]';
                C̃[fmm_idx[k+1][2i-1]:fmm_idx[k+1][2i]-1,fmm_idx[k+1][2i+4]:fmm_idx[k+1][2i+5]-1] =
                    BigU_C[k][2i-1]*B_C[k][2i-1][2i+4]*BigV_C[k][2i+4]';
                C̃[fmm_idx[k+1][2i]:fmm_idx[k+1][2i+1]-1,fmm_idx[k+1][2i+3]:fmm_idx[k+1][2i+4]-1] =
                    BigU_C[k][2i]*B_C[k][2i][2i+3]*BigV_C[k][2i+3]'; #here
```

```
                C̃[fmm_idx[k+1][2i]:fmm_idx[k+1][2i+1]-1,fmm_idx[k+1][2i+4]:fmm_idx[k+1][2i+5]-1] =
                    BigU_C[k][2i]*B_C[k][2i][2i+4]*BigV_C[k][2i+4]';

                C̃[fmm_idx[k+1][2i+2]:fmm_idx[k+1][2i+3]-1,fmm_idx[k+1][2i-1]:fmm_idx[k+1][2i]-1] =
                    BigU_C[k][2i+2]*B_C[k][2i+2][2i-1]*BigV_C[k][2i-1]';
                C̃[fmm_idx[k+1][2i+3]:fmm_idx[k+1][2i+4]-1,fmm_idx[k+1][2i-1]:fmm_idx[k+1][2i]-1] =
                    BigU_C[k][2i+3]*B_C[k][2i+3][2i-1]*BigV_C[k][2i-1]';
                C̃[fmm_idx[k+1][2i+4]:fmm_idx[k+1][2i+5]-1,fmm_idx[k+1][2i-1]:fmm_idx[k+1][2i]-1] =
                    BigU_C[k][2i+4]*B_C[k][2i+4][2i-1]*BigV_C[k][2i-1]'; #here
                C̃[fmm_idx[k+1][2i+3]:fmm_idx[k+1][2i+4]-1,fmm_idx[k+1][2i]:fmm_idx[k+1][2i+1]-1] =
                    BigU_C[k][2i+3]*B_C[k][2i+3][2i]*BigV_C[k][2i]';
                C̃[fmm_idx[k+1][2i+4]:fmm_idx[k+1][2i+5]-1,fmm_idx[k+1][2i]:fmm_idx[k+1][2i+1]-1] =
                    BigU_C[k][2i+4]*B_C[k][2i+4][2i]*BigV_C[k][2i]';
            end

        end

    end

C̃
end
#############################################################################################

function Copy_UV(fmm :: FMM, U :: OffsetArray{Array{Float64}}, V :: OffsetArray{Array{Float64}}, i ::
    Int64)
    if isa(fmm, Leaf)
        U[i] = fmm.U;
        V[i] = fmm.V;

        U,V

    elseif isa(fmm,Node)
        U1,V1 = Copy_UV(fmm.fmmUL,U,V,2*i-1)
        U2,V2 = Copy_UV(fmm.fmmLR,U1,V1,2*i)

        U2,V2

    end
end

function Copy_RW(fmm :: FMM, R :: Array{OffsetArray{Array{Float64}}}, W :: Array{OffsetArray{Array{
    Float64}}}, l :: Int64, i :: Int64)
    if isa(fmm.fmmUL, Leaf) #assuming a complete tree
        R[l][2i-1] = fmm.Rl;
        R[l][2i] = fmm.Rr;

        W[l][2i-1] = fmm.Wl;
        W[l][2i] = fmm.Wr;

        R,W
    elseif isa(fmm.fmmUL, Node) #assuming a complete tree
        R[l][2i-1] = fmm.Rl;
```

```
        R[l][2i] = fmm.Rr;

        W[l][2i-1] = fmm.Wl;
        W[l][2i] = fmm.Wr;
        #println("2i-1 = ", 2i-1)

        R1,W1 = Copy_RW(fmm.fmmUL,R,W,l+1,2i-1);
        R2,W2 = Copy_RW(fmm.fmmLR,R,W,l+1,2i);

        R2,W2
    end
end


function Copy_D(fmm :: FMM, D :: OffsetArray{OffsetArray{Array{Float64}}}, i :: Int64, depth)
    if isa(fmm, Leaf)

        if i == 1
            D[i][i] = fmm.D_0;
            D[i][i+1] = fmm.D_r;
        elseif i == 2 .^depth
            D[i][i-1] = fmm.D_l;
            D[i][i] = fmm.D_0;
        else
            D[i][i-1] = fmm.D_l;
            D[i][i] = fmm.D_0;
            D[i][i+1] = fmm.D_r;
        end

        D

    elseif isa(fmm,Node)
        D1 = Copy_D(fmm.fmmUL,D,2*i-1,depth)
        D2 = Copy_D(fmm.fmmLR,D1,2*i,depth)

        D2
    end
end

function Copy_B(fmm :: FMM, B :: Array{OffsetArray{OffsetArray{Array{Float64}}}}, k :: Int64, i ::
    Int64) #Copy_B(fmm :: FMM, B :: Array{OffsetArray{Array{Array{Float64}}}}, k :: Int64, i :: Int64)

    if isa(fmm.fmmUL, Leaf) #assuming a complete tree
        # No B's store at the leaf.  Return
        B

    elseif isa(fmm.fmmUR, Node) #assuming a complete tree

        B1 = Copy_B_OffDiag(fmm.fmmUR,fmm.fmmLL,B,k+1,i)
        B2 = Copy_B(fmm.fmmUL,B1,k+1,2i-1);
        B3 = Copy_B(fmm.fmmLR,B2,k+1,2(i+1)-1);
```

```
        B3
    end
end

function Copy_B_OffDiag(fmmUR :: FMM, fmmLL :: FMM, B :: Array{OffsetArray{OffsetArray{Array{Float64
    }}}}, k :: Int64, i :: Int64)#Copy_B_OffDiag(fmmUR :: FMM, fmmLL :: FMM, B :: Array{OffsetArray{
    Array{Array{Float64}}}}, k :: Int64, i :: Int64)
    if isa(fmmUR, LeafOffDiag) #assuming a complete tree

        #No B's stored at the leaf.  Return.
        B

    elseif isa(fmmUR, Node) #assuming a complete tree
        #this will place all B's between cousins in the array

        i_new = 2i-1;

        B[k+1][i_new][i_new+2] = fmmUR.B13;
        B[k+1][i_new][i_new+3] = fmmUR.B14;
        B[k+1][i_new+1][i_new+3] = fmmUR.B24;

        B[k+1][i_new+2][i_new] = fmmLL.B31;
        B[k+1][i_new+3][i_new] = fmmLL.B41;
        B[k+1][i_new+3][i_new+1] = fmmLL.B42;

        B1 = Copy_B_OffDiag(fmmUR.fmmLL,fmmLL.fmmUR,B,k+1,i_new+1)

        B1
    end

end

function get_fmm_indices(fmm :: FMM,fmm_idx :: Array{Array{Int64}},fmm_m :: Array{Array{Int64}},k ::
    Int64,i :: Int64, depth :: Int64)
#stores indices given in the fmm structure into an array of arrays
#INPUT:                 fmm (FMM) fmm structure
#                       fmm_idx (Array{Array{Array{Int64}}}) Empty array of arrays
#                       k (Int64) level we are at in the tree/fmm structure
#                       i (Int64) current node at the kth level
#                       depth (Int64) total depth of the fmm tree
#OUTPUT                 fmm_idx (Array{Array{Int64}}) Array of arrays containing fmm indices
#                       fmm_idx (Array{Array{Int64}}) Array of arrays containing fmm partition
    dimensions

    if isa(fmm, Leaf)

        fmm_idx[depth-k+1][i] = fmm.col_idx;
        fmm_m[depth-k+1][i] = fmm.m;

        fmm_idx,fmm_m
    elseif isa(fmm, Node)
```

```
        fmm_idx[depth-k+2][2i-1] = fmm.fmmUL.col_idx;
        fmm_m[depth-k+2][2i-1] = fmm.fmmUL.m;

        fmm_idx[depth-k+2][2i] = fmm.fmmLR.col_idx;
        fmm_m[depth-k+2][2i] = fmm.fmmLR.m;

        fmm_idx,fmm_m = get_fmm_indices(fmm.fmmUL,fmm_idx,fmm_m,k-1,2i-1,depth)
        fmm_idx,fmm_m = get_fmm_indices(fmm.fmmLR,fmm_idx,fmm_m,k-1,2i,depth)

        fmm_idx,fmm_m
    end

end
```

A.2.4 FMM Script

This section contains code which gives examples of how to call the functions in appendix A.2.

```
push!(LOAD_PATH, pwd())
using FMM_Types
using Polynomials
using OffsetArrays
using LinearAlgebra
# include("/home/klessel/Dropbox/Julia_home/Tree_Functions.jl")
# include("/home/klessel/Dropbox/Julia_home/svd_ranktol.jl")
# include("/home/klessel/Dropbox/Julia_home/FMM_Construction.jl")
# include("/home/klessel/Dropbox/Julia_home/FMM_Multiply.jl")
# include("/home/klessel/Dropbox/Julia_home/FMM_FMM_Multiply_Final.jl")
# include("/home/klessel/Dropbox/Julia_home/FMM_Functions.jl")

include("C:\\Users\\klessel\\Dropbox\\Julia_home\\Tree_Functions.jl")
include("C:\\Users\\klessel\\Dropbox\\Julia_home\\svd_ranktol.jl")
include("C:\\Users\\klessel\\Dropbox\\Julia_home\\FMM_Construction.jl")
include("C:\\Users\\klessel\\Dropbox\\Julia_home\\FMM_Multiply.jl")
include("C:\\Users\\klessel\\Dropbox\\Julia_home\\FMM_FMM_Multiply_Final.jl")
include("C:\\Users\\klessel\\Dropbox\\Julia_home\\FMM_Functions.jl")

N = Int64;
r = Int64;
d = Int64;
minPart = Int64;
r=3; #hankel block rank
N = 32768;
minPart = 3*r;

d = 1;
tree, dummy = Gen_Tree_Complete(minPart,N,d);

#tree = Gen_Tree_Mtree(minPart,N);
#tree, dummy = label_tree(tree,1);
#fmm  = FMM_Construction(tree,r);
```

```
# #Complete Dummy Test Tree - 3 level

# #complete test tree
# N = 160;
# leaf1 = Leaf_Spine(4,-1,-1);
# leaf2 = Leaf_Spine(8,-1,-1);
# node1 = Node_Spine(leaf1,leaf2,12,-1,-1);

# leaf3 = Leaf_Spine(7,-1,-1);
# leaf4 = Leaf_Spine(21,-1,-1);
# node2 = Node_Spine(leaf3,leaf4,28,-1,-1);

# leaf5 = Leaf_Spine(14,-1,-1);
# leaf6 = Leaf_Spine(16,-1,-1);
# node3 = Node_Spine(leaf5,leaf6,30,-1,-1);

# leaf7 = Leaf_Spine(25,-1,-1);
# leaf8 = Leaf_Spine(65,-1,-1);
# node4 = Node_Spine(leaf7,leaf8,90,-1,-1);

# node5 = Node_Spine(node1,node2,40,-1,-1);
# node6 = Node_Spine(node3,node4,120,-1,-1);

# tree = Node_Spine(node5,node6,160,-1,-1);


# tree, dummy = label_tree(tree,1);
# #fmm::Fmm = FMM_Construction(tree,r);
# fmm = FMM_Construction(tree,r);


################################################################################
#Complete Dummy Test Tree - 4 level

# N = 340

# leaf1 = Leaf_Spine(4,-1,-1);
# leaf2 = Leaf_Spine(8,-1,-1);
# node1 = Node_Spine(leaf1,leaf2,12,-1,-1);

# leaf3 = Leaf_Spine(7,-1,-1);
# leaf4 = Leaf_Spine(21,-1,-1);
# node2 = Node_Spine(leaf3,leaf4,28,-1,-1);

# leaf5 = Leaf_Spine(14,-1,-1);
# leaf6 = Leaf_Spine(16,-1,-1);
# node3 = Node_Spine(leaf5,leaf6,30,-1,-1);

# leaf7 = Leaf_Spine(29,-1,-1);
# leaf8 = Leaf_Spine(61,-1,-1);

# node4 = Node_Spine(leaf7,leaf8,90,-1,-1);
```

```
# node5 = Node_Spine(node1,node2,40,-1,-1);
# node6 = Node_Spine(node3,node4,120,-1,-1);

# leaf8 = Leaf_Spine(10,-1,-1);
# leaf9 = Leaf_Spine(15,-1,-1);
# node7 = Node_Spine(leaf8,leaf9,25,-1,-1);

# leaf10 = Leaf_Spine(23,-1,-1);
# leaf11 = Leaf_Spine(32,-1,-1);
# node8 = Node_Spine(leaf10,leaf11,55,-1,-1);

# leaf12 = Leaf_Spine(9,-1,-1);
# leaf13 = Leaf_Spine(26,-1,-1);
# node9 = Node_Spine(leaf12,leaf13,35,-1,-1);

# leaf14 = Leaf_Spine(24,-1,-1);
# leaf15 = Leaf_Spine(41,-1,-1);
# node10 = Node_Spine(leaf14,leaf15,65,-1,-1);

# node11 = Node_Spine(node7,node8,80,-1,-1);
# node12 = Node_Spine(node9,node10,100,-1,-1);

# node13 = Node_Spine(node5,node6,160,-1,-1);
# node14 = Node_Spine(node11,node12,180,-1,-1);

# tree = Node_Spine(node13,node14,340,-1,-1);

#############################################################################################

# FMM Construction
tree, dummy = label_tree(tree,1);
fmmA  = @time FMM_Construction2(tree,r,f); #this one takes in the function f() as a variable
fmmB  = @time FMM_Construction2(tree,r,g);

# # #FMM Vector Multiply
# x = rand(N,1);
# x = x/norm(x,2);
# b =  FMM_Multiply(fmmB,x)
#
# B1 = g(1:N,1:N,N);
# b2 = B1*x;
#
# norm(b-b2,2)/norm(b2,2)

#############################################################################################

# FMM x FMM Multiply
D,U,R,B,W,V =  Copy_FMM(fmmA)
D̃,Ũ,R̃,B̃,W̃,Ṽ =  Copy_FMM(fmmB)

depth = fmmA.depth;
```

```
print("FMM x FMM Multiply time: \n")
D_C, U_C, R_C, B_C, W_C, V_C =  @time FMM_FMM_Multiply(D,U,R,B,W,V,D̃,Ũ,R̃,B̃,W̃,Ṽ,depth);

#Store indices of each diagonal block and partition dimensions in arrays
fmm_idx = Array{Array{Int64}}(undef,depth+1);
fmm_m = Array{Array{Int64}}(undef,depth+1);
i = 1;
for k = 0:depth
    fmm_idx[k+1] = Array{Int64}(undef,2^k);
    fmm_m[k+1] = Array{Int64}(undef,2^k);
end
fmm_idx[1][1] = fmmA.col_idx;
fmm_m[1][1] = fmmA.m;

fmm_idx, fmm_m = get_fmm_indices(fmmA,fmm_idx,fmm_m,depth,i,depth) #didn't end up needing fmm_m

C̃ = FivePointFMMtoMatrix(D_C, U_C, R_C, B_C, W_C, V_C,fmm_idx,depth)

###############################################################################################

#Generate the Original matrix to compare to
# A1 = f(1:N,1:N,N);
# A2 = g(1:N,1:N,N);

# C = A1*A2; # 8.115646911570535e-6

#use the FMM multiply routine to generate A, so that we remove the error that was contributed
#by the construction to isolate the error that is induced by the matrix multiply.
A1 = zeros(N,0);
for i = 1:N
    e = zeros(N,1);
    e[i] = 1;
    a =  FMM_Multiply(fmmA,e)
    global A1 = [A1 a];
end

A2 = zeros(N,0);
for i = 1:N
    e = zeros(N,1);
    e[i] = 1;
    a =  FMM_Multiply(fmmB,e)
    global A2 = [A2 a];
end

print("Standard Multiply time: \n")
C = @time A1*A2
print("Relative FMM xFMM Multiply Error: \n")
norm(C̃-C,Inf)/norm(C,Inf)
```

A.2.5 FMM Functions

```
function Gen_Tree_Complete(minPart::Int64,N::Int64,d::Int64)
#function(minPart,N)
#This code is for testing purposes only.  Generates a tree
#structure that is not associated with any function.  Function takes in the
#size of the desired matrix and generates a tree by splitting this
#repeatedly in half on the left and right for a 'complete' tree.
#INPUT:              minPart     (Int64) minimum number of partitions
#                                type of tree that will be generated
#                    N           (Uint) number of points in desired
#                                function
#OUTPUT:             tree        (Union(Node_Spine,Leaf_Spine) contains split dimensions and depth at
#                                each level

    if N/2 < minPart # leaf
        depth = 0;
        leafL = Leaf_Spine(N,d,depth);

        d = d + N;
        leafL, depth, d
    else # node
        treeL, depth_l, d_l = Gen_Tree_Complete(minPart,convert(Int64,N/2),d);
        treeR, depth_r, d_r = Gen_Tree_Complete(minPart,convert(Int64,N/2),d_l);

        depth = 1 + max(depth_l,depth_r);
        tree = Node_Spine(treeL,treeR,N,d,depth);
        tree, depth, d_r
    end
end

function Gen_Tree_Mtree(minPart::Int64,N::Int64)
#function [tree] = Gen_TestTree(tree,N)
#This code is for testing purposes only. Artificually generates a tree
#structure that is not associated with any function.  Function takes in the
#number of desired nodes and generates a Worst case memory tree
# must have the line 'using Polynomials' in the main routine
#
#INPUT:
#                    N           (Int64) number of points in desired
#                                function
#                    minPart     (Int64) minimum number of partitions
#OUTPUT:             tree        (Tree) contains split dimensions at
#                                each level, as well as whether or not the
#                                node is a leaf


    # number of nodes
    n = 2*floor(N/minPart) -1; # n = N_N = N_L -1, and N_L = N/minPart.

    #depth of tree
    nodeRoots = roots(Poly([1-n,1,1]));
    d_exact  = nodeRoots[nodeRoots.>0]; #choose the positive root.
```

```
    #find maximum depth, d_max, of the tree we will generate
    d_max = convert(Int64,ceil(d_exact[1]));

    #Call the function initially on the root node of the tree.
    #   .<-
    #  / \
    # /\ /\
    #/\ /\/\
    tree = Gen_MTree_Right(N,minPart,d_max);
    tree
end

function Gen_MTree_Right(m,minPart,depth)
#Descend into right branch of the root of the current subtree (pictured below)
#    .
#  / \<-
# /\ /\
#/\ /\/\
#INPUT:
#                    m           (Int64) partition dimension of current
#                                node
#                    minPart     (Int64) minimum number of partitions
#                    depth       (Int64) depth of current node
#                    pB          (Bool) a flag that denotes whether the
#                                current node is on the prime branch
#OUTPUT:             tree        (Tree) contains split dimensions at
#                                each level, as well as whether or not the
#                                node is a leaf
    if !(m < 2*minPart)
    #Case 1: If we do have enough rows to split into 2 blocks of minimum
    #partition size split and recurse on both children.  If we do not, then do
    #not partition further; Return two leaf nodes.

        #max number of blocks of minimum partition size we can have on this left subtree
        numBlocks = convert(Int64,floor(m/minPart));
        #remaining depth of the left subtree.
        r_depth = numBlocks-1;

        #if we cannot generate a full left subtree - only generate children
        #corresponding to the the number of partitions we have left.
        if m <(depth +1)*minPart
            mL = r_depth*minPart;
        else
            mL = depth*minPart;
        end

        #Left Subtree Call
        treeL = Gen_MTree_Left(mL,minPart,depth);

        #Right Subtree Call
        #do we have enough rows to partition into two blocks of minumum partition size?
        if  m >= 2*minPart && m < 3*minPart
```

```
            #Case A: do we only have enough to split into two blocks and no
            #more?(blocks must be of at least minimum partition size and no
            #more than 2 times the minimum partition size)
            #Ex: minPart = 60
            #       /\170
            #   110  60
            #

            mL = m- minPart; #extra rows tacked onto left child
            leafL = Leaf_Spine(mL,-1,-1);

            mR = minPart;
            leafR = Leaf_Spine(mR,-1,-1);

            tree  = Node_Spine(leafL,leafR,m,-1,-1);
            tree
        elseif m<(depth+1)*minPart
            #Case B: do we have enough to split into more than two blocks, but cannot
            #generate a full left going subtree?
            #Ex:(N = 4096) Depth of full tree = 12. MinPart = 60.
            #(Though here I am only showing partition  dimensions for 3 levels.
            #Dots indicate a part of the tree not shown. )
            #   . . /\
            #  .     /\204
            #  . 144/\ 60
            #  .  60  84
            # /\

            mR = m - (numBlocks-1)*minPart;

            treeR = Leaf_Spine(mR,-1,-1);
            tree = Node_Spine(treeL,treeR,m,-1,-1);
            tree
        else #(depth+1)*minPart < m
            #Case C: We have enough rows to generate a full leftgoing subtree of depth d_max

            mR = m- depth*minPart;
            treeR = Gen_MTree_Right(mR,minPart,depth);

            tree = Node_Spine(treeL,treeR,m,-1,-1);
            tree
        end

    else #Case 2: we did not have enough rows to partition - label node as a leaf and return
        leaf = Leaf_Spine(m,-1,-1);
        leaf

    end

end

function Gen_MTree_Left(m,minPart,depth)
```

```
#Generate left branch of the root of the current subtree (pictured below)
#      .
#     / \
# ->/\ /\
#  /\ /\/\
#INPUT:
#                    m           (Int64) partition dimension of current
#                                node
#                    minPart     (Int64) minimum number of partitions
#                    depth       (Int64) depth of current node
#                    pB          (Bool) a flag that denotes whether the
#                                current node is on the prime branch
#OUTPUT:             tree        (Tree) contains split dimensions at
#                                each level, as well as whether or not the
#                                node is a leaf

    if m < 2*minPart  # if Leaf

        #pB = false;
        leafL = Leaf_Spine(m,-1,-1);

        leafL
    else #node
        depth -= 1;
        mL = m - minPart;
        treeL = Gen_MTree_Left(mL,minPart,depth);

        treeR = Leaf_Spine(minPart,-1,-1);

        tree = Node_Spine(treeL,treeR,m,-1,-1);
        tree
    end

end

function label_tree(tree::Tree,col_idx)
#This function labels each node with the starting index (row/col value)
# of its corresponding diagonal block. Can also label the depth if the
# commented lines are uncommented
#
#INPUT:              tree    (Tree) contains partition dimensions for
#                            each subdivision of the input matrix, for each
#                            of which there are a left and right child
#                            structure
#                    col_idx       (Int64) row and col index which corresponds to
#                            the first element in the current diagonal block
#                            at every node.
#OUTPUT:             tree    (Tree) same as input,  that contains valid
#                            diagonal (row/col) index at each node
#                            (and depth if commented lines are uncommented.
#
# Author: Kristen Lessel - Sept 2014
```

```
    if !isa(tree,Leaf_Spine) # if not a leaf node

        tree.col_idx = col_idx;
        tree.treeL,col_idx = label_tree(tree.treeL,col_idx);
        depth_l = tree.treeL.depth;

        tree.treeR,col_idx = label_tree(tree.treeR,col_idx);
        depth_r = tree.treeR.depth;

        tree.depth = 1 + max(depth_l,depth_r);
        tree, col_idx
    else
        tree.depth = 0;
        tree.col_idx = col_idx;
        col_idx = col_idx + tree.m;
        tree, col_idx
    end
end

function create_empty_tree(depth)
#creates a complete tree with partition dimensions of 0 at each node
#INPUT:                 depth (Int64) denoting depth of tree)
#OUTPUT                 empty_tree (Tree) empty partition tree with given depth
    if depth == 0
    empty_leaf = Leaf_Spine(0,1,depth)

    empty_leaf
    else
        depth = depth-1;
        empty_treeL = create_empty_tree(depth);
        empty_treeR = create_empty_tree(depth);

        empty_tree = Node_Spine(empty_treeL,empty_treeR,0,1,depth+1);
        empty_tree
    end

end

function create_empty_fmm(depth)
    #creates a complete fmm representation  with partition dimensions of 0 at each node
    #The fmmUR and fmmUL nodes are flipped for use with the FMM_Multiply()
    #Creates an empty FMM Representation of this form:
    #    ---------------
    #   \        \   \ x \
    #   \        \-------\
    #   \        \   \   \
    #   \---------------\
    #   \   \   \        \
    #   \-------\        \
    #   \ x \   \        \
    #    ---------------
```

```
    #
    #The nodes on the diagonal could effectively be empty because they aren't used,
    #but this code assigns empty nodes/leaves/arrays to thesse as well.
    #
    #
    #INPUT:                 depth (Int64) denoting depth of tree)
    #OUTPUT                 empty_tree (Tree) empty partition tree with given depth

    if depth == 0
        empty_leaf = Leaf(-1,-1,depth,Array(Float64,0,0),Array(Float64,0,0),Array(Float64,0,0),Array(
            Float64,0,0),Array(Float64,0,0));
        empty_leaf

    else
        depth = depth-1;
        empty_treeUR,empty_treeLL = create_empty_fmm_offdiag(depth);
        empty_treeL = create_empty_fmm(depth);
        empty_treeR = create_empty_fmm(depth);

        empty_tree = Node(empty_treeL,empty_treeR,empty_treeLL,empty_treeUR,-1,-1,-1,-1,depth+1,
                        Array(Float64,0,0),Array(Float64,0,0),Array(Float64,0,0),Array(Float64,0,0),
                        Array(Float64,0,0),Array(Float64,0,0),Array(Float64,0,0),Array(Float64,0,0),
                        Array(Float64,0,0),Array(Float64,0,0));

        empty_tree
    end

end

function create_empty_fmm_offdiag(depth)
    if depth == 0
        empty_leaf_offdiag = LeafOffDiag(-1,-1,-1,-1,0,Array(Float64,0,0),Array(Float64,0,0),
                                        Array(Float64,0,0),Array(Float64,0,0),Array(Float64,0,0),
                                            Array(Float64,0,0));

        empty_leaf_offdiag, empty_leaf_offdiag

    else

        depth = depth-1;
        empty_treeUR, empty_treeLL = create_empty_fmm_offdiag(depth);

        empty_leaf = LeafOffDiag(-1,-1,-1,-1,-1,Array(Float64,0,0),Array(Float64,0,0),
                                    Array(Float64,0,0),Array(Float64,0,0),Array(Float64,0,0),Array(
                                        Float64,0,0));
        empty_treeUR2 = Node(empty_leaf, empty_leaf, empty_leaf,empty_treeUR,-1,-1,-1,
                    -1,depth+1,Array(Float64,0,0),Array(Float64,0,0),Array(Float64,0,0),Array(Float64
                        ,0,0),
                    Array(Float64,0,0),Array(Float64,0,0),Array(Float64,0,0),Array(Float64,0,0),
                    Array(Float64,0,0),Array(Float64,0,0));
        empty_treeLL2 = Node(empty_leaf, empty_leaf,empty_treeLL,empty_leaf,-1,-1,-1,
                    -1,depth+1,Array(Float64,0,0),Array(Float64,0,0),Array(Float64,0,0),Array(Float64
```

```
                        ,0,0),
                    Array(Float64,0,0),Array(Float64,0,0),Array(Float64,0,0),Array(Float64,0,0),
                    Array(Float64,0,0),Array(Float64,0,0));

        empty_treeUR2, empty_treeLL2
    end

end

function create_empty_gtree(depth)
#creates a complete tree with partition dimensions of 0 at each node
#INPUT:                depth (Int64) denoting depth of tree)
#OUTPUT                empty_tree (Tree) empty partition tree with given depth
    if depth == 0
        g = Array(Float64,0,1); #i have this as 3, but do i need to give an input fmm structure in
            general to match the size of the g's? Does the size of the empty matrix matter, is it used
             at all?
        empty_gleaf = Leaf_Gtree(0,1,depth,g)

        empty_gleaf
    else
        depth = depth-1;
        empty_gtreeL = create_empty_gtree(depth);
        empty_gtreeR = create_empty_gtree(depth);

        g = Array(Float64,0,1);
        empty_gtree = Node_Gtree(empty_gtreeL,empty_gtreeR,0,1,depth+1,g);
        empty_gtree
    end
end

function svd_ranktol(A::Array{Float64,2},r::Int64)
#INPUT:             A       Array{Float64}(2)
#                   r       (Uint64) largest allowable rank of hankel blocks -
#                           this determines the amount of compression, and
#                           should be compatible with your input tree.
#                           (if the maximum allowable rank is p, then the
#                           tree partitions should be no smaller than 3p)
#                           Corresponding singular vectors below this rank
#                           will be dropped.
#OUTPUT:            Uhat    (Array{Float64}(2)) left singular vectors of A
#                           corresponding to singular values higher than
#                           the chosen tolerance
#                   Vhat    (Array{Float64}(2)) right singular vectors of A
#                           corresponding to singular values higher than
#                           the chosen tolerance
#                   Shat    (Array{Float64}(1)) array containing the singular
#                           values of A higher than the chosen tolerance
    U, S, V = svd(A);
    if isempty(A)
        ## julia handles svd's of empty matrices incorrectly at this time.
        # These if statements correct that
```

```
        m, n = size(A);
        if size(A,2) == 0
            Uhat = Matrix{Float64}(I,m,n); #eye(m,n);
            Shat = Array{Float64,1}(undef,0); #for some reason the function diagm() (used in the fmm
                construction code) will only accept this input to produce a 0x0 matrix, and errors
                otherwise
            Vhat = Array{Float64,2}(undef,0,0);
        elseif size(A,1) == 0
            Uhat = Array{Float64,2}(undef,0,0);
            Shat = Array{Float64,1}(undef,0);
            Vhat = Matrix{Float64}(I,n,m);#eye(n,m);
        end
    else
        Uhat = U[:,1:r];
        Shat = S[1:r];
        Vhat = V[:,1:r];
    end
    Uhat, Shat, Vhat
end

function f(rowIdx,colIdx,N)
#function f(rowIdx::Array{Int64,1},colIdx::Array{Int64,1},N::Int64)
#generates blocks of a matrix defined by input parameters. This function
#is not efficient in matlab
#INPUT          sr: (Int64) start row
#               sc: (Int64) start column
#               ml: (Int64) number of rows
#               nl: (Int64) number of columns
#               sInt: (Int64) start of interval
#               eInt: (Int64) end of interval
#               N: (Int64) grid size
#
#OUTPUT         H: (Array{Float64,2}) matrix defined by given inputs
    sInt = 0;
    eInt = 1;

    # number of rows
    ml = length(rowIdx);

    # number of columns
    nl = length(colIdx);

    H = Array{Float64,2}(undef,ml,nl);

    if ml != 0 || nl != 0
        x = range(sInt,stop=eInt,length=N);
        for ii = 1:ml
            for jj = 1:nl
                i_idx = rowIdx[ii];
                j_idx = colIdx[jj];
                H[ii,jj] = sqrt(abs(x[i_idx]-x[j_idx]));
                #H[ii,jj] = log(1+abs(x(ii+sr)-x(jj+sc)));
```

```
            end
        end
    end
    H
end

## FMM_Types.jl
# 1-16-18

module FMM_Types

export Leaf, LeafOffDiag, Node, FMM, Leaf_Spine, Node_Spine, Tree, Leaf_Gtree, Node_Gtree, Gtree

mutable struct Leaf{T1 <: Integer, T2 <: Number}
    m::T1 #dimension of dense diagonal (square) matrix at current node
    col_idx::T1 #column index number of the matrix at the current node
    depth::T1
    U::Array{T2,2}
    V::Array{T2,2}
    D_l::Array{T2,2}
    D_0::Array{T2,2}
    D_r::Array{T2,2}
end

mutable struct LeafOffDiag{T1 <: Integer, T2 <: Number}
    m::T1
    n::T1
    col_idx::T1
    row_idx::T1
    depth::T1
    B13::Array{T2,2}
    B14::Array{T2,2}
    B24::Array{T2,2}
    B31::Array{T2,2}
    B41::Array{T2,2}
    B42::Array{T2,2}
end

mutable struct Node{T1 <: Integer, T2 <: Number}
    fmmUL::Union{Node, Leaf, LeafOffDiag}
    fmmLR::Union{Node, Leaf, LeafOffDiag}
    fmmUR::Union{Node, Leaf, LeafOffDiag}
    fmmLL::Union{Node, Leaf, LeafOffDiag}
    m::T1
    n::T1
    col_idx::T1
    row_idx::T1
    depth::T1
    B13::Array{T2,2}
    B14::Array{T2,2}
    B24::Array{T2,2}
    B31::Array{T2,2}
```

```
    B41::Array{T2,2}
    B42::Array{T2,2}
    Rl::Array{T2,2}
    Rr::Array{T2,2}
    Wl::Array{T2,2}
    Wr::Array{T2,2}
end
#Define FMM structure
FMM = Union{Leaf,Node,LeafOffDiag}

# #Define partition(spine) tree
mutable struct Leaf_Spine{T <: Integer}
    m::T
    col_idx::T
    depth::T
end

mutable struct Node_Spine{T <: Integer}
    treeL::Union{Node_Spine{T},Leaf_Spine{T}}
    treeR::Union{Node_Spine{T},Leaf_Spine{T}}
    m::T
    col_idx::T
    depth::T
end

Tree = Union{Leaf_Spine,Node_Spine}

mutable struct Leaf_Gtree{T1 <: Integer, T2 <: Number}
    m::T1
    col_idx::T1
    depth::T1
    g::Array{T2,2}
end

mutable struct Node_Gtree{T1 <: Integer, T2 <: Number}
    gtreeL::Union{Node_Gtree,Leaf_Gtree}
    gtreeR::Union{Node_Gtree,Leaf_Gtree}
    m::T1
    col_idx::T1
    depth::T1
    g::Array{T2,2}
end

Gtree = Union{Leaf_Gtree, Node_Gtree}

end
```

Bibliography

[1] J. Xia, *Efficient structured multifrontal factorization for general large sparse matrices*, *SIAM Journal on Scientific Computing* **35** (2013), no. 2 A832–A860.

[2] J. Xia, Y. Xi, and M. Gu, *A superfast structured solver for toeplitz linear systems via randomized sampling*, *SIAM Journal on Matrix Analysis and Applications* **33** (2012), no. 3 837–858.

[3] L. Greengard and V. Rokhlin, *A fast algorithm for particle simulations*, *Journal of computational physics* **73** (1987), no. 2 325–348.

[4] S. Chandrasekaran, M. Gu, and T. Pals, *A fast ulv decomposition solver for hierarchically semiseparable representations*, *SIAM Journal on Matrix Analysis and Applications* **28** (2006), no. 3 603–622.

[5] J. Xia, S. Chandrasekaran, M. Gu, and X. S. Li, *Fast algorithms for hierarchically semiseparable matrices*, *Numerical Linear Algebra with Applications* **17** (2010), no. 6 953–976.

[6] P.-G. Martinsson, *A fast randomized algorithm for computing a hierarchically semiseparable representation of a matrix*, *SIAM Journal on Matrix Analysis and Applications* **32** (2011), no. 4 1251–1274.

[7] F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov, *A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization*, *arXiv preprint arXiv:1503.05464* (2015).

[8] J. Xia, *Randomized sparse direct solvers*, *SIAM Journal on Matrix Analysis and Applications* **34** (2013), no. 1 197–227.

[9] F.-H. Rouet, X. S. Li, P. Ghysels, and A. Napov, *A distributed-memory package for dense hierarchically semi-separable matrix computations using randomization*, *ACM Transactions on Mathematical Software (TOMS)* **42** (2016), no. 4 1–35.

[10] G. Agnarsson and R. Greenlaw, *Graph theory: Modeling, applications, and algorithms*. Prentice-Hall, Inc., 2006.

[11] H. A. Van der Vorst and C. Vuik, *The superlinear convergence behaviour of gmres*, *Journal of computational and applied mathematics* **48** (1993), no. 3 327–341.

[12] Y. Saad and M. H. Schultz, *Gmres: A generalized minimal residual algorithm for solving nonsymmetric linear systems*, *SIAM Journal on scientific and statistical computing* **7** (1986), no. 3 856–869.

[13] P. Coulier, H. Pouransari, and E. Darve, *The inverse fast multipole method: using a fast approximate direct solver as a preconditioner for dense linear systems*, *SIAM Journal on Scientific Computing* **39** (2017), no. 3 A761–A796.

[14] B. Carpentieri, I. S. Duff, L. Giraud, and G. Sylvand, *Combining fast multipole techniques and an approximate inverse preconditioner for large electromagnetism calculations*, *SIAM Journal on Scientific Computing* **27** (2005), no. 3 774–792.

[15] K. Lessel, M. Hartman, and S. Chandrasekaran, *A fast memory efficient construction algorithm for hierarchically semi-separable representations*, *SIAM Journal on Matrix Analysis and Applications* **37** (2016), no. 1 338–353.

[16] W. Hackbusch and S. Börm, *Data-sparse approximation by adaptive H2-matrices*, *Computing* **69** (2002), no. 1 1–35.

[17] M. Gu and S. C. Eisenstat, *Efficient algorithms for computing a strong rank-revealing qr factorization*, *SIAM Journal on Scientific Computing* **17** (1996), no. 4 848–869.

[18] Y. P. Hong and C.-T. Pan, *Rank-revealing qr factorizations and the singular value decomposition*, *Mathematics of Computation* **58** (1992), no. 197 213–232.

[19] S. Chandrasekaran and I. C. Ipsen, *On rank-revealing factorisations*, *SIAM Journal on Matrix Analysis and Applications* **15** (1994), no. 2 592–622.

[20] E. Liberty, F. Woolfe, P.-G. Martinsson, V. Rokhlin, and M. Tygert, *Randomized algorithms for the low-rank approximation of matrices*, *Proceedings of the National Academy of Sciences* **104** (2007), no. 51 20167–20172.

[21] L. Miranian and M. Gu, *Strong rank revealing lu factorizations*, *Linear algebra and its applications* **367** (2003) 1–16.

[22] C.-T. Pan, *On the existence and computation of rank-revealing lu factorizations*, *Linear Algebra and its Applications* **316** (2000), no. 1 199–222.

[23] S. Chandrasekaran and K. Lessel, "Scientific computing website." `http://scg.ece.ucsb.edu/software.html`.

[24] M. Hochbruck and A. Ostermann, *Exponential integrators*, *Acta Numerica* **19** (2010) 209–286.

[25] G. H. Golub and C. F. Van Loan, *Matrix computations*, vol. 3. JHU press, 2012.

[26] J. Bezanson, V.B. Shah, S. Karpinski, A.Edelman, "Julia benchmarks." `https://julialang.org/benchmarks/`.

[27] S. Chandrasekaran, E. Epperly, and N. Govindarajan, *Graph-induced rank structures and their representations*, *arXiv preprint arXiv:1911.05858* (2019).