

## ABSTRACT

Title of thesis:       **THE FAST MULTIPOLE METHOD  
FOR 2D COULOMBIC PROBLEMS:  
ANALYSIS, IMPLEMENTATION  
AND VISUALIZATION**

Yang Wang, Master of Science, 2005

Thesis directed by:  **Professor Ramani Duraiswami  
Institute for Advanced Computer Studies**

The author reviews the Fast Multipole Method (FMM) for solving the Coulombic potential problem. An implementation of the FMM algorithm based on the Java programming language is presented, and the process of solving the Coulombic problem using the FMM is illustrated in a Java applet. It is hoped that the applet can be of pedagogical value for students new to the FMM algorithm, that the Java code can be easily extended to solve matrix-vector multiplications of other types, and that the code developed in this project can serve as a basis for a full set of Application Programming Interfaces (API) to promote and aide the integration of the FMM algorithm into larger Java-based scientific computing packages. A live demonstration of the applet, the source code of the current developer's build of the Java-FMM package, and other related material including API documentation can be found online at <http://www.umiacs.umd.edu/~wpwy/fmm>.

THE FAST MULTIPOLE METHOD FOR  
2D COULOMBIC PROBLEMS:  
ANALYSIS, IMPLEMENTATION AND VISUALIZATION

by

Yang Wang

Thesis submitted to the Faculty of the Graduate School of the  
University of Maryland, College Park in partial fulfillment  
of the requirements for the degree of  
Master of Science  
2005

Advisory Committee:

Professor Ramani Duraiswami, Chair/Advisor  
Professor Nail Gumerov  
Professor C. David Levermore

© Copyright by  
Yang Wang  
2005

# DEDICATION

To my family.

## ACKNOWLEDGMENTS

First and foremost I would like to thank my advisor, Dr. Ramani Duraiswami, for introducing me to the Fast Multipole Method, a fascinating algorithm to study and to implement. I am constantly amazed by how fast Dr. Duraiswami replies my emails whenever I have a question, how welcoming he is everytime I show up at his office with even more questions, and how tolerant he is towards my ignorance and never-ending questionsxx.

I also owe immense gratitude to Dr. and Mrs. Clopper Almon, whose generosity made my education in the United States financially possible and who have become parent-like figures and role models to me over the years. Although a thesis at the Ph.D. level would have pleased them more, I am eager to use the first available opportunity to put in words my appreciation for their guidance.

I would also like to thank my employer, the Institute for Advanced Computer Studies, for its support and understanding without which I couldn't have completed this thesis while employed fulltime.

Last but not the least, I owe my deepest thanks to my family. Words cannot express my gratitude, and therefore to them I dedicate this thesis.

## TABLE OF CONTENTS

List of Tables	vii
List of Figures	vii
1 Introduction	1
1.1 Overview . . . . .	1
1.2 The 2D Coulomb system . . . . .	2
1.3 Outline of the Thesis . . . . .	4
2 The Fast Multipole Method	6
2.1 Overview . . . . .	6
2.2 Expansions . . . . .	8
2.2.1 Multipole Expansion ( <b>S</b> -Expansion) . . . . .	8
2.2.2 Local Expansion ( <b>R</b> -Expansion) . . . . .	9
2.3 Translation . . . . .	9
2.3.1 Multipole-to-Multipole ( <b>S S</b> ) Translation . . . . .	10
2.3.2 Multipole-to-Local ( <b>S R</b> ) Translation . . . . .	11
2.3.3 Local-to-Local ( <b>R R</b> ) Translation . . . . .	12
2.3.4 Translation Error . . . . .	13
2.4 The Multilevel FMM (MLFMM) Algorithm . . . . .	14

2.4.1	Space Partition . . . . .	14
2.4.2	Upward Pass . . . . .	18
2.4.3	Downward Pass . . . . .	20
2.4.4	Final Summation . . . . .	21
2.4.5	Complexity of the Multilevel FMM Algorithm . . . . .	22
2.5	Adaptive Multilevel FMM . . . . .	24
2.5.1	Space Partition: D-Tree and C-Forest . . . . .	24
2.5.2	Upward Pass, Downward Pass and the Final Summation . . . . .	26
2.6	Structured Matrices and Related Algorithms . . . . .	26
3	Mathematical Analysis . . . . .	30
3.1	Basic Requirements for the FMM Algorithm . . . . .	30
3.2	Expansion and Translation Equations for the Coulombic System . . . . .	32
4	Implementation . . . . .	36
4.1	Language Considerations . . . . .	36
4.2	System Design . . . . .	37
4.2.1	The PotentialBuilder Object . . . . .	39
4.2.2	The FMM Blackbox . . . . .	40
4.2.3	Code Example . . . . .	42
5	Empirical Analysis . . . . .	44

5.1	The Accuracy of FMM and the Effect of Truncation Number $p$ . . . . .	44
5.2	The Accuracy of FMM and the Problem Size $N$ . . . . .	47
5.3	The Time Complexity of FMM and Problem Size $N$ . . . . .	49
6	Visualizing the FMM Algorithm	52
6.1	The Animated Visualization . . . . .	52
6.1.1	The FMM Builder and Space Partitioning . . . . .	53
6.1.2	Interaction with Partitions . . . . .	54
6.1.3	Animation of the FMM Solver . . . . .	55
6.1.4	Animation of the Adaptive FMM Algorithm . . . . .	56
A	Tables of Numerical Results	62
B	Technical Notes on the Software	76
B.1	License . . . . .	76
B.2	System Requirements . . . . .	76
B.3	Obtaining and running the code . . . . .	76
	Bibliography	78



## LIST OF TABLES

A.1	FMM Error and Computing Time for Various Truncation Numbers . . . . .	63
A.2	Minimum truncation number required to achieve $10^{-5}$ accuracy for various problem sizes . . . . .	66
A.3	Comparison of computing time (in milliseconds) for FMM and direct methods for various problem sizes . . . . .	71

## LIST OF FIGURES

2.1	Multipole Expansion . . . . .	9
2.2	Local Expansion . . . . .	10
2.3	Multipole-to-multipole Translation . . . . .	11
2.4	Multipole-to-local Translation . . . . .	12
2.5	Local-to-local Translation . . . . .	13
2.6	Space Partition and Morton-ordered Indices, $L = 2$ . . . . .	15
2.7	Space Partition and Box Labeling Example, $L = 3$ . . . . .	15
2.8	Hierarchical Spatial Domains . . . . .	17
2.9	Multilevel FMM Algorithm, Upward Pass Step 1 . . . . .	18
2.10	Multilevel FMM Algorithm, Upward Pass Step 2 . . . . .	19
2.11	Multilevel FMM Algorithm, Downward Pass Step 1 . . . . .	21
2.12	Multilevel FMM Algorithm, Downward Pass Step 2 . . . . .	22
2.13	Multilevel FMM Algorithm, Final Summation . . . . .	23
3.1	Plot of the $\Phi$ function as a variable of $z = \mathbf{y}_j - \mathbf{x}_i$ . . . . .	33
4.1	System Design . . . . .	38
4.2	The Potential Builder Object . . . . .	40
4.3	The FMM Blackbox . . . . .	41

5.1	FMM accuracy with respect to various truncation numbers . . . . .	46
5.2	FMM running time with respect to various truncation numbers . . . . .	47
5.3	Minimum truncation numbers needed to achieve $10^{-5}$ accuracy for various problem sizes. . . . .	48
5.4	FMM running time with respect to various problem sizes . . . . .	50
6.1	Application launched and initialized . . . . .	53
6.2	Source and target points generated and space partition performed accordingly . . . . .	54
6.3	User clicks inside Box (3, 27). . . . .	55
6.4	Displaying the properties of Box (3, 27). Its $E4$ neighborhood boxes are highlighted. . . . .	56
6.5	Animated FMM solver in progress, performing multipole-to-local trans- lation for Box (3, 27). . . . .	57
6.6	FMM solver completed. The computed error is displayed in the status panel. . . . .	57
6.7	Space Partitioning of Adaptive FMM: C-Forest . . . . .	58
6.8	Space Partitioning of Adaptive FMM: D-Tree . . . . .	58
6.9	Adaptive FMM Algorithm: Upward Pass . . . . .	59
6.10	Adaptive FMM Algorithm: R—R Translation . . . . .	59
6.11	Adaptive FMM Algorithm: S—R Translation . . . . .	60

6.12 Adaptive FMM Algorithm is completed. The software reports the number of translations and re-expansions used in the adaptive algorithm as well as what would the non-adaptive algorithm have required. In this particular example, the adaptive version needed 3853 (re)expansions, whereas the non-adaptive version needed 34674. . . . 61

## Chapter 1

### Introduction

#### 1.1 Overview

The Fast Multipole Method, introduced by Rokhlin and Greengard in [1] has been acclaimed as one of the greatest algorithms of the 20th century. The FMM algorithm dramatically reduces the complexity of matrix-vector multiplication involving a certain type of dense matrix, which can arise out of many physical systems.

FMM is a complicated algorithm. It involves many different types of operations in a recursive manner, which has made it very challenging for someone new to the algorithm to understand.

To that end, the author has developed a set of Java-based applications that visually demonstrates the FMM algorithm. It is hoped that the animations contained in these applications will assist future teaching and learning of the FMM algorithm.

These applications can also serve as a research tool, in that they provide user interfaces to allow the creation of custom data, as well as a detailed break-down of the operation counts for the regular and adaptive FMM algorithms.

Furthermore, the core functions of these Java-based applications have been

bundled into a programming toolkit for the FMM algorithm. The toolkit is simple to use and highly extensible, and can serve as the basis for other research projects in this area.

## 1.2 The 2D Coulomb system

Given the matrix

$$\Phi = \begin{bmatrix} \Phi_{11} & \Phi_{12} & \cdots & \Phi_{1N} \\ \Phi_{21} & \Phi_{22} & \cdots & \Phi_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ \Phi_{M1} & \Phi_{M2} & \cdots & \Phi_{MN} \end{bmatrix}, \quad (1.1)$$

where

$$\Phi_{ji} = \log \|\mathbf{y}_j - \mathbf{x}_i\|, \quad (1.2)$$

and  $\mathbf{y}$  and  $\mathbf{x}$  are arrays containing  $M$  and  $N$  points, respectively, in the complex plane, we seek to achieve a fast computation of matrix-vector multiplications of the form  $\Phi \mathbf{u} = \mathbf{v}$  where  $\mathbf{u}$  is a given vector of length  $N$ .

The structured matrix  $\Phi$  arises from solving the Coulomb system of charged particles. Given a point charge of unit strength at point  $\mathbf{x}_0$  in the complex plane, then for any  $\mathbf{x}$  also in the complex plane with  $\mathbf{x} \neq \mathbf{x}_0$ , the potential at point  $\mathbf{x}$  due to the charge of  $\mathbf{x}_0$  is given by

$$\Phi_{\mathbf{x}_0}(\mathbf{x}) = -\log \|\mathbf{x} - \mathbf{x}_0\|. \quad (1.3)$$

For the clarity of analysis, we drop the negative sign and note the similarity of 1.3 to the elements of 1.2. Indeed, when we are given charged particles at points  $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N$  with charge potential of  $u_1, u_2, \dots, u_N$  respectively, in order to compute the potentials  $v_1, v_2, \dots, v_M$  at points  $\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_M$ , we would need to compute the matrix-vector product

$$\begin{bmatrix} \Phi_{11} & \Phi_{12} & \cdots & \Phi_{1N} \\ \Phi_{21} & \Phi_{22} & \cdots & \Phi_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ \Phi_{M1} & \Phi_{M2} & \cdots & \Phi_{MN} \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_N \end{bmatrix}, \quad (1.4)$$

where  $\Phi_{ji} = \log \|\mathbf{y}_j - \mathbf{x}_i\|$ .

Direct computation of this problem requires  $O(MN)$  operations, or, in the case where  $M \sim N$ , the order of complexity is  $O(N^2)$ . For a given precision  $\epsilon$ , the Fast Multipole Method can accelerate this computation substantially and achieve a complexity of  $O(N \log N)$  in the “building” step and  $O(N)$  in the “solving” step. Since most iterative methods for solving linear systems involve matrix-vector multiplications, this accelerated algorithm can also speed up such iterative methods involving appropriately structured matrices.

In [1, 2], Rokhlin and Greengard discussed this physical model of charged particles with the potential and force obtained as the sum of pairwise interaction from Coulomb’s Law. These papers presented the analysis necessary for the Fast

Multipole Method to be applicable for this particular problem and introduced the FMM algorithm. In [3], Tang further explained and illustrated the algorithm from a different perspective.

### 1.3 Outline of the Thesis

In Chapter 2, we present the Fast Multipole Method based on [4, 5]. The key components of the FMM are expansion (multipole and local), translation (multipole-to-multipole, multipole-to-local and local-to-local) and space partition. We then outline the FMM algorithm, followed by a brief complexity analysis. An adaptive version of the FMM algorithm is then discussed in the final section of the chapter.

In Chapter 3, we state several slightly modified results from [1]. All mathematical analysis needed to perform the FMM algorithm on the Coulombic system described above is presented in this chapter, including the expansion equations and translation matrices.

In Chapter 4, several implementation issues are discussed. Java is chosen as the programming language for this project due to its strengths as an object-oriented language and portability to various platforms. We then discuss the system design of this Java-FMM implementation, especially the design and implementation of the “FMM Blackbox” and “PotentialBuilder” objects, and present some simple code examples that utilize the Java-FMM routine.



In Chapter 5, we conduct several numerical experiments using code developed in Chapter 4 to measure the performance of the FMM algorithm under various conditions. We will see that the FMM algorithm can be made arbitrarily accurate, subject to the limitations of machine precision, and that it indeed has an advantage in time complexity when compared to direct computation.

In Chapter 6, we discuss and demonstrate an animated Java applet that simulates the FMM solver in action. The design, usage and limitations of this applet is then presented in detail.

## Chapter 2

### The Fast Multipole Method

#### 2.1 Overview

The Fast Multipole Method, acclaimed as one of the ten most significant algorithms in scientific computation discovered in the 20th century [6], represents a revolutionary view towards algorithm design for computational tasks. It is an example of a class of algorithms that trades accuracy for reduced complexity. The algorithm allows the product of a dense matrix and a vector to be approximated in  $O(N \log N)$  operations within a pre-established error bound  $\epsilon$ . Since many scientific computations only require a certain accuracy (up to the machine precision), FMM is well suited for extremely large computational problems and offer improved efficiency and reduced memory requirement.

How does FMM reduce the computational complexity for large-scale matrix multiplications? The following is an analogy of how FMM works.

Suppose that several customers are ordering dinner at a restaurant.

Customer A: I would like to have a beer, a salad and a steak.

Customer B: A glass of water, a salad and a steak.

Customer C: A beer, a soup and a steak.

Customer D: A beer, a soup and a steak.

The waiter can record the orders sequentially as 12 separately prepared items, or, as most seasoned waiters will do, will say the following:

Waiter: So that will be 3 beers, 1 water, 2 salads, 2 soups and 4 steaks.

Thus, the 12 items are partitioned into 5 groups before the waiter sends the order to the bar and the kitchen. Multiple orders of the same food can be prepared together, thus increasing efficiency. Once the food and drinks are ready, the waiter un-groups the items and distributes them among the customers.

Efficiency can be further increased by performing regrouping at a higher level, that is, if orders from multiple tables are combined and regrouped before reported to the kitchen.

The Fast Multipole Method works in a similar fashion. It achieves regrouping by operations called “expansions”, which expand the expression to be evaluated at one evaluation point into the sum of a series. The terms in this series will be similar to those of the expansions from other evaluation points and thus can be regrouped before actual calculations are carried out. Combined with operations called “translations”, which translates the terms in the series from one center to another, the reduction in computational complexity is achieved by computing these terms only once and reusing them for all points in the domain where the expansion

is valid.

Similar to the restaurant analogy above, the efficiency of FMM can also be increased by regrouping at a higher level. This is achieved by recursively using translations to group together more evaluations and is commonly referred to as Multi-Level FMM (MLFMM).

The following sections define these “expansions” and “translations” rigorously and explain the MLFMM algorithm in greater details.

## 2.2 Expansions

**R** and **S** expansions are fundamental to the FMM algorithm. By “expansion”, we attempt to rewrite the elements of the matrix  $\Phi$  as the sum of an infinite series. There are two types of expansions, far-field and near-field, each used in different stages of the FMM algorithm.

### 2.2.1 Multipole Expansion (**S**-Expansion)

Let  $\mathbf{x}_* \in \mathbf{C}$  be a point other than  $\mathbf{x}_i$  in the complex plane. We call the expansion of the form

$$\Phi(\mathbf{y}, \mathbf{x}_i) = \sum_{m=0}^{\infty} b_m(\mathbf{x}_i, \mathbf{x}_*) \mathbf{S}_m(\mathbf{y} - \mathbf{x}_*),$$

far-field expansion (or **S**-Expansion) outside a circle of radius  $R_*$  centered at  $\mathbf{x}_*$  if the series converges  $\forall \mathbf{y}$  such that  $\|\mathbf{y} - \mathbf{x}_*\| > R_*$ , as shown in Figure 2.1.

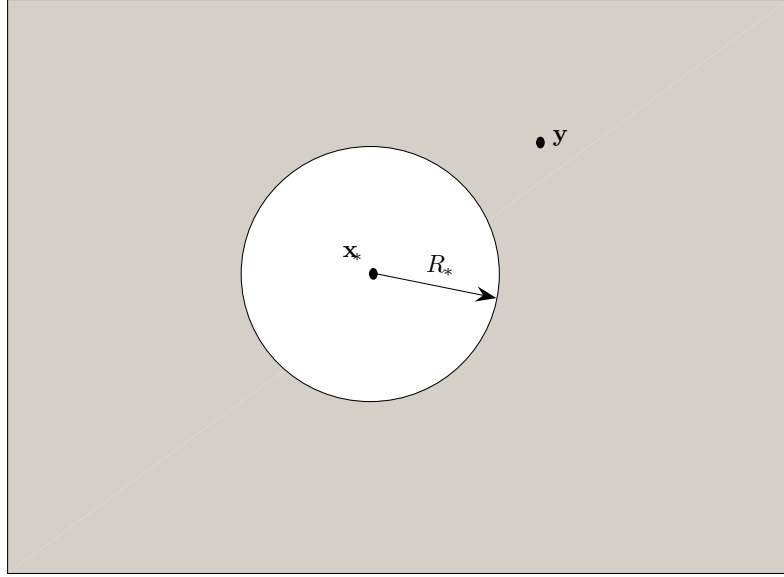


Figure 2.1: Multipole Expansion

### 2.2.2 Local Expansion ( $\mathbf{R}$ -Expansion)

Let  $\mathbf{x}_* \in \mathbf{C}$  be a point other than  $\mathbf{x}_i$  in the complex plane. We call the expansion

$$\Phi(\mathbf{y}, \mathbf{x}_i) = \sum_{m=0}^{\infty} a_m(\mathbf{x}_i, \mathbf{x}_*) \mathbf{R}_m(\mathbf{y} - \mathbf{x}_*),$$

near-field expansion (or  $\mathbf{R}$ -Expansion) inside a circle of radius  $R_*$  centered at  $\mathbf{x}_*$  if the series converges  $\forall \mathbf{y}$  such that  $\|\mathbf{y} - \mathbf{x}_*\| < R_*$ , as shown in Figure 2.2.

## 2.3 Translation

Translation, or re-expansion, is used to further reduce the steps required in the computation. There are three types of translations used in FMM, namely,  $\mathbf{S}|\mathbf{S}$

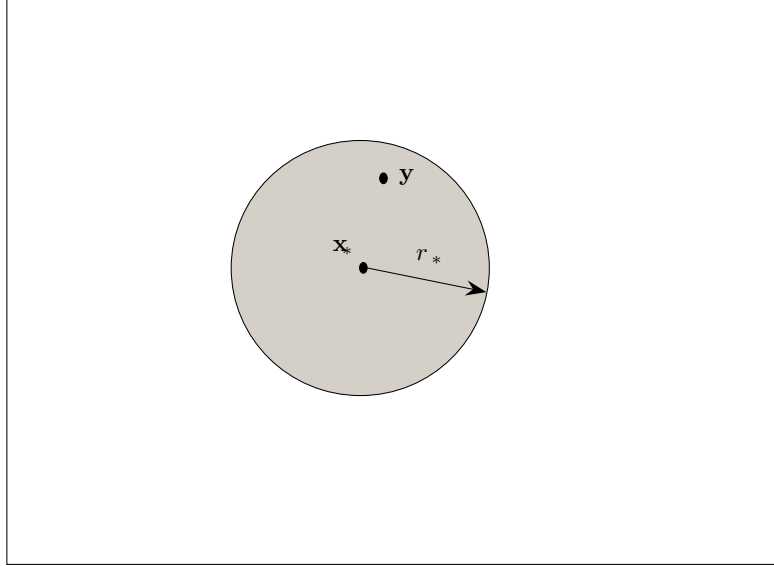


Figure 2.2: Local Expansion

translation,  $\mathbf{S}|\mathbf{R}$  translation and  $\mathbf{R}|\mathbf{R}$  translation.

### 2.3.1 Multipole-to-Multipole ( $\mathbf{S}|\mathbf{S}$ ) Translation

Let  $\mathbf{y} - \mathbf{x}_*$  and  $\mathbf{y} - \mathbf{x}_* + \mathbf{t} \in \Omega_r(\mathbf{x}_*) \subset C$ ,  $\Omega_r(\mathbf{x}_*) : |\mathbf{y} - \mathbf{x}_*| > r$  and

$$S_n(\mathbf{y} - \mathbf{x}_* + \mathbf{t}) = \sum_{l=0}^{\infty} (\mathbf{S}|\mathbf{S})_{ln}(\mathbf{t}) S_l(\mathbf{y} - \mathbf{x}_*).$$

The infinite matrix

$$(\mathbf{S}|\mathbf{S})(\mathbf{t}) = \begin{bmatrix} (\mathbf{S}|\mathbf{S})_{00} & (\mathbf{S}|\mathbf{S})_{10} & \cdots \\ (\mathbf{S}|\mathbf{S})_{10} & (\mathbf{S}|\mathbf{S})_{11} & \cdots \\ \cdots & \cdots & \cdots \end{bmatrix}, \quad (2.1)$$

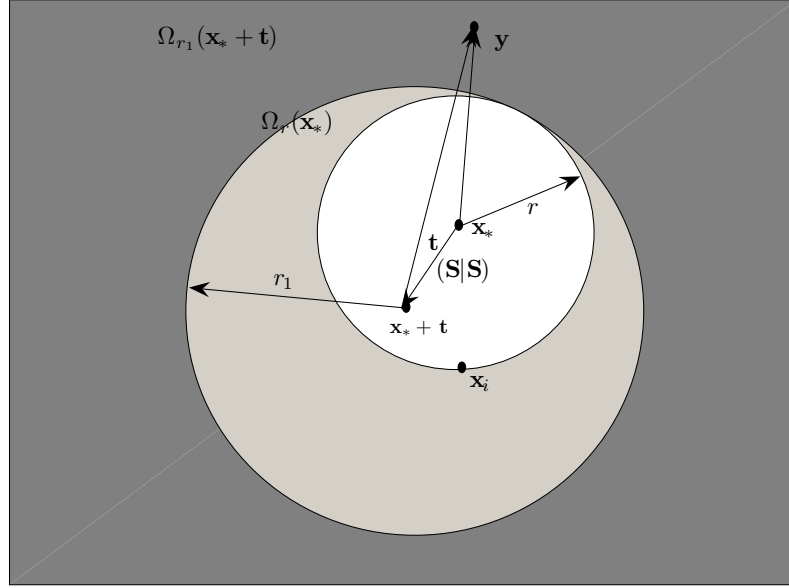


Figure 2.3: Multipole-to-multipole Translation

is called  $\mathbf{S}|\mathbf{S}$  translation matrix. The  $\mathbf{S}|\mathbf{S}$  translation is shown in Figure 2.3.

### 2.3.2 Multipole-to-Local ( $\mathbf{S}|\mathbf{R}$ ) Translation

Let  $\mathbf{y} - \mathbf{x}_* \in \Omega_r(\mathbf{x}_*) \subset C$ ,  $\Omega_r(\mathbf{x}_*) : |\mathbf{y} - \mathbf{x}_*| > r$  and  $\mathbf{y} - \mathbf{x}_* + \mathbf{t} \in \Omega_{r_1}(\mathbf{x}_*) \subset C$ ,  
 $\Omega_{r_1}(\mathbf{x}_*) : |\mathbf{y} - \mathbf{x}_*| < r_1$  and  $\Omega_{r_1} \in \Omega_r$

$$R_n(\mathbf{y} - \mathbf{x}_n + \mathbf{t}) = \sum_{l=0}^{\infty} (\mathbf{S}|\mathbf{R})_{ln}(\mathbf{t}) R_l(\mathbf{y} - \mathbf{x}_*).$$

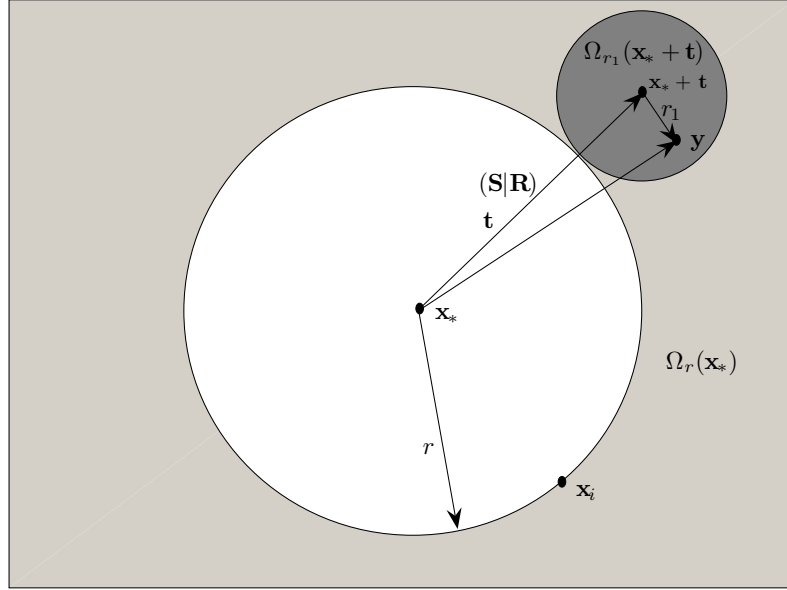


Figure 2.4: Multipole-to-local Translation

The infinite matrix

$$(\mathbf{S}|\mathbf{R})(\mathbf{t}) = \begin{bmatrix} (\mathbf{S}|\mathbf{R})_{00} & (\mathbf{S}|\mathbf{R})_{10} & \cdots \\ (\mathbf{S}|\mathbf{R})_{10} & (\mathbf{S}|\mathbf{R})_{11} & \cdots \\ \cdots & \cdots & \cdots \end{bmatrix}, \quad (2.2)$$

is called  $\mathbf{S}|\mathbf{R}$  translation matrix. The  $\mathbf{S}|\mathbf{R}$  translation is shown in Figure 2.4.

### 2.3.3 Local-to-Local $(\mathbf{R}|\mathbf{R})$ Translation

Let  $\mathbf{y} - \mathbf{x}_*$  and  $\mathbf{y} - \mathbf{x}_* + \mathbf{t} \in \Omega_r(\mathbf{x}_*) \subset C$ ,  $\Omega_r(\mathbf{x}_*) : |\mathbf{y} - \mathbf{x}_*| < r$  and

$$R_n(\mathbf{y} - \mathbf{x}_* + \mathbf{t}) = \sum_{l=0}^{\infty} (\mathbf{R}|\mathbf{R})_{ln}(\mathbf{t}) R_l(\mathbf{y} - \mathbf{x}_*).$$



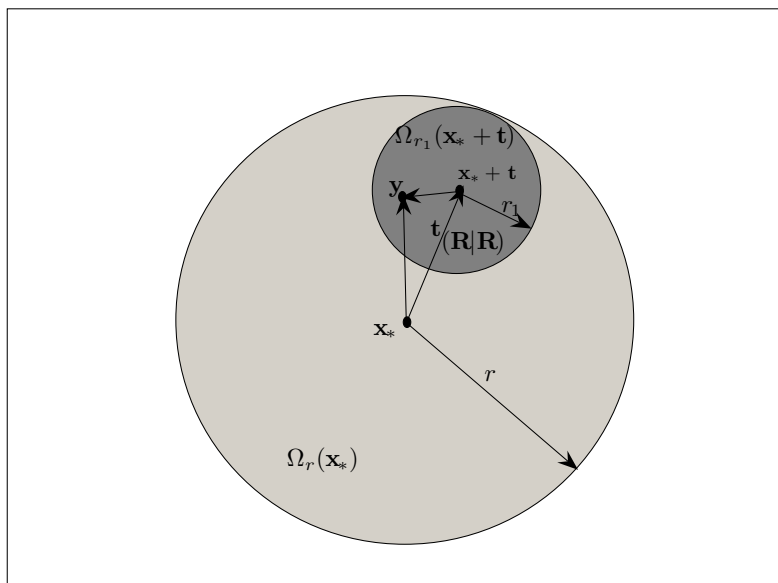


Figure 2.5: Local-to-local Translation

The infinite matrix

$$(\mathbf{R}|\mathbf{R})(\mathbf{t}) = \begin{bmatrix} (\mathbf{R}|\mathbf{R})_{00} & (\mathbf{R}|\mathbf{R})_{10} & \cdots \\ (\mathbf{R}|\mathbf{R})_{10} & (\mathbf{R}|\mathbf{R})_{11} & \cdots \\ \cdots & \cdots & \cdots \end{bmatrix}, \quad (2.3)$$

is called  $\mathbf{R}|\mathbf{R}$  translation matrix. The  $\mathbf{R}|\mathbf{R}$  translation is shown in Figure 2.5.

#### 2.3.4 Translation Error

The translation matrices presented above are infinite in dimension and the translations are exact operations. However, in the actual FMM algorithm, a truncated matrix of size  $p \times p$  is used in lieu of the actual transformation matrix, thus intro-

ducing an error in the computation. This truncation is desirable because it reduces the computational complexity, and it can be shown that [7, 5, 8], for convergent series, the error introduced in this step is bounded and can be reduced to an arbitrarily small number by increasing the truncation number,  $p$ . It is also possible to determine what truncation number  $p$  to use in order to achieve a desired level of accuracy [7, 5].

## 2.4 The Multilevel FMM (MLFMM) Algorithm

The multilevel FMM algorithm can be divided into four distinct steps: space partition, upward pass, downward pass and final summation.

### 2.4.1 Space Partition

Space partition divides the unit square in the complex plane into  $4^L$  equally sized boxes, where  $L$  is the level of partition. With  $l$  varying between 0 and  $L$ , at level  $l$ , there are  $4^l$  equally sized boxes, each of which is then assigned an index number according to Morton-order (Figure 2.6) for quadtrees and 2-dimensional matrices [9]. This way, each box in the partition is uniquely identified by  $l$ , its level, and  $n$ , its index number at that level. Thus, we can use  $(n, l)$  as the global index of this box. For example, the dark-shaded box in Figure 2.7 has the global index  $(15, 3)$  and the light-shaded box is  $(10, 2)$ .

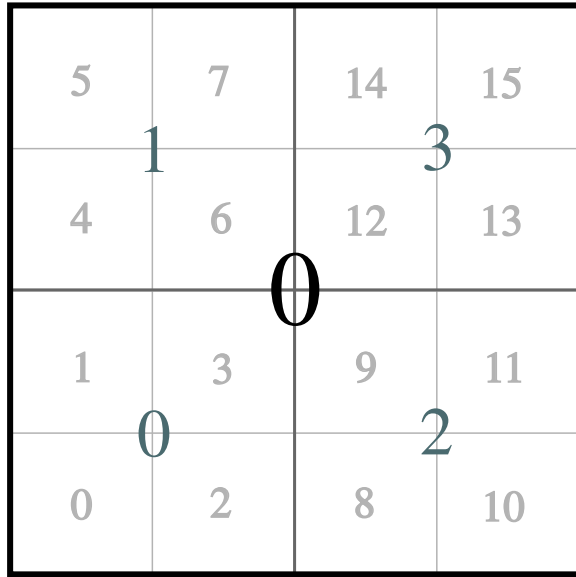


Figure 2.6: Space Partition and Morton-ordered Indices,  $L = 2$

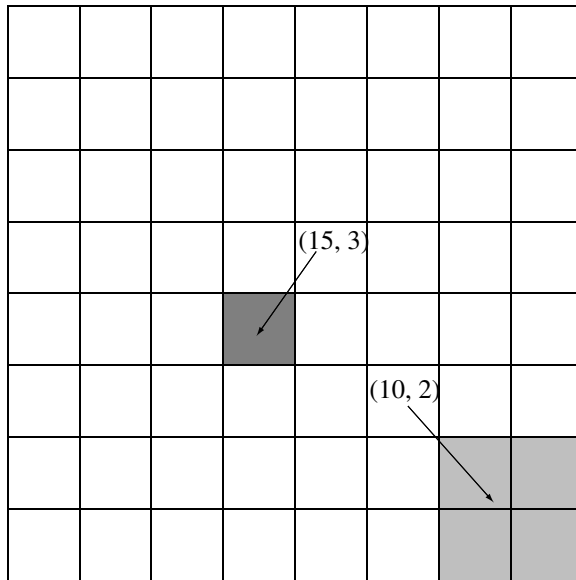


Figure 2.7: Space Partition and Box Labeling Example,  $L = 3$

In addition, we define the following sets for box  $(n, l)$ :

Set  $E_1(n, l)$  is defined as all spatial points in the domain (Figure 2.8(a))

$$I_1(n, l) = (n, l);$$

Set  $E_2(n, l)$  is defined as all spatial points in the domain (Figure 2.8(b))

$$I_2(n, l) = \{Neighbors(n, l) \cup I_1(n, l)\};$$

Set  $E_3(n, l)$  is defined as (Figure 2.8(c))

$$E_3(n, l) = E_1(0, 0) \setminus E_2(n, l);$$

Set  $E_4(n, l)$  is defined as (Figure 2.8(d))

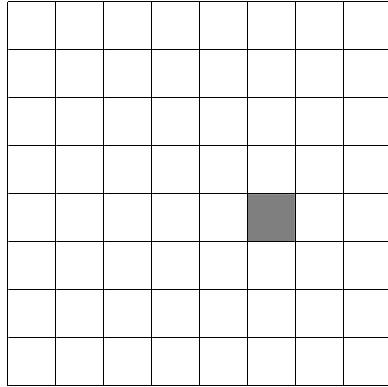
$$E_4(n, l) = E_2(Parent(n), l - 1) \setminus E_2(n, l).$$

We also define the potentials due to sources in each of the domains above:

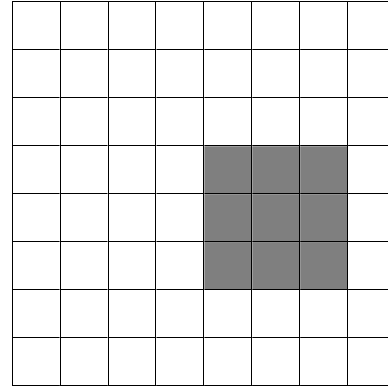
$$\begin{aligned}\Phi_1^{(n,l)}(\mathbf{y}) &= \sum_{\mathbf{x}_i \in E_1(n,l)} u_i \Phi(\mathbf{y}, \mathbf{x}_i); \\ \Phi_2^{(n,l)}(\mathbf{y}) &= \sum_{\mathbf{x}_i \in E_2(n,l)} u_i \Phi(\mathbf{y}, \mathbf{x}_i); \\ \Phi_3^{(n,l)}(\mathbf{y}) &= \sum_{\mathbf{x}_i \in E_3(n,l)} u_i \Phi(\mathbf{y}, \mathbf{x}_i); \\ \Phi_4^{(n,l)}(\mathbf{y}) &= \sum_{\mathbf{x}_i \in E_4(n,l)} u_i \Phi(\mathbf{y}, \mathbf{x}_i).\end{aligned}$$

Since sets  $E_2(n, l)$  and  $E_3(n, l)$  are complementary, for arbitrary  $n$  and  $l$ ,

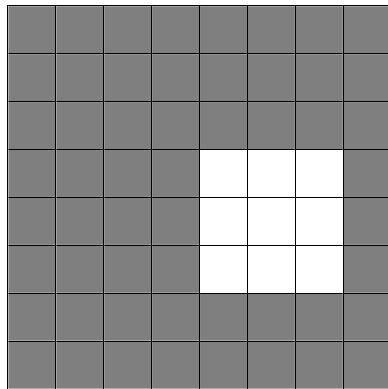
$$\begin{aligned}\Phi(\mathbf{y}) &= \sum_{i=1}^N u_i \Phi(\mathbf{y}, \mathbf{x}_i) \\ &= \sum_{\mathbf{x}_i \in E_2(n,l) \cup E_3(n,l)} u_i \Phi(\mathbf{y}, \mathbf{x}_i) \\ &= \Phi_2^{(n,l)}(\mathbf{y}) + \Phi_3^{(n,l)}(\mathbf{y}).\end{aligned}$$



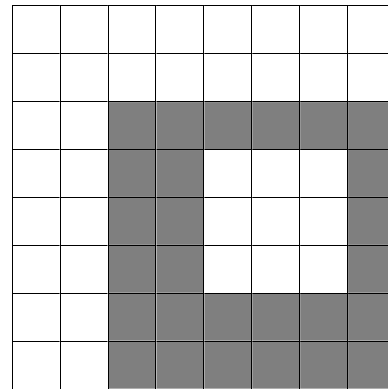
(a)  $E_1(n, l)$



(b)  $E_2(n, l)$



(c)  $E_3(n, l)$



(d)  $E_4(n, l)$

Figure 2.8: Hierarchial Spatial Domains

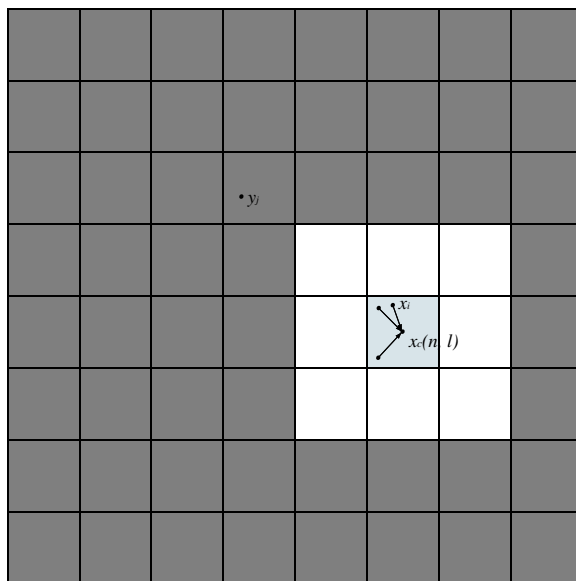


Figure 2.9: Multilevel FMM Algorithm, Upward Pass Step 1

### 2.4.2 Upward Pass

#### Step 1. Multipole Expansion

At the finest level of space subdivision, build multipole expansion for source points inside each box  $(n, L)$  near  $\mathbf{x}_c^{(n,L)}$ , the center of that box (Figure 2.9):

$$\Phi_1^{(n,L)}(\mathbf{y}) = \mathbf{C}^{(n,L)} \mathbf{S}(\mathbf{y} - \mathbf{x}_c^{(n,L)}),$$

and

$$\mathbf{C}^{(n,L)} = \sum_{x_i \in E_1(n,l)} u_i \mathbf{B}(\mathbf{x}_i, \mathbf{x}_c^{(n,L)}).$$

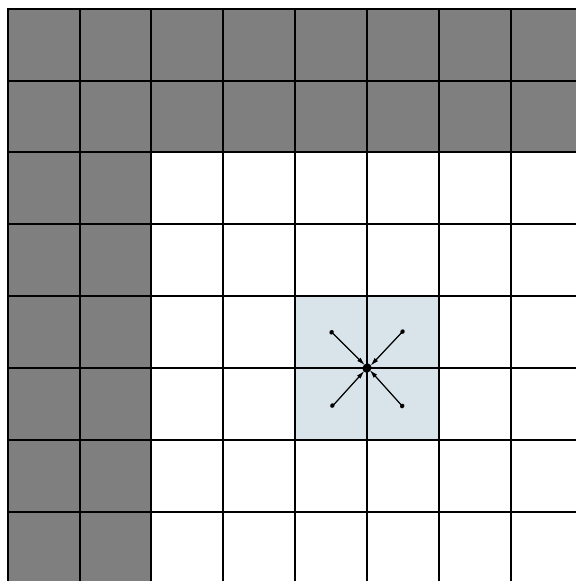


Figure 2.10: Multilevel FMM Algorithm, Upward Pass Step 2

Step 2. Multipole-to-Multipole Translation

For  $l = L - 1, \dots, 2$ , recursively form  $\Phi_1^{(n,l)}(\mathbf{y})$  by translating  $\Phi_1^{Children(n),l+1}(\mathbf{y})$  to near the center of the parent box and summing up the contributions of all child boxes (Figure 2.10):

$$\Phi_1^{(n,l)}(\mathbf{y}) = \mathbf{C}^{(n,l)} \mathbf{S}(\mathbf{y} - \mathbf{x}_c^{(n,l)}),$$

where

$$\mathbf{C}^{(n,l)} = \sum_{n' \in Children(n)} (\mathbf{S}|\mathbf{S}) (\mathbf{x}_c^{(n',l+1)} - \mathbf{x}_c^{(n,l)}) \mathbf{C}^{(n',l+1)}.$$

### 2.4.3 Downward Pass

During the downward pass, steps 1 and 2 should be performed recursively for levels  $l = 2, \dots, L$  of space subdivision.

#### Step 1. Multipole-to-Local Translation

At this step, form coefficients of regular expansion for function  $\Phi_4^{(n,l)}(\mathbf{y})$  (Figure 2.11).

$$\Phi_4^{(n,l)}(\mathbf{y}) = \tilde{D}^{n,l} R(\mathbf{y} - \mathbf{x}_c^{(n,l)}),$$

where

$$\tilde{D}^{n,l} = \sum_{m \in E_4(n,l)} \mathbf{S} | \mathbf{R} \left( \mathbf{x}_c^{(n,l)} - \mathbf{x}_c^{(m,l)} \right) \mathbf{C}^{(m,l)}.$$

#### Step 2. Local-to-local Translation

At  $l = 2$ , we have

$$\Phi_3^{(n,2)}(\mathbf{y}) = \Phi_4^{(n,2)}(\mathbf{y}),$$

and

$$D^{(n,2)} = \tilde{D}^{(n,2)}.$$

Form  $\Phi_3^{(n,l)}(\mathbf{y})$  by adding  $\Phi_4^{(Parent(n),l-1)}(\mathbf{y})$  to  $(\mathbf{R} | \mathbf{R})$ -translated coefficients of the parent box to the child center (Figure 2.12):

$$\Phi_3^{(n,l)}(\mathbf{y}) = D^{(n,l)} R(\mathbf{y} - \mathbf{x}_c^{(n,l)}),$$



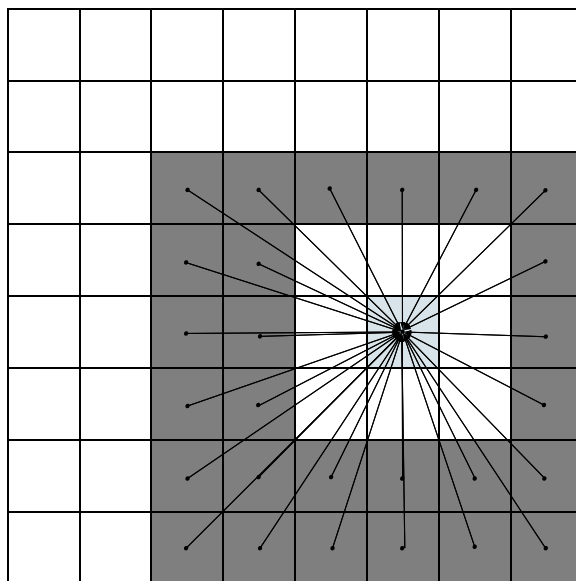


Figure 2.11: Multilevel FMM Algorithm, Downward Pass Step 1

where

$$D^{(n,l)} = \tilde{D}^{(n,l)} + (\mathbf{R}|\mathbf{R}) \left( \mathbf{x}_c^{(n,l)} - \mathbf{x}_c^{(m,l-1)} \right) D^{(m,l-1)},$$

and  $m = \text{Parent}(n)$ .

#### 2.4.4 Final Summation

As soon as coefficients  $D^{(n,L)}$  are determined, total potential can be computed for any point  $\mathbf{y}_j \in E_1(0,0)$ , where  $\Phi_2^{(n,l)}(\mathbf{y})$  can be computed directly (Figure 2.13).

Therefore,

$$\begin{aligned} v_j &= \Phi(\mathbf{y}_j) \\ &= \sum_{\mathbf{x}_i \in E_2(n,L)} u_i \Phi(\mathbf{y}_j, \mathbf{x}_i) + D^{(n,L)} R(\mathbf{y}_j - \mathbf{x}_c^{(n,L)}), \end{aligned}$$

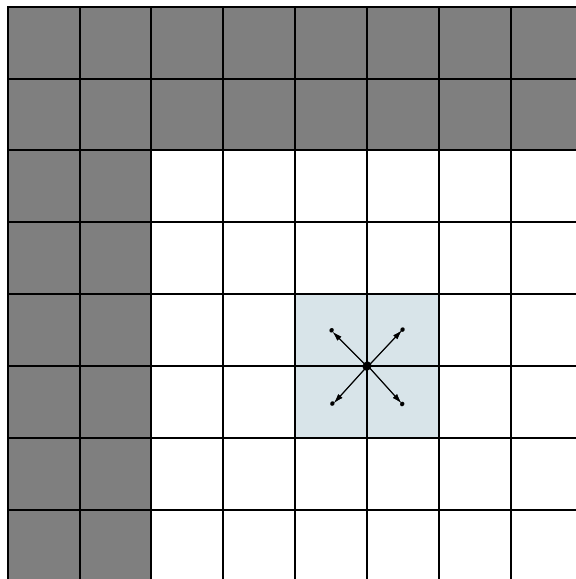


Figure 2.12: Multilevel FMM Algorithm, Downward Pass Step 2

where  $\mathbf{y}_j \in E_1(n, L)$ .

#### 2.4.5 Complexity of the Multilevel FMM Algorithm

Assuming that there are approximately as many source ( $\mathbf{X}$ ) points as there are target ( $\mathbf{Y}$ ) points, i.e.,  $N \sim M$ , the overall time complexity of the multilevel FMM algorithm is  $O(N \log N)$ , as is rigorously proven in [8]. This is considerable improvement over the regular  $O(N^2)$  operations needed by direct multiplication.

In practice, the FMM algorithm is often used in two steps and can enjoy even further optimization in certain applications:

- MLFMM Constructor

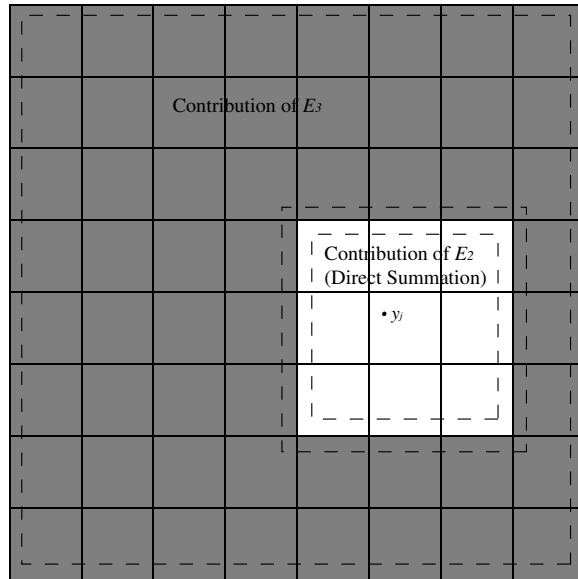


Figure 2.13: Multilevel FMM Algorithm, Final Summation

In this step, the expansion and translation coefficients for each level of space partition are computed and stored in appropriate data structures. No knowledge about the vector  $u$  being multiplied is required at this step. The complexity of this step is shown [5] to be  $O(N \log N)$ .

- MLFMM Solver

Once the vector  $u$  is known, we can perform the upward and downward passes as outlined above. It can be shown [5] that the complexity for the upward and downward passes are  $O(N)$ .

In situations when we need to repeatedly compute the product of a matrix with different vectors, we can perform the operations in “MLFMM Constructor” only once

and invoke the “MLFMM Solver” repeatedly. With a large number of repetitions, the time complexity of MLFMM approaches  $O(N)$ . This is particularly useful in iterative methods for solving linear systems and eigenvalues.

## 2.5 Adaptive Multilevel FMM

The Multilevel FMM algorithm above partitions the domain into a tree-like structure that can be classified as a complete quad-tree, that is, all branches and leaves of the tree are present. This works well for problems with uniformly distributed source and target points in the domain because most boxes at the finest level are populated. However, for other problems this may not always be true, and further optimization can be achieved by partitioning the domain adaptively. For regions with fewer points, the level of partitioning can be smaller than those with a higher density of points.

Many variations of the adaptive FMM algorithm have been proposed. We briefly introduce one of them as follows. A more detailed treatment of the subject can be found in [10].

### 2.5.1 Space Partition: D-Tree and C-Forest

To reduce the number of partitions while maintaining the efficiency of FMM, we require that each leaf node in the upward-pass tree, which we will call “D-Tree”, to

satisfy the following requirement: the number of target points in the neighborhood of this leaf box must be less than a pre-established clustering parameter,  $q$ . That is, for box  $(n, l)$ , we require that

$$N_n < q,$$

where  $N_n$  is the number of  $\mathbf{Y}_i$  such that  $\mathbf{Y}_i \in E_1(n, l)$ .

It is shown ([10]) that the optimum clustering parameter is

$$q_{opt} = 3^d 2^{-dl_{max}^{opt}} N,$$

where  $d$  is the dimension of the problem,  $N$  is the problem size, and  $l_{max}^{opt}$  is the optimal number of levels as determined in the regular multilevel FMM algorithm.

The algorithm for space partitioning according to this rule is a recursive one.

Step 1. Partition the domain according to the non-adaptive algorithm using  $l_{max} = 2$ .

Step 2. For each leaf-node  $(n, 2)$  in the tree from Step 1, compute  $N_n$  as specified above.

Step 3. If  $N_n \leq q$ , then no further partitioning is required on this node. Otherwise equally divide this box into  $2^d$  child boxes, where  $d$  is the dimension of this problem.

Step 4. Perform steps 2 and 3 on each of the newly created child boxes from Step 3.

The resulting “D-tree” is the tree that the Adaptive FMM algorithm will follow during the downward pass.

The “C-Forest”, the structure that the Adaptive FMM algorithm follows during the upward pass, is derived from the “D-Tree” as the following: For each node in the D-Tree, mark nodes in its  $E4$  neighborhood as a member of the “C-Forest” if they contain source points. After such all such nodes are located and marked, traverse through them to identify any parent-child relationship. The resulting structure is a collection of trees, i.e., a forest structure which we call the “C-Forest.”

### 2.5.2 Upward Pass, Downward Pass and the Final Summation

Upward Pass is performed in the same manner as in the regular non-adaptive FMM algorithm but using the “C-Forest” as the path of traversal, skipping over all nodes that are not in the forest. Downward Pass is carried out similarly with the “D-Tree” as the traversal path.

Final summation is performed on each leaf node of the “D-Tree” in exactly the same way as the non-adaptive FMM algorithm.

## 2.6 Structured Matrices and Related Algorithms

The Fast Multipole Method is one of many algorithms designed to work with a class of matrices called “structured matrices” [3]. A dense matrix of order  $N \times N$  is

called “structured” if its entries depend on only  $O(N)$  parameters. As we can see, the matrix derived from the Coulomb problem depends on  $M + N$  parameters and is indeed a structured matrix. Other commonly seen structured matrices include:

- Fourier Matrix

$$\begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w_n & w_n^2 & \cdots & w_n^{n-1} \\ 1 & w_n^2 & w_n^4 & \cdots & w_n^{2(n-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w_n^{n-1} & w_n^{2(n-1)} & \cdots & w_n^{(n-1)(n-1)} \end{bmatrix},$$

where

$$w_n = e^{-\frac{2\pi i}{n}}.$$

- Toeplitz Matrix

$$\begin{bmatrix} x_0 & x_1 & x_2 & \cdots & x_{n-1} \\ x_{-1} & x_0 & x_1 & \cdots & x_{n-2} \\ x_{-2} & x_{-1} & x_0 & \cdots & x_{n-3} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{-n+1} & x_{-n+2} & x_{-n+3} & \cdots & x_0 \end{bmatrix},$$

- Hankel Matrix

$$\begin{bmatrix} x_{-n+1} & x_{-n+2} & x_{-n+3} & \cdots & x_0 \\ x_{-n+2} & x_{-n+3} & x_{-n+4} & \cdots & x_1 \\ x_{-n+3} & x_{-n+4} & x_{-n+5} & \cdots & x_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_0 & x_1 & x_2 & \cdots & x_{n-1} \end{bmatrix},$$

- Vandermonde Matrix

$$\begin{bmatrix} 1 & 1 & \cdots & 1 \\ x_0 & x_1 & \cdots & x_{n-1} \\ \vdots & \vdots & \ddots & \vdots \\ x_0^{n-1} & x_1^{n-1} & \cdots & x_{n-1}^{n-1} \end{bmatrix},$$

One of the best-known matrix-related algorithm is the Fast Fourier Transform (FFT) [11]. FFTs were first discussed by Cooley and Tukey [12], although Gauss had actually described the critical factorization step as early as 1805 [13, 14]. The FFT algorithm also reduces the complexity for an N-point problem from  $O(N^2)$  to  $O(N \log N)$ .

Another related algorithm is the Fast Gauss Transform (FGT). Introduced by Greengard and Strain [15], it is an important variant of the more general fast



multipole method. It efficiently evaluates the sum of Gaussians:

$$G(y_j) = \sum_{i=1}^N q_i e^{-\frac{\|y_j - x_i\|^2}{h^2}}, \quad j = 1, \dots, M,$$

which is equivalent to the matrix-vector product

$$\begin{bmatrix} G(y_1) \\ G(y_2) \\ \vdots \\ G(y_M) \end{bmatrix} = \begin{bmatrix} \Phi_{11} & \Phi_{21} & \cdots & \Phi_{N1} \\ \Phi_{12} & \Phi_{22} & \cdots & \Phi_{N2} \\ \vdots & \vdots & \ddots & \vdots \\ \Phi_{1M} & \Phi_{2M} & \cdots & \Phi_{NM} \end{bmatrix} \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_N \end{bmatrix},$$

where

$$\Phi_{ij} = e^{-\frac{\|y_j - x_i\|^2}{h^2}}.$$

The fast Gauss transform is widely applied in many areas including option pricing, computer vision and pattern recognition [16, 17, 18].

## Chapter 3

### Mathematical Analysis

#### 3.1 Basic Requirements for the FMM Algorithm

For the FMM algorithm to be applicable to a matrix-vector multiplication, the following requirements must be satisfied:

- We have two sets of points in a vector space:

$$\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N\}, \mathbf{x}_i \in R^d, i = 1, \dots, N,$$

$$\mathbf{Y} = \{\mathbf{y}_1, \mathbf{y}_2, \dots, \mathbf{y}_M\}, \mathbf{y}_j \in R^d, j = 1, \dots, M;$$

- We have potentials functions (or, “mother functions”):

$$\Phi(\mathbf{x}_i, \mathbf{y}) : R^d \rightarrow R, \mathbf{y} \in R^d, i = 1, \dots, N;$$

- These functions have multipole expansion:

$$\Phi(\mathbf{x}_i, \mathbf{y}) = \mathbf{A}(\mathbf{x}_i, \mathbf{x}_*) \circ \mathbf{R}(\mathbf{y} - \mathbf{x}_*), \quad |\mathbf{y} - \mathbf{x}_*| < r < |\mathbf{x}_i - \mathbf{x}_*|, \quad i = 1, \dots, N;$$

- These functions have local expansion:

$$\Phi(\mathbf{x}_i, \mathbf{y}) = \mathbf{B}(\mathbf{x}_i, \mathbf{x}_*) \circ \mathbf{S}(\mathbf{y} - \mathbf{x}_*), \quad |\mathbf{y} - \mathbf{x}_*| > r > |\mathbf{x}_i - \mathbf{x}_*|, \quad i = 1, \dots, N;$$

- The product  $\circ$  is distributive operation with respect to addition:

$$(u_1 \mathbf{A}_1 + u_2 \mathbf{A}_2) \circ \mathbf{F} = u_1 \mathbf{A}_1 \circ \mathbf{F} + u_2 \mathbf{A}_2 \circ \mathbf{F}, \quad \mathbf{F} = \mathbf{S}, \mathbf{R};$$

- The local expansion coefficients can be  $R|R$ -translated (i.e., local-to-local translation):

$$|\mathbf{x} - \mathbf{x}_{*2}| < |\mathbf{x}_i - \mathbf{x}_{*1}| - |\mathbf{x}_{*1} - \mathbf{x}_{*2}|,$$

$$\mathbf{A}(\mathbf{x}_i, \mathbf{x}_{*2}) = (R|R)(\mathbf{x}_{*2} - \mathbf{x}_{*1})\mathbf{A}(\mathbf{x}_i - \mathbf{x}_{*1});$$

- The multipole expansion coefficients can be  $S|S$ -translated (i.e., multipole-to-multipole translation):

$$|\mathbf{x} - \mathbf{x}_{*2}| > |\mathbf{x}_i - \mathbf{x}_{*1}| + |\mathbf{x}_{*1} - \mathbf{x}_{*2}|,$$

$$\mathbf{B}(\mathbf{x}_i, \mathbf{x}_{*2}) = (S|S)(\mathbf{x}_{*2} - \mathbf{x}_{*1})\mathbf{B}(\mathbf{x}_i - \mathbf{x}_{*1});$$

- The multipole expansion coefficients can be  $S|R$ -translated (i.e., multipole-to-local translation):

$$|\mathbf{x} - \mathbf{x}_{*2}| < |\mathbf{x}_i - \mathbf{x}_{*1}| + |\mathbf{x}_{*1} - \mathbf{x}_{*2}|,$$

$$\mathbf{A}(\mathbf{x}_i, \mathbf{x}_{*2}) = (S|R)(\mathbf{x}_{*2} - \mathbf{x}_{*1})\mathbf{B}(\mathbf{x}_i - \mathbf{x}_{*1});$$

- We would like to compute the sum:

$$v_j = \sum_{i=1}^N u_i \Phi(\mathbf{y}_j, \mathbf{x}_i), \quad j = 1, \dots, M.$$

- Some generalizations are possible. For example, instead of  $\Phi(\mathbf{y}_j, \mathbf{x}_i)$  we can rewrite as  $\Phi_i(\mathbf{y}_j)$ , etc.

### 3.2 Expansion and Translation Equations for the Coulombic System

Below we will simply state the relevant results discussed in [1].

- Mother function

The so-called “mother function” is

$$\begin{aligned}\Phi_{ji} &= \Phi(\mathbf{y}_j, \mathbf{x}_i) \\ &= \log \sqrt{(\Re(\mathbf{y}_j) - \Re(\mathbf{x}_i))^2 + (\Im(\mathbf{y}_j) - \Im(\mathbf{x}_i))^2},\end{aligned}$$

where  $\Re$  and  $\Im$  stand for the real and imaginary parts of a complex number, respectively.

Note that the right-hand side of the equation above is equivalent to  $\Re(\log(\mathbf{y}_j - \mathbf{x}_i))$ , therefore, we can follow the standard practice and write the mother function as:

$$\Phi_{ji} = \log(\mathbf{y}_j - \mathbf{x}_i).$$

If we view  $\mathbf{y}_j - \mathbf{x}_i$  as a variable  $z$ , then  $\Phi$  is a function of  $z$  with a singularity at  $z = 0$ . The plot of this function is shown in Figure 3.1.

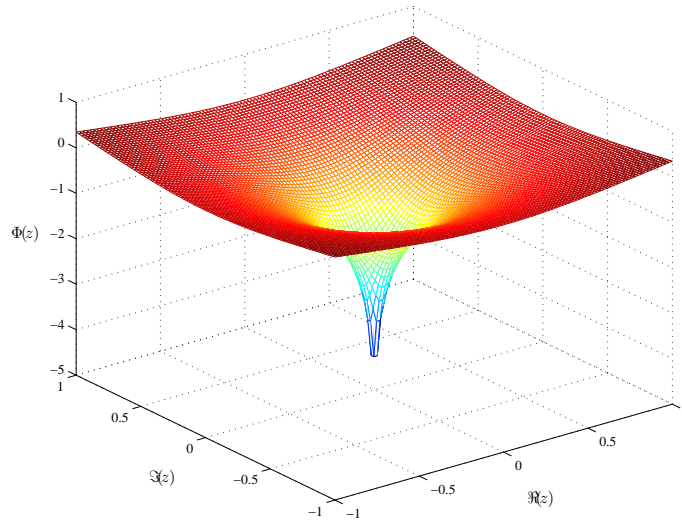


Figure 3.1: Plot of the  $\Phi$  function as a variable of  $z = \mathbf{y}_j - \mathbf{x}_i$

- S-Expansion

$\log(y - x_i) = \sum_{m=0}^{\infty} b_m \cdot S_m$  where

$$b_m = \begin{cases} 1 & \text{if } m = 0; \\ -\frac{(x_i - x_*)^m}{m} & \text{if } m \geq 1. \end{cases}$$

$$S_m = \begin{cases} \log(y - x_*) & \text{if } m = 0; \\ -\frac{1}{(y - x_*)^m} & \text{if } m \geq 1. \end{cases}$$

- R-Expansion

$\log(y - x_i) = \sum_{m=0}^{\infty} a_m \cdot R_m$  where

$$a_m = \begin{cases} \log(x_* - x_i) & \text{if } m = 0; \\ -\frac{1}{m(x_i - x_*)^m} & \text{if } m \geq 1. \end{cases}$$

$$R_m = \begin{cases} 1 & \text{if } m = 0; \\ (y - x_*)^m & \text{if } m \geq 1. \end{cases}$$

- S|S-Translation

$$b_0 = b_0,$$

$$b_l = \left( \sum_{k=1}^l b_k (-t)^{l-k} C_{l-1}^{k-1} \right) - \frac{b_0 (-t)^l}{l}.$$

Therefore, the  $p$ -truncated translation matrix,  $S|S$ , is as follows:

$$SS(t) = \begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ t & 1 & 0 & 0 & \cdots & 0 \\ -\frac{1}{2}t^2 & -t & 1 & 0 & \cdots & 0 \\ \frac{1}{3}t^3 & -t^2 & 2t & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \frac{1}{p-1}(-t)^{p-1} & -C_{p-2}^0 t^{p-2} & C_{p-2}^1 t^{p-3} & -C_{p-2}^2 t^{p-4} & \cdots & 1 \end{bmatrix};$$

- S|R-Translation

$$b_0 = \sum_{k=0}^{\infty} \frac{a_k}{t^k} + a_0 \log t,$$

$$b_l = \left( \frac{1}{(-t)^l} \sum_{k=1}^{\infty} \frac{a_k}{t^k} C_{l+k-1}^{k-1} \right) - \frac{a_0}{l(-t)^l}.$$

Therefore, the  $p$ -truncated translation matrix,  $S|R$ , is as follows:

$$SR(t) = \begin{bmatrix} \log t & \frac{1}{t} & \frac{1}{t^2} & \frac{1}{t^3} & \cdots & \frac{1}{t^{p-1}} \\ \frac{1}{t} & -\frac{1}{t^2} & -\frac{2}{t^3} & -\frac{3}{t^4} & \cdots & -\frac{p-1}{t^p} \\ -\frac{1}{2t^2} & \frac{1}{t^3} & \frac{3}{t^4} & \frac{6}{t^5} & \cdots & \frac{p(p-1)}{2t^{p+1}} \\ \frac{1}{3t^3} & -\frac{1}{t^4} & -\frac{4}{t^5} & -\frac{10}{t^6} & \cdots & \frac{(p+1)p(p-1)}{-6t^{p+2}} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ -\frac{1}{(p-1)(-t)^{p-1}} & \frac{C_{p-1}^0}{(-1)^{p-1}t^p} & \frac{C_p^1}{(-1)^{p-1}t^{p+1}} & \frac{C_{p+1}^2}{(-1)^{p-1}t^{p+2}} & \cdots & \frac{C_{2p-3}^{p-2}}{(-1)^{p-1}t^{2p-2}} \end{bmatrix};$$

- R|R-Translation

$$a_l = \sum_{k=l}^n a_k t^{k-l} C_k^l.$$

$$RR(t) = \begin{bmatrix} 1 & t & t^2 & t^3 & \cdots & t^{p-1} \\ 0 & 1 & 2t & 3t^2 & \cdots & (p-1)t^{p-2} \\ 0 & 0 & 1 & 3t & \cdots & \frac{(p-1)(p-2)}{2}t^{p-3} \\ 0 & 0 & 0 & 1 & \cdots & \frac{(p-1)(p-2)(p-3)}{6}t^{p-4} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & 1 \end{bmatrix}.$$

## Chapter 4

### Implementation

#### 4.1 Language Considerations

There are several reasons for Java to be the choice of programming language for this project.

- Java is ideal for 2-D animated visualization thanks to its built-in GUI functionalities ([19], [20]) and multi-threading capabilities.
- Applets written in Java can run on most major hardware and software platforms and can be downloaded and launched with only a Web browser, thus making this course project somewhat useful to other students.
- Java is a strongly object-oriented programming language, therefore it is very well suited for building complex data structures, especially when compared to Matlab.
- Java's polymorphism feature allows an abstract and fully extensible implementation of the potential function, so that code written for this project can serve as a basis for a future FMM application programming interface (API).



With the benefits also come the costs. The most apparent drawback in this case is that Java's automated and unpredictable memory garbage collection inevitably affects the performance of the algorithm, and also makes it more difficult to accurately measure and chart the CPU time it takes the program to solve problems of various sizes. Therefore, the rate of growth of computation time obtained this way may not be exactly inline with the theoretical prediction. In addition, Java's limited syntax does not allow operator-overloading. This has become a noticeable annoyance during the course of project development, as what could have been a single inline operator in C++ or Matlab has to be fully spelled out with method calls.

## 4.2 System Design

Figure 4.1 shows the main components of the system and their relationship between each other. The "FMM Blackbox" takes the following arguments:

- `int level`

The number of levels of space subdivision for the FMM algorithm. A unit cube with no division is at level 0. FMM requires a level of at least 2.

- `Point[] X`

The list of source points in the space.

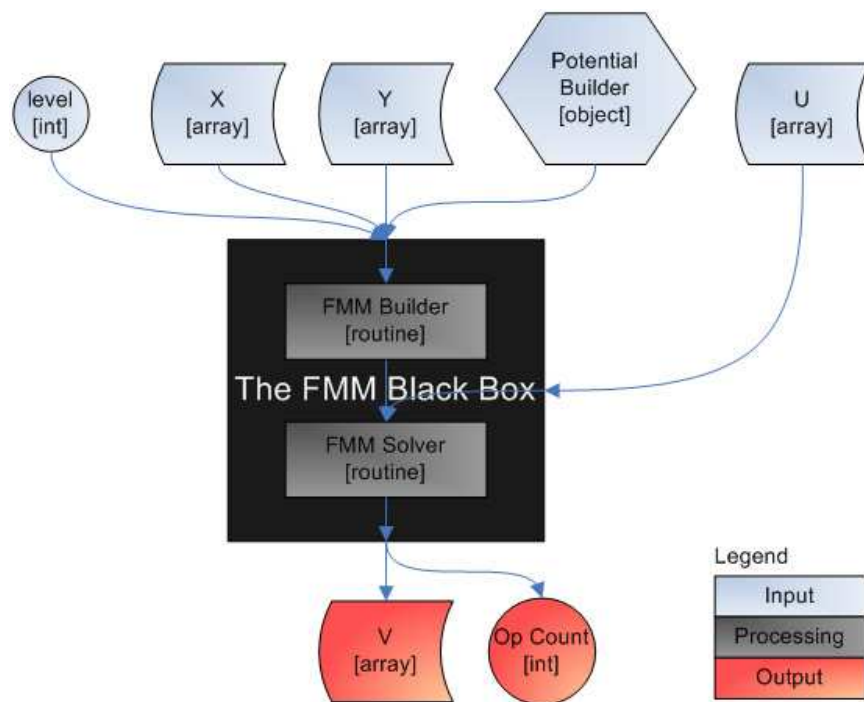


Figure 4.1: System Design

- `Point[] Y`

The list of target points in the space.

- `Object PotentialBuilder`

An object that exposes a certain set of methods. This will be explained in greater detail below.

- `double[] U`

The vector to which the matrix should be multiplied.

The blackbox returns the following output:

- `double[] V`

The desired matrix-vector product.

- `int opCount`

The number of operations used to solve this problem. In the current implementation, this is only an estimate.

#### 4.2.1 The PotentialBuilder Object

The `PotentialBuilder` object (Figure 4.2) is an abstraction of the mother function of the matrices that can be evaluated using FMM. All such functions share a common set of characteristics. Furthermore, the FMM algorithm does not require any additional information about the mother function other than requiring that it

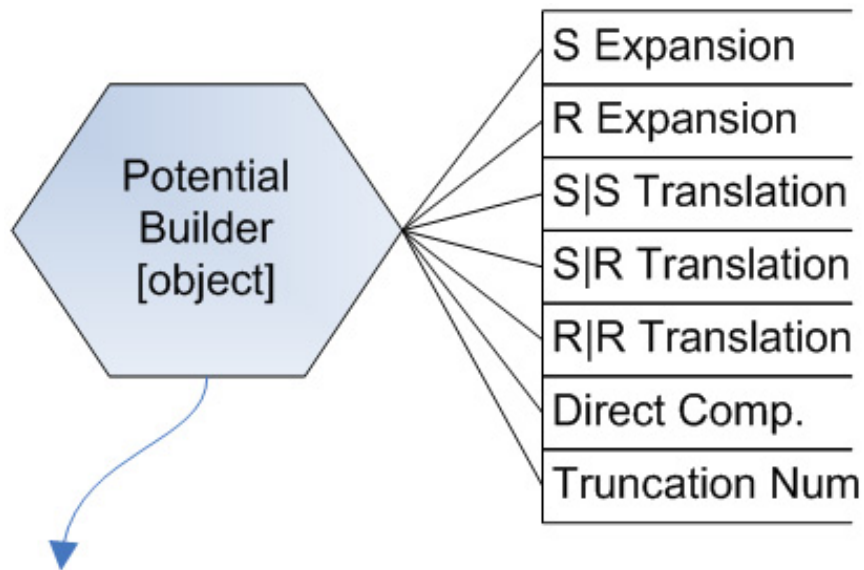


Figure 4.2: The Potential Builder Object

satisfy the set of characteristics. Therefore, using a Java interface that defines these “characteristics” as abstract methods will allow all Java classes implementing this `PotentialBuilder` interface to be used with the Java-FMM package.

#### 4.2.2 The FMM Blackbox

The FMM Blackbox can be divided into two parts: the FMM Builder and the FMM solver (Figure 4.3).

- The FMM Builder corresponds to the step in the FMM algorithm where the level of space subdivision is determined and the data structure is set. This step is only required once for every matrix.

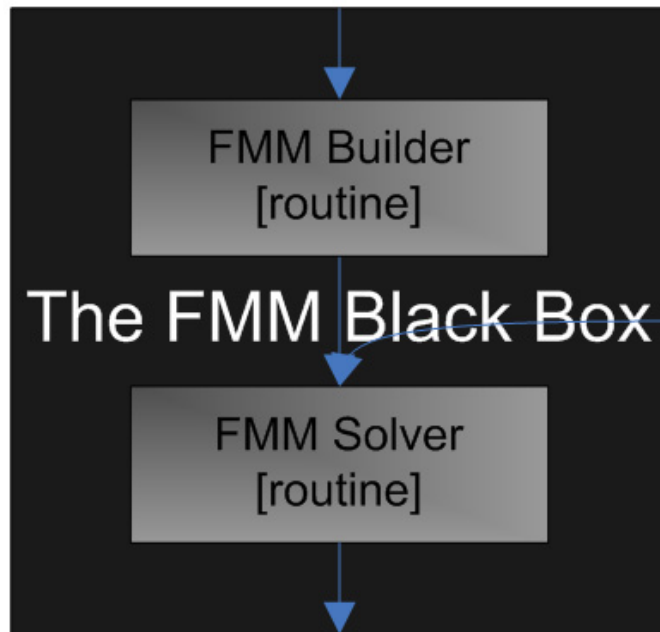


Figure 4.3: The FMM Blackbox

- The FMM Solver handles the upward and downward passes leading to the matrix-vector product, and is run every time a new vector ( $\mathbf{u}$ ) is sent as input.

In the actual Java-FMM implementation, the FMM Builder is encapsulated in the constructor of the `FmmTree` object. This is because the constructor is called only once during the lifespan of the object, in a way similar to the FMM Builder as an initialization step of the data structure is only used once in the FMM algorithm. The FMM Solver takes the form of the `solve()` method of the `FmmTree` class and can be called repeatedly as necessary.

### 4.2.3 Code Example

Below is a demonstration of how the Java-FMM package would be used in an actual program.

```
//where the Java-FMM package is
import edu.umiacs.fmm.*;

//Some code to create the appropriate numOfLevels,
//x, y and p variables
//...(omitted)

//Build FMM Data Structure
FmmTree t = new FmmTree(numOfLevels, x, y,
                        new Potential(p));

//Some code to generate the u vector here
//...(omitted)

//Solve the problem, using u as the input.
double[] fmmAns = t.solve(u);
```

```
//and count how many steps were used.  
long numOfOperations = t.getNumOps();
```

## Chapter 5

### Empirical Analysis

In the previous chapter, we mentioned that due to the choice of programming language, it is very difficult to obtain an accurate measure of the algorithm's running time. The data presented below therefore may include time spent by the Java Virtual Machine in memory garbage collection. However, since the real-world performance of a Java-FMM package will inevitably be affected by Java's built-in memory management as well, the experiments and comparisons presented below are valid approximations of how well the FMM algorithm will perform in practice.

All data obtained from these experiments are also presented in tables in Appendices, in addition to being plotted below.

#### 5.1 The Accuracy of FMM and the Effect of Truncation Number $p$

FMM is not an exact algorithm: It trades accuracy for efficiency. The error in the Fast Multipole Method is introduced by the truncation number  $p$  in the translation steps. Increasing  $p$  will reduce the truncation error incurred in the translation steps, but will also increase the running time of the algorithm due to the increased size of translation matrices, and vice versa.



Below is an experiment designed to study the effect of varying  $p$  on the accuracy and running time of FMM. For this experiment, we choose a problem size ( $N$ ) of 1000 and randomly generate the 1000 source ( $x$ ) and target ( $y$ ) points in the unit square. Next, we randomly generate 1000 charge potentials ( $u$ ). Finally, we vary the truncation number ( $p$ ) from 2 to 50, run the FMM solver with each, and record the following two data:

- Error. Error is computed by taking the largest element from the difference between the directly computed result and the FMM result, i.e., if  $\mathbf{v}_{direct}$  and  $\mathbf{v}_{fmm}$  are the two results obtained, then the error is computed as

$$\epsilon = \|\mathbf{v}_{direct} - \mathbf{v}_{fmm}\|_{\infty}.$$

Note that due to limitations of machine precision,  $\mathbf{v}_{direct}$  itself is an approximation of the true value. Therefore, the error  $\epsilon$  obtained this way is only an estimate of the true error.

- Time. Timer is set when the FMM routine starts with building the FMM data structure for each  $p$ . It continues through the FMM solver and is stopped when the FMM routine produces the result. The time spent on the error calculation above is not included. The system is forced to perform a garbage collection immediately before the timer is set.

The errors associated with various  $p$  values are plotted in Figure 5.1. In this

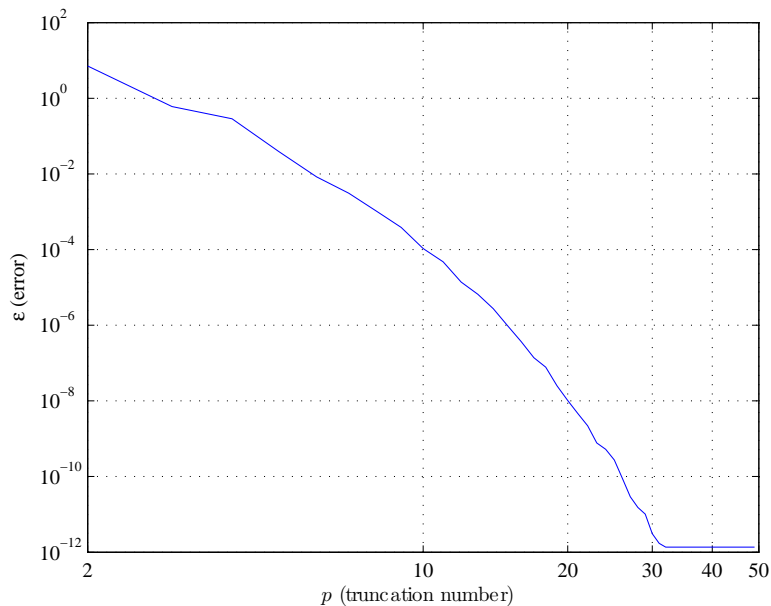


Figure 5.1: FMM accuracy with respect to various truncation numbers

log-log plot, we notice that the error decreases rapidly as  $p$  increases. For  $p > 31$ , the error is about constant at  $10^{-12}$ . This is due to the limitations of the machine precision and the gradual build-up of the loss of precision during computation.

For the same experiment, the times (in milliseconds) used by the FMM algorithm for each truncation number are plotted in Figure 5.2. We find that the computation time increases as we increase the  $p$  value. By comparison to Figure 5.1, we also find that while the error plunges from about  $10^1$  to  $10^{-12}$ , the computation time only increased from about  $10^3$  to under  $10^5$ . The gain in computation time is insignificant compared to the improvement in accuracy.

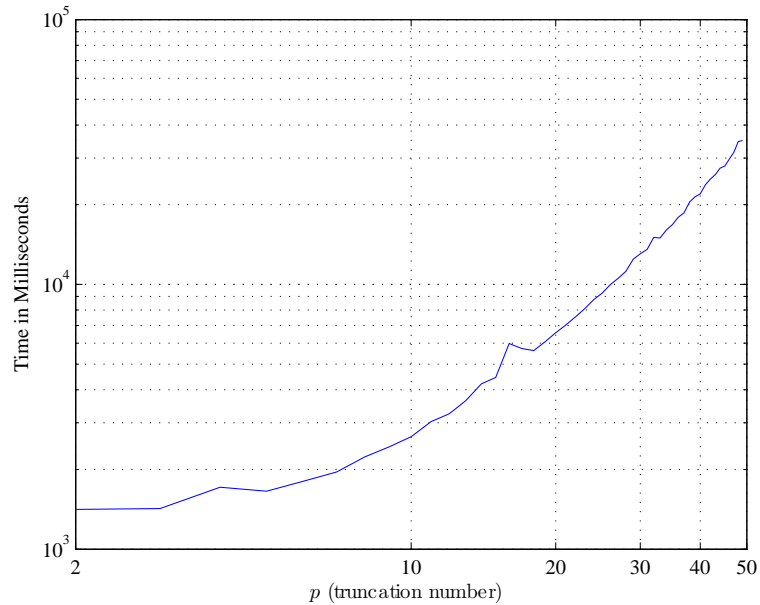


Figure 5.2: FMM running time with respect to various truncation numbers

## 5.2 The Accuracy of FMM and the Problem Size $N$

$N$ , the problem size, is fixed in the experiment above. A natural question to ask is how will the truncation number and the error vary when the problem size increases. For a pre-determined accuracy requirement, there should be an optimal truncation number such that it is large enough FMM will produce a result within this tolerance, but just large enough so that no computation cycles are wasted in gaining unnecessary accuracy.

In this experiment, we try to find empirically this optimal truncation number. The tolerance is set at  $10^{-5}$ . For each problem size  $N$ , we begin with a truncation

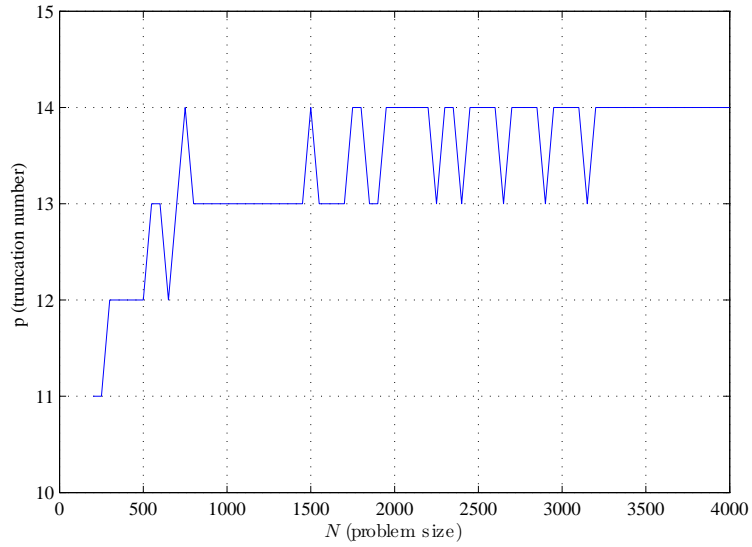


Figure 5.3: Minimum truncation numbers needed to achieve  $10^{-5}$  accuracy for various problem sizes.

number of 5 and gradually increase it until the computed error first falls in the tolerance. The result is plotted in Figure 5.3.

We see that the truncation number increases as the problem size increases, but at a much slower rate. From  $N = 500$  to  $N = 4000$ , the problem size grows 8-fold, but the truncation number required to maintain the same accuracy ( $10^{-5}$ ) increases only by 2.

### 5.3 The Time Complexity of FMM and Problem Size $N$

Having shown that FMM can be made to be arbitrarily accurate, subject to the limitations of machine precision, we turn to study the time complexity of FMM. We know that FMM will take longer to execute for smaller problems compared to the direct method, since FMM requires extra steps to set up the space partition and data structure. However, as the problem size  $N$  increases, FMM should compare more and more favorably to the direct method. We are interested in finding out the growth rate of both the FMM method and direct method, as well as the equilibrium point where the two methods take about the same amount of time to solve the problem. FMM will out-perform the direct method in problems with a size larger than this equilibrium.

In this experiment, the truncation number is fixed at  $p = 10$ .  $N$ , the problem size, varies between 500 and 4000 in increments of 50. For each value of  $N$ , a set of source and target points as well as the charged potentials are randomly generated. We then calculate the appropriate number of levels for space partition in order to optimize the FMM algorithm as shown in [7]. With these parameters, the FMM algorithm is performed for each  $N$ , after which the direct method is used to compute the result again. The times it takes to complete each computation are recorded and plotted in Figure 5.4.

In Figure 5.4, the solid lines are the actual running times for each method.

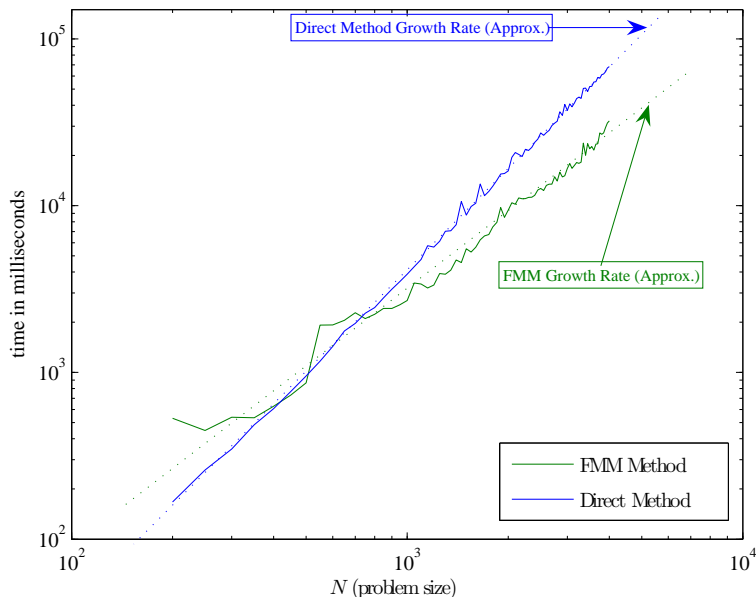


Figure 5.4: FMM running time with respect to various problem sizes

Due to the randomness in generating the data sets for each  $N$  as well as the variance in computer system performance, these lines are not as smooth as theoretical results would predict. However, the growth rate, i.e., complexity, of each method is clear, as noted in the plot by the dotted lines.

As one would expect, the dotted line representing the growth rate of the direct method has a slope of approximately 2 in this log-log plot, corresponding to the theoretical result of  $O(N^2)$  complexity. The slope (and thus the growth rate) of the FMM method is considerably lower and is consistent with the theoretical result of  $O(N \log N)$ .

The equilibrium point is around 800 in this experiment. For any  $N$  greater than this equilibrium, FMM offers an advantage in computation time. At  $N \approx 2000$ , the FMM method uses only half of what's needed by the direct method.

We also note that the FMM method appears to have more fluctuation in computation time than the direct method as  $N$  increases. This is likely due to the nature of space partitioning. Since the number of levels of partition is always an integer and is therefore discrete, the computation time will vary by a somewhat large amount when the number of levels needed by the FMM algorithm increases from  $L$  to  $L + 1$  as  $N$  increases.

## Chapter 6

### Visualizing the FMM Algorithm

The Fast Multipole Method is a complex algorithm and understanding it can be a daunting task to someone new to this algorithm. Being able to visualize and interactively explore how FMM performs space partition, expansion and translation at different levels will undoubtedly assist in the learning process. To that end, we developed the following application.

#### 6.1 The Animated Visualization

The software is implemented with the programming interface described in Chapter 4. It visualizes the process of solving the two-dimensional Coulombic problem outlined in Chapter 1, with the source and target points being generated randomly. Problem size ( $N$ ) and truncation number ( $p$ ) are specified by the user, with default values being 500 and 15, respectively.

The technical notes for acquiring and running the software are detailed in Appendix B. Once launched, the software consists of three main visual components (Figure 6.1): a toolbar with several buttons, a grid representing the unit square being partitioned, and a status panel. Upon initialization, the unit square is partitioned



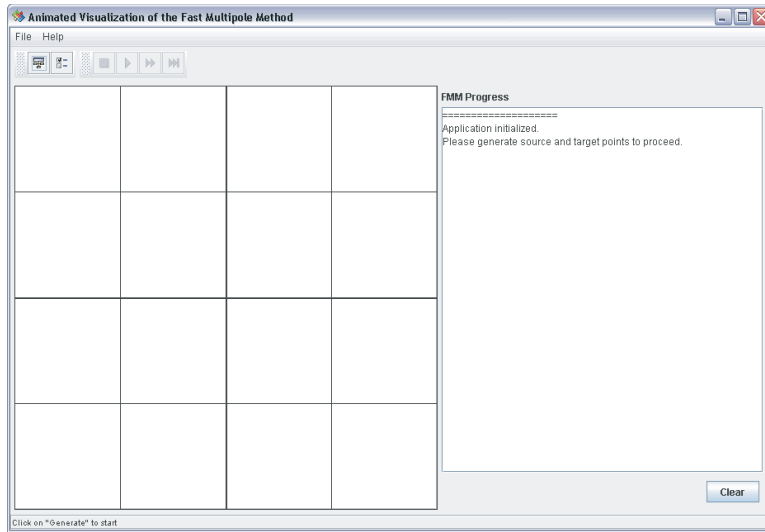


Figure 6.1: Application launched and initialized

with  $L = 2$ , since this is the minimum number of levels required by the FMM algorithm. Thicker borders divide boxes at a higher level (smaller  $l$  number), and as the level number increases, the dividing grid-lines become thinner. The status panel is cleared and prompts the user to click on the “generate” button to generate source and target points to use with the FMM algorithm.

### 6.1.1 The FMM Builder and Space Partitioning

The FMM builder is invoked once the “generate” button is clicked. It randomly generates the source and target points within the unit square and performs space partitioning accordingly. Once completed, the grid area will show the newly partitioned unit square, along with the source and target points, represented by red

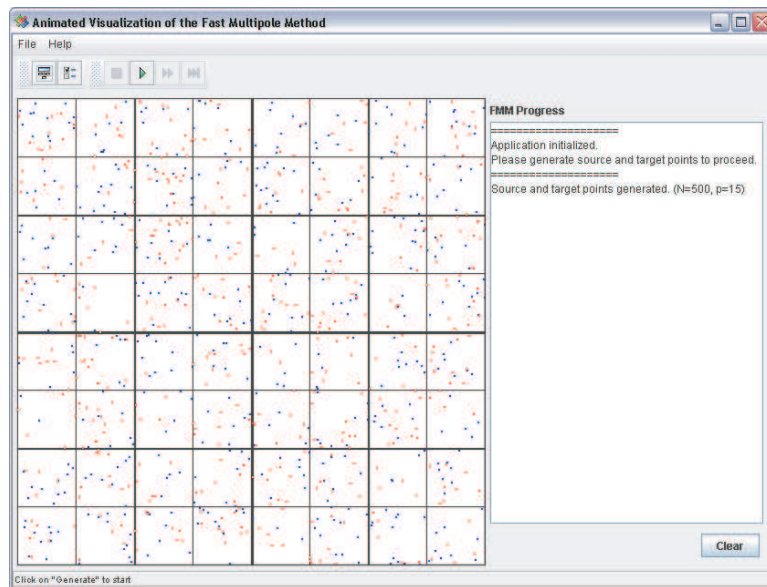


Figure 6.2: Source and target points generated and space partition performed accordingly

and blue dots scattered across the domain (Figure 6.2). At this point, the user can choose to either interact with the partitioned boxes, or start the animated FMM solver.

### 6.1.2 Interaction with Partitions

By clicking anywhere in the grid, a series of FMM boxes are highlighted. These are the boxes at different levels that contain the coordinates of this mouse click. Moving the mouse over these highlighted boxes will reveal their level number and index at that level (Figure 6.3).

Double-clicking inside a box or right-clicking then selecting from the pop-up

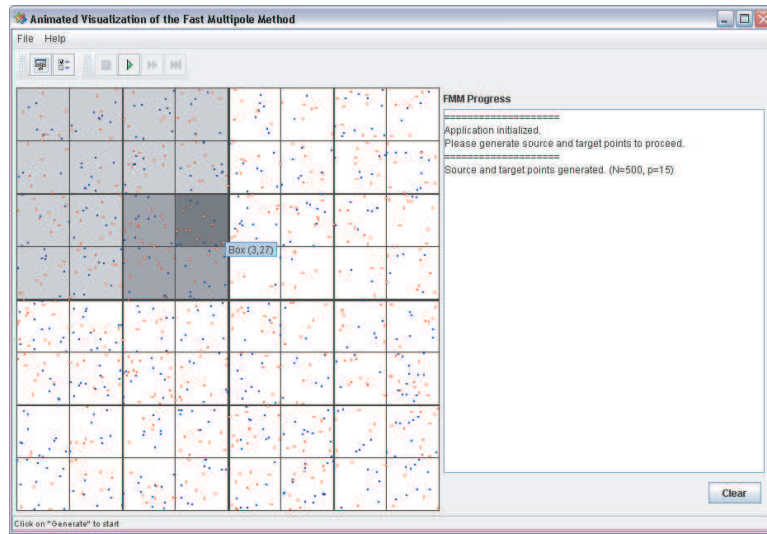


Figure 6.3: User clicks inside Box (3, 27).

menu brings up the properties window of that box. Major properties of interest to the FMM algorithm are displayed in a tree structure, including children, parent, neighbors,  $E_4$  neighborhood, as well as source and target points contained in this box. To aide visualization, clicking on the tree nodes will highlight the corresponding item in the box (Figure 6.4).

### 6.1.3 Animation of the FMM Solver

The animation step is started by clicking on the toolbar button “run”. Starting with the initial multipole expansions, the animation continues all the way through the final summation step while displaying the current action in the status panel to the right on the animation panel. You may pause, fast forward or stop the animation

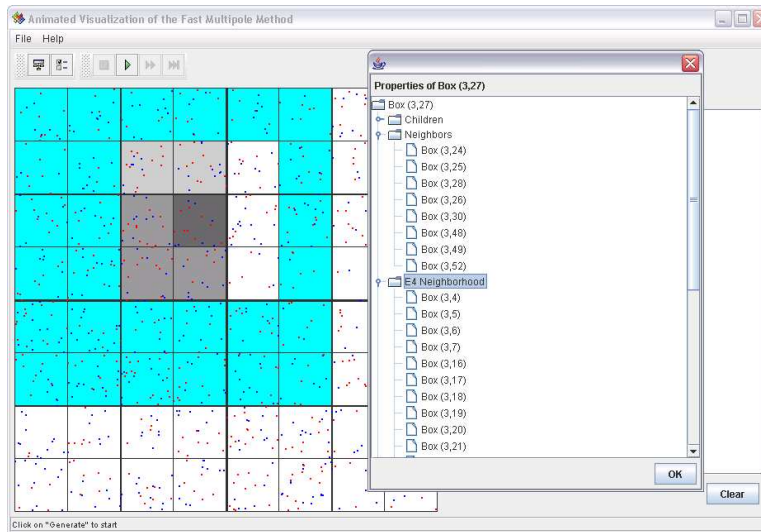


Figure 6.4: Displaying the properties of Box (3, 27). Its  $E_4$  neighborhood boxes are highlighted.

at any time before it ends.

When the animation stops, the FMM solver is completed. The computed error is then displayed in the status panel.

#### 6.1.4 Animation of the Adaptive FMM Algorithm

An adaptive version of the FMM animation software has also been developed. Figures 6.7 through 6.12 show various stages of the application.

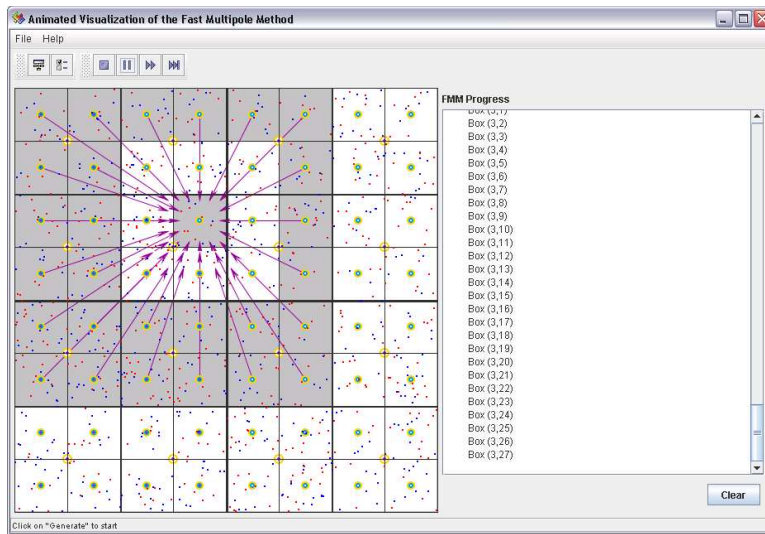


Figure 6.5: Animated FMM solver in progress, performing multipole-to-local translation for Box (3, 27).

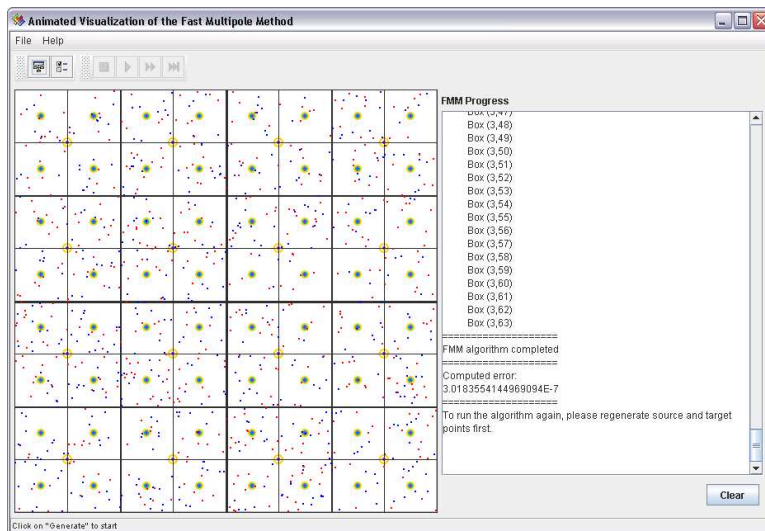


Figure 6.6: FMM solver completed. The computed error is displayed in the status panel.

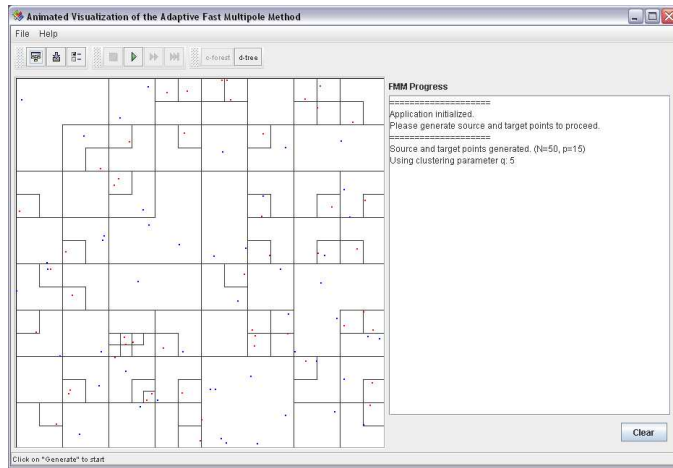


Figure 6.7: Space Partitioning of Adaptive FMM: C-Forest

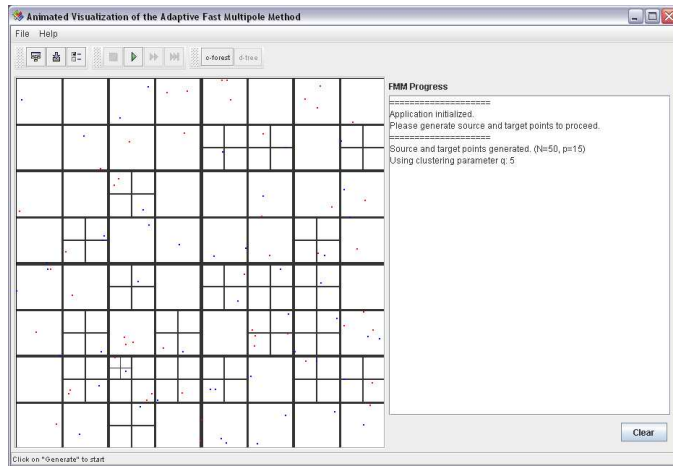


Figure 6.8: Space Partitioning of Adaptive FMM: D-Tree



Figure 6.9: Adaptive FMM Algorithm: Upward Pass

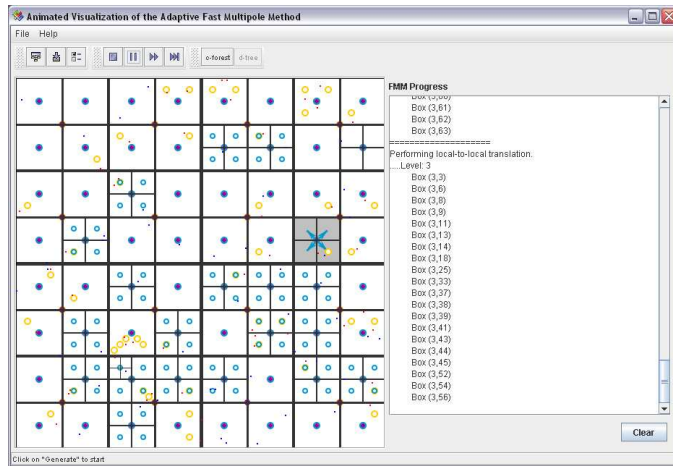


Figure 6.10: Adaptive FMM Algorithm: R—R Translation

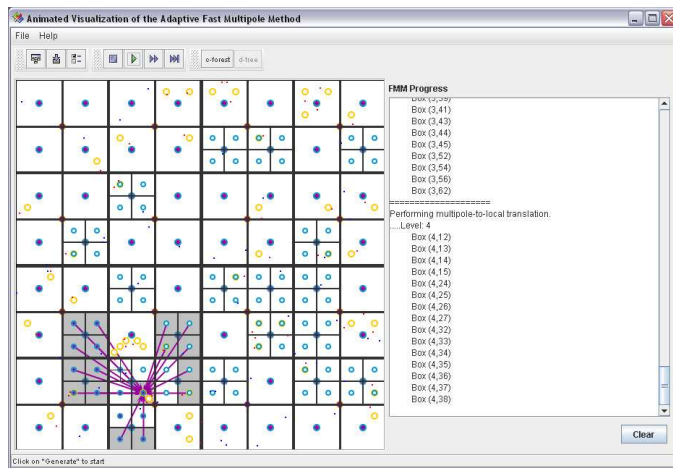


Figure 6.11: Adaptive FMM Algorithm: S—R Translation



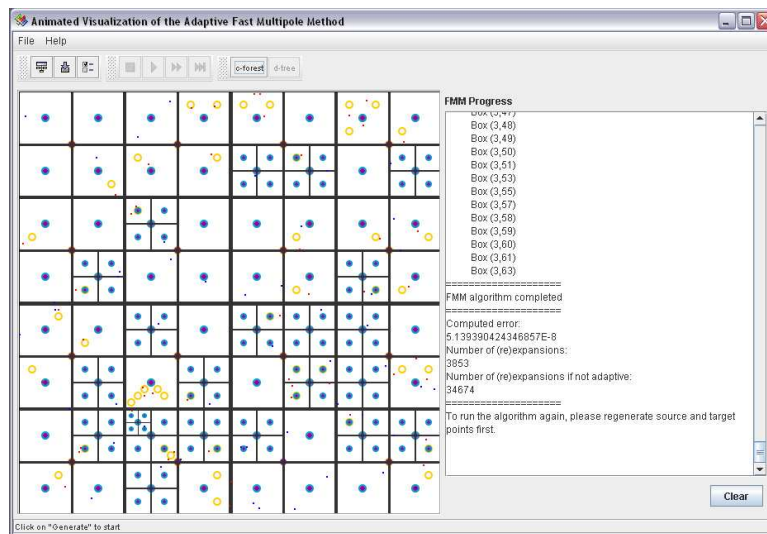


Figure 6.12: Adaptive FMM Algorithm is completed. The software reports the number of translations and re-expansions used in the adaptive algorithm as well as what would the non-adaptive algorithm have required. In this particular example, the adaptive version needed 3853 (re)expansions, whereas the non-adaptive version needed 34674.

## Appendix A

### Tables of Numerical Results

All experiments are performed on the same workstation with the following configurations:

- CPU: Pentium III 866Mhz
- RAM: 384MB
- Swap: 1GB
- HDD: Maxtor IDE 40GB / ATA100
- OS: Mandrake Linux 10 / Kernel 2.6
- JVM: Sun J2SE 1.5.0 (build 1.5.0-b64)

To ensure best results, all non-essential tasks on this workstation are either terminated or suspended before the tests. Test programs are invoked on the command console and output is redirected to file. The results of the numerical experiments are presented in the tables below.

Table A.1: FMM Error and Computing Time for Various  
Truncation Numbers

<b>Truncation Number</b> ( $p$ )	<b>Error</b> ( $\epsilon$ )	<b>Time</b> ( $t$ , in milliseconds)
2	7.142040526027E+00	1414
3	6.043577294839E-01	1425
4	2.876825796328E-01	1713
5	3.922249063550E-02	1657
6	8.343043238369E-03	1811
7	3.098313853513E-03	1957
8	1.035121598079E-03	2230
9	3.842355300776E-04	2437
10	1.065776338578E-04	2661
11	4.750127436637E-05	3033
12	1.389593575141E-05	3246
13	6.558698032677E-06	3640
14	2.730674736995E-06	4209
15	9.589101637175E-07	4452
16	3.706864504238E-07	5984
Continued on next page		

Table A.1 – continued from previous page

Truncation Number ( $p$ )	Error ( $\epsilon$ )	Time ( $t$ , in milliseconds)
17	1.380445837640E-07	5730
18	7.775400945320E-08	5625
19	2.478225269442E-08	6070
20	1.027478901960E-08	6573
21	4.609830739355E-09	7024
22	2.194781245635E-09	7550
23	7.699441084696E-10	8102
24	5.256310942059E-10	8751
25	2.732463144639E-10	9274
26	8.924416761147E-11	9983
27	2.921751729446E-11	10550
28	1.529087967356E-11	11200
29	1.017497197608E-11	12458
30	3.069544618484E-12	13047
31	1.705302565824E-12	13558
32	1.364242052659E-12	15031
33	1.364242052659E-12	14980
Continued on next page		

Table A.1 – continued from previous page

Truncation Number ( $p$ )	Error ( $\epsilon$ )	Time ( $t$ , in milliseconds)
34	1.364242052659E-12	16054
35	1.364242052659E-12	16824
36	1.364242052659E-12	17950
37	1.364242052659E-12	18617
38	1.364242052659E-12	20492
39	1.364242052659E-12	21397
40	1.364242052659E-12	21978
41	1.364242052659E-12	23790
42	1.364242052659E-12	24985
43	1.364242052659E-12	25997
44	1.364242052659E-12	27468
45	1.364242052659E-12	28000
46	1.364242052659E-12	29750
47	1.364242052659E-12	31538
48	1.364242052659E-12	34620
49	1.364242052659E-12	34929

Table A.2: Minimum truncation number required to achieve  $10^{-5}$  accuracy for various problem sizes

<b>Problem Size (<math>N</math>)</b>	<b>Truncation Number (<math>p</math>)</b>	<b>Error (<math>\epsilon</math>)</b>
200	11	7.041187330969E-06
250	11	7.022761707276E-06
300	12	8.927634837619E-06
350	12	9.605649154310E-06
400	12	6.352896605222E-06
450	12	7.953947601891E-06
500	12	9.396703944731E-06
550	13	3.366135246097E-06
600	13	5.787635871002E-06
650	12	8.341518551447E-06
700	13	6.307310741249E-06
750	14	4.432796686160E-06
800	13	2.432844325995E-06
850	13	3.595948214752E-06
900	13	8.110734142974E-06
Continued on next page		

Table A.2 – continued from previous page

Problem Size ( $N$ )	Truncation Number ( $p$ )	Error ( $\epsilon$ )
950	13	8.306316601647E-06
1000	13	6.143656037239E-06
1050	13	8.059167214469E-06
1100	13	5.035360231886E-06
1150	13	4.565287099467E-06
1200	13	4.374229092718E-06
1250	13	7.292289069483E-06
1300	13	5.577112233368E-06
1350	13	5.965986019874E-06
1400	13	9.325244263891E-06
1450	13	6.448535714298E-06
1500	14	2.431479515508E-06
1550	13	6.177602926982E-06
1600	13	9.912295695358E-06
1650	13	9.076994388124E-06
1700	13	9.722795653033E-06
1750	14	5.676148020939E-06
Continued on next page		

Table A.2 – continued from previous page

Problem Size ( $N$ )	Truncation Number ( $p$ )	Error ( $\epsilon$ )
1800	14	3.213633362975E-06
1850	13	5.808242121930E-06
1900	13	7.115891435205E-06
1950	14	4.110851818950E-06
2000	14	5.324059884515E-06
2050	14	2.670802246030E-06
2100	14	3.712859324878E-06
2150	14	3.936011466976E-06
2200	14	7.168588012973E-06
2250	13	9.417148930879E-06
2300	14	8.555599606552E-06
2350	14	5.282794745654E-06
2400	13	9.833391231950E-06
2450	14	8.782166787569E-06
2500	14	3.685536512421E-06
2550	14	8.090593610177E-06
2600	14	8.278871689527E-06
Continued on next page		



Table A.2 – continued from previous page

Problem Size ( $N$ )	Truncation Number ( $p$ )	Error ( $\epsilon$ )
2650	13	8.643616411064E-06
2700	14	7.967861392899E-06
2750	14	4.702105911747E-06
2800	14	5.751091748607E-06
2850	14	7.719208952039E-06
2900	13	9.443562362321E-06
2950	14	4.982841232959E-06
3000	14	8.393277084906E-06
3050	14	7.991714937816E-06
3100	14	4.433992444319E-06
3150	13	9.805245099415E-06
3200	14	6.417822760341E-06
3250	14	3.348814743731E-06
3300	14	4.433467893250E-06
3350	14	7.064818419167E-06
3400	14	4.113119757676E-06
3450	14	3.961659786000E-06
Continued on next page		

Table A.2 – continued from previous page

Problem Size ( $N$ )	Truncation Number ( $p$ )	Error ( $\epsilon$ )
3500	14	6.770501613573E-06
3550	14	8.304355787914E-06
3600	14	6.189411806190E-06
3650	14	6.308871434157E-06
3700	14	7.232324833240E-06
3750	14	7.109581702025E-06
3800	14	4.792581194124E-06
3850	14	7.337976057897E-06
3900	14	3.830982905129E-06
3950	14	5.345888894226E-06
4000	14	5.211456027610E-06

Table A.3: Comparison of computing time (in milliseconds) for FMM and direct methods for various problem sizes

<b>Problem Size (<math>N</math>)</b>	<b>Time (FMM) (<math>t_{fmm}</math>)</b>	<b>Time (Direct) (<math>t_{direct}</math>)</b>
200	531	167
250	449	260
300	539	347
350	535	485
400	627	607
450	732	770
500	866	952
550	1921	1169
600	1927	1431
650	2057	1773
700	2282	1971
750	2104	2265
800	2234	2447
850	2423	2798

Continued on next page

Table A.3 – continued from previous page

Problem Size ( $N$ )	Time (FMM) ( $t_{fmm}$ )	Time (Direct) ( $t_{direct}$ )
900	2422	3156
950	2545	3505
1000	2700	3879
1050	3441	4310
1100	3389	4757
1150	3211	5755
1200	3344	5651
1250	3918	6124
1300	3880	7033
1350	4110	7085
1400	4729	7685
1450	4559	10600
1500	5522	8774
1550	5276	9900
1600	5629	10336
1650	6245	13488
1700	6594	11480
Continued on next page		

Table A.3 – continued from previous page

Problem Size ( $N$ )	Time (FMM) ( $t_{fmm}$ )	Time (Direct) ( $t_{direct}$ )
1750	6714	12194
1800	7462	13091
1850	7950	14210
1900	9749	15491
1950	8515	15587
2000	9415	16152
2050	10433	19520
2100	10196	20849
2150	11145	20302
2200	10999	19714
2250	11022	21803
2300	11216	21436
2350	11240	22387
2400	11542	23979
2450	12409	25078
2500	12676	27346
2550	12337	26424

Continued on next page

Table A.3 – continued from previous page

Problem Size ( $N$ )	Time (FMM) ( $t_{fmm}$ )	Time (Direct) ( $t_{direct}$ )
2600	13256	27427
2650	13463	28283
2700	13468	30227
2750	14921	31242
2800	14065	32137
2850	15891	36537
2900	14887	34834
2950	15120	40476
3000	16764	37265
3050	17923	40764
3100	16736	39213
3150	17451	41334
3200	18163	43921
3250	17955	44777
3300	18231	43978
3350	23748	50526
3400	20043	50729
Continued on next page		

Table A.3 – continued from previous page

Problem Size ( $N$ )	Time (FMM) ( $t_{fmm}$ )	Time (Direct) ( $t_{direct}$ )
3450	23555	48413
3500	21946	51804
3550	22666	52015
3600	21538	55220
3650	23431	55358
3700	23522	58860
3750	27332	58067
3800	26828	61078
3850	27324	61723
3900	29380	63618
3950	31372	67074
4000	32281	68258

## Appendix B

### Technical Notes on the Software

#### B.1 License

The applet accompanying this thesis is released under the LGPL license (<http://www.opensource.org/licenses/lgpl-license.php>). By downloading, compiling, executing or editing the source code, binary code or file package of the FMM applet developed by the author, you agree to the LGPL license.

#### B.2 System Requirements

The software is compiled with JDK1.5.0 and requires a minimum of Java Runtime Environment (JRE) 1.5.0 to run. It cannot be compiled or run with lower versions of the JDK or JRE.

#### B.3 Obtaining and running the code

There are several ways to obtain and run the application.

- Java WebStart

For most installations of Java 1.5 and higher, the Java WebStart functionality



is enabled by default. Simply open <http://www.umiacs.umd.edu/~wpwy/fmm> in your Java-enabled browser to launch the application via Java WebStart.

- Java Archive File

If for some reason your WebStart does not execute the code (most likely due to local security manager settings), please follow the link given in

<http://www.umiacs.umd.edu/~wpwy/fmm>. You will be prompted to download a “jar” file. Once the download is complete, simply double click on the file or run

```
java -jar fmm.jar
```

to start the program.

- Source Code

Source code is available upon request.

## BIBLIOGRAPHY

- [1] L. Greengard and V. Rokhlin. A fast algorithm for particle simulations. *Journal of Computational Physics*, 73(2):325–348, 1987.
- [2] L. Greengard. *The Rapid Evaluation Of Potential Fields in Particle Systems*. MIT Press, Cambridge, Massachusetts, 1998.
- [3] Zhihui Tang. *Fast Transforms Based on Structured Matrices with Applications to the Fast Multipole Method*. PhD thesis, University of Maryland, College Park, 2003.
- [4] Nail A. Gumerov and Ramani Duraiswami. An introduction to the fast multipole method with applications. in preparation.
- [5] Ramani Duraiswami and Nail A. Gumerov. Lecture notes on the fast multipole method. for course AMSC698R at the University of Maryland, College Park.
- [6] J.J. Dongarra and F. Sullivan. The top 10 algorithms. *Computing in Science & Engineering*, 2:22–23, 2000.
- [7] Nail A. Gumerov, Ramani Duraiswami, and Eugene A. Borovikov. Data structures, optimal choice of parameters, and complexity results for generalized multilevel fast multipole methods in  $d$  dimensions. Technical report, University of Maryland Institute for Advanced Computer Studies, 2003.

- [8] Eric Darve. The fast multipole method I: Error analysis and asymptotic complexity. *SIAM Journal on Numerical Analysis*, 38(1):98–128, 2000.
- [9] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Comput. Surv.*, 16(2):187–260, 1984.
- [10] Nail A. Gumerov and Ramani Duraiswami. *Fast Multipole Methods for the Helmholtz Equation in Three Dimensions*. Elsevier Science, Oxford, 2005.
- [11] Eric W. Weisstein. "fast fourier transform." from mathworld—a wolfram web resource.
- [12] J. W. Cooley and O. W. Tukey. An algorithm for the machine calculation of complex fourier series. *Math. Comput.*, 19:297–301, 1965.
- [13] G. D. Gergkand. A guided tour of the fast fourier transform. *IEEE Spectrum*, 6:41–52, July 1969.
- [14] G. Strang. Wavelet transforms versus fourier transforms. *Bull. Amer. Math. Soc.*, 28:288–305, 1993.
- [15] Leslie Greengard and John Strain. The fast gauss transform. *SIAM J. Sci. Stat. Comput.*, 12(1):79–94, 1991.

- [16] A. Elgammal, R. Duraiswami, and L. Davis. Efficient nonparametric adaptive color modeling using fast gauss transform. In *Proc. IEEE Conf. Computer Vision and Pattern Recognition*, volume 2, pages 563–570, 2001.
- [17] Mark Broadie and Yusaku Yamamoto. Application of the fast gauss transform to option pricing. In *5th Columbia JAFEE Conference on Mathematics of Finance*, New York, 2002.
- [18] Leslie Greengard and John Strain. A fast algorithm for the evaluation of heat potentials. *Comm. Pure Appl. Math.*, 43:949–963, 1990.
- [19] Robert Eckstein, Mark Loy, and Dave Wood. *Java Swing*. O’Reilly & Associates, Inc., Sebastopol, California, 1998.
- [20] Jonathan Knudsen. *Java 2D Graphics*. O’Reilly & Associates, Inc., Sebastopol, California, 1999.