Theses and Dissertations

May 2019

Hermite Interpolation in the Treecode Algorithm

Benjamin St. Aubin
University of Wisconsin-Milwaukee

HERMITE INTERPOLATION IN THE TREECODE ALGORITHM

by

Benjamin J St. Aubin

A Thesis Submitted in
Partial Fulfillment of the
Requirements for the Degree of

Master of Science
in Mathematics

at

The University of Wisconsin-Milwaukee
May 2019

ABSTRACT

HERMITE INTERPOLATION IN
THE TREECODE ALGORITHM

by

Benjamin J St. Aubin

The University of Wisconsin-Milwaukee, 2019
Under the Supervision of Professor Lei Wang

In this thesis, a treecode implementing Hermite interpolation is constructed to approximate a summation of pairwise interactions on large data sets. Points are divided into a hierarchical tree structure and the interactions between points and well-separated clusters are approximated by interpolating the kernel function over the cluster. Performing the direct summation takes $O(N^2)$ time for system size N, and evidence is presented to show the method presented in this paper scales with $O(N \log N)$ time. Comparisons between this method and existing ones are made, highlighting the relative simplicity and adaptability of this process. Parallelization of the computational step is implemented by splitting the data set into pieces whose interactions are independently calculated on separate CPU cores. Additionally, steps are taken to make this approximation more efficient, allowing greater precision to be achieved without increasing completion time. Results are presented for the 3D $1/r$ and Screened Coulomb Potential e^{-kr}/r kernels on random data sets in size up to 10^7.

Thank You.

To Lei Wang, who graciously shared her time and knowledge of this topic with me,

To my committee members Istvan Lauko, Gabriella Pinter, and Dexuan Xie, who took time out of their busy schedules to provide me with feedback,

And

To my friends and colleagues, who provided encouragement and spell-checking.

<div style="text-align: right">Benjamin J St. Aubin</div>

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Introduction and Motivation

This chapter introduces the treecode algorithm, some comparable algorithms, and recent developments. The thesis is motivated by identifying a possible way to extend the functionality of treecode.

I.1   Treecode History and Development

The need for an algorithm like treecode arises when dealing with many physical systems. For example, in the classic N-body problem, the gravitational force on one body is calculated as a sum of pairwise interactions between the other $N-1$ bodies. Calculating the result of these interactions over the entire system requires $O(N^2)$ operations, which is prohibitively costly when dealing with systems on an astronomical scale. Examples of other problems that have similar behavior exist across the fields of physics (e.g. fluid dynamics, see Lindsay (1997)), computer science (e.g. machine learning as in Lee et al. (2012)), and mathematics (e.g. k nearest neighbor problem, see LeJeune et al. (2019)). Treecodes reduce the complexity of these systems by creating a hierarchical tree structure from the objects, so that many particle-particle interactions can be approximated by a single particle-cluster interaction.

One of the first popular uses of treecode is by Barnes and Hut (1986). This method simplified N-body calculations by approximating clusters of points as a single "heavy" point at the center of the cluster. If a point was deemed to be well-separated from a cluster, the effect of the cluster was approximated by a single calculation using the heavy point. For points that were not well-separated, the normal summation was used. This method reduces

the completion time for the N-body calculations from $O(N^2)$ to $O(N \log N)$ for distributions of particles that are roughly uniform. The Barnes and Hut algorithm is still in use due its generality and the ease in which it is applied. However, a significant error is introduced by approximating clusters by a single point at their centers. A brief but interesting look at the impact of error introduced by this method is given in Hernquist et al. (1993).

Seeking other ways to approximate the particle-cluster interactions led to an algorithm called the Fast Multipole Method. This method also utilizes a tree structure, but approximations are made by employing multipole expansions of the kernel function over each cluster of points. This method adds significant complexity, but in many applications can reduce $O(N^2)$ computations to $O(N)$. This method also offers the benefit of being able to specify a desired precision on the front end of computation. Unfortunately, FMMs are difficult to formulate and costly in terms of memory usage. Despite their unrivaled performance, simpler tree methods continue to be developed. For detailed descriptions of this method, see Greencard and Rokhlin (1987) and Beatson and Greengard (1997).

Another way to approximate the interactions between a target point and a well-separated cluster is to apply a kth order Taylor expansion of a kernel about the center of each cluster of points. Effectively, this produces a Taylor polynomial for each cluster that can be used to approximate the effect of the cluster on a target point. This method lowers the operation count to $O(N \log N)$ and can result in high-accuracy approximations. However, while the algorithm is less complex than the FMM in general, calculating the Taylor coefficients for each cluster can take significant time. To calculate these efficiently, it is often possible to derive recurrence relations between the coefficients. However, these derivations must be reworked for new kernel functions and can be difficult for complex kernels. For an example of treecode implemented in this way, see Deng and Driscoll (2012).

A simpler method, and the one examined in this thesis, is to approximate well-separated particle-cluster interactions by interpolating the kernel functions over the cluster at grid points. Similar to the Taylor method, this allows for pre-calculating and storing cluster

moments that may be used for all interactions involving the cluster. Lagrange interpolation has been applied to treecode in this way, and the main benefit over Taylor expansions is that the interpolation process can be generalized to work with a variety of kernel functions without modification; one need only supply the analytic formula for each new kernel. Unfortunately, the simplicity of this method comes at the cost of precision. To obtain a desired accuracy of approximation, a high-order interpolation is often required. This requires more grid points in each cluster and increases the amount of calculations required during approximation. This method also scales as $O(N \log N)$ but is generally outperformed in terms of completion time and precision by other standard methods. For a recent treecode implementation involving barycentric Lagrange interpolation, see Wang et al. (2019).

I.2   Thesis Goal

The goal of this thesis is to achieve a lower error of approximation for a given CPU time than is possible with Lagrange interpolation, while retaining much of the simplicity and adaptability from this method. A natural way to attempt this is by using Hermite interpolation. Since Hermite interpolation matches the value of a function and its derivative at each interpolation node, it induces less error than the Lagrange method for the same number of nodes. This thesis will adapt the treecode algorithm to use Hermite interpolation and examine the performance of this method. Additionally, we design the algorithm around CPU multiprocessing and perform some optimization of the treecode method around interpolation.

Hermite Interpolation

In the treecode algorithm, interpolation is carried out while calculating far field components to approximate a kernel function for well-separated point-cluster interactions. To investigate the precision and stability of Hermite interpolation in treecode, a variety of kernel functions were interpolated on a cluster of points and tested for accuracy and stability. The barycentric formulation of the Hermite interpolant was considered for usage but ultimately abandoned due to its negligible impact on stability.

II.1   Formulation

In one dimension, a function is approximated by a Hermite polynomial of degree 2n+1 in the following way:

Given values for a function f and its derivative f' at n+1 nodes $t_i, i = 0, 1, ..., n$, the Hermite interpolating polynomial as given by Burden et al. (2014) is

$$H(x) = \sum_{i=0}^{n}(1 - 2(x - t_i)l_i'(t_i))l_i^2(x)f(t_i) + \sum_{i=0}^{n}(x - t_i)l_i^2(x)f'(t_i)$$

where

$$l_i(x) = \prod_{j=0,j\neq i}^{n} \frac{x - t_j}{t_i - t_j}$$

is the ith Lagrange basis polynomial for $i = 0, 1, ..., n$ and

$$l_i'(t_i) = \sum_{j=0, j \neq i}^{n} \frac{1}{t_i - t_j}$$

is the first derivative of the ith Lagrange basis polynomial evaluated at t_i for $i = 0, 1, ..., n$. For simplicity, we write this as

$$H(x) = \sum_{i=0}^{n} f(t_i) h_i^{(1)}(x) + \sum_{i=0}^{n} f'(t_i) h_i^{(2)}(x)$$

for

$$h_i^{(1)}(x) = (1 - 2(x - t_i) l_i'(t_i)) l_i^2(x)$$

$$h_i^{(2)}(x) = (x - t_i) l_i^2(x)$$

To interpolate a function f in three dimensions, the interpolant takes the form

$$H(x_1, x_2, x_3) = \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{n} f(t_{1i}, t_{2j}, t_{3k}) h_i^{(1)}(x_1) h_j^{(1)}(x_2) h_k^{(1)}(x_3)$$

$$+ f_{x_1}'(t_{1i}, t_{2j}, t_{3k}) h_i^{(2)}(x_1) h_j^{(1)}(x_2) h_k^{(1)}(x_3) + f_{x_2}'(t_{1i}, t_{2j}, t_{3k}) h_i^{(1)}(x_1) h_j^{(2)}(x_2) h_k^{(1)}(x_3)$$

$$+ f_{x_3}'(t_{1i}, t_{2j}, t_{3k}) h_i^{(1)}(x_1) h_j^{(1)}(x_2) h_k^{(2)}(x_3) + f_{x_1, x_2}''(t_{1i}, t_{2j}, t_{3k}) h_i^{(2)}(x_1) h_j^{(2)}(x_2) h_k^{(1)}(x_3)$$

$$+ f_{x_1, x_3}''(t_{1i}, t_{2j}, t_{3k}) h_i^{(2)}(x_1) h_j^{(1)}(x_2) h_k^{(2)}(x_3) + f_{x_2, x_3}''(t_{1i}, t_{2j}, t_{3k}) h_i^{(1)}(x_1) h_j^{(2)}(x_2) h_k^{(2)}(x_3)$$

$$+ f_{x_1, x_2, x_3}'''(t_{1i}, t_{2j}, t_{3k}) h_i^{(2)}(x_1) h_j^{(2)}(x_2) h_k^{(2)}(x_3)$$

where f_{x_1}' is the partial derivative of f in the x_1 direction and t_{dn} is the component of the nth node in the x_d direction, with $h_m^{(1)}(x)$ and $h_m^{(2)}(x)$ as described above for $m = i, j, k$.

II.2   Testing Stability

The first consideration to make when interpolating a function is which method will be used to generate the interpolation nodes over a given interval. It is generally well-known that generating nodes uniformly in the desired interval leads to poor performance, as demonstrated in Figure II.1.
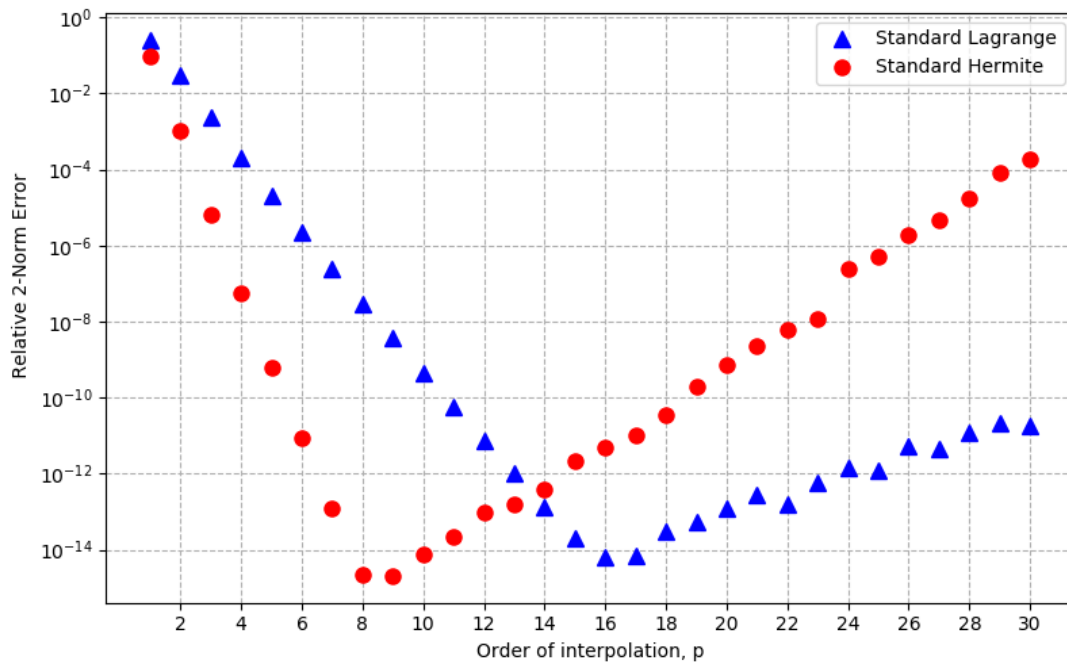
Figure II.1: Characteristic behavior of interpolating functions using uniform nodes. Error vs order p is compared for standard Hermite and Lagrange interpolants generated using uniform nodes on the interval [2,3] for the function $f(x) = 1/x$. Relative 2-norm error was calculated by comparing the value of function and interpolant at 100 evenly spaced points in the interval.

To create this figure, p nodes were generated in the interval $[2, 3]$ and the function $1/x$ was approximated according to the standard Hermite and Standard Lagrange methods. The relative 2-Norm error was calculated for the interpolant at the points $z = 2 : 0.01 : 3$, where

relative 2-Norm error between a function f and approximation function P is given by

$$\text{Error} = \left(\frac{\sum_{i=0}^{n} (f(z_i) - P(z_i))^2}{\sum_{i=0}^{n} f(z_i)^2} \right)^{1/2}$$

for test points $z_i, i = 1, 2, ..., n$

As the order of interpolation increases, approximation error falls toward machine epsilon. The error nears this point, but then begins to increase instead of stabilizing. Obviously, this is not a desirable behavior when approximating a given function, so a better method of selecting nodes must be chosen.

One such method is to generate nodes by calculating the roots of Chebyshev polynomials and mapping the results to the desired interval. Nodes generated in this manner are referred to as Chebyshev nodes, and they are well-known for reducing error in polynomial interpolation. The following simple formula from Burden et al. (2014) generates p nodes in the interval $[-1, 1]$.

$$c_k = \cos\left(\frac{2k-1}{2p}\pi\right) \text{ for each } k = 1, 2, .., p$$

To map these to an arbitrary interval $[a, b]$ perform the linear transformation

$$c_k = \frac{1}{2}(a + b) + \frac{1}{2}(b - a) \cos\left(\frac{2k-1}{2p}\pi\right), k = 1, 2, .., p$$

In the case of functions of two or more dimensions, these nodes are expanded into a tensor grid of the desired dimension. To demonstrate the effectiveness of Hermite interpolation with Chebyshev nodes, six functions with different properties were chosen to interpolate over the interval $[0.1, 1.1]$. Figure II.2 shows the result of this experiment.

These plots indicate that both Lagrange and Hermite interpolation show favorable stability properties when using Chebyshev nodes. Error steadily declines to machine epsilon in both cases and stabilizes as p is increased. As expected, approximating a function with the Hermite method induces less error for a given order of interpolation than Lagrange.
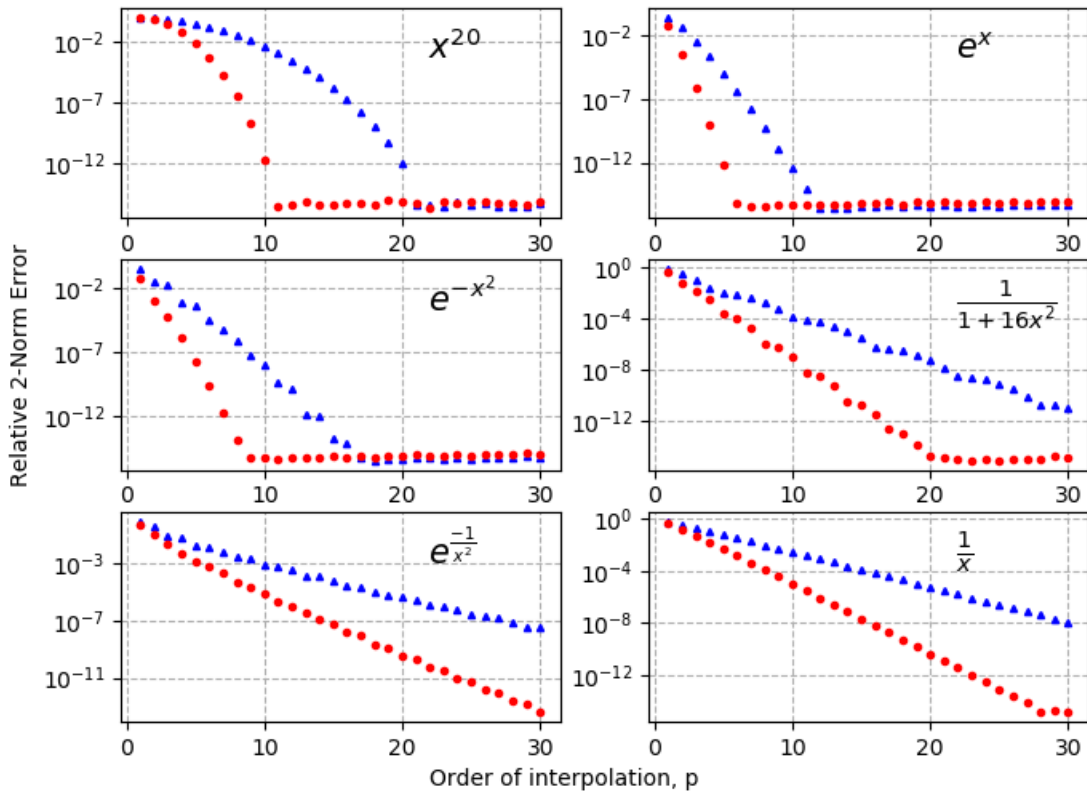
Figure II.2: Error of interpolating with standard Hermite (red circles) and standard Lagrange (blue triangles) for six functions on the interval [0.1,1.1] with order of interpolation varying from 1 to 30.

II.3   Barycentric Interpolation

In numerical applications, the standard formulations of the Hermite and Lagrange inter-polants are not often used due to concerns about stability. There are two main issues with the Lagrange basis polynomial, used in both Lagrange and Hermite interpolation. The first issue is that given two data points with the same x-coordinate but different values, a zero term appears in the denominator. The second issue is that as nodes get closer together, error increases due to values close to zero in the denominator. One way to address these issues is to use the barycentric formulation of the Lagrange and Hermite polynomials. As given by

Corless et al. (2008), the formula for the barycentric interpolant is

$$f(x) = \frac{\sum_{i=1}^{n} \sum_{j=0}^{s_i-1} \frac{\gamma_{i,j}}{(x-t_i)^{j+1}} \sum_{k=0}^{j} \frac{f^{(k)}(t_i)}{k!}}{\sum_{i=1}^{n} \sum_{j=0}^{s_i-1} \frac{\gamma_{i,j}}{(x-t_i)^{j+1}}}$$

where t_i for $i = 1, 2, ..., n$ are the nodes, $f^{(k)}(t_i)$ is the kth derivative of f evaluated at t_i, s_i is the nonnegative confluency of node t_i, and $\gamma_{i,j}$ are known as barycentric weights.

For this general form, the confluencies s_i enumerate the number of known values for the derivatives of f at node t_i. An $s_i = 1$ means only $f^{(0)}(t_i)$ is known, or just the value of the function at node t_i. This means that different amounts of information about f can be known at each node, but in practice we often consider all nodes to share the same confluency. Choosing $s_i = 1$ for all nodes gives the barycentric form of the Lagrange interpolant, while choosing $s_i = 2$ for each node corresponds to Hermite interpolation.

To create the interpolant, barycentric weights must be generated according to one of several possible methods. The method applied in this thesis is according to the one outlined in Corless and Fillion (2013). As it may be of interest, a short python implementation of a function to generate these weights is provided in Appendix A. To test the performance of the barycentric formulations, the six functions from section II.2 were interpolated using Chebyshev nodes. The results of one such test is shown in Figure II.3. Each of the experiments yielded similar results: error steadily decreased to machine epsilon, with corresponding error between the standard and barycentric formulations being very similar (but not identical). After the error neared machine epsilon, increasing p resulted in slightly less error for the barycentric versions.

These results make sense given the reasons the barycentric form is generally preferred. First, since we are generating the nodes according to the Chebyshev formula, we enforce a certain separation between nodes. Second, the data and weights are generated to floating point precision and the random generation of data ensures a data point will never exactly match a node. Together, these facts mean we avoid the issue of zeros and largely avoid the
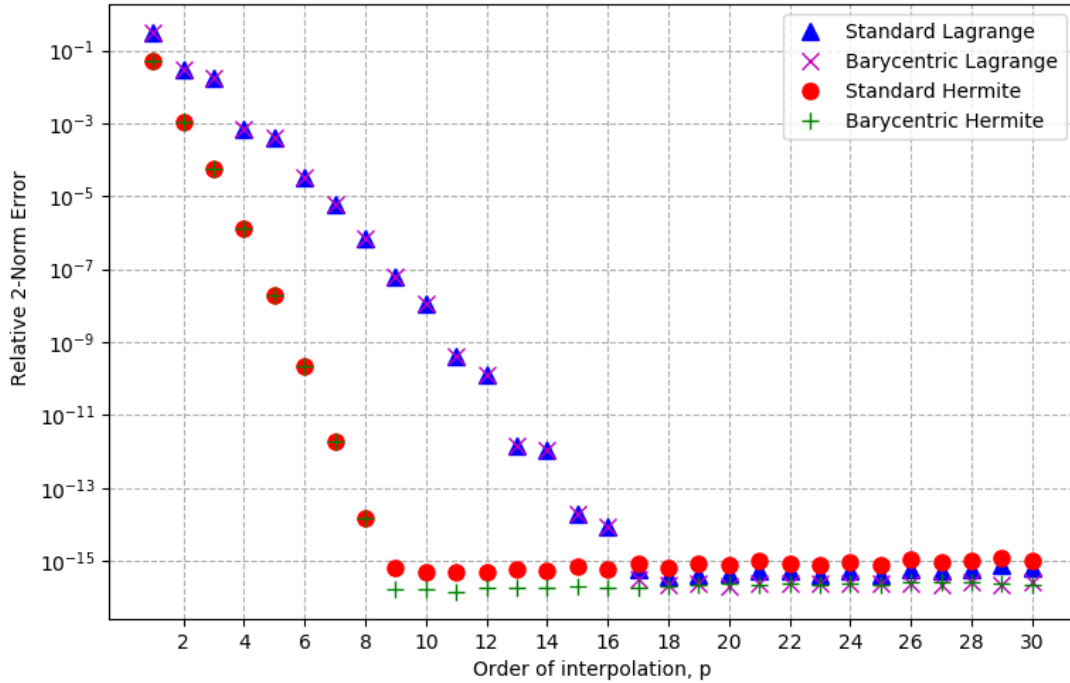
Figure II.3: A comparison of the behavior of the standard and barycentric forms of the Lagrange and Hermite interpolants of $f(x) = e^{-x^2}$ on the interval [0.1,1.1]. Interpolants were created with p Chebyshev nodes, and relative 2-Norm error was calculated at 100 evenly spaced points in the interval.

issue of very small values in the denominator of the Lagrange polynomial.

Since the small benefits to stability and precision from the barycentric formulation were largely outweighed by the added complexity to interpolation (and later far field expansions), the standard Hermite formulation was chosen for usage in this thesis.

Hermite Far Field Expansion

In this chapter we introduce the criterion for applying a far field expansion during treecode and derive the formula for a 1D and 3D Hermite far field expansion. We include data weights in this formulation due to their ubiquity in application. The precision of approximating a direct summation with far field expansion vs order of interpolation is tested for several simple kernel functions.

III.1   Particle-Cluster Separation Criterion

In treecode, we approximate the interaction between a target particle and a source cluster using a far field expansion if the target and source are well-separated. To determine if the target and source cluster are well-separated, the distance R between the target and center of source cluster is compared to the source cluster radius r. The ratio r/R is compared to a user-inputted variable θ, called the MAC number. If $r/R < \theta$, the target and cluster are considered well-separated and the interactions between them are approximated. Figure III.4 illustrates this process.

The selection of θ has a large impact on the speed and precision of treecode. On one extreme, if $\theta = 0$, r/R will always be greater than θ. In this case, direct summations are always performed and treecode offers no benefit over direct calculations. On the other hand, if $\theta = 1$ or higher, particles would be considered well-separated from every cluster they are not contained within. This would result in a fast treecode approximation with significant error. Generally, MAC values between 0.5 and 0.9 offer acceptable tradeoffs for time and
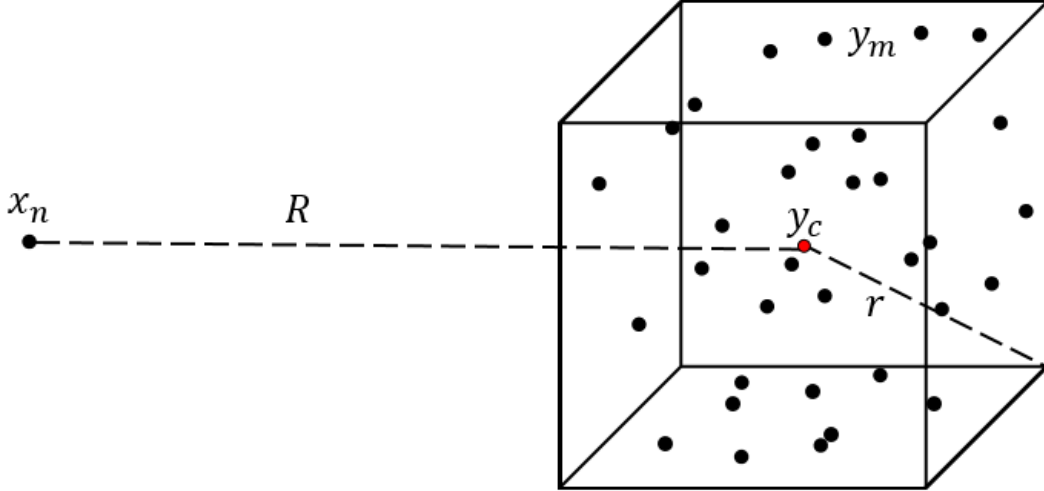
Figure III.4: Illustrates particle-cluster interaction between a target particle x_n and a cluster of particles y_m. The cluster center is y_c, cluster radius is r, and the particle-cluster distance is R. If the ratio $r/R < \theta$, where $\theta \in [0,1]$ is the user-defined MAC parameter, we consider target and cluster well-separated.

precision, but optimal choices for θ vary based on characteristics of the data set and other treecode parameters. We will revisit the selection of these parameters in a later section.

III.2 Far Field Formulation

Performing a far field expansion requires a node grid be generated for each cluster. We do this according to the Chebyshev formula described in section II.2, expanding the node grid to the desired dimension. To be relevant in many applications, we must also include data weights in this formulation. In application, weights may represent quantities from charge or mass to quadrature weights in numeric integration. To perform a far field expansion of a kernel function ϕ with target point x_i over a cluster of N points y_j, we begin with the formula for a direct summation and approximate the kernel function with a pth order Hermite interpolant. After this, we can change the order of summations to get the following result:

$$f(x_i) = \sum_{j=1}^{N} \lambda_j \phi(x_i, y_j)$$

$$\approx \sum_{j=1}^{N} \lambda_j H(x_i, y_j)$$

$$= \sum_{j=1}^{N} \lambda_j \left(\sum_{v=0}^{p} \phi(x_i, t_v) h_v^{(1)}(y_j) + \sum_{v=0}^{p} \phi'(x_i, t_v) h_v^{(2)}(y_j) \right)$$

$$= \sum_{j=1}^{N} \lambda_j \sum_{v=0}^{p} \phi(x_i, t_v) h_v^{(1)}(y_j) + \sum_{j=1}^{N} \lambda_j \sum_{v=0}^{p} \phi'(x_i, t_v) h_v^{(2)}(y_j)$$

$$= \sum_{v=0}^{p} \phi(x_i, t_v) \sum_{j=1}^{N} \lambda_j h_v^{(1)}(y_j) + \sum_{v=0}^{p} \phi'(x_i, t_v) \sum_{j=1}^{N} \lambda_j h_v^{(2)}(y_j)$$

$$= \sum_{v=0}^{p} \phi(x_i, t_v) M_{j1} + \sum_{v=0}^{p} \phi'(x_i, t_v) M_{j2}$$

for cluster moments

$$M_{j1} = \sum_{j=1}^{N} \lambda_j h_v^{(1)}(y_j)$$

$$M_{j2} = \sum_{j=1}^{N} \lambda_j h_v^{(2)}(y_j)$$

This process is much the same in three dimensions, with t_{dn} representing the component of the nth node in the x_d direction, target point $\boldsymbol{x}_n = < x_1, x_2, x_3 >$, and cluster of N points $\boldsymbol{y}_m = < y_{m1}, y_{m2}, y_{m3} >$ for $m = 1, 2, ...N$.

$$f(\boldsymbol{x}_n) = \sum_{m=1}^{N} \lambda_m \phi(\boldsymbol{x}_n, \boldsymbol{y}_m)$$

$$\approx \sum_{m=1}^{N} \lambda_m H(\boldsymbol{x}_n, \boldsymbol{y}_m)$$

$$= \sum_{m=1}^{N} \lambda_m \sum_{i=1}^{p} \sum_{j=1}^{p} \sum_{k=1}^{p} \phi(\boldsymbol{x}_n, t_{1i}, t_{2j}, t_{3k}) h_i^{(1)}(x_1) h_j^{(1)}(x_2) h_k^{(1)}(x_3)$$

13

$$+\phi'_{x_1}(\boldsymbol{x}_n, t_{1i}, t_{2j}, t_{3k})h_i^{(2)}(x_1)h_j^{(1)}(x_2)h_k^{(1)}(x_3) + \phi'_{x_2}(\boldsymbol{x}_n, t_{1i}, t_{2j}, t_{3k})h_i^{(1)}(x_1)h_j^{(2)}(x_2)h_k^{(1)}(x_3)$$

$$+\phi'_{x_3}(\boldsymbol{x}_n, t_{1i}, t_{2j}, t_{3k})h_i^{(1)}(x_1)h_j^{(1)}(x_2)h_k^{(2)}(x_3) + \phi''_{x_1,x_2}(\boldsymbol{x}_n, t_{1i}, t_{2j}, t_{3k})h_i^{(2)}(x_1)h_j^{(2)}(x_2)h_k^{(1)}(x_3)$$

$$+\phi''_{x_1,x_3}(\boldsymbol{x}_n, t_{1i}, t_{2j}, t_{3k})h_i^{(2)}(x_1)h_j^{(1)}(x_2)h_k^{(2)}(x_3) + \phi''_{x_2,x_3}(\boldsymbol{x}_n, t_{1i}, t_{2j}, t_{3k})h_i^{(1)}(x_1)h_j^{(2)}(x_2)h_k^{(2)}(x_3)$$

$$+\phi'''_{x_1,x_2,x_3}(\boldsymbol{x}_n, t_{1i}, t_{2j}, t_{3k})h_i^{(2)}(x_1)h_j^{(2)}(x_2)h_k^{(2)}(x_3)$$

Switching the order of summation and simplifying the expression leads to

$$f(\boldsymbol{x}_n) \approx \sum_{i=1}^p \sum_{j=1}^p \sum_{k=1}^p \phi(\boldsymbol{x}_n, t_{1i}, t_{2j}, t_{3k})M_{ijk}^{111} + \phi'_{x_1}(\boldsymbol{x}_n, t_{1i}, t_{2j}, t_{3k})M_{ijk}^{211}$$

$$+ \phi'_{x_2}(\boldsymbol{x}_n, t_{1i}, t_{2j}, t_{3k})M_{ijk}^{121} + \phi'_{x_3}(\boldsymbol{x}_n, t_{1i}, t_{2j}, t_{3k})M_{ijk}^{121}$$

$$+ \phi''_{x_1,x_2}(\boldsymbol{x}_n, t_{1i}, t_{2j}, t_{3k})M_{ijk}^{221} + \phi''_{x_1,x_3}(\boldsymbol{x}_n, t_{1i}, t_{2j}, t_{3k})M_{ijk}^{212}$$

$$+ \phi''_{x_2,x_3}(\boldsymbol{x}_n, t_{1i}, t_{2j}, t_{3k})M_{ijk}^{122} + \phi'''_{x_1,x_2,x_3}(\boldsymbol{x}_n, t_{1i}, t_{2j}, t_{3k})M_{ijk}^{222}$$

for cluster moments

$$M_{ijk}^{111} = \sum_{m=1}^N \lambda_m h_i^{(1)}(x_1)h_j^{(1)}(x_2)h_k^{(1)}(x_3) \qquad M_{ijk}^{211} = \sum_{m=1}^N \lambda_m h_i^{(2)}(x_1)h_j^{(1)}(x_2)h_k^{(1)}(x_3)$$

$$M_{ijk}^{121} = \sum_{m=1}^N \lambda_m h_i^{(1)}(x_1)h_j^{(2)}(x_2)h_k^{(1)}(x_3) \qquad M_{ijk}^{112} = \sum_{m=1}^N \lambda_m h_i^{(1)}(x_1)h_j^{(1)}(x_2)h_k^{(2)}(x_3)$$

$$M_{ijk}^{221} = \sum_{m=1}^N \lambda_m h_i^{(2)}(x_1)h_j^{(2)}(x_2)h_k^{(1)}(x_3) \qquad M_{ijk}^{212} = \sum_{m=1}^N \lambda_m h_i^{(2)}(x_1)h_j^{(1)}(x_2)h_k^{(2)}(x_3)$$

$$M_{ijk}^{122} = \sum_{m=1}^N \lambda_m h_i^{(1)}(x_1)h_j^{(2)}(x_2)h_k^{(2)}(x_3) \qquad M_{ijk}^{222} = \sum_{m=1}^N \lambda_m h_i^{(2)}(x_1)h_j^{(2)}(x_2)h_k^{(2)}(x_3)$$

Notice that in each case, the moments depend only on the weights, nodes, and cluster points. With regard to treecode, this means that after the points are divided into clusters, these moments can be calculated for each cluster and stored in the tree structure. Whenever an interaction between a target point and a cluster is to be approximated, all that needs to be calculated are the values of the kernel functions and derivative at the target point and

nodes.

III.3   Testing Kernel Functions

The same six 1D kernel functions tested in section II.2 were tested for error using a far field expansion. The difference in this case is that they are presented as functions of r, where r represents the distance between two input points.

A cluster of source points were randomly generated using a uniform distribution in the $[2, 3]$ interval. A target point placed at the origin represented a well-separated point and weights for the source points were randomly generated in the interval $[-1, 1]$. The direct summation was compared to far field estimations of increasing order in p, resulting in plots in Figure III.5. Each of these plots show decreasing error in p, falling more rapidly for the Hermite interpolation than Lagrange.

Additional testing was performed for the 2D and 3D cases, with a target at the origin and source points in a plane or cube. In every case, performing a Lagrange or Hermite far field expansion resulted in stable errors that decreased with increasing order of interpolation p. Additionally, a desired precision is always reached for a smaller p when using the Hermite method.
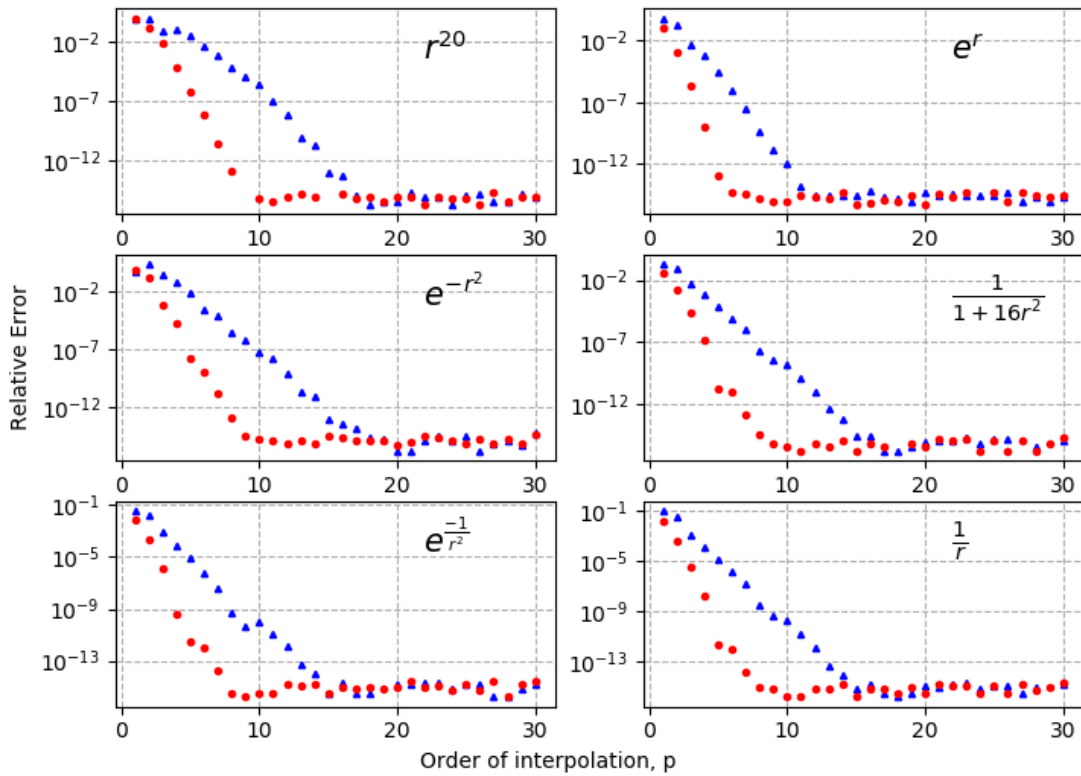
Figure III.5: Error of approximating particle-cluster interactions with standard Hermite (red circles) and standard Lagrange (blue triangles) for six functions. Target point is at 0, cluster is randomly generated in [2,3], and particle weights are random uniform in [-1,1].

Hermite Treecode Implementation

This chapter outlines the treecode algorithm with Hermite interpolation. Some modifications to the standard method of computing interactions are introduced and comparisons are made between the standard and modified performance. Additionally, the implementation of parallel processing is explained and the parallel efficiency of this algorithm is examined.

IV.1   Formation of the Tree Structure

The tree structure treecode gets its name from is formed starting with all points bounded inside an initial region of equal dimensions, called the root of the tree. A user-defined parameter n is compared to the number of points in this region. If the number of points in the root exceeds n it is split in half along each Cartesian dimension, resulting in 2^d children of level 1, where d is the number of dimensions. Each child is again compared to n and split in the same fashion, resulting in children of one level higher. Whenever a child has less than n points, it is called a leaf of the tree and is no longer split. During this process, information about each child, such as its center and radius, is calculated and stored to a data structure called a panel. The tree itself is a collection of all these panels. Figure IV.6 illustrates this process in two dimensions for $N = 2000$ nonuniform points with a max leaf size of $n = 25$. Panels of different level can be identified by their size, with the first image comprising the level 0 root of the tree. Each time a region is subdivided the level of its children increases by one. Once a panel has less than n points it is considered a leaf, but the leaves may be of any level 1 or higher. The last image in this figure shows all the leaves in the tree, but it is

important to remember that the tree structure contains all intermediate levels as well.
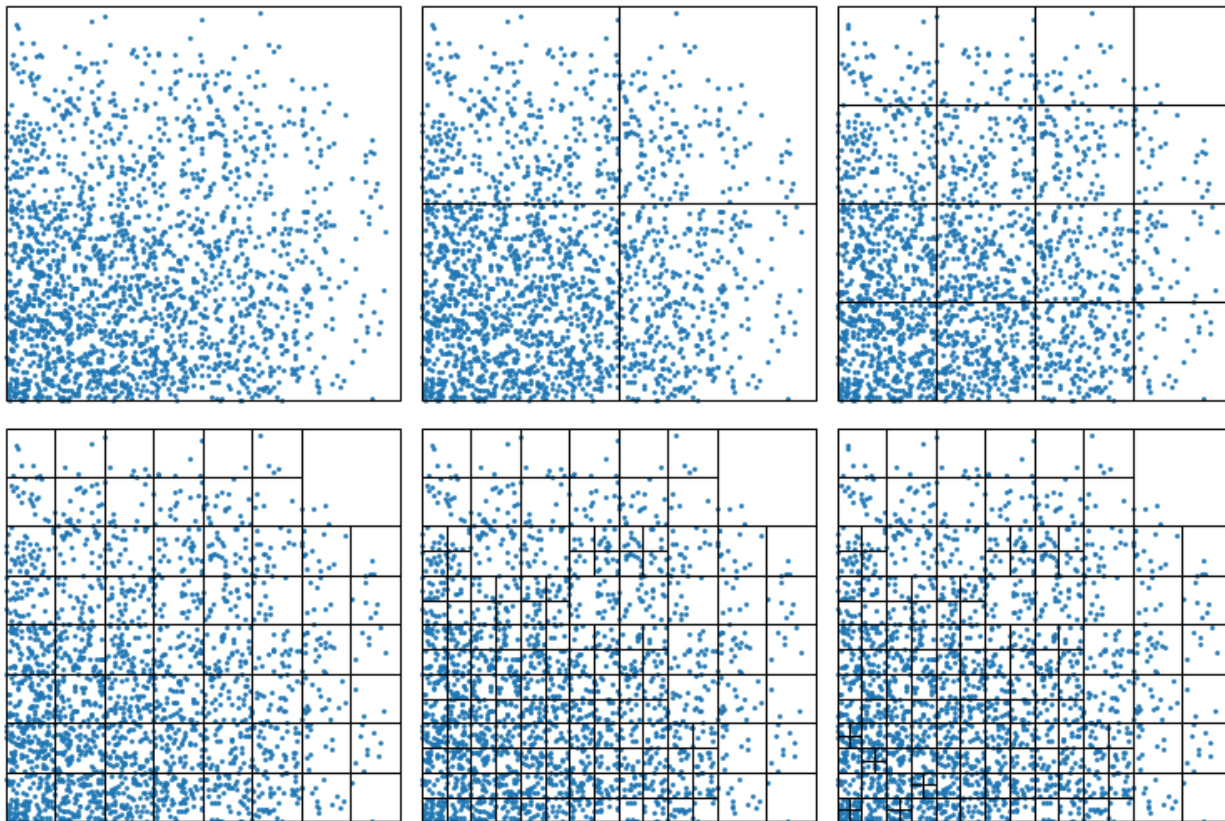
Figure IV.6: 2D tree structure creation process for $N = 2000$ nonuniform points with a max leaf size of $n = 25$. The first image represents the root cluster of the tree at level 0. At each stage, panels containing more than 25 points are split into four children of one level higher. The final image shows all leaves of the tree, or panels containing 25 points or less.

After the tree structure is formed, cluster moments are calculated for each panel. To do this, a Chebyshev node grid of the appropriate dimension is generated for each panel. Then, the cluster moments are calculated according to the procedure in section III.2. The nodes and moments are stored in the tree structure for each panel, to be accessed when approximating a particle-cluster interaction using a far field expansion. No nodes or moments are generated for the root cluster because no target point can be well-separated from the entire data set. This means that during the search for well-separated clusters, the tree will always be traversed to at least level 1 before a far field expansion is performed.

18

IV.2 Computing Interactions

Once the tree structure is complete, approximations for each point can begin. The beauty of treecode is that the entire process of calculating all interactions between particles now requires only a single function with a recursive call. For each target point in the data set, we call the function 'compute interaction' for the target and the root panel of the cluster. This function first applies the separation criterion from section III.1. If the target point is well-separated from the cluster a far field expansion is performed. Else, if the panel is a leaf a direct summation is performed. Else, the compute interaction function is called for the target and all children of the current panel. An outline of this process is provided in Algorithm 1.

Algorithm 1 Treecode
---
    input data, weights, MAC, p, n
    construct tree
    calculate cluster moments
    for x_i in data **do**
        Compute Interaction $(x_i, \text{Root Panel})$
    function COMPUTE INTERACTION (x_i, Panel)
        if target and cluster well-separated **then**
            return Far Field Expansion (x_i, Panel)
        else if Panel is leaf **then**
            return Direct Summation (x_i, Panel)
        else
            for children of Panel **do**
                Compute Interaction (x_i, Child)

Following this process for a particle involves traveling deeper into the tree structure, performing calculations at a cluster when it is well-separated or when a leaf is reached, until the entire tree has been traversed. Clusters that are farther away from the target point will be approximated by Hermite far field expansion while still relatively large, saving a great deal of computation time. When closer to the target point the function will have to go deeper into the tree to find well-separated clusters. Depending on θ, the size of the neighborhood

that will be directly calculated varies quite a bit, but interactions between the target point and the leaf it is contained within will always be directly summed.

After this process is completed an approximate error for the system can be calculated. Finding the true global error is usually not an option because it would require the exact results for each particle. Instead, we can perform the direct summation for a subset of the particles and approximate global error with the results. An estimated direct summation time for the entire system can also be acquired from this step by extrapolating from the time required to complete summations for the particle subset. Error and time estimates in this thesis were calculated by performing a direct summation for 1500 randomly selected data points.

IV.3   Modifications

Initial results of Hermite treecode revealed that in 3 dimensions the CPU time rose rapidly as the order of interpolation p was increased. An example of this behavior is shown in Figure IV.7 for the screened coulomb potential kernel on 10^6 random points in a cube. For each MAC number θ, completion time of Hermite treecode with small p remains below the time required for a direct summation. However, as p is increased to reduce error, the time required to complete the treecode approximation quickly overtakes the direct summation time. As seen in Figure IV.7, the effect is pronounced enough that achieving global errors less than 10^{-7} is not possible without taking longer than the exact calculations.

Obviously this is not suitable behavior for an approximation method, and without fixing it there would be no use for this treecode method. Revisiting the Hermite far field expansion formula revealed the reason this was occurring. For every particle-cluster interaction that is approximated by far field expansion, the value of kernel function and each of its derivatives must be evaluated for the target point and every node. This results in eight function evaluations for each of the p^3 nodes in three dimensions, or $O(8p^3)$ operations. However,

Figure IV.7: CPU time vs error for Hermite treecode over 10^5 points on the screened coulomb potential kernel. Max leaf size $n = 1000$, $\theta = 0.8$, and order of approximation $p = 1 : 1 : 10$ from right to left. This figure represents completion times that are unacceptable for an approximation method as they rapidly overtake the direct summation time.

performing a direct summation over a cluster of k points only results in $O(k)$ operations. Therefore, for any cluster with size k that satisfies $k < 8p^3$, performing the direct summation instead of a far field expansion should theoretically save time and be more precise. In the standard method of computing interactions, the first condition that is checked for each panel is if the target and cluster are well-separated. If instead we first check to see if the cluster contains less than $8p^3$ points, we get the modified subroutine described in Algorithm 2.

The results of implementing this change are shown in Figure IV.8. The figure shows the same data as in Figure IV.7, but with the addition of data obtained after the subroutine was modified. It can be seen that compared to the standard formulation, CPU time to complete the program levels out to something below the direct summation time instead of rapidly overtaking it. Additionally, the treecode error in the modified case is always less than that

---

Algorithm 2 Modified Compute Interaction

 1: **function** COMPUTE INTERACTION (x_i, Panel)
 2:     **if** panel size $< 8p^3$ **then**
 3:         **return** Direct Summation (x_i, Panel)
 4:     **else if** target and cluster well-separated **then**
 5:         **return** Far Field Expansion (x_i, Panel)
 6:     **else if** Panel is leaf **then**
 7:         **return** Direct Summation (x_i, Panel)
 8:     **else**
 9:         **for** children of Panel **do**
10:             Compute Interaction (x_i, Child)

---

of the standard formulation. Another benefit to this formulation is a decreased memory usage in the tree structure. Since direct summations will be calculated for any panel with less than $8p^3$ points, interpolation nodes and cluster moments need not be generated and stored in the tree structure for panels containing less than this amount of points. Nodes and moments are stored as floating-point arrays, so reducing the number of these objects in the tree saves a significant amount of memory. In practice, this leads to less overhead while running treecode and the ability to handle larger systems. With these modifications to the treecode algorithm it becomes a valid method to approximate pairwise interactions, and all further results presented in this thesis implement this modification.

One last use for Figure IV.8 is to help decide upon which treecode parameters may be optimal for data collection. Recall that n, p, and θ may all be adjusted, resulting in different precision and CPU time. If we decide to hold $n = 1000$, we can pick a desired error and find which θ and p result in the fastest time. Plotting time vs error to select treecode parameters in this way is common practice, as one need only look along the lower envelope of the plots to find an acceptable tradeoff between time and error. A target error of 10^{-7} was selected, leading to the choice of $\theta = 0.8$ and $p = 7$ for the remainder of this thesis.
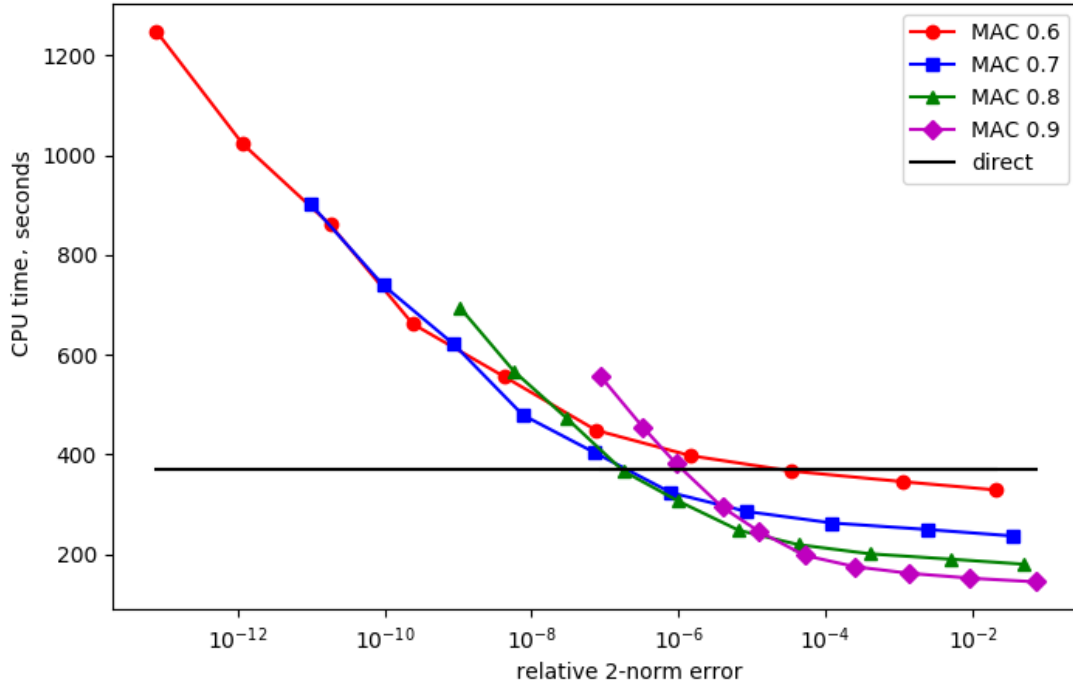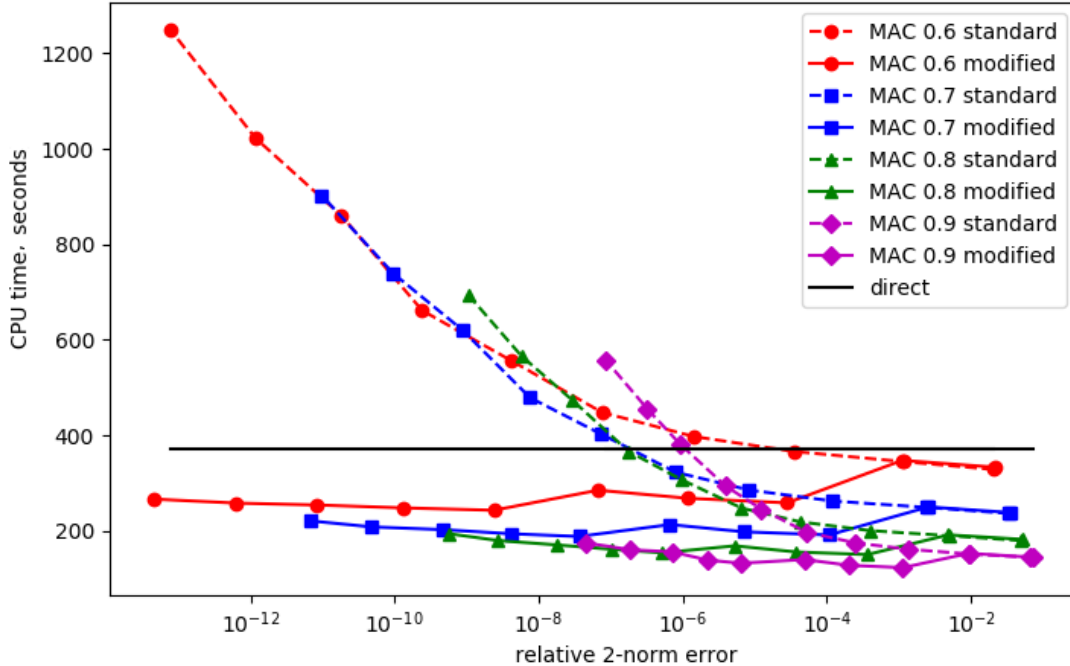
Figure IV.8: CPU time vs error for Hermite treecode over 10^5 points on the screened coulomb potential kernel. Max leaf size $n = 1000$, $\theta = 0.8$, and order of approximation $p = 1 : 1 : 10$ from right to left. After modifications to the compute interaction function as described in this section, treecode completion times stay below direct summation time.

IV.4 Parallelization

For each particle x_n, the compute-interaction phase is independent from every other particle. This allows for easy parallelization by splitting the data set into a desired number of pieces and starting a process for each of those to calculate all interactions. For this thesis, multiprocessing was handled by the **multiprocessing** python package. To accomplish treecode multiprocessing with c CPU cores, we divide the N data points into c chunks and pass the indices for each chunk to an intermediate function in each process. This function loads the data, weights, and tree structure into each process and performs the compute-interaction stage for its chunk of the data. After each process is complete, the results are assembled into a single array. This process is outlined in Algorithm 3.

Algorithm 3 Parallel Treecode

1: **in main process:**
2:      input data, weights, treecode parameters
3:      construct tree, save tree to file
4:      split particle indices and map to c processes
5: **in spawned processes**:
6:      load tree, data, weights
7:      compute interaction for particle chunk
8:      send results to main process

To test the parallel efficiency, both the treecode algorithm and the direct summation were performed utilizing 1-7 CPU cores. The points were randomly generated using a uniform distribution in a cube on the interval $[0, 1]$. For the treecode component, order 7 Hermite interpolation was used with MAC 0.8 and a max leaf size of 1000. This was completed on an AMD FX-8350 CPU running at a base clock of 4.0 GHz. The results for this experiment are shown in Table IV.1. Results indicate that in this formulation and on this machine, treecode has a higher parallel efficiency than performing a direct summation. As a result, the speedup from doing treecode over direct summation increases with the number of processes c.

c	d CPU (S)	d_1/d_c	d PE%	t CPU (S)	t_1/t_c	t PE%	d/t
1	977.5	1.00	100.0	676.5	1.00	100	1.45
2	656.7	1.49	74.4	346.5	1.95	97.6	1.89
3	522.7	1.87	62.3	264.0	2.56	85.4	1.98
4	441.2	2.22	55.4	212.4	3.18	79.6	2.08
5	388.3	2.52	50.3	184.2	3.67	73.4	2.11
6	354.4	2.76	46.0	163.7	4.13	68.9	2.16
7	340.3	2.87	41.0	149.4	4.52	64.7	2.28

Table IV.1:   Hermite Treecode parallel efficiency results for 10^5 Random points in a cube with Screened Coulomb Potential Kernel. Parameters are $n = 1000$, $\theta = 0.8$, p $= 7$, resulting in error 1.9e-7. Number of processes c, CPU time (d = direct, t = treecode), ratio of CPU time for one process and c processes ($d_1/d_c, t_1/t_c$), parallel efficiency (ratio/c), and treecode speedup d/t.

The relatively low parallel efficiency for the direct summation was unexpected. This may be in part due to the memory requirements of doing a direct summation over the treecode algorithm on my machine. To do a direct summation, calculations over the entire data set

must be performed for each target particle. In contrast, treecode works through the data set in clusters, only performing calculations on pieces of the data set when doing a far field expansion or direct summation on a given cluster. Additionally, spawning and initializing processes takes an amount of time that may not be negligible for a data set of this size. Hardware or software limitations on the test pc may also have played a part, but the cause of this issue was not investigated further. All further data in this thesis was collected utilizing 6 CPU cores.

Results

In this chapter results for Hermite Treecode are presented for 3D $1/r$ and e^{-kr}/r kernel functions. Random data was generated in two shapes: uniformly distributed in the $[0, 1]$ cube and uniformly distributed on the surface of a sphere with radius 0.5 centered at $(0.5, 0.5)$. Spherical data was created by generating arrays of three-dimensional points according to standard normal distributions. Normalizing these points to a desired radius results in a uniform distribution on the surface of a sphere. Data sets ranging in size from 10^4 to 10^7 were tested, with point weights randomly distributed on the interval $[-1, 1]$. All code for this thesis was written in python and is provided at the link in Appendix B.

V.1   Case 1: 1/r Kernel

Figure V.9 shows the results of Hermite treecode applied to the 3D $1/r$ kernel function. The max leaf size was chosen to remain constant at 1000, with $\theta = 0.8$ and order of interpolation $p = 7$ to target a global error near 10^-7. In the figure of CPU time vs N, the lower bound line represents $O(N \log N)$ time and the upper bound is $O(N^2)$. As expected, the direct summation time for both data sets is parallel to the $O(N^2)$ line, indicating the process scales as $O(N^2)$. The completion times for treecode vary somewhat, but do appear to level out to $O(N \log N)$ as the data sets increase. Regrettably, running treecode on larger data sets was not possible for this thesis due to time constraints. Treecode error is as expected for both data sets, with some slight increase for larger N. The low error for $N = 10,000$ is due to how the treecode algorithm behaves after the modification described in section IV.3. For $p = 7$,

a direct summation will be performed by default for clusters with less than $8*7^3 = 3584$ points. Due to the even distributions of points, every level 1 panel has less than this amount of points for the smallest data set and the entire tree will be directly calculated.
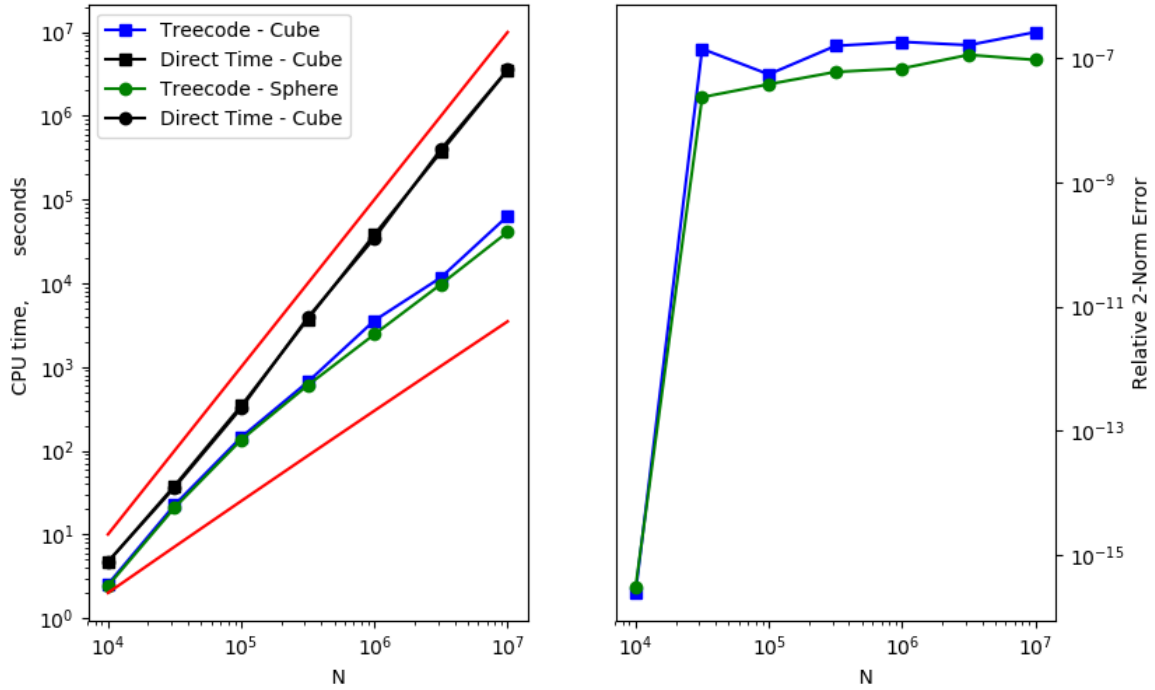

Figure V.9: Plots of CPU time and error vs N for points randomly generated in a cube on interval $[0,1]$ and on the surface of a sphere with radius $1/2$. Results are for $1/r$ kernel using order 7 Hermite Treecode with $\theta = 0.8$. Red lines indicate $O(N \log N)$ and $O(N^2)$ time.

V.2 Case 2: Screened Coulomb Potential Kernel

Figure V.10 shows results for Hermite treecode on the 3D screened coulomb potential kernel e^{-kr}/r. Parameters were max leaf size $n = 1000$, $\theta = 0.8$ and order of interpolation $p = 7$. The plot of CPU time vs N has lines representing $O(N \log N)$ and $O(N^2)$ time. Results were very similar to the $1/r$ function, indicating time treecode scaling that trends to $O(N \log N)$ for large N. Estimated error increases slightly with N, but remains near the target of 10^{-7}.
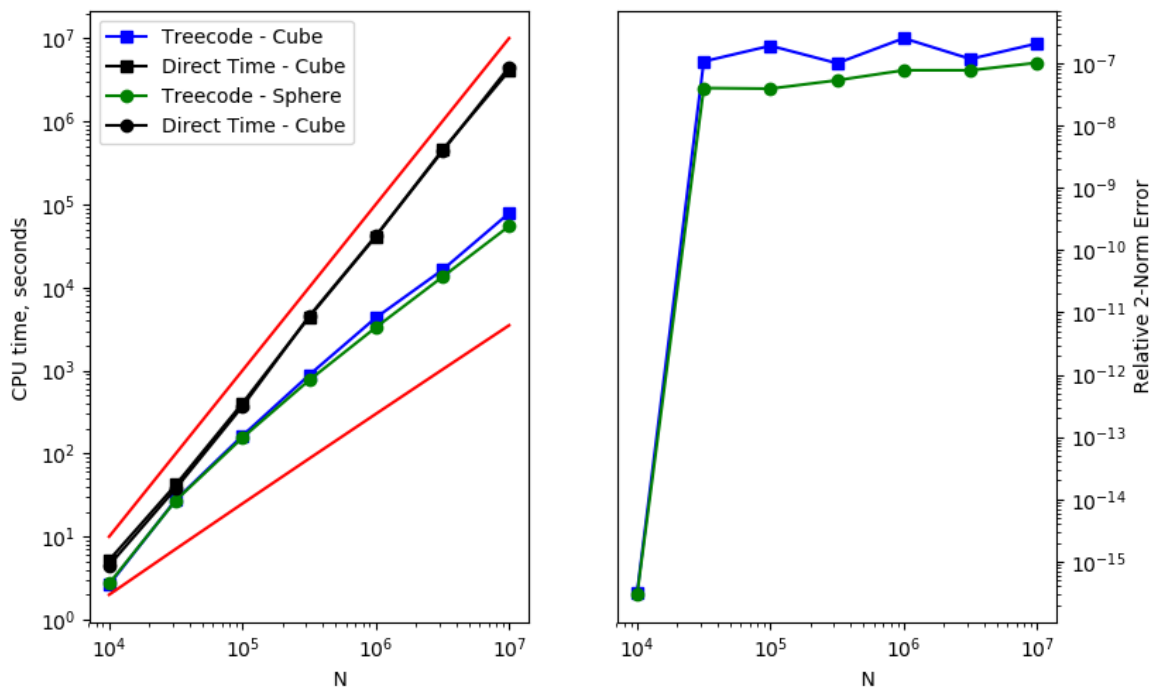
Figure V.10: Plots of CPU time and error vs N for points randomly generated in a cube on interval $[0, 1]$ and on the surface of a sphere with radius $1/2$. Results are for screened coulomb potential kernel using order 7 Hermite Treecode with $\theta = 0.8$. Red lines indicate $O(N \log N)$ and $O(N^2)$ time.

Summary and Outlook

In this chapter, the procedure and results of this thesis are summarized. Areas for further research are laid out, along with some comments on the potential utility for the method outlined in this paper.

VI.1 Summary of Results

The results of this thesis indicate Hermite interpolation offers another valid application of treecode to approximate systems of pairwise interactions. Hermite interpolation and far field expansions are numerically stable over a wide variety of kernel functions when using Chebyshev grids, and the Hermite treecode algorithm achieves stable error with completion time that appears to scale $O(N \log N)$ for system size N. Parallel processing is easily accomplished using the standard python multiprocessing package, achieving a significant reduction in completion time even on a standard home computer. More testing is needed to confirm these results for a wider variety of kernels and data shapes, but this method is easily applied to other kernels by providing formulas for partial derivatives. Other methods exist that have better time scaling (eg FMM), but the adaptability of this method and simple means of parallel processing may make it an attractive choice in some applications. Additional refinement of this process is likely possible, with faster runtimes to be expected in a compiled language vs its current python implementation.

VI.2   Outlook

There are several components of this method that remain to be investigated. It is difficult to judge the utility of this method without comparing its performance to existing ones, so testing should be performed in multiple applications. This process would also help investigate the effect of changing treecode parameters max leaf size n, MAC number θ, and approximation order p on data sets that are less uniform than those tested in this thesis. Additionally, applications often involve data sets that are highly nonuniform, and a thorough testing of this treecode should be performed over a wider range of data shapes.

The parallel processing component of this thesis was limited to utilizing multiple CPU cores in a single machine. Adapting this program to run on a cluster computer with several computing nodes would offer another way to increase computational speed. While the standard python multiprocessing package used in this thesis does allow for shared computing across multiple nodes, many other open-source packages exist that are designed especially for cluster-based computing. This would also allow for testing this algorithm on data sets that are much larger to better analyze parallel efficiency and memory usage.

Bibliography

Barnes, J. and Hut, P. (1986). A hierarchical o(n log n) force-calculation algorithm. *Nature*, 324:446–449.

Beatson, R. and Greengard, L. (1997). *A short course on fast multipole methods*, pages 1–37. Numerical Mathematics and Scientific Computation. Oxford University Press.

Burden, R., Faires, D., and Burden, A. (2014). *Numerical Analysis*. Cenege Learning, 10 edition.

Corless, R. and Fillion, N. (2013). *A Graduate Introduction to Numerical Analysis: From the Viewpoint of Backward Error Analysis*. Springer, 1 edition.

Corless, R., Shakoori, A., Aruliah, D., and Gonzalez-Vega, L. (2008). Barycentric hermite interpolants for event location in initial-value problems. *Journal of Numerical Analysis, Industrial and Applied Mathematics*, 3(1-2):1–16.

Deng, Q. and Driscoll, T. A. (2012). A fast treecode for multiquadratic interpolation with varying shape parameters. *SIAM*, 34(2):A1126–A1140.

Greencard, L. and Rokhlin, V. (1987). A fast algorithm for particle simulations. *Journal of Computational Physics*, 73:325–348.

Hernquist, L., Hut, P., and Makino, J. (1993). Discreteness noise versus force errors in n-body simulations. *The Astrophysical Journal*, 402:L85–L88.

Lee, D., Vuduc, R., and Gray, A. G. (2012). *A Distributed Kernel Summation Framework for General-Dimension Machine Learning*, pages 391–402. Wiley.

LeJeune, D., Baraniuk, R. G., and Heckel, R. (2019). Adaptive estimation for approximate k-nearest-neighbor computations. *CoRR*, abs/1902.09465.

Lindsay, K. (1997). *A Three-Dimensional Cartesian Tree-Code and Applications to Vortex Sheet Roll-Up*. PhD thesis, University of Michigan.

Wang, L., Krasny, R., and Tlupova, S. (2019). A kernel-independent treecode based on barycentric lagrange interpolation. Available at https://arxiv.org/abs/1902.02250".

Appendix

A Python Code for Barycentric Weights

```python
"""
This function calculates barycentric weights given the nodes
and integer confluencies s_in for each node. This is adapted
from "genbaryweights.m," a matlab file created by Robert M.
Corless. For an explanation of this algorithm, see Algorithm
2.4 in "A Graduate Introduction to Numerical Methods" by
Robert Corless and Nicolas Fillion

Ben St. Aubin 10/19/2018
"""
import numpy as np

def baryweights(nodes, s_in):
        n = len(nodes)
        s = s_in * np.ones(n)

        n_diff = (np.transpose(np.tile(nodes,(n,1))) -
                          np.tile(nodes,(n,1)) +
                          np.eye(n))

        u = np.zeros((n, s_in))
        nu = np.zeros((n, s_in))
        nu[:,0]=1
        node_diff_recip = n_diff**-1

        for m in range(1,s_in):
                u[:,m] = sum(np.dot(np.diag(s),
                                    node_diff_recip**(m)-np.eye(n)))
                for i in range(n):
                        nu[i,m] = (1/(m))*np.dot(u[i,0:m+1],
                                        np.transpose(np.flip(nu[i,0:m+1],0)))

        node_diff_recip = node_diff_recip**s_in

        w = np.flip(np.dot(np.diag(np.prod(node_diff_recip,
                axis = 1)),nu),1)

        return w
```

B Link to Code Repository

Much of the code created for this thesis is available at the link below. You are free to download, modify, and use this code for any purpose.

https://github.com/benjstaubin/Hermite-Treecode