Dongryeol Lee

New Algorithmic Techniques for Generalized N-body Problems

Senior Honors Research Thesis

CARNEGIE MELLON UNIVERSITY

Bachelor of Science in Computer Science

April 2005

Advised by **Alexander Gray** and **Andrew Moore**

Abstract

This research focuses on algorithmic analysis and implementations of state-of-the-art algorithms for generalized N-body problems.

Informally, in a N-body problem, we need to consider each pair (or n-tuple) of points formed from N points in a metric space. These problems arise in areas (computational statistics/physics, machine learning, database systems, computer graphics and computer vision) where the dataset has many points of high dimensionality. As a result, the brute-force algorithm with $O(N^n)$ scales poorly with the number of data points and the dimensionality. Fortunately, we can use the following techniques to reduce the computational time. First, adaptive partitioning via hierarchial data structures lets us process data points in "chunks." Secondly, we sometimes only require the computed answer to be within the user specified precision ϵ in many cases, so decomposition of the kernel function (as done in fast multipole methods) lets us speed up the computation.

Instead of surveying the entire class of generalized N-body problem, we will limit ourselves to "pair-wise" kernel functions ($n = 2$). The newly developed techniques in this thesis will focus on "pair-wise" N-body problems only.

The purpose of this thesis research is to develop a unified approach to these problems. A good solution would provide performance scalability with respect to the number of data points and the dimensionality of each point, and high adaptation to arbitrary data distribution.

As an example of newly developed techniques, I propose a new fast kernel density estimation algorithm combining two successful approaches in reducing the computational cost involved in nonparametric density estimation: a fully-recursive approach utilizing adaptive hierarchical data structures in computational geometry (dual-tree recursion) and an analytical approach in approximation theory (fast multipole methods). The technique developed here is general enough to be applied to other problems in which fast evaluations of "pair-wise" kernel functions are required. In demonstrating the effectiveness of the newly developed techniques, I will provide experimental results against current state-of-the-art algorithms.

Contents

List Of Figures

Chapter 1

Kernel Density Estimation (KDE)

Kernel Density Estimation is a popular technique used for nonparametric density estimation in which users make no assumption about the underlying distribution of the dataset. We are given a D-dimensional query dataset $X_Q = \{x_1, x_2, ..., x_{N_Q}\}$ of size N_Q and a D-dimensional reference dataset $X_R = \{x_1, x_2, ..., x_{N_R}\}$ of size N_R. KDE computes the density estimate $\hat{p}(x_q)$ at each $x_q \in X_Q$:

$$\hat{p}(x_q) = \frac{1}{N_R V_{Dh}} \sum_{r=1}^{N_R} K\left(x_q, x_r\right) \tag{1.1}$$

where the bandwidth h controls the degree of smoothing, normalizing constant V_{Dh} depends on the dimension and the bandwidth. The kernel functinon $K()$ is a function centered at each reference data point, and decreases mononically away from its center.

Theoretically, if $p(x)$ is the true density estimate function, increasing the size of the reference dataset will increase accuracy of the density estimate computed by KDE (and hence comes the demand for a fast KDE algorithm):

$$\lim_{N_R \to \infty} \int_{-\infty}^{\infty} |\hat{p}(x) - p(x)| dx = 0 \tag{1.2}$$

For our purposes, we will consider the "monochromatic" case in which the query dataset and the reference dataset are the same ($X = X_Q = X_R$, $N = N_Q = N_R$), although the algorithms analyzed in this thesis can handle the "bichromatic" case ($X_Q \neq X_R$). We will also choose the

Gaussian kernel $K_h(x_q, x_r) = e^{\frac{-||x_q - x_r||^2}{2h^2}}$, as it is one of the most popular kernels. One variant of kernel density estimation allows placeing different weights at each refrence point.

$$\hat{p}(x_q) = \frac{1}{N_R V_{Dh}} \sum_{r=1}^{N_R} w_r K\left(x_q, x_r\right) \tag{1.3}$$

The algorithms we discuss also handle these cases, but for simplicity we will only worry about $w_r = 1$ for $1 \leq r \leq N_R$. Nevertheless, it is the case that the techniques developed for kernel density estimation problem can be easy extended to allow different weights.

We also define the density estimate error ϵ for a query point x_q as the percentage deviation from the density estimate computed by the trivial naive algorithm. That is,

$$\epsilon = \frac{|\hat{p}_{alg}(x_q) - \hat{p}_{naive}(x_q)|}{\hat{p}_{naive}(x_q)} \tag{1.4}$$

Current state-of-the-art KDE algorithms use the following three main *techniques* for achieving fast speed.

1. Exact algebraic computataion

2. Decomposition of kernel function into a Taylor/multipole series

3. Fast adaptive node-to-node comparison via computational geometry

But more importantly, we are interested in designing fast algorithms that allow users to bound ϵ freely, thus bounding the error level for the density estimates for all query points. Note that this is a stronger defintion of error tolerance than one used in [4, 9, 12, 14, 5] in which users bound the actual deviation from the density estimate computed by the trivial naive algorithm is measured.

1.1 Trivial Naive Algorithm

For clarity, we present the naive algorithm for computing density estimate using KDE technique. Though effective on a small dataset and requiring no additional storage space, its time complexity is $O(DN^2)$. Note that this algorithm is simple to code for any arbitrary kernel function.

```
NaiveKDE($X$)
  for each point $x_q \in X$
    $\hat{p}(x_q) = 0$
    for each point $x_r \in X,\ x_r \neq x_q$
      $\hat{p}(x_q) += K_h(x_q,\ x_r)$
    end
    $\hat{p}(x_q) /= (N \cdot V_{DH})$
  end
```

Figure 1.1: A method using two nested loops.

1.2 Multidimensional Fast Fourier Transform

Fast Fourier Transform is often quoted as the solution for the computational cost in KDE. Kernel density estimation using FFT is described in [22] and [27]. [22] discusses the implementation of KDE only in a univariate case, while [27] extends Silverman's algorithm to handle more than one dimension.

1.2.1 Data Structure

We first compute the $M_1 \times \cdots \times M_D$ matrix by binning the data assigning the raw data to neighboring grid points using one of the binning rules. This involves computing the minimum and maximum coordinate values $(g_{i,M_i}, g_{i,1})$, and the grid width $\delta_i = \frac{g_{i,M_i} - g_{i,1}}{M_i - 1}$ for each i-th dimension. This essentially divides each dimension into $M - 1$ intervals of equal length.

In particular, [27] discusses two different types of binning rules - linear binning, which is recommended by Silverman, and nearest-neighbor binning. [27] states that nearest-neighbor binning rule performs poorly, so this thesis will test the implementation using the linear binning rule, as recommended by both authors.

In addition, we compute the $L_1 \times \cdots \times L_d$ kernel weight matrix, where $L_i = \min(\left\lfloor \frac{\tau h}{\delta_i} \right\rfloor, M_i - 1), with \tau \approx 4$ and $K_l = \prod_{k=1}^{d} e^{\frac{-0.5 l_k \delta_k}{h^2}}$, $-L_k \leq l_k \leq L_k$, for $l = (l_1, ..., l_D) \in \mathbb{Z}^D$.

To reduce the wrap-around effects of fast Fourier transform near the dataset boundary, it is essential to appropriately zero-pad the grid count and the kernel weight matrices to two matrices of the dimensionality $P_1 \times \cdots P_d$, where $P_i = 2^{\log_2 \lceil M_i + L_i \rceil}$.

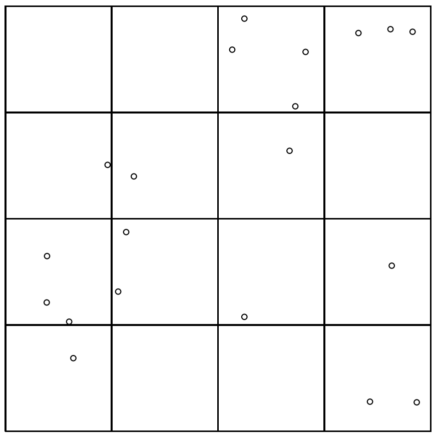


Figure 1.2: Two dimensional gridding of data points. Here $M_1 = M_2 = 5$.

1.2.2 Algorithm

The key ingredient in this method is the use of Convolution Theorem for Fourier transforms. The structure of the computed grid count matrix and the kernel weight matrix is cleverly crafted to take advantage of the fast Fourier transform. Basically we want $\tilde{s_k}(g_j) = \sum_{l_1=-L_1}^{L_1} \cdots \sum_{l_d=-L_d}^{L_d} c_{j-l} K_{k,l}$, for every grid point $g = (g_{1j_1}, ..., g_{dj_d})$.

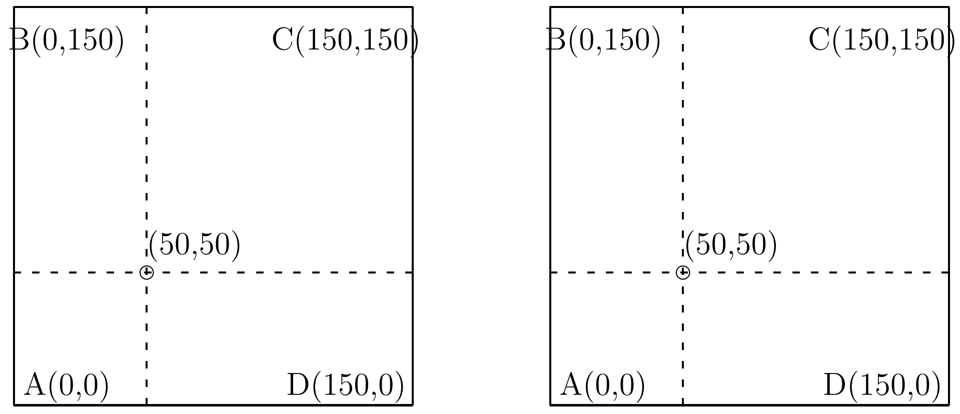
Let two functions $h(t)$ and $g(t)$, and their corresponding Fourier transforms $H(f)$ and $G(f)$ be given. Then, the convolution of the two functions is defined by:

$$g * h \equiv \int_{-\infty}^{\infty} g(\tau)h(t-\tau)d\tau \tag{1.5}$$

Then, the Convolution Theorem [20] states that the Fourier transform of the convolution of two functions is the product of the individual Fourier transforms. That is,

$$g * h \equiv G(f)H(f) \tag{1.6}$$

After the necessary convolution of two matrices, the $M_1 \times \cdots \times M_d$ submatrix in the upper left corner of the resultant matrix contains the kernel density estimate of the grid points. Then,



(a) Nearest Neighbor Binning Rule ($A = 1, B = C = D = 0$)

(b) Linear Binning Rule ($A = \frac{4}{9}, B = \frac{2}{9}, C = \frac{1}{9}, D = \frac{2}{9}$)

Figure 1.3: Two possible binning rules for KDE using multidimensional fast Fourier transform. Consider a data point falling in a two-dimensional rectangle. In 1.3(a), the entire weight is assigned to the nearest grid point. In 1.3(b), the weight is distributed to all neighboring grid points by linear interpolation.

the density estimate of the actual data point is linearly interpolated using the density estimates of neighboring grid points.

The multidimensional FFT and inverse FFT has been adapted from [20, 10]. A detailed pseudocode is presented in Appendix A.1.

1.2.3 Free Parameters

- M_i (indirect): The number of grid points along the i-th dimension. Since it is cumbersome to specify M_i's for every dimension, we instead choose $M = M_1 = M_2 = \cdots = M_D$.

1.2.4 Algorithm Cost

Because of the explicit dependence on the dimensionality D and the number of grid points along each dimension M, this method has never been tested above three dimensions. M also has been limited to values less than 50 [27].

$$\text{The grid count matrix: } c^Z = \begin{pmatrix} c_{1,1} & \cdots & c_{1,M_2} & \\ \vdots & \ddots & \vdots & \mathbf{0} \\ c_{M_1,1} & \cdots & c_{M_1,M_2} & \\ & \mathbf{0} & & \mathbf{0} \end{pmatrix}$$

$$\text{The kernel weight matrix: } \mathbf{K}^Z = \begin{pmatrix} K_{00} & \cdots & K_{0L_2} & & K_{0L_2} & \cdots & K_{01} \\ \vdots & \ddots & \vdots & \mathbf{0} & \vdots & \ddots & \vdots \\ K_{L_10} & \cdots & K_{L_1L_2} & & K_{L_1L_2} & \cdots & K_{L_11} \\ & \mathbf{0} & & \mathbf{0} & & \mathbf{0} & \\ K_{L_10} & \cdots & K_{L_1L_2} & & K_{L_1L_2} & \cdots & K_{L_11} \\ \vdots & \ddots & \vdots & \mathbf{0} & \vdots & \ddots & \vdots \\ K_{10} & \cdots & K_{1L_2} & & K_{1L_2} & \cdots & K_{11} \end{pmatrix}$$

$$\text{where } K_{l_1,l_2} = e^{\frac{-0.5((l_1\delta_1)^2+(l_2\delta_2)^2)}{h^2}}.$$

Figure 1.4: The grid count and the kernel weight matrix formed for a two-dimensional dataset. They are formed by appropriately zero-padding for taking the boundary-effects of fast Fourier transform based algorithms into account.

1.2.4.1 Runtime

The first part of the algorithm takes $O(2^D N)$ for gridding each data point. Computing the necessary convolution of the grid count matrix and the kernel weight matrix takes two invocations of fast Fourier transforms of two matrices and the intverse fast Fourier transform of the element-wise products of two transformed matrices. This takes $O((M \log M)^D)$. Finally, retrieving the final density estimate of each data point requires a linear interpolation of the density estimates at the neighboring grid points, hence taking $O(2^D N)$. Thus, the algorithm is of order $O((M \log M)^D + 2^D N)$, suffering from the *curse of dimensionality.*

1.2.4.2 Space Cost

The uniform grids used to bind the data points uses exponential space in terms of the dimension D, $O(M^D)$. Because the grid-based data structure is not adaptive, there will be many empty grid boxes for high-dimensional/non-uniform dataset.

1.2.5 Error Control

Performing a calculation on equally-spaced grid points introduces artificats at the bounaries of the data [11]. In addition, the linear interpolation of the data points by assigning to neighboring grid points introduce further errors. In general, M_i's act as an indirect parameter as increasing it will provide more accuracy. However, due to the limitations on floating point representation, there exists M_L such that any $M > M_L$ will not provide any additional accuracty. It is also impossible to provide a tight bound on ϵ in a mathematical relationship involving M_i's.

1.2.6 Summary

Using the multidimensional fast Fourier transform in kernel density estimation setting, as analyzed in this section, does not seem to be a good idea. Although the exact algebraic nature of the transform may be attractive in achieving high accuracy, the algorithm relies on periodicity of the dataset with each data point on a grid scheme. Most importantly, we do not have full error control, as this method does not provide the parameter to do so.

1.3 Improved Fast Gauss Transform

This algorithm is a derivative of the one described in [12], but *does not follow the exact methodology developed in the original method.* The key difference involves use of adaptive space partitioning of reference data points, and the different mathematical tools to approximate the contribution of a chunk of reference data points as a truncated Taylor series of order p. Readers are advised to note that this method has nothing in similarity as the original method developed in [12], and advised to skip to Section 1.5, for the description of the real fast multipole method-based Gauss transform and the explanations for terminologies used in this section.

1.3.1 Data Structure

Figure 1.5 shows the spherical partitioning of the reference dataset X_R used in the algorithm. These groupings are an improvement over the grid scheme/oct-tree/quad-tree used in multidimensional fast Fourier transform and fast multipole methods.

The reference points beloning to a cluster are combined to form a representative "Taylor" expansion centered at a representative center x_R, summarizing the positions of these points.

The authors of [29] state that the space subdivision task is modeled by a k-center problem: given a set of N_R reference point and a number of the clusters k, find a partition of the points into clusters $X_{R_1}, ..., X_{R_k}$ and their centers $x_{R_1}, ..., x_{R_k}$ such that we minimize $\max\limits_{1 \leq i \leq k} \max\limits_{x_r \in X_{R_i}} ||x_r - x_{R_i}||$.

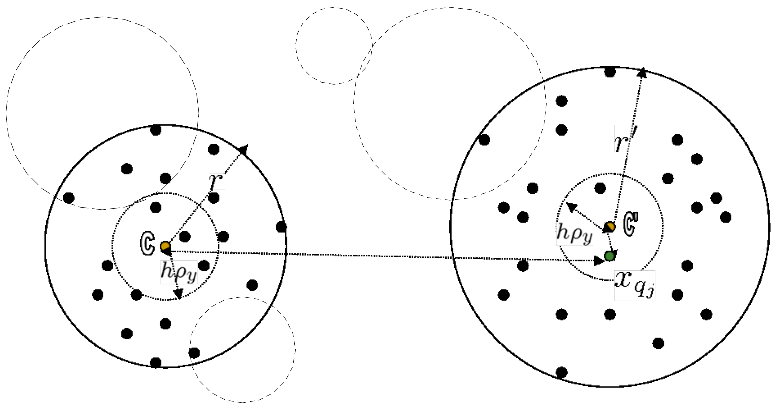


Figure 1.5: A simple k-center algorithm is used to produce spherical groupings of reference points in improved fast Gauss transform algorithm [29]. Here, a query point x_{q_j} is evaluated at the Taylor series whose radius is r' since it is within $h\rho_y$ from its center. However, it is not evaluated for the other spherical grouping whose radius is r since it is too far away from the center.

That is, to minimize the maximum radius of clusters. The authors of [29] believe the clustering algorithm used in improved fast Gauss transform eliminates the need of having to maintain the *interaction list* for each group entity (a cluster in this case, a box in a grid-based algorithm, a node in a tree-based algorithm).

1.3.2 Different Factorization of the Gaussian Kernel

The authors of [29] believe that expanding the Gaussian kernel along each direction is the bottleneck in the applicability of fast multipole methods in higher dimensional setting. They claim that because the Gaussian kernel decays rapidly as the distance from its source increases, the contributions outside of a certain radius can be safely ignored. They also claim there is no need to perform the multipole expansions which account for the far-field contributions. They advocate viewing the multivariate Gaussian kernel not as a product of univariate Gaussian kernel, but as a vector function. Then, we have the following useful factorization.

$$e^{-\frac{||x_{q_j}-x_{r_i}||^2}{2h^2}} = e^{-\frac{||\Delta x_{q_j}||^2}{2h^2}} e^{-\frac{||\Delta x_{r_i}||^2}{2h^2}} e^{\frac{\Delta x_{q_j} \Delta x_{r_i}}{h^2}} \tag{1.7}$$

where $\Delta x_{q_j} = x_{q_j} - x_R$, $\Delta x_{r_i} = x_{r_i} - x_R$.

Then, we can express the unnormalized Gaussian sum as:

$$G(x_{q_j}) = \sum_{\alpha \geq 0} C_\alpha e^{-||x_{q_j}-x_R||^2/h^2} \left(\frac{x_{q_j} - x_R}{h} \right)^\alpha \tag{1.8}$$

where $C_\alpha = \frac{2^{|\alpha|}}{\alpha!} \sum\limits_{i=1}^{N_R} e^{-||x_{r_i}-x_R||^2/h^2} \left(\frac{x_{r_i}-x_R}{h} \right)^\alpha$.

In this representation, the trucation of the Taylor expansion after tht order $p-1$ requires $\binom{p+D-1}{D}$. For $D \to \infty$ and moderate value of p, the number of terms becomes $O(D^p)$.

1.3.3 Algorithm

Here is an informal description of the algorithm.

1. Input tweak parameters: p, K, ρ_y.
2. Assign the reference data points into K clusters using the farthest-point clustering algorithm.
   - Pick an arbitrary point v_0 as the center of the first cluster and add it to the center set C.
   - For $i = 1$ to k, in i-th iteration, for every point, compute its distance to the set C: $d_i(v, C) = \min\limits_{c \in C} ||v - c||$.
   - Add the point farthest away from C, v_i, to the set C, and let the new center v_i "steal" the points closer to it from the other centers.
   - Note the maximum radius of a cluster $R_{MAX} = h\rho_x$.
3. Choose p sufficiently large such that the error estimate is less than the desired precision ϵ: $|E(y)| \leq (\sum |q_j|) \left(\frac{2^p}{p!} \rho_x^p \rho_y^p + e^{-(\rho_y - \rho_x)^2} \right) \leq \epsilon$
4. For each cluster S_k with center c_k, compute the coefficients given by the expression: $C_\alpha^k = \frac{2^{|\alpha|}}{\alpha!} \sum\limits_{x_{r_i} \in S_k} q_i e^{-||x_{r_i}-c_k||^2/h^2} \left(\frac{x_{r_i}-c_k}{h} \right)^\alpha$.
5. For each $x_{q_j} \in X_Q$, compute: $G(x_{q_j}) = \sum\limits_{||x_{q_j}-c_k|| \leq h\rho_y} \left[\sum\limits_{|\alpha|<p} C_\alpha^k e^{-||x_{q_j}-c_k||^2/h^2} \left(\frac{x_{q_j}-c_k}{h} \right)^\alpha \right]$.

1.3.4 Free Parameters

- K: The number of clusters to group n data points using the "k center algorithm"
- ρ_y: For a given query data point, x_{q_j}, a contribution from a cluster will be "pruned" if it is not within $h\rho_y$ from its centroid.
- p: The order of truncation for the Taylor expansion stored at each cluster.

1.3.5 Algorithm Cost

1.3.5.1 Runtime

The time complexity is $O(D^p)$, when each Taylor expansion is evaluated at each query point.

1.3.5.2 Space Cost

The required space complexity is dominated by storing the Taylor coefficients for all K clusters, $O(\binom{p+D-1}{D}K)$. So it is $O\left((K\binom{p+d-1}{d}\right)$.

1.3.6 Error Control

This algorithm gives no control on specifying the error tolerance directly. The authors claim that the number of clusters can be less than $\sqrt{N_R}$, and the number of Taylor terms needed is less than 10. On page 8 of [29], the authors show how the 3-tuple $(p, \rho_x.\rho_y)$ (where ρ_x is the largest bounding radius of the clusters created by the k-center algorithm and ρ_y is the cut-off ratio of the radius to the included cluster in the gaussian summation calculation, to the bandwidth h) can satisfy the following error bound for a given target point, y. Because we are using a slightly different Gaussian kernel from one used in the authors' implementation and each source point has an equal weight of 1, the authors claim we can solve for τ explicitly in the following expression:

$$\begin{aligned}|E(y)| &\leq Q[\frac{\rho_x^p \rho_y^p}{p!} + e^{-0.5(\rho_y-\rho_x)^2}] \\ &= N_R[\frac{\rho_x^p \rho_y^p}{p!} + e^{-0.5(\rho_y-\rho_x)^2}] \\ &< \tau\end{aligned}$$

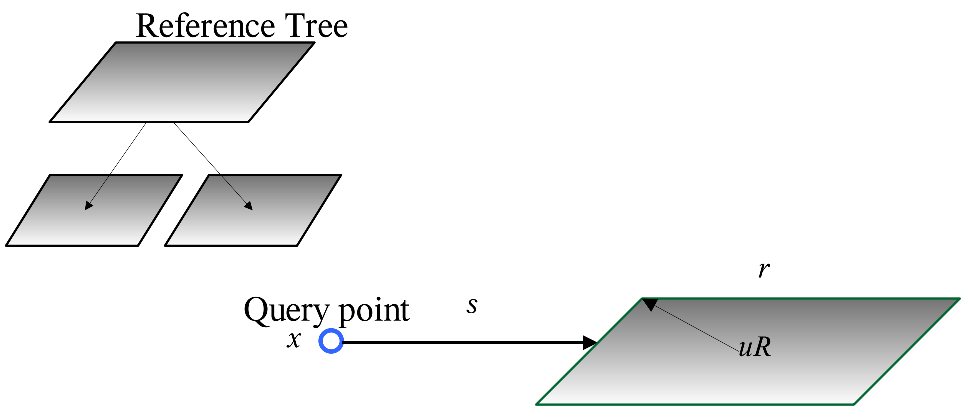


Figure 1.6: In Barnes-Hut algorithm, we only build the tree on the reference data points. For every query point x_q, we do a depth-first traversal of the tree. We specify θ as an indirect parameter controlling the pruning rule. If $s > \frac{r}{\theta}$, then the contribution of the reference points in the node is approximated by $N_R K_h(x_q, \mu_R)$.

However, the empirical results seem to indicate that this error bound analysis is incorrect. Our current guess is that the authors of [29] have made errors in analyzing the Gaussian kernel using the vector version of the Taylor series. This observation was echoed independently by another group attempting to use the method.

1.3.7 Summary

Improved fast Gauss transform attempted to eliminate the grid structure used in multidimensional fast Fourier transform and the original fast Gauss transform [12] and replaced it with adaptive clustering structures. However, the authors unfortunately introduced three *tweak parameters* that made error bound analysis and usage of the algorithm very difficult. We suspect that the algorithm itself does not take into account the property of Taylor series and the properties of the Gaussian kernel, as described in Section 1.5.2.2.

1.4 Dual-tree KDE

The conventional tree-based KDE algorithms are what we call "single-tree" algorithms; we process each query point $x_q \in X_Q$ at a time and do a sequence of depth-first traversal of the tree containing the reference dataset X_R. Such algorithms (such as Barnes-Hut algorithm) have complexity of at least $O(N \log N)$.

On the other hand, fast multpole methods [12] introduced the concept of putting the query dataset in an organized datastructure (grid or an octtree/quadtree). This allowed processing a chunk of query points and a chunk of reference points at a time to allow for fast approximation. For each node/grid box containing any reference points, an interaction list of query nodes/grid boxes is created. Maintaining these lists in high dimensional setting is quite daunting and impossible. [11] describes a fast, easy-to-use implementation that eliminates the need for maintaining such lists by introducing the concept of *higher-order divide-and-conquer*. This is the first tree-based KDE algorithm we will analyze, and will form the basis of the new algorithm.

The original algorithm handles any arbitrary kernel function easily, but again we shall be focusing our attention on the Gaussian kernel:

$$K_h(x_q, x_r) = e^{\frac{-||x_q - x_r||^2}{2h^2}} \tag{1.9}$$

1.4.1 Data Structure

In this algorithm, adaptive data structures such as *kd*-trees and metric trees [18, 19, 6] divide the data points into hierarhical groups. Instead of maintaining interaction list for each node/box as done in [12], a simultaneous depth-first traversal of two different pointers (query tree and reference tree) to a single tree representing the data points. In addition, a hard lower/upper bound on the density estimate on any pair of query/reference point is obtained via simple geometric information of the query/reference node. The density estimate computed by this method is guaranteed to be at most ϵ away from the density estimates computed by the naive method.

In the implementation we tested, a high-dimensional *sphere-rectangle tree* [18] was constructed for the reference dataset and the query dataset, using a simplified version of *anchors hierarchy*.

1.4.1.1 Advantages of Tree Data Structure

A tree is an inductively defined data structure. For example, a binary tree using the following inductive definition:

1. An empty node is a binary tree.

2. If t_{left} and t_{right} are binary trees, then the node whose left and right pointers point to t_{left} and t_{right} is a binary tree.

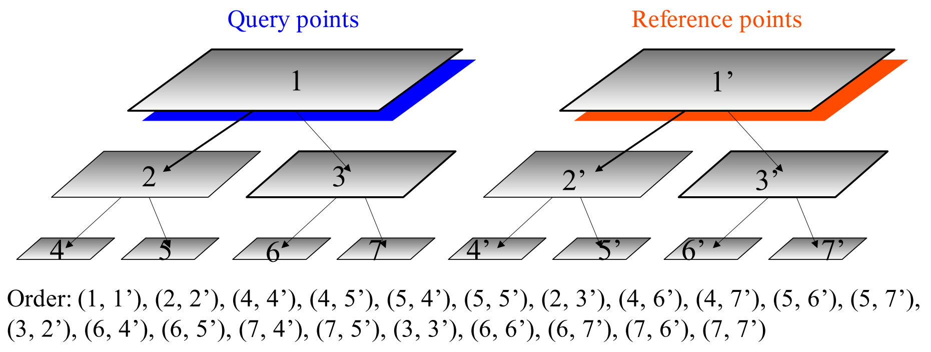

Query points
Reference points
1
2
3
4
5
6
7
1'
2'
3'
4'
5'
6'
7'
Order: (1, 1'), (2, 2'), (4, 4'), (4, 5'), (5, 4'), (5, 5'), (2, 3'), (4, 6'), (4, 7'), (5, 6'), (5, 7'), (3, 2'), (6, 4'), (6, 5'), (7, 4'), (7, 5'), (3, 3'), (6, 6'), (6, 7'), (7, 6'), (7, 7')

Figure 1.7: Traversal order using two trees.

This inductive definition allows usage of *cached sufficient statistics* [17], which allows a form of dynamic programming. We shall see how information gain on a given node can be propagated to its children or its parent, as in the case of up-down propagation in dual-tree KDE or the set of translation operators in fast multipole methods.

1.4.2 Algorithm

The high-level description of the algorithm is given in Figure 1.10. Basically, the four-way recursion is done with each pair of reference node and query node is considered for "pruning." Note that if a pair of reference node and query node can be "pruned," then the recursion terminates.

1.4.2.1 Maintaining Hard Bounds

For each node, we maintain the bounds ϕ_q^{min} and ϕ_q^{max} on the unnormalized Gaussian sum. These bounds start in a pessimistic way: $\phi_q^{min} = 0$ and $\phi_q^{max} = N_R$, since we know that the Gaussian kernel is bounded by 0 and 1 for any pairs of query point and reference point. The lower bound increases monotonically and the upper bound decreases monotonically.

When a reference node R is considered and pruning is performed, we update the bounds in the following way:

$$\begin{aligned} \forall q \in Q, \Phi_q^{min} &+= dl, dl = N_R K_h(\delta_{QR}^{max}) \\ \forall q \in Q, \Phi_q^{max} &+= du, du = N_R [K_h(\delta_{QR}^{min}) - 1] \end{aligned} \tag{1.10}$$

For every query point in a given node X_Q, its density estimate is always between Φ_q^{min} and Φ_q^{max}.

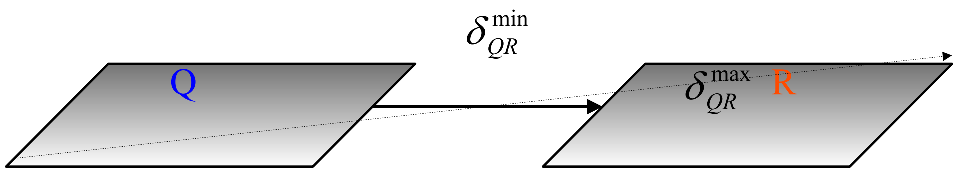


Figure 1.8: A hard lower/upper bound on the density estimate is obtained from simple geometric information about the query node and the reference node. Here, the maximum contribution of the reference node on any query point is $N_R K_h(\delta_{QR}^{min})$. Similarly the minimum contribution of the reference node on any query point is $N_R K_h(\delta_{QR}^{max})$.

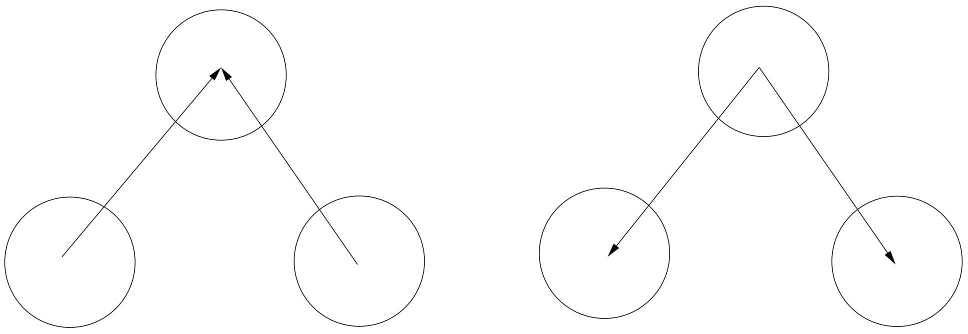


Figure 1.9: Information gain due to pruning for a give query node should be transmitted to its parent and its children. These operations are analogous to *multipole-to-multipole translation* and *local-to-local translation* operators respectively.

1.4.2.2 Maintaining Error Control

Suppose we consider a pair of a query node X_Q and a reference node X_R, and let δ_{QR}^{min} and δ_{QR}^{max} be the minimum and maximum distance between two nodes. Intuitively, we can approximate the reference node contribution by $N_R \bar{K_h}$ where $\bar{K_h} = \frac{K_h(\delta_{QR}^{max}) + K_h(\delta_{QR}^{min})}{2}$ if $K_h(\delta_{QR}^{max})$ and $K_h(\delta_{QR}^{min})$ are close.

Therefore, the error of the approximation using $N_R \bar{K_h}$ with respect to any query point $x_q \in X_Q$:

$$e_{QR} = \frac{N_R(K_h(\delta_{QR}^{min}) - K_h(\delta_{QR}^{max}))}{2} \tag{1.11}$$

If we want to make sure the density estimate of every query point $\phi(x_q)$ meets the user-specified ϵ, we obtain the following local pruning criterion which ensures the global error tolerance ϵ:

$$|K_h(\delta_{QR}^{min}) - K_h(\delta_{QR}^{max})| \leq \frac{2\epsilon}{N_R} \phi_Q^{min} \tag{1.12}$$

1.4.2.3 Delayed Summation Technique

Rather than updating each Φ_q^{min} and Φ_q^{max} for each query data point, we maintain Φ_Q^{min} and Φ_Q^{max} for all of the query points Q. In this way, each update is $O(1)$ rather than $O(N_Q)$. This is very similar to the *multipole-to-local translation operator* works in fast multipole methods.

Of course, using delayed summation requires a single pre-order traversal of the query tree, which simply adds any contributions stored in these variables to the query points in the relevant nodes. In fast multipole methods, this is exactly what the *local-to-local translation operator* does.

1.4.2.4 Up-down mass propagation

Delayed summation technique causes a problem since each local lower/upper bound update is known only to the query node that is pruned. For example, when a reference node X_R's contribution is stored, in the bounds for X_Q, this same information is not transmitted to its subtree. [11] states two ways to solve this problem:

- *Downward propagation*: pass dl and du to the entire subtree below X_Q whenever the new mass is obtained.
- *Upward propagation*: takes min/max of the children's bounds to tighten the lower/bounds of the query node.

In order to avoid full downward propagation on every pruning, the algorithm in [11] uses the *deferred propagation technique* in which pruning information on a given query node is passed to its immediate children and its parent only.

1.4.3 Free Parameters

- τ: used as a local pruning rule for error control for ensuring the global error tolerance ϵ. Using $\tau \leq \epsilon$ suffices.

1.4.4 Algorithm Cost

1.4.4.1 Runtime

It takes roughly $O(N)$ for a dual-tree KDE computation empirically but we have not yet proved this.

```
KDE(Q, R)
  Do up/down mass propagation.
  Try inclusion/exclusion pruning rule.

  if leaf(Q) and leaf(R)
    Run a naive quadratic algorithm on every pair of points in Q and R
  else
    KDE(Q.left, R.left)
    KDE(Q.left, R.right)
    KDE(Q.right, R.left)
    KDE(Q.right, R.right)
```

Figure 1.10: A pseudocode doing a depth-first traversal of two trees is presented here.

1.4.4.2 Space Cost

It requires $O(N)$ for space complexity to store the required trees.

1.4.5 Error Control

Using the simple pruning criterion developed in 1.4.2.2, we can guarantee that every density estimate satisfies the user-specified ϵ criterion.

1.4.5.1 Summary

In this section, we have seen how trees can be used in KDE setting. Using two trees (each for the reference dataset and and for the query dataset) according to [11] gives huge improvement over conventional single-tree-based algorithms. In the next section, we will discuss a powerful technique from approximation theory and later show how it could be combined with dual-tree KDE to create a new fast algorithm.

1.5 Fast Multipole Methods

1.5.1 Brief Overview of Fast Multipole Methods

Fast multipole methods (FMM) are a class of powerful techniques developed in computational physics. Unlike most N-body methods, they come with rigorous error bound on the estimate and were one of the first methods to achieve $O(N)$ time complexity. In these class of methods, the kernel function in consideration is expanded into a truncated Taylor/multipole expansion of order p.

In contast with fast Fourier transform-based methods (which perform computations by taking advantage of the periodicity of the dataset), fast multipole methods are approximate, based on analytic considerations, and insensitive to the distribution of the reference data set X_R [5]. Interested readers are invited to consult a good, concise introduction to this general class of algorithms [5].

Despite their success in reducing the computational cost in many physical settings, the FMMs have not been utilized outside specialized communities because it is hard to visualize the mapping between the theory and the real working implementation.

Our goal in this section is to clarify any confusion and to discuss their potentials in tackling pair-wise N-body problems. We will be focusing on analysis and derivation of kernel-specific fast multipole methods. Readers interested in FMM variants that are kernel-independent are encouraged to refer to [2] and other literatures.

1.5.1.1 Notations

As we are working in a Euclidean space of any arbitrary dimension, it is convenient to use a multi-index notation defined in [12].

We define a multi-index $\alpha = (\alpha_1, \alpha_2, ..., \alpha_D)$ as a D-tuple of nonnegative integers. For any multi-index α and any $x \in \mathbb{R}^D$, define:

$$\begin{aligned} |\alpha| &= \alpha_1 + \alpha_2 + \cdots + \alpha_D \\ \alpha! &= \alpha_1!\alpha_2! \cdots \alpha_D! \\ x^\alpha &= x^{\alpha_1} x^{\alpha_2} \cdots x^{\alpha_D} \\ D^\alpha &= \partial_1^{\alpha_1} \partial_2^{\alpha_2} \cdots \partial_D^{\alpha_D} \end{aligned}$$

where ∂_i is a partial derivative with respect to the D-th coordinate direction. If $\alpha \geq p$ for an integer p, then $\alpha_i \geq p$ for $1 \leq i \leq D$.

We in addition also define the following multi-index arithmetic operations. For two multi-indices $\alpha = (\alpha_1, ..., \alpha_D)$ and $\beta = (\beta_1, ..., \beta_D)$, we define the sum of two multi-indices as $\alpha + \beta = (\alpha_1 + \beta_1, \alpha_2 + \beta_2, .., \alpha_D + \beta_D)$. We define for two multi-indices and α and β such that $\alpha_i \geq \beta_i$ for $1 \leq i \leq D$, the difference as $\alpha - \beta = (\alpha_1 - \beta_1, \alpha_2 - \beta_2, ..., \alpha_D - \beta_D)$.

1.5.1.2 Multipole Expansion

Recall that our goal is to evaluate the density estimate $\hat{p}(x_q)$ at each $x_q \in X_Q$:

$$\hat{p}(x_q) = \frac{1}{N_R V_{Dh}} \sum_{r=1}^{N_R} K\left(x_q, x_r\right) \tag{1.13}$$

Suppose the kernel function $K(x_q, x_r)$ is infinitely differentiable in most of $\mathbb{R}^D$. Then, it can be expressed as a truncated series of order p about a carefully chosen center x_R:

$$K(x_q, x_r) = \sum_{\alpha \geq 0} \phi_\alpha(x_q, x_R)\psi_\alpha(x_r, x_R) \tag{1.14}$$

where ϕ_k's are functions dependent only on x_q, x_R and ψ_k's are functions dependent only on x_r, x_R. If we truncate the infinite series after p terms along each direction, we will have p^D terms for the approximation.

$$\begin{aligned}
\hat{p}(x_q) &= \frac{1}{N_R V_{Dh}} \sum_{r=1}^{N_R} K\left(x_q, x_r\right) \\
&= \frac{1}{N_R V_{Dh}} \sum_{r=1}^{N_R} \sum_{\alpha \geq 0} \phi_\alpha(x_q, x_R)\psi_\alpha(x_r, x_R) \\
&= \frac{1}{N_R V_{Dh}} \left(\sum_{r=1}^{N_R} \sum_{\alpha < p} \phi_\alpha(x_q, x_R)\psi_\alpha(x_r, x_R) + \epsilon_M(p) \right) \\
&= \frac{1}{N_R V_{Dh}} \left(\sum_{\alpha < p} \left(\sum_{r=1}^{N_R} \psi_\alpha(x_r, x_R) \right) \phi_\alpha(x_q, x_R) + \epsilon_M(p) \right)
\end{aligned} \tag{1.15}$$

Note that by switching the summation between the one iterating through all the reference data points and the one controlling the order of the series, we can pre-compute $A_\alpha = \sum\limits_{r=1}^{N_R} \psi_\alpha(x_r, x_R)$ for all $0 \leq \alpha < p$. Because A_α's depend only on the reference data points and a representative point x_R (a centroid in a node case), no other knowledge is needed to pre-compute these "moments." This expansion, as shown in Figure 1.11 is called a multipole expansion or a reference-side expansion [11].

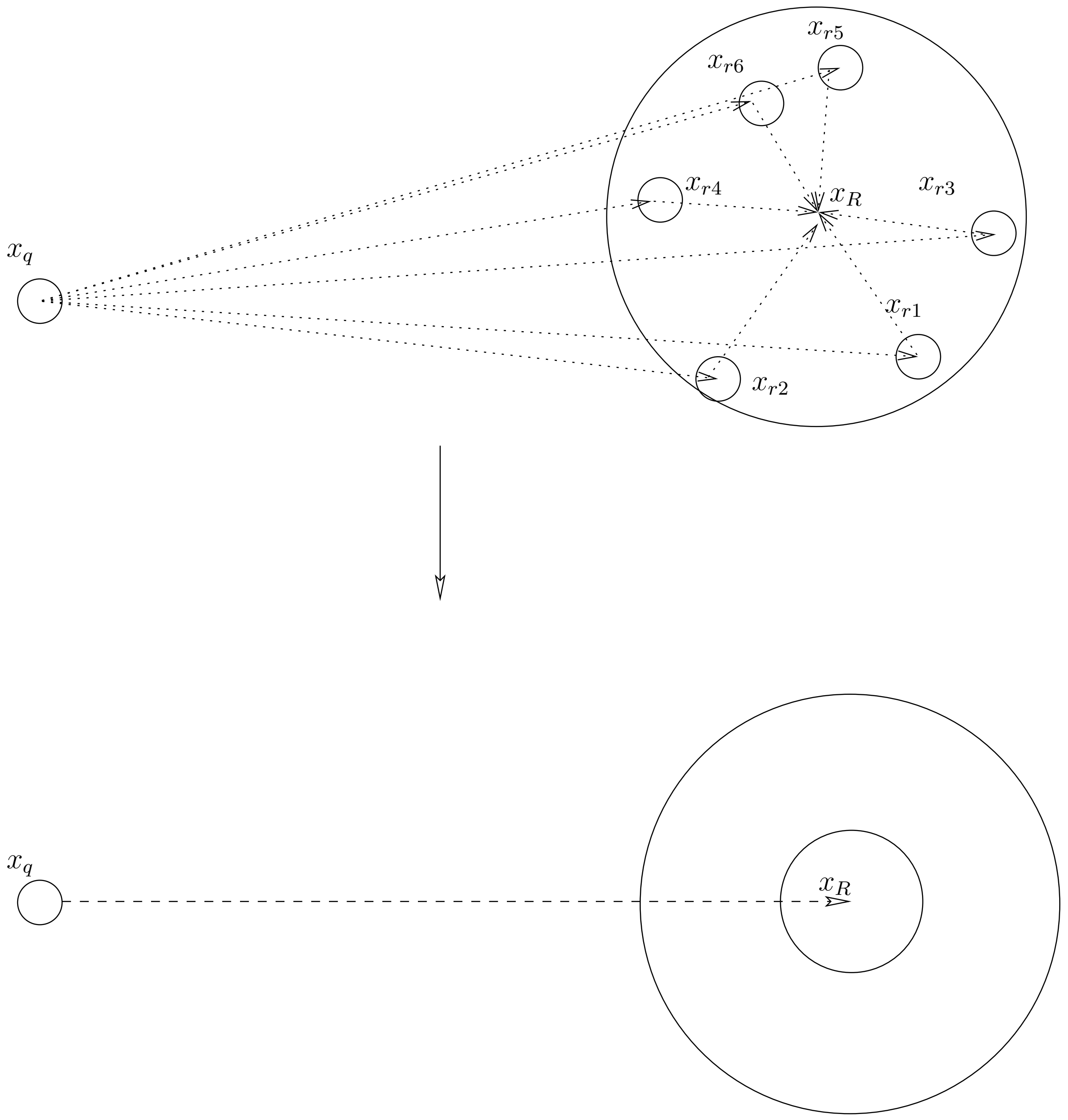


Figure 1.11: Here we are given a query point x_q and a reference node containing six reference data points. A multipole expansion summarizes the six individual interactions between the query point x_q and each of the six x_{ri}'s by expanding the six reference data points about the centroid X_R. Then a single function of order p can capture the entire contribution of the reference node for x_q.

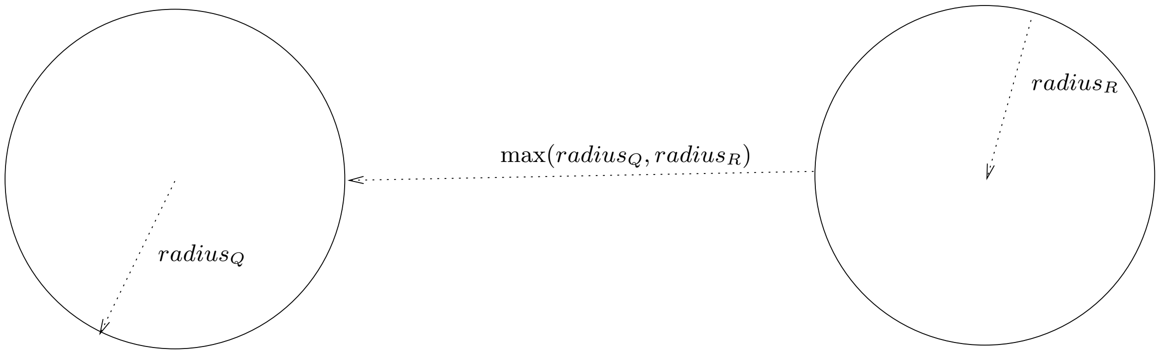


Figure 1.12: A reference node is well-separated from a query node if the minimum distance between the two is at least $\max(radius_Q, radius_R)$.

There are two types of error bound functions $\epsilon_M(p)$. For example, $\epsilon_M(p)$ for a Coloumbic kernel [9] depends both on the desired precision and the well-separated-ness of x_q and x_r's as defined in [7]. In contrast, $\epsilon_M(p)$ for a Gaussian kernel, by bounding the size of the reference node (via use of a grid structure), may only depend on the desired precision and be independent of the location of x_q [4, 12].

1.5.1.3 Local Expansion

A local expansion or a query-side expansion as introduced in [9, 12] is basically a Taylor expansion of a multipole expansion. This is essential for achieving $O(N)$, whereas the asymptotic complexity of tree-based codes that do not use local expansion [3] is at least $O(N \log N)$.

Intuitively, a local expansion lets us approximate the potential at each x_q in a query node in terms of the centroid x_Q.

$$\begin{aligned}\hat{p}(x_q) &= \frac{1}{N_R V_{Dh}} \sum_{r=1}^{N_R} K\left(x_q, x_r\right) \\ &= \frac{1}{N_R V_{Dh}} \left(\sum_{\beta < p} B_\beta \phi_\beta(x_q, x_Q) + \epsilon_L(p) \right)\end{aligned} \tag{1.16}$$

where $B_\beta = \sum\limits_{\alpha<p} A_\alpha \omega_{\alpha+\beta}(x_Q, x_R)$. Basically, the coefficients of the local expansion is formed by taking a convolution sum between each multipole coefficient and functions dependent on x_Q and x_R. Because the multipole expansion is truncated at p^D terms before being converted to a local expansion, the error approximation $\epsilon_L(p)$ is larger than $\epsilon_M(p)$.

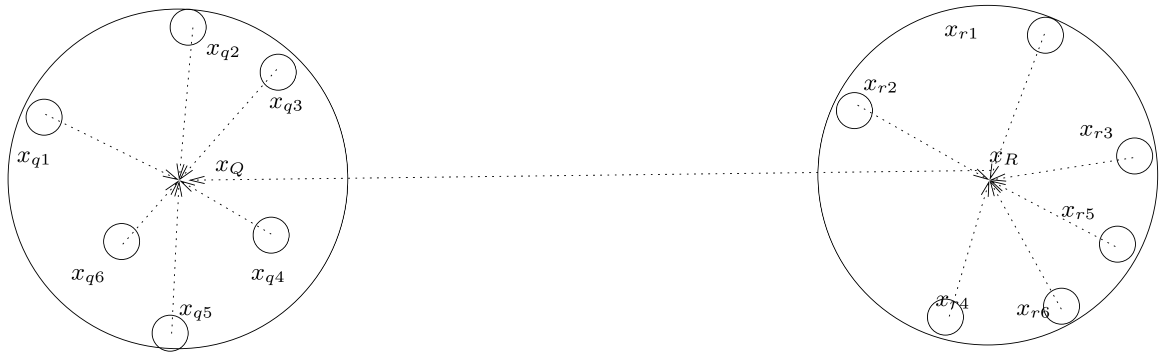


Figure 1.13: Instead of evaluating the multipole expansion formed from x_{ri}'s at each x_{qi}'s, we can form a Taylor expansion of a multipole expansion (local expansion) about x_Q, a representative point in the query node. This lets us collect the contribution of reference nodes to each query point in the query node in a form of Taylor coefficients.

1.5.1.4 Translation Operators

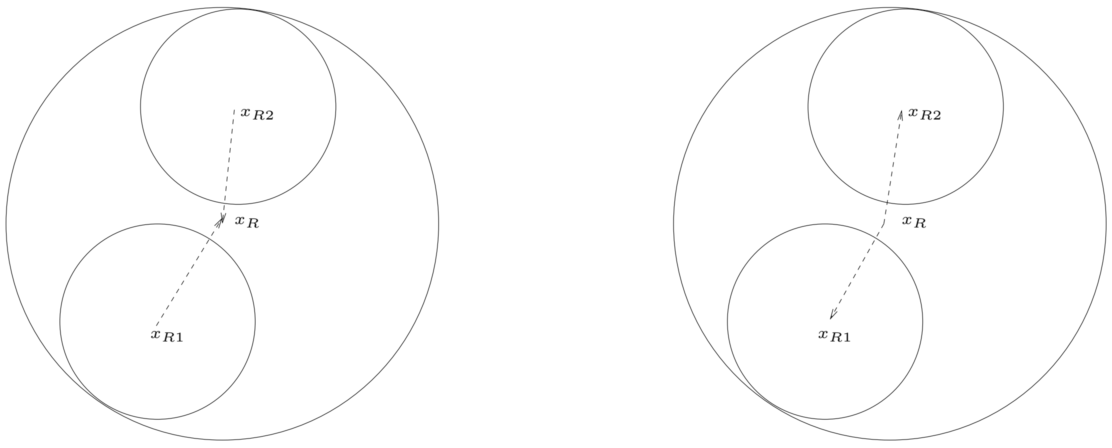
Greengard and Rokhlin introduced the set of translation operators for converting between multipole and local expansion; these operators are crucial components in achieving $O(N)$ in FMMs. The concrete example using these abstract operators in the Cartesian setting will be presented in 1.6.

1. *Multipole to multipole translation operator (M2M operator)*: This is a useful operator in a tree-based FMM. In forming the multipole expansion of a parent node, the multipole expansions of its children nodes are shifted to the centroid of the parent node and summed up. This can be interpreted as a form of cached sufficient statistics calculated in a bottomup fashion during construction of the tree.

   Suppose a parent node has c children nodes: $X_1, ..., X_c$, each with $N_1, ..., N_c$ reference data points. Without the help of M2M operator, the cost of computing a multipole expansion of the parent node is $O\left(p^D\left(\sum\limits_{i=1}^{c} N_i\right)\right)$. However, the M2M operator lets us compute the same expansion in $O(c \cdot p^D)$.

2. *Multipole to local translation operator (M2L operator)*: This operator is used to convert a multipole expansion of a reference node to form a local expansion centered at the centroid of the query node, as explained in 1.5.1.3.

3. *Local to local translation operator (L2L operator)*: This operator, in contrast with M2M operator, acts as a "clean-up" crew in a tree-based FMM. In a tree structure comprising the query dataset X_Q, we are left with a local expansion in each node of the tree at various depth



(a) M2M Translation Operator (b) L2L Translation Operator

Figure 1.14: In 1.14(a), the multipole expansions of the two children nodes centered at x_{R1} and x_{R2} are shifted to the centroid of the parent node x_R. In 1.14(b), the local expansion of the parent node is transmitted to each of the two children nodes.

levels. By performing a breadth-first traversal of the query tree, the L2L operator shifts a local expansion of a parent node to the children node's centroid and adds to the children's local expansions.

It is worthwhile to note that each query point is evaluated only once at a local expansion of the leaf node it belongs to, for computing the far-field contributions outside.

1.5.1.5 Algorithm

Here we present the *recursive generalization* of $O(N)$ tree-based fast multipole methods (for arbitrary kernel function). Details of the approach using the Coluombic kernel are outlined in [9].

1. Create a oct-tree/quad-tree of $X_Q \cup X_R$, the combined dataset containing both the query points and the reference points.

   - When forming a leaf node, compute the truncated multipole expansion of order p of the reference data points belonging to it.
   - When forming an internal node, compute the truncated multipole expansion due to all reference points in it by applying the $M2M$ translation operator to the multipole expansions of the children nodes.

2. Traverse down the tree in the breadth-first manner and form the interaction list of each node b containing any reference points. The interaction list of b is formed by considering the *well-separated* nodes of the same refinement level. For each node c in the interaction list, we convert the multipole expansion of b to the Taylor expansion about the center of c. For immediate near-neighboring query nodes, the contribution of b is computed directly using the naive method.

3. Do a pre-order traversal of the tree again and apply the $L2L$ translation operator to any Taylor coefficients on a given node, transmitting the information down to its children nodes. Once on a leaf level, we evaluate the accumulated Taylor expansion at every single query point.

1.5.1.6 Limitation of Fast Multipole Methods

- *Restriction on Kernel Function.* The kernel function needs to be analytic and have infinite number of non-zero derivatives. The Columobic, multiquadric, and Gaussian functions satisfy the necessary condition (and have fast multipole methods using those kernels). However, it is impossible to apply the techniques of fast multipole methods using other useful kernel functions such as spherical kernel ($K_h(||x_q - x_r||) = 1$ if $||x_q - x_r|| < h$, otherwise 0), Epanechnikov kernel ($K_h(||x_q - x_r||) = 1 - ||x_q - x_r||^2$ if $||x_q - x_r|| < h$, otherwise 0).

- *New Derivation for Each Kernel Function.* Each kernel function has unique multipole and Taylor expansions, and a new error bound has to be derived by algorithm designers as a result [1, 21].

- *Curse of Dimensionality.* Expanding the kernel function into a truncated seried of order p in each direction requires $O(p^D)$ space and runtime complexity. In addition, each hierarchial entity (a node or a grid box) needs to generate a list of neighboring partners, which is an expensive process in higher dimenisions. As a result, fast multipole methods have never been used beyond a three-dimensional setting.

1.5.1.7 Potential Improvements

- *Efficiency of Translation Operators.* One of the most expensive part of computations in fast multipole methods involves the translation operators: M2M (multipole to multipole), M2L (multipole to Taylor), L2L (Taylor to Taylor). Some work has been pursued to reduce the cost, as done in [13] for improving the original fast Gauss transform [12], by replacing original Hermite and Taylor expansions with an expansion in terms of plane waves.

- *Use of Adaptive Hierarchial Data Structures.* The utilization of hierarchial division in fast multipole methods is limited to quad-trees/oct-trees (semi-adaptive) [9, 14] and grid-scheme (non-adpative) [12]. Adaptive hierarchial data structures, such as k-d trees [6] and metric trees [18, 19], can be used to cluster data points more in tigher groupings than grids, quad-trees/oct-trees. [15] states that these adaptive data structures have not been fully utilized in fast multipole methods. Perhaps, the asymptotic building cost of $O(NlogN)$ discouraged designers of fast multipole methods for ever pursuing this avenue. Nevertheless, we will show how adaptive data structures can be usedful in developing fast algorithms.

1.5.2 Fast Gauss Transform

This algorithm was first introduced in [12]. This is just a fast multipole method using the Gaussian kernel

$K_h(x_q, x_r) = e^{\frac{-||x_q - x_r||^2}{2h^2}}$.

Unlike the original algorithm, we will assume that the weights placed at each reference points are equally weighted. That is, $q_j = 1$ for $1 \leq j \leq N_R$.

1.5.2.1 Data Structure

Strangely enough, no tree-based fast multipole method has been developed for this kernel, unlike other kernel functions. The data points are first scaled to fit in the unit hypercube $[0, 1]^D$ which is divided into a uniform grid of hypercubes. In contrast with the grid scheme used in fast Fourier transform method, which sets the number of grid boxes in advance, the fast Gauss transform limits the size of each grid box instead. Each side of a hypercube in the fast Gauss transform mesh has a maximum length of h, the smoothing parameter (bandwidth).

1.5.2.2 Properties of the Gaussian Kernel

Fast Gauss transform algortihm takes advantage of the analytical and numerical properties of the Gaussian kernel. I will summarize the mechanism developed in [12].

Our development begins with one-dimensional setting and generalizes to multi-dimensional setting we are interested in.

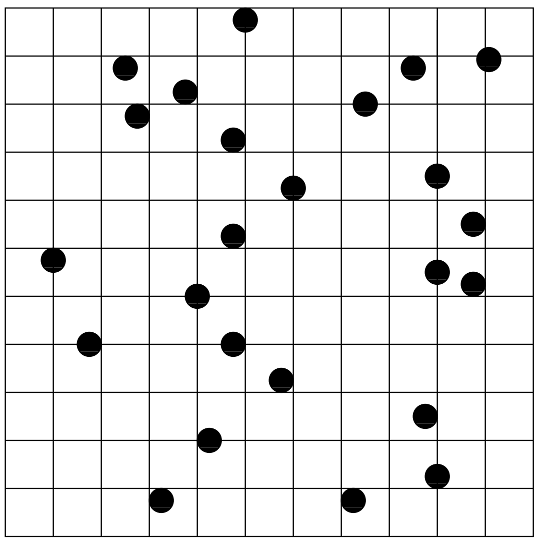


Figure 1.15: Uniform grid scheme is used for the fast multipole method using the Gaussian kernel.

We define the Hermite polynomials by the Rodrigues formula:

$$H_n(t) = (-1)^n e^{t^2} D^n e^{-t^2}, t \in \mathbb{R}^1 \tag{1.17}$$

The first few polynomials include [28]: $H_0(t) = 1$, $H_1(t) = 2t$, $H_2(t) = 4t^2 - 2$.
The generating function for Hermite polynomials is defined by:

$$e^{2ts-s^2} = \sum_{n=0}^{\infty} \frac{s^n}{n!} H_n(t) \tag{1.18}$$

Let us define the Hermite functions $h_n(t)$ by

$$h_n(t) = e^{-t^2} H_n(t) \tag{1.19}$$

Multiplying both sides by e^{-t^2} yields:

$$e^{-(t-s)^2} = \sum_{n=0}^{\infty} \frac{s^n}{n!} h_n(t) \tag{1.20}$$

We would like to use a “scaled and shifted” version of this derivation for taking the bandwidth h into account.

$$\begin{aligned} e^{\frac{-(t-s)^2}{2h^2}} &= e^{\frac{-((t-s')-(s-s'))^2}{2h^2}} \\ &= \sum_{n=0}^{\infty} \frac{1}{n!} \left(\frac{s-s'}{\sqrt{2h^2}}\right)^n h_n\left(\frac{t-s'}{\sqrt{2h^2}}\right) \end{aligned} \tag{1.21}$$

Note that our D-dimensional multivariate Gaussian kernel can be expressed as a product of D one-dimensional Gaussian kernel. Similarly, the multidimensional Hermite functions can be written as a product of one-dimensional Hermite functions using the following identity for any $t \in \mathbb{R}^{\mathbb{D}}$.

$$\begin{aligned} H_\alpha(t) &= H_{\alpha_1}(t_1) \cdots H_{\alpha_D}(t_D) \\ h_\alpha(t) &= e^{-||t^2||} H_\alpha(t) = h_{\alpha_1}(t_1) \cdots h_{\alpha_D}(t_D) \end{aligned} \tag{1.22}$$

where $||t^2|| = t_1^2 + \cdots + t_D^2$.

$$\begin{aligned} e^{\frac{-||t-s||^2}{2h^2}} &= e^{\frac{-(t_1-s_1)^2-(t_2-s_2)^2-\cdots-(t_D-s_D)^2}{2h^2}} \\ &= e^{\frac{-(t_1-s_1)^2}{2h^2}} e^{\frac{-(t_2-s_2)^2}{2h^2}} \cdots e^{\frac{-(t_D-s_D)^2}{2h^2}} \end{aligned} \tag{1.23}$$

Therefore, we can express the multivariate Gaussian kernel as

$$\begin{aligned} e^{\frac{-||t-s||^2}{2h^2}} &= \left(\sum_{n_1=0}^{\infty} \frac{1}{n_1!} \left(\frac{s_1-s'_1}{\sqrt{2h^2}}\right)^{n_1} h_{n_1}\left(\frac{t_1-s'_1}{\sqrt{2h^2}}\right)\right) \cdots \left(\sum_{n_D=0}^{\infty} \frac{1}{n_D!} \left(\frac{s_D-s'_D}{\sqrt{2h^2}}\right)^{n_D} h_{n_D}\left(\frac{t_D-s'_D}{\sqrt{2h^2}}\right)\right) \\ &= \sum_{\alpha \geq 0} \frac{1}{\alpha!} \left(\frac{s-s'}{\sqrt{2h^2}}\right)^{\alpha} h_\alpha\left(\frac{t-s'}{\sqrt{2h^2}}\right) \end{aligned} \tag{1.24}$$

By replacing t and s with our familiar query/reference point pair x_q, x_r, x_Q, x_R, we have

$$e^{\frac{-||x_q-x_r||^2}{2h^2}} = \sum_{\alpha \geq 0} \frac{1}{\alpha!} \left(\frac{x_r-x_R}{\sqrt{2h^2}}\right)^{\alpha} h_\alpha\left(\frac{x_q-x_R}{\sqrt{2h^2}}\right) \tag{1.25}$$

1.25 lets us evaluate the kernel value for a query/reference point pair x_q and x_r, $e^{\frac{-||x_q-x_r||^2}{2h^2}}$ as an *Hermite (multipole or reference-side)* expansion centered at a representative point x_R in the reference node.

Consider the following *unnormalized* Gaussian sum for a given query point x_q. By switching the summation signs, we obtain the *multipole* moements in a bracket. These momenets are only dependent on the reference data points and the representative centroid x_R in the reference node, so they can be computed when the tree/grid structure is constructed for the reference dataset.

$$\begin{aligned} G(x_q) &= \sum_{r=1}^{N_R} e^{\frac{-||x_q-x_r||^2}{2h^2}} \\ &= \sum_{r=1}^{N_R} \sum_{\alpha \geq 0} \frac{1}{\alpha!} \left(\frac{x_r - x_R}{\sqrt{2h^2}} \right)^{\alpha} h_{\alpha} \left(\frac{x_q - x_R}{\sqrt{2h^2}} \right) \\ &= \sum_{\alpha \geq 0} \left[\sum_{r=1}^{N_R} \frac{1}{\alpha!} \left(\frac{x_r - x_R}{\sqrt{2h^2}} \right)^{\alpha} \right] h_{\alpha} \left(\frac{x_q - x_R}{\sqrt{2h^2}} \right) \end{aligned} \tag{1.26}$$

We note that 1.25 can be re-written as

$$e^{\frac{-||x_q-x_r||^2}{2h^2}} = \sum_{\alpha \geq 0} \frac{1}{\alpha!} h_{\alpha} \left(\frac{x_r - x_Q}{\sqrt{2h^2}} \right) \left(\frac{x_q - x_Q}{\sqrt{2h^2}} \right)^{\alpha} \tag{1.27}$$

and hence, we get the unnormalized Gaussian kernel sum as

$$\begin{aligned} G(x_q) &= \sum_{r=1}^{N_R} e^{\frac{-||x_q-x_r||^2}{2h^2}} \\ &= \sum_{r=1}^{N_R} \sum_{\alpha \geq 0} \frac{1}{\alpha!} h_{\alpha} \left(\frac{x_r - x_Q}{\sqrt{2h^2}} \right) \left(\frac{x_q - x_Q}{\sqrt{2h^2}} \right)^{\alpha} \\ &= \sum_{\alpha \geq 0} \left[\sum_{r=1}^{N_R} \frac{1}{\alpha!} h_{\alpha} \left(\frac{x_r - x_Q}{\sqrt{2h^2}} \right) \right] \left(\frac{x_q - x_Q}{\sqrt{2h^2}} \right)^{\alpha} \end{aligned} \tag{1.28}$$

This expresses a Gaussian kernel sum as a *Taylor (local)* expansion about a nearby representative centroid x_Q in the query node.

The final property is the recurrence relation of the one-dimensional Hermite function

$$h_{n+1}(t) = 2t \cdot h_n(t) - 2n \cdot h_{n-1}(t), t \in \mathbb{R}^1 \tag{1.29}$$

and the Taylor expansion of the Hermite function $h_\alpha(t)$ about an arbitrary point $t_0 \in \mathbb{R}^{\mathbb{D}}$.

$$h_\alpha(t) = \sum_{\beta \geq 0} \frac{(t - t_0)^\beta}{\beta!} (-1)^{|\beta|} h_{\alpha+\beta}(t_0) \tag{1.30}$$

1.5.2.3 Translation Operators

Now we have all the machinery required to build a fast method. I will state (as derived in [12, 13]) the multipole-to-local translation operator for the Gaussian kernel. In addition, I will present the derivations of the two newly invented translation operators necessary for moving the grid-based fast Gauss transform to a tree-based one. From now on, we will only consider the unnormalized part of the kernel summation for each query point, as it is trivial to multiply the unnormalized density estimate by the required factor $\frac{1}{N_R V_{Dh}}$.

Lemma 1.5.1. *Multipole-to-multipole Translation Operator for Gaussian: Suppose we are given the Hermite expansion centered at a centroid $x_{R'}$ in a reference node $X_{R'}$:*

$$G(x_q) = \sum_{\alpha \geq 0} A'_\alpha h_\alpha \left(\frac{x_q - x_{R'}}{\sqrt{2h^2}} \right)$$

Then, this same multipole expansion shifted to a new location x_R of the parent node of X_R is given by the following:

$$G(x_q) = \sum_{\gamma \geq 0} A_\gamma h_\gamma \left(\frac{x_q - x_R}{\sqrt{2h^2}} \right)$$

where $A_\gamma = \sum\limits_{0 \leq \alpha \leq \gamma} \frac{1}{(\gamma-\alpha)!} A'_\alpha \left(\frac{x_{R'} - x_R}{\sqrt{2h^2}} \right)^{\gamma-\alpha}$.

Proof. We simply replace the Hermite function part of the multipole expansion by a new Taylor series.

$$
\begin{aligned}
G(x_q) &= \sum_{\alpha\geq 0} A'_\alpha h_\alpha \left(\frac{x_q - x_{R'}}{\sqrt{2h^2}}\right) \\
&= \sum_{\alpha\geq 0} A'_\alpha \sum_{\beta\geq 0} \frac{1}{\beta!} \left(\frac{x_R - x_{R'}}{\sqrt{2h^2}}\right)^\beta (-1)^{|\beta|} h_{\alpha+\beta} \left(\frac{x_q - x_R}{\sqrt{2h^2}}\right) \\
&= \sum_{\alpha\geq 0} \sum_{\beta\geq 0} A'_\alpha \frac{1}{\beta!} \left(\frac{x_R - x_{R'}}{\sqrt{2h^2}}\right)^\beta (-1)^{|\beta|} h_{\alpha+\beta} \left(\frac{x_q - x_R}{\sqrt{2h^2}}\right) \\
&= \sum_{\alpha\geq 0} \sum_{\beta\geq 0} A'_\alpha \frac{1}{\beta!} \left(\frac{x_{R'} - x_R}{\sqrt{2h^2}}\right)^\beta h_{\alpha+\beta} \left(\frac{x_q - x_R}{\sqrt{2h^2}}\right) \\
&= \sum_{\gamma\geq 0} \left[\sum_{0\leq\alpha\leq\gamma} \frac{1}{(\gamma-\alpha)!} A'_\alpha \left(\frac{x_{R'} - x_R}{\sqrt{2h^2}}\right)^{\gamma-\alpha}\right] h_\gamma \left(\frac{x_q - x_R}{\sqrt{2h^2}}\right)
\end{aligned}
$$

where $\gamma = \alpha + \beta$. □

Lemma 1.5.2. *Multipole-to-local Translation Operator for Gaussian (as presented in Lemma 2.2 in [12, 13]): Suppose we are given a reference node X_R and a query node X_Q, and given the Hermite (multipole) expansion centered at a centroid x_R of X_R:*

$$G(x_q) = \sum_{\alpha\geq 0} A_\alpha h_\alpha \left(\frac{x_q - x_R}{\sqrt{2h^2}}\right)$$

Then, the Taylor expansion of the Hermite expansion at the centroid x_Q of the query node X_Q is given by:

$$G(x_q) = \sum_{\beta\geq 0} B_\beta \left(\frac{x_q - x_Q}{\sqrt{2h^2}}\right)^\beta \tag{1.31}$$

where $B_\beta = \frac{(-1)^{|\beta|}}{\beta!} \sum\limits_{\alpha\geq 0} A_\alpha h_{\alpha+\beta} \left(\frac{x_Q - x_R}{\sqrt{2h^2}}\right)$

Proof. We simply replace the Hermite function part of the multipole expansion by its Taylor series.

$$
\begin{aligned}
G(x_q) &= \sum_{\alpha \geq 0} A_\alpha h_\alpha \left(\frac{x_q - x_R}{\sqrt{2h^2}} \right) \\
&= \sum_{\alpha \geq 0} A_\alpha \sum_{\beta \geq 0} \frac{(-1)^{|\beta|}}{\beta!} h_{\alpha+\beta} \left(\frac{x_Q - x_R}{\sqrt{2h^2}} \right) \left(\frac{x_q - x_Q}{\sqrt{2h^2}} \right)^\beta \\
&= \sum_{\beta \geq 0} \left[\frac{(-1)^{|\beta|}}{\beta!} \sum_{\alpha \geq 0} A_\alpha h_{\alpha+\beta} \left(\frac{x_Q - x_R}{\sqrt{2h^2}} \right) \right] \left(\frac{x_q - x_Q}{\sqrt{2h^2}} \right)^\beta
\end{aligned}
$$

□

Lemma 1.5.3. *Local-to-local Translation Operator for Gaussian: Suppose you are given a Taylor expansion centered at a centroid $x_{Q'}$ of a query node $X_{Q'}$:*

$$
G(x_q) = \sum_{\beta \geq 0} B_\beta \left(\frac{x_q - x_{Q'}}{\sqrt{2h^2}} \right)^\beta
$$

Then, the Taylor expansion obtained by shifting this expansion to the new centroid x_Q of the child node X_Q is:

$$
G(x_q) = \sum_{\alpha \geq 0} \left[\sum_{\beta \geq \alpha} \frac{\beta!}{\alpha!(\beta - \alpha)!} B_\beta \left(\frac{x_Q - x_{Q'}}{\sqrt{2h^2}} \right)^{\beta - \alpha} \right] \left(\frac{x_q - x_Q}{\sqrt{2h^2}} \right)^\alpha
$$

Proof. We expand the part involving x_q and $x_{Q'}$ in a new Taylor series centered at a new center x_Q. Then, we obtain:

$$\begin{aligned}
G(x_q) &= \sum_{\beta \geq 0} B_\beta \left(\frac{x_q - x_{Q'}}{\sqrt{2h^2}} \right)^\beta \\
&= \sum_{\beta \geq 0} B_\beta \sum_{\alpha \geq 0} \frac{1}{\alpha!} \left(D^\alpha \left[\left(\frac{x_q - x_{Q'}}{\sqrt{2h^2}} \right)^\beta \right] (x_Q) \right) (x_q - x_Q)^\alpha \\
&= \sum_{\beta \geq 0} B_\beta \sum_{\alpha \leq \beta} \frac{1}{\alpha!} \left(\frac{1}{\sqrt{2h^2}} \right)^\alpha \left(\prod_{d=1}^{D} \beta_D(\beta_D - 1) \cdots (\beta_D - \alpha_D + 1) \right) \left(\frac{x_Q - x_{Q'}}{\sqrt{2h^2}} \right)^\beta (x_q - x_Q)^\alpha \\
&= \sum_{\beta \geq 0} \sum_{\alpha \leq \beta} B_\beta \frac{\beta!}{\alpha!(\beta - \alpha)!} \left(\frac{x_Q - x_{Q'}}{\sqrt{2h^2}} \right)^{\beta - \alpha} \left(\frac{x_q - x_Q}{\sqrt{2h^2}} \right)^\alpha \\
&= \sum_{\alpha \geq 0} \left[\sum_{\beta \geq \alpha} B_\beta \frac{\beta!}{\alpha!(\beta - \alpha)!} \left(\frac{x_Q - x_{Q'}}{\sqrt{2h^2}} \right)^{\beta - \alpha} \right] \left(\frac{x_q - x_Q}{\sqrt{2h^2}} \right)^\alpha
\end{aligned}$$

$\square$

1.5.2.4 Error Bound on Translation Operators

Because the multipole and the Taylor expansion are truncated after taking p^D terms, we incur an error in approximation. The original error bounds for the Gaussian kernel in [12, 13] were wrong and corrections have been made in [4]. The authors in [4] presented corrections to one of the error bounds presented in [12]. Here, I will present all necessary three error bounds incurred in performing translation operators, incorporating an improved upper bound of one-dimensional Hermite functions [26].

$$\frac{1}{n!}|h_n(x)| \leq \frac{2^{\frac{n}{2}}}{\sqrt{n!}} e^{\frac{-x^2}{2}}, n \geq 0, x \in \mathbb{R}^1 \tag{1.32}$$

Lemma 1.5.4. Error Bound for Truncating a Hermite (Multipole) Expansion (as presented in [4])*: Suppose we are given a multipole expansion of a reference node about its centroid x_R.*

$$G(x_q) = \sum_{\alpha \geq 0} A_\alpha h_\alpha \left(\frac{x_q - x_R}{\sqrt{2h^2}} \right)$$

where $A_\alpha = \sum\limits_{r=1}^{N_R} \frac{1}{\alpha!} \left(\frac{x_r - x_R}{\sqrt{2h^2}} \right)^\alpha$.

For a fixed query point x_q, the error due to truncating the series after the first p^D term is

$$|\epsilon_M(p)| \leq \frac{N_R}{(1-r)^D} \sum_{k=0}^{D-1} \binom{D}{k} (1-r^p)^k \left(\frac{r^p}{\sqrt{p!}}\right)^{D-k}$$

where each reference data point x_r in the reference node satisfies $||x_r - x_R||_\infty < rh$ for $r < 1$.

Proof. We expand the Hermite expansion as a product of one-dimensional Hermite functions by using the following notations for $1 \leq i \leq D$.

$$\begin{aligned}
u_p(x_{q_i}, x_{r_i}, x_{R_i}) &= \sum_{n_i=0}^{p-1} \frac{1}{n_i!} \left(\frac{x_{r_i} - x_{R_i}}{\sqrt{2h^2}}\right)^{n_i} h_{n_i}\left(\frac{x_{q_i} - x_{R_i}}{\sqrt{2h^2}}\right) \\
v_p(x_{q_i}, x_{r_i}, x_{R_i}) &= \sum_{n_i=p}^{\infty} \frac{1}{n_i!} \left(\frac{x_{r_i} - x_{R_i}}{\sqrt{2h^2}}\right)^{n_i} h_{n_i}\left(\frac{x_{q_i} - x_{R_i}}{\sqrt{2h^2}}\right) \\
e^{\frac{-||x_q - x_r||^2}{2h^2}} &= \prod_{i=1}^{D} \left(u_p(x_{q_i}, x_{r_i}, x_{R_i}) + v_p(x_{q_i}, x_{r_i}, x_{R_i})\right)
\end{aligned}$$

Using 1.32, we obtain for $1 \leq i \leq D$

$$\begin{aligned}
u_p(x_{q_i}, x_{r_i}, x_{R_i}) &\leq \sum_{n_i=0}^{p-1} \frac{1}{n_i!} \left|\frac{x_{r_i} - x_{R_i}}{\sqrt{2h^2}}\right|^{n_i} \left|h_{n_i}\left(\frac{x_{q_i} - x_{R_i}}{\sqrt{2h^2}}\right)\right| \leq \sum_{n_i=0}^{p-1} \left|\frac{rh}{\sqrt{2h^2}}\right|^{n_i} \frac{2^{\frac{n_i}{2}}}{\sqrt{n_i!}} \left(e^{-\frac{(x_{q_i} - x_{R_i})^2}{4h^2}}\right) \\
&\leq \sum_{n_i=0}^{p-1} r^{n_i} \leq \frac{1-r^p}{1-r} \\
v_p(x_{q_i}, x_{r_i}, x_{R_i}) &\leq \sum_{n_i=p}^{\infty} \frac{1}{n_i!} \left|\frac{x_{r_i} - x_{R_i}}{\sqrt{2h^2}}\right|^{n_i} \left|h_{n_i}\left(\frac{x_{q_i} - x_{R_i}}{\sqrt{2h^2}}\right)\right| \leq \sum_{n_i=p}^{\infty} \left|\frac{rh}{\sqrt{2h^2}}\right|^{n_i} \frac{2^{\frac{n_i}{2}}}{\sqrt{n_i!}} \left(e^{-\frac{(x_{q_i} - x_{R_i})^2}{4h^2}}\right) \\
&\leq \frac{1}{\sqrt{p!}} \sum_{n_i=p}^{\infty} r^{n_i} \leq \frac{1}{\sqrt{p!}} \frac{r^p}{1-r}
\end{aligned}$$

Therefore,

$$\begin{aligned}
\left|e^{\frac{-||x_q - x_r||^2}{2h^2}} - \prod_{i=1}^{D} u_p(x_{q_i}, x_{r_i}, x_{R_i})\right| &\leq (1-r)^{-D} \sum_{k=0}^{D-1} \binom{D}{k} (1-r^p)^k \left(\frac{r^p}{\sqrt{p!}}\right)^{D-k} \\
\left|\sum_{r=1}^{N_R} e^{\frac{-||x_q - x_r||^2}{2h^2}} - \sum_{\alpha < p} A_\alpha h_\alpha \left(\frac{x_q - x_R}{\sqrt{2h^2}}\right)\right| &\leq \frac{N_R}{(1-r)^D} \sum_{k=0}^{D-1} \binom{D}{k} (1-r^p)^k \left(\frac{r^p}{\sqrt{p!}}\right)^{D-k}
\end{aligned}$$

$\square$

Lemma 1.5.5. Error Bound for Truncating a Taylor Expansion Converted from a Hermite Expansion of Infinite Order*: Suppose we are given the following Taylor expansion about the centroid x_Q of a query node*

$$G(x_q) = \sum_{\beta \geq 0} B_\beta \left(\frac{x_q - x_Q}{\sqrt{2h^2}} \right)^\beta$$

where $B_\beta = \frac{(-1)^{|\beta|}}{\beta!} \sum_{\alpha \geq 0} A_\alpha h_{\alpha+\beta} \left(\frac{x_Q - x_R}{\sqrt{2h^2}} \right)$ *and A_α's are the coefficients of the Hermite expansion centered at the reference node centroid x_R.*

Truncating the series after p^D terms satisfies the error bound

$$|\epsilon_L(p)| \leq \frac{N_R}{(1-r)^D} \sum_{k=0}^{D-1} \binom{D}{k} (1 - r^p)^k \left(\frac{r^p}{\sqrt{p!}} \right)^{D-k}$$

where $||x_q - x_Q||_\infty < rh$ for $r < 1$.

Proof. Note from the definition of Taylor expansion of Hermite function that

$$\begin{aligned}
e^{\frac{-||x_q - x_r||^2}{2h^2}} &= \sum_{\beta \geq 0} \frac{(-1)^{|\beta|}}{\beta!} \sum_{\alpha \geq 0} \frac{1}{\alpha!} \left(\frac{x_r - x_R}{\sqrt{2h^2}} \right)^\alpha h_{\alpha+\beta} \left(\frac{x_Q - x_R}{\sqrt{2h^2}} \right) \left(\frac{x_q - x_Q}{\sqrt{2h^2}} \right)^\beta \\
&= \sum_{\beta \geq 0} \frac{(-1)^{|\beta|}}{\beta!} \sum_{\alpha \geq 0} \frac{1}{\alpha!} \left(\frac{x_R - x_r}{\sqrt{2h^2}} \right)^\alpha (-1)^{|\alpha|} h_{\alpha+\beta} \left(\frac{x_Q - x_R}{\sqrt{2h^2}} \right) \left(\frac{x_q - x_Q}{\sqrt{2h^2}} \right)^\beta \\
&= \sum_{\beta \geq 0} \frac{(-1)^{|\beta|}}{\beta!} h_\beta \left(\frac{x_Q - x_r}{\sqrt{2h^2}} \right) \left(\frac{x_q - x_Q}{\sqrt{2h^2}} \right)^\beta
\end{aligned}$$

As done in the proof of the truncation error for multipole expansion, we define the following univariate functional notations such that $e^{\frac{-||x_q - x_r||^2}{2h^2}} = \prod_{i=1}^{D} (u_p(x_{q_i}, x_{r_i}, x_{Q_i}) + v_p(x_{q_i}, x_{r_i}, x_{Q_i}))$ for $1 \leq i \leq D$.

$$\begin{aligned}
u_p(x_{q_i}, x_{r_i}, x_{Q_i}) &= \sum_{n_i=0}^{p-1} \frac{(-1)^{n_i}}{n_i!} h_{n_i} \left(\frac{x_{Q_i} - x_{r_i}}{\sqrt{2h^2}} \right) \left(\frac{x_{q_i} - x_{Q_i}}{\sqrt{2h^2}} \right)^{n_i} \\
v_p(x_{q_i}, x_{r_i}, x_{Q_i}) &= \sum_{n_i=p}^{\infty} \frac{(-1)^{n_i}}{n_i!} h_{n_i} \left(\frac{x_{Q_i} - x_{r_i}}{\sqrt{2h^2}} \right) \left(\frac{x_{q_i} - x_{Q_i}}{\sqrt{2h^2}} \right)^{n_i}
\end{aligned}$$

These two functions satisfy the following bounds for $1 \leq i \leq D$:

$$u_p(x_{q_i}, x_{r_i}, x_{Q_i}) \leq \frac{1 - r^p}{1 - r}$$
$$v_p(x_{q_i}, x_{r_i}, x_{Q_i}) \leq \frac{1}{\sqrt{p!}} \frac{r^p}{1 - r}$$

So the error bound is the same as the one obtained in 1.5.4. □

Lemma 1.5.6. Error Bound for Truncating a Taylor Expansion Converted from an Already Truncated Hermite Expansion*: A truncated Hermite expansion centered about the centroid x_R of a reference node*

$$G(x_q) = \sum_{\alpha < p} A_\alpha h_\alpha \left(\frac{x_q - x_R}{\sqrt{2h^2}} \right)$$

has the following Taylor expansion about the centroid x_Q of a query node:

$$G(x_q) = \sum_{\beta \geq 0} C_\beta \left(\frac{x_q - x_Q}{\sqrt{2h^2}} \right)^\beta$$

where the coefficients C_β are given by

$$C_\beta = \frac{(-1)^{|\beta|}}{\beta!} \sum_{\alpha < p} A_\alpha h_{\alpha + \beta} \left(\frac{x_Q - x_R}{\sqrt{2h^2}} \right)$$

Truncating the series after p^D terms satisfies the error bound

$$|\epsilon_L(p)| \leq \frac{N_R}{(1 - 2r)^{2D}} \sum_{k=0}^{D-1} \binom{D}{k} ((1 - (2r)^p)^2)^k \left(\frac{((2r)^p)(2 - (2r)^p)}{\sqrt{p!}} \right)^{D-k}$$

for a query node X_Q for which $||x_q - x_Q||_\infty < rh$, and a reference node X_R for which $\forall x_r \in x_R, ||x_r - x_R||_\infty < rh$ for $r < \frac{1}{2}$.

Proof. We define the following for $1 \leq i \leq D$:

$$u_p(x_{q_i}, x_{r_i}, x_{Q_i}, x_{R_i}) = \sum_{n_i=0}^{p-1} \frac{(-1)^{n_i}}{n_i!} \sum_{n_j=0}^{p-1} \frac{1}{n_j!} \left(\frac{x_{R_i} - x_{r_i}}{\sqrt{2h^2}} \right)^{n_j} (-1)^{n_j} h_{n_i+n_j} \left(\frac{x_{Q_i} - x_{R_i}}{\sqrt{2h^2}} \right) \left(\frac{x_{q_i} - x_{Q_i}}{\sqrt{2h^2}} \right)^{n_i}$$

$$v_p(x_{q_i}, x_{r_i}, x_{Q_i}, x_{R_i}) = \sum_{n_i=0}^{p-1} \frac{(-1)^{n_i}}{n_i!} \sum_{n_j=p}^{\infty} \frac{1}{n_j!} \left(\frac{x_{R_i} - x_{r_i}}{\sqrt{2h^2}} \right)^{n_j} (-1)^{n_j} h_{n_i+n_j} \left(\frac{x_{Q_i} - x_{R_i}}{\sqrt{2h^2}} \right) \left(\frac{x_{q_i} - x_{Q_i}}{\sqrt{2h^2}} \right)^{n_i}$$

$$w_p(x_{q_i}, x_{r_i}, x_{Q_i}, x_{R_i}) = \sum_{n_i=p}^{\infty} \frac{(-1)^{n_i}}{n_i!} \sum_{n_j=0}^{\infty} \frac{1}{n_j!} \left(\frac{x_{R_i} - x_{r_i}}{\sqrt{2h^2}} \right)^{n_j} (-1)^{n_j} h_{n_i+n_j} \left(\frac{x_{Q_i} - x_{R_i}}{\sqrt{2h^2}} \right) \left(\frac{x_{q_i} - x_{Q_i}}{\sqrt{2h^2}} \right)^{n_i}$$

Note that $e^{\frac{-||x_q - x_r||^2}{2h^2}} = \prod_{i=1}^{D} \left(u_p(x_{q_i}, x_{r_i}, x_{Q_i}, x_{R_i}) + v_p(x_{q_i}, x_{r_i}, x_{Q_i}, x_{R_i}) + w_p(x_{q_i}, x_{r_i}, x_{Q_i}, x_{R_i}) \right)$ for $1 \leq i \leq D$.

Using the bound for Hermite functions and the property of geometric series, we obtain the following upper bounds

$$\begin{aligned}
u_p(x_{q_i}, x_{r_i}, x_{Q_i}, x_{R_i}) &\leq \sum_{n_i=0}^{p-1} \frac{1}{n_i!} \sum_{n_j=0}^{p-1} \frac{1}{n_j!} \left| \frac{x_{R_i} - x_{r_i}}{\sqrt{2h^2}} \right|^{n_j} \left| h_{n_i+n_j} \left(\frac{x_{Q_i} - x_{R_i}}{\sqrt{2h^2}} \right) \right| \left| \frac{x_{q_i} - x_{Q_i}}{\sqrt{2h^2}} \right|^{n_i} \\
&\leq \sum_{n_i=0}^{p-1} \sum_{n_j=0}^{p-1} \frac{(n_i+n_j)!}{n_i! n_j!} \left| \frac{x_{R_i} - x_{r_i}}{\sqrt{2h^2}} \right|^{n_j} \frac{1}{(n_i+n_j)!} \left| h_{n_i+n_j} \left(\frac{x_{Q_i} - x_{R_i}}{\sqrt{2h^2}} \right) \right| \left| \frac{x_{q_i} - x_{Q_i}}{\sqrt{2h^2}} \right|^{n_i} \\
&\leq \sum_{n_i=0}^{p-1} \sum_{n_j=0}^{p-1} 2^{n_i+n_j} \left(\frac{rh}{\sqrt{2h^2}} \right)^{n_j} 2^{\frac{n_i+n_j}{2}} \frac{1}{\sqrt{(n_i+n_j)!}} e^{\frac{-(x_{Q_i} - x_{R_i})^2}{4h^2}} \left(\frac{rh}{\sqrt{2h^2}} \right)^{n_i} \\
&\leq \sum_{n_i=0}^{p-1} \sum_{n_j=0}^{p-1} (2r)^{n_i} (2r)^{n_j} = \left(\frac{1 - (2r)^p)}{1 - 2r} \right)^2
\end{aligned}$$

$$\begin{aligned}
v_p(x_{q_i}, x_{r_i}, x_{Q_i}, x_{R_i}) &\leq \sum_{n_i=0}^{p-1} \sum_{n_j=p}^{\infty} 2^{n_i+n_j} \left(\frac{rh}{\sqrt{2h^2}} \right)^{n_j} 2^{\frac{n_i+n_j}{2}} \frac{1}{\sqrt{(n_i+n_j)!}} e^{\frac{-(x_{Q_i} - x_{R_i})^2}{4h^2}} \left(\frac{rh}{\sqrt{2h^2}} \right)^{n_i} \\
&\leq \frac{1}{\sqrt{p!}} \sum_{n_i=0}^{p-1} \sum_{n_j=p}^{\infty} (2r)^{n_i} (2r)^{n_j} = \frac{1}{\sqrt{p!}} \left(\frac{1 - (2r)^p}{1 - 2r} \right) \left(\frac{(2r)^p}{1 - 2r} \right)
\end{aligned}$$

$$\begin{aligned}
w_p(x_{q_i}, x_{r_i}, x_{Q_i}, x_{R_i}) &\leq \sum_{n_i=p}^{\infty} \sum_{n_j=0}^{\infty} 2^{n_i+n_j} \left(\frac{rh}{\sqrt{2h^2}} \right)^{n_j} 2^{\frac{n_i+n_j}{2}} \frac{1}{\sqrt{(n_i+n_j)!}} e^{\frac{-(x_{Q_i} - x_{R_i})^2}{4h^2}} \left(\frac{rh}{\sqrt{2h^2}} \right)^{n_i} \\
&\leq \frac{1}{\sqrt{p!}} \sum_{n_i=p}^{\infty} \sum_{n_j=0}^{\infty} (2r)^{n_i} (2r)^{n_j} = \frac{1}{\sqrt{p!}} \left(\frac{1}{1 - 2r} \right) \left(\frac{(2r)^p}{1 - 2r} \right)
\end{aligned}$$

Therefore,

$$\left| e^{\frac{-||x_q - x_r||^2}{2h^2}} - \prod_{i=1}^{D} u_p(x_{q_i}, x_{r_i}, x_{R_i}) \right| \leq (1-2r)^{-2D} \sum_{k=0}^{D-1} \binom{D}{k} ((1-(2r)^p)^2)^k \left(\frac{((2r)^p)(2-(2r)^p)}{\sqrt{p!}} \right)^{D-k}$$

$$\left| \sum_{r=1}^{N_R} e^{\frac{-||x_q - x_r||^2}{2h^2}} - \sum_{\beta < p} C_\beta \left(\frac{x_q - x_Q}{\sqrt{2h^2}} \right)^\beta \right| \leq \frac{N_R}{(1-2r)^{2D}} \sum_{k=0}^{D-1} \binom{D}{k} ((1-(2r)^p)^2)^k \left(\frac{((2r)^p)(2-(2r)^p)}{\sqrt{p!}} \right)^{D-k}$$

□

We note that these error bounds place limits on the size of the query node and the reference node. This is unfortunately due to the fast growth of Hermite functions (formed from taking partial derivatives of the Gaussian kernel along some coordinate directions). The error bounds are derived by expanding the multivariate Gaussian kernel as a product of one-dimensional Gaussians, and requires the tails of infinite geometric sums to converge.

1.5.4 implies that evaluating a truncated multipole expansion for a single query point requires each reference point that was used to form the expansion to be within h away from the representative point X_R. Similarly, 1.5.5 implies that evaluting a truncated Taylor expansion for a single query point requires each query point to be within h away from the representative point X_R.

1.5.6 has the strictest requirement of all, requiring the maximum side length of the query node and the reference node to be less than h. This is exactly why [12] proposed using a grid structure. It is highly likely that the error bounds derived here are quite loose, but this is the best we can do without breaking any laws of mathematics.

1.5.2.5 Another Derivation of Error Bound on Translation Operators

[24] had an interesting idea of using Stirling's formula to lift the node size constraint by truncating the multipole/Taylor series after taking enough terms. Recall that Stirling's formula implies for any non-negative integer n:

$$\left(\frac{n+1}{e} \right)^n \leq n! \tag{1.33}$$

This might be of a potential help to develop a full-fledged tree-based fast Gauss transform method. Unfortunately, the error bounds developed in [24] were also wrong. I will provide the necessary corrections for using three translation operators based on the techniques introduced in [4].

Lemma 1.5.7. Error Bound for Truncating a Hermite (Multipole) Expansion*: Suppose we are given a multipole expansion of a reference node about its centroid x_R.*

$$G(x_q) = \sum_{\alpha \geq 0} A_\alpha h_\alpha \left(\frac{x_q - x_R}{\sqrt{2h^2}} \right)$$

where $A_\alpha = \sum_{r=1}^{N_R} \frac{1}{\alpha!} \left(\frac{x_r - x_R}{\sqrt{2h^2}} \right)^\alpha$.

For a fixed query point x_q, the error due to truncating the series after the first p^D term is

$$|\epsilon_M(p)| \leq N_R \sum_{k=0}^{D-1} \binom{D}{k} \left(\frac{1 - {r_r}^p}{1 - r_r} \right)^k \left(\frac{{r_{rr}}^p}{1 - r_{rr}} \right)^{D-k}$$

where each reference data point x_r in the reference node satisfies $||x_r - x_R||_\infty \leq r_r h$ and $r_{rr} = \frac{r_r \sqrt{e}}{\sqrt{p+1}} < 1$ for sufficiently high p.

Proof. We use the same notation as in 1.5.4.

$$\begin{aligned} u_p(x_{q_i}, x_{r_i}, x_{R_i}) &\leq \sum_{n_i=0}^{p-1} \frac{1}{n_i!} \left| \frac{x_{r_i} - x_{R_i}}{\sqrt{2h^2}} \right|^{n_i} \left| h_{n_i} \left(\frac{x_{q_i} - x_{R_i}}{\sqrt{2h^2}} \right) \right| \leq \sum_{n_i=0}^{p-1} \left| \frac{r_r h}{\sqrt{2h^2}} \right|^{n_i} \frac{2^{\frac{n_i}{2}}}{\sqrt{n_i!}} \left(e^{-\frac{(x_{q_i} - x_{R_i})^2}{4h^2}} \right) \\ &\leq \sum_{n_i=0}^{p-1} {r_r}^{n_i} \leq \frac{1 - {r_r}^p}{1 - r_r} \\ v_p(x_{q_i}, x_{r_i}, x_{R_i}) &\leq \sum_{n_i=p}^{\infty} \frac{1}{n_i!} \left| \frac{x_{r_i} - x_{R_i}}{\sqrt{2h^2}} \right|^{n_i} \left| h_{n_i} \left(\frac{x_{q_i} - x_{R_i}}{\sqrt{2h^2}} \right) \right| \leq \sum_{n_i=p}^{\infty} \left| \frac{rh}{\sqrt{2h^2}} \right|^{n_i} \frac{2^{\frac{n_i}{2}}}{\sqrt{n_i!}} \left(e^{-\frac{(x_{q_i} - x_{R_i})^2}{4h^2}} \right) \\ &\leq \sum_{n_i=p}^{\infty} \frac{{r_r}^{n_i}}{\sqrt{\left(\frac{n_i+1}{e} \right)^{n_i}}} \leq \frac{{r_{rr}}^p}{1 - r_{rr}} \end{aligned}$$

Then,

$$\left| e^{\frac{-||x_q - x_r||^2}{2h^2}} - \prod_{i=1}^{D} u_p(x_{q_i}, x_{r_i}, x_{R_i}) \right| \leq \sum_{k=0}^{D-1} \binom{D}{k} \left(\frac{1 - {r_r}^p}{1 - r_r} \right)^k \left(\frac{{r_{rr}}^p}{1 - r_{rr}} \right)^{D-k}$$

$$\left| \sum_{r=1}^{N_R} e^{\frac{-||x_q - x_r||^2}{2h^2}} - \sum_{\alpha < p} A_\alpha h_\alpha \left(\frac{x_q - x_R}{\sqrt{2h^2}} \right) \right| \leq N_R \sum_{k=0}^{D-1} \binom{D}{k} \left(\frac{1 - {r_r}^p}{1 - r_r} \right)^k \left(\frac{{r_{rr}}^p}{1 - r_{rr}} \right)^{D-k}$$

□

Lemma 1.5.8. Error Bound for Truncating a Taylor Expansion Converted from a Hermite Expansion of Infinite Order*: Suppose we are given the following Taylor expansion about the centroid x_Q of a query node*

$$G(x_q) = \sum_{\beta \geq 0} B_\beta \left(\frac{x_q - x_Q}{\sqrt{2h^2}} \right)^\beta$$

where $B_\beta = \frac{(-1)^{|\beta|}}{\beta!} \sum_{\alpha \geq 0} A_\alpha h_{\alpha+\beta} \left(\frac{x_Q - x_R}{\sqrt{2h^2}} \right)$ *and* A_α*'s are the coefficients of the Hermite expansion centered at the reference node centroid* x_R*.*

Truncating the series after p^D *terms satisfies the error bound*

$$|\epsilon_L(p)| \leq N_R \sum_{k=0}^{D-1} \binom{D}{k} \left(\frac{1 - {r_q}^p}{1 - r_q} \right)^k \left(\frac{{r_{qq}}^p}{1 - r_{qq}} \right)^{D-k}$$

where $||x_q - x_Q||_\infty \leq r_q h$ *and* $r_{qq} = \frac{r_q \sqrt{e}}{\sqrt{p+1}} < 1$ *for sufficiently high* p*.*

Proof. If we use the same notations defined in 1.5.5 and the techniques in 1.5.7, we get

$$u_p(x_{q_i}, x_{r_i}, x_{Q_i}) \leq \sum_{n_i=0}^{p-1} {r_q}^{n_i} \leq \frac{1 - {r_q}^p}{1 - r_q}$$

$$v_p(x_{q_i}, x_{r_i}, x_{Q_i}) \leq \sum_{n_i=p}^{\infty} \frac{{r_q}^{n_i}}{\sqrt{\left(\frac{n_i+1}{e} \right)^{n_i}}} \leq \frac{{r_{qq}}^p}{1 - r_{qq}}$$

The derivation of the error bounds follows as a result. □

Lemma 1.5.9. Error Bound for Truncating a Taylor Expansion Converted from an Already Truncated Hermite Expansion*: A truncated Hermite expansion centered about the centroid* x_R *of a reference node*

$$G(x_q) = \sum_{\alpha < p} A_\alpha h_\alpha \left(\frac{x_q - x_R}{\sqrt{2h^2}} \right)$$

has the following Taylor expansion about the centroid x_Q *of a query node:*

$$G(x_q) = \sum_{\beta \geq 0} C_\beta \left(\frac{x_q - x_Q}{\sqrt{2h^2}} \right)^\beta$$

where the coefficients C_β are given by

$$C_\beta = \frac{(-1)^{|\beta|}}{\beta!} \sum_{\alpha<p} A_\alpha h_{\alpha+\beta}\left(\frac{x_Q - x_R}{\sqrt{2h^2}}\right)$$

Truncating the series after p^D terms satisfies the error bound

$$|\epsilon_L(p)| \le N_R \sum_{k=0}^{D-1} \binom{D}{k} \left(\left(\frac{1-(2r_q)^p)}{1-2r_q}\right)\left(\frac{1-(2r_r)^p)}{1-2r_r}\right)\right)^k$$
$$\left(\left(\frac{1-(2r_q)^p}{1-2r_q}\right)\left(\frac{(r_{rr})^p}{1-r_{rr}}\right) + \left(\frac{{r_{rr}}^p}{1-r_{rr}} + \frac{1-(2r_r)^p}{1-2r_r}\right)\left(\frac{{r_{qq}}^p}{1-r_{qq}}\right)\right)^{D-k}$$

for a query node X_Q for which $\forall x_q \in X_Q, ||x_q - x_Q||_\infty \le r_q h$, and a reference node X_R for which $\forall x_r \in X_R, ||x_r - x_R||_\infty \le r_r h$, and $r_{qq} = \frac{2r_q\sqrt{e}}{\sqrt{p+1}} < 1$, $r_{rr} = \frac{2r_r\sqrt{e}}{\sqrt{p+1}} < 1$ for sufficiently high p.

Proof. We use the same notations used in 1.5.6.

$$\begin{aligned}
u_p(x_{q_i}, x_{r_i}, x_{Q_i}, x_{R_i}) &\le \sum_{n_i=0}^{p-1} \frac{1}{n_i!} \sum_{n_j=0}^{p-1} \frac{1}{n_j!} \left|\frac{x_{R_i} - x_{r_i}}{\sqrt{2h^2}}\right|^{n_j} \left|h_{n_i+n_j}\left(\frac{x_{Q_i} - x_{R_i}}{\sqrt{2h^2}}\right)\right| \left|\frac{x_{q_i} - x_{Q_i}}{\sqrt{2h^2}}\right|^{n_i} \\
&\le \sum_{n_i=0}^{p-1} \sum_{n_j=0}^{p-1} \frac{(n_i+n_j)!}{n_i! n_j!} \left|\frac{x_{R_i} - x_{r_i}}{\sqrt{2h^2}}\right|^{n_j} \frac{1}{(n_i+n_j)!} \left|h_{n_i+n_j}\left(\frac{x_{Q_i} - x_{R_i}}{\sqrt{2h^2}}\right)\right| \left|\frac{x_{q_i} - x_{Q_i}}{\sqrt{2h^2}}\right|^{n_i} \\
&\le \sum_{n_i=0}^{p-1} \sum_{n_j=0}^{p-1} 2^{n_i+n_j} \left(\frac{r_r h}{\sqrt{2h^2}}\right)^{n_j} 2^{\frac{n_i+n_j}{2}} \frac{1}{\sqrt{(n_i+n_j)!}} e^{\frac{-(x_{Q_i} - x_{R_i})^2}{4h^2}} \left(\frac{r_q h}{\sqrt{2h^2}}\right)^{n_i} \\
&\le \sum_{n_i=0}^{p-1} \sum_{n_j=0}^{p-1} (2r_q)^{n_i} (2r_r)^{n_j} = \left(\frac{1-(2r_q)^p)}{1-2r_q}\right)\left(\frac{1-(2r_r)^p)}{1-2r_r}\right)
\end{aligned}$$

$$\begin{aligned}
v_p(x_{q_i}, x_{r_i}, x_{Q_i}, x_{R_i}) &\le \sum_{n_i=0}^{p-1} \sum_{n_j=p}^{\infty} 2^{n_i+n_j} \left(\frac{r_r h}{\sqrt{2h^2}}\right)^{n_j} 2^{\frac{n_i+n_j}{2}} \frac{1}{\sqrt{(n_i+n_j)!}} e^{\frac{-(x_{Q_i} - x_{R_i})^2}{4h^2}} \left(\frac{r_q h}{\sqrt{2h^2}}\right)^{n_i} \\
&\le \sum_{n_i=0}^{p-1} \sum_{n_j=p}^{\infty} 2^{n_i+n_j} \left(\frac{r_r h}{\sqrt{2h^2}}\right)^{n_j} 2^{\frac{n_i+n_j}{2}} \frac{1}{\sqrt{n_i!}\sqrt{n_j!}} e^{\frac{-(x_{Q_i} - x_{R_i})^2}{4h^2}} \left(\frac{r_q h}{\sqrt{2h^2}}\right)^{n_i} \\
&\le \sum_{n_i=0}^{p-1} \sum_{n_j=p}^{\infty} (2r_q)^{n_i} \left(\frac{2r_r}{\sqrt{\frac{n_j+1}{e}}}\right)^{n_j} = \left(\frac{1-(2r_q)^p}{1-2r_q}\right)\left(\frac{(r_{rr})^p}{1-r_{rr}}\right)
\end{aligned}$$

$$
\begin{aligned}
w_p(x_{q_i}, x_{r_i}, x_{Q_i}, x_{R_i}) &\leq \sum_{n_i=p}^{\infty} \sum_{n_j=0}^{\infty} 2^{n_i+n_j} \left(\frac{r_r h}{\sqrt{2h^2}}\right)^{n_j} 2^{\frac{n_i+n_j}{2}} \frac{1}{\sqrt{(n_i+n_j)!}} e^{\frac{-(x_{Q_i}-x_{R_i})^2}{4h^2}} \left(\frac{r_q h}{\sqrt{2h^2}}\right)^{n_i} \\
&\leq \sum_{n_i=p}^{\infty} \sum_{n_j=0}^{\infty} 2^{n_i+n_j} \left(\frac{r_r h}{\sqrt{2h^2}}\right)^{n_j} 2^{\frac{n_i+n_j}{2}} \frac{1}{\sqrt{n_i!}\sqrt{n_j!}} \left(\frac{r_q h}{\sqrt{2h^2}}\right)^{n_i} \\
&\leq \sum_{n_i=p}^{\infty} \sum_{n_j=0}^{p-1} \frac{(2r_r)^{n_j}}{\sqrt{n_j!}} \frac{(2r_q)^{n_i}}{\sqrt{n_i!}} + \sum_{n_i=p}^{\infty} \sum_{n_j=p}^{\infty} \frac{(2r_r)^{n_j}}{\sqrt{n_j!}} \frac{(2r_q)^{n_i}}{\sqrt{n_i!}} \\
&\leq \sum_{n_i=p}^{\infty} \sum_{n_j=0}^{p-1} (2r_r)^{n_j} \left(\frac{2r_q\sqrt{e}}{\sqrt{n_i+1}}\right)^{n_i} + \sum_{n_i=p}^{\infty} \sum_{n_j=p}^{\infty} \left(\frac{2r_r\sqrt{e}}{\sqrt{n_j+1}}\right)^{n_j} \left(\frac{2r_q\sqrt{e}}{\sqrt{n_i+1}}\right)^{n_i} \\
&\leq \left(\frac{1-(2r_r)^p}{1-2r_r}\right) \left(\frac{{r_{qq}}^p}{1-r_{qq}}\right) + \left(\frac{{r_{qq}}^p}{1-r_{qq}}\right) \left(\frac{{r_{rr}}^p}{1-r_{rr}}\right)
\end{aligned}
$$

Therefore,

$$
\left| e^{\frac{-||x_q-x_r||^2}{2h^2}} - \prod_{i=1}^{D} u_p(x_{q_i}, x_{r_i}, x_{R_i}) \right| \leq \sum_{k=0}^{D-1} \binom{D}{k} \left(\left(\frac{1-(2r_q)^p)}{1-2r_q}\right)\left(\frac{1-(2r_r)^p)}{1-2r_r}\right)\right)^k
\left(\left(\frac{1-(2r_q)^p}{1-2r_q}\right)\left(\frac{(r_{rr})^p}{1-r_{rr}}\right) + \left(\frac{{r_{rr}}^p}{1-r_{rr}} + \frac{1-(2r_r)^p}{1-2r_r}\right)\left(\frac{{r_{qq}}^p}{1-r_{qq}}\right)\right)^{D-k}
$$

$$
\left| \sum_{r=1}^{N_R} e^{\frac{-||x_q-x_r||^2}{2h^2}} - \sum_{\beta<p} C_\beta \left(\frac{x_q-x_Q}{\sqrt{2h^2}}\right)^\beta \right| \leq N_R \sum_{k=0}^{D-1} \binom{D}{k} \left(\left(\frac{1-(2r_q)^p)}{1-2r_q}\right)\left(\frac{1-(2r_r)^p)}{1-2r_r}\right)\right)^k
\left(\left(\frac{1-(2r_q)^p}{1-2r_q}\right)\left(\frac{(r_{rr})^p}{1-r_{rr}}\right) + \left(\frac{{r_{rr}}^p}{1-r_{rr}} + \frac{1-(2r_r)^p}{1-2r_r}\right)\left(\frac{{r_{qq}}^p}{1-r_{qq}}\right)\right)^{D-}
$$

$\square$

1.5.2.6 The Original Fast Gauss Transform Algorithm

Finally, we present the description of the original algorithm using the mechanisms developed so far [12] in a slightly different way. Note that the algorithm developed in [12] and described here does not use all three of the translation operators I derived. In addition, we do not assume that all the points are pre-scaled to fit in the unit hypercube $[0,1]^D$. We also note that it is not necessary for the grid data structure to be discretized into hyper-cubes. Hyper-rectangles with bounded maximum side length less than the bandwidth h would suffice here.

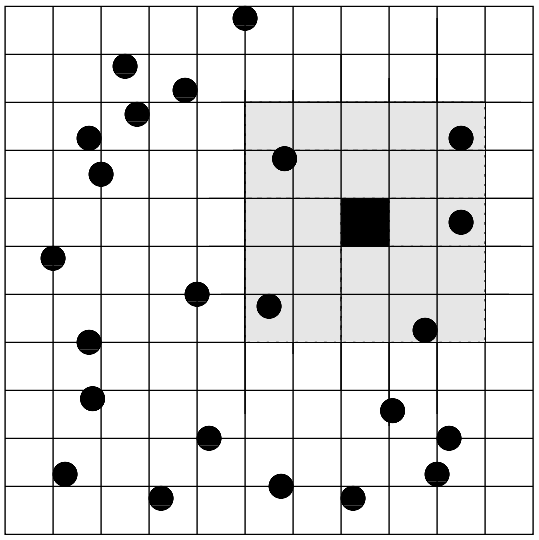


Figure 1.16: Here, the grid box containing some reference points is marked black, and its 2-ring neighboring grid boxes are included in its interaction list, for a total of 25 boxes.

The algorithm processes each "query" box X_Q in the interaction list of a given "reference" box X_R. The number of boxes in the interaction list is constant dependent only on the precision ϵ. The idea is that the far-away boxes can be eliminated from the interaction list as the Gaussian kernel decays rapidly. Details are available in [12]. Thus, we only consider n-nearest ring neighbors of each query box for a total of at most $(2n+1)^D$ boxes in the interaction list.

In the end, we want to compute the contributions of all reference points in X_R to each of the query points in X_Q. There are four possible ways in which X_R can influence X_Q.

1. *$O(DN_RN_Q)$ algorithm:* Run a naive quadratic algorithm on all pairs of query/reference points.

2. *$O(p^DN_R)$ algorithm:* Accumulate each $x_r \in X_R$ as the Taylor series about the center x_Q of X_Q using Equation 1.28.

3. *$O(p^DN_Q)$ algorithm:* Evaluate each x_q at the Hermite series centered about x_R of X_R using Equation 1.26.

4. *$O(Dp^{D+1})$ algorithm:* Convert the Hermite series centered about x_R of X_R to the Taylor series centered about x_Q of X_Q using 1.31.

Choosing an optimal strategy for a given pair of a query/reference box is determined by choosing cutoff parameters $N_F = O(p^{D-1})$ and $M_L = O(p^{D-1})$. Intuitively, the Fast Gauss transform *chooses* *the least amount of work to do at a given moment of time by analyzing the costs of each operation in advance*. We will show how we can use this idea in a tree-based algorithm.

There are four different cases and outcomes listed above depending on the number of reference points N_R and the number of query points N_Q.

1. *$N_R \leq N_F, N_Q \leq M_L$:* Run the first algorithm.
2. *$N_R \leq N_F, N_Q > M_L$:* Run the second algorithm.
3. *$N_R > N_F, N_Q \leq M_L$:* Run the third algorithm.
4. *$N_R > N_F, N_Q > M_L$:* Run the fourth algorithm.

The algorithm is divided into three clear steps as shown in Figures 1.18, 1.19, 1.20.

1.5.2.7 Free Parameters

- τ: The maximum deviation of the density estimate of each query point from the true estimate computed by the naive algorithm. Note that $\tau \neq \epsilon$, which is the *maximum percentage deviation*.

1.5.2.8 Runtime Cost

Time complexity is dominated by evaluating Taylor and multipole expansions and applying the translation operators for switching between two representations, all of which are around $O(p^D)$. The authors of [12, 13] consider all of these costs as a big constant in front of the number of query points and the reference points, hence claiming the linear complexity. Theoretically, the algorithm should perform poorly on uniformly distributed dataset (e.g. every reference point and query point fall on a different grid box) and may degrade to quadratic running time.

1.5.2.9 Space Cost

Space complexity is dominated by the space needed for storing multipole and Taylor coefficients, and the grid data structure for reference points and query points. Storing multipole and Taylor coefficients of order p for a given box is of order $O(p^D)$. However, maintaining the grid structure for highly structured/clustered dataset results in many empty grid boxes. The effects of *curse of dimensionality* dominate here.

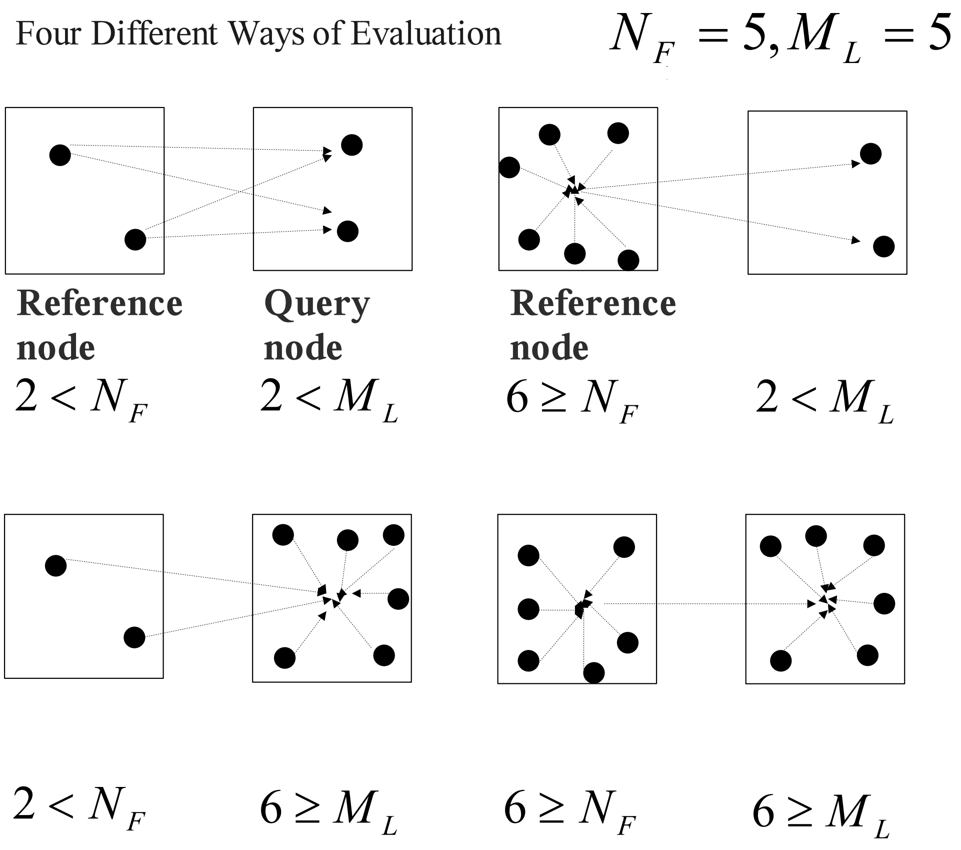


Figure 1.17: The original fast Gauss transform chooses some cutoff parameters N_F and M_L depending on the costs of creating multipole and Taylor expansions of order p. The algorithm efficiently chooses which of the four evaluation methods to use depending on the number of query points and the number of reference points. In the top left box, the naive algorithm is used. In the top right box, the Hermite expansion of the reference points is formed then evaluated at each query point. In the lower left box, each reference point is directly accumulated as the Taylor series about the center X_Q in the query node and evaluated at each query point. In the lower right box, the Hermite expansion of reference points is converted to the Taylor expansion in the query node, which is evaluated at each query point.

```
FGTPreprocess($X_Q$, $X_R$)
  vec $mincoord = (\infty, \cdots \infty)$
  vec $maxcoord = (-\infty, \cdots -\infty)$

  for each point $x_q \in X_Q$
    $\hat{p}(x_q) = 0$

  for each point $x \in X_Q \cup X_R$
    for $d = 0$ to $D - 1$ step $1$
      if mincoord[d] > x[d]
        mincoord[d] = x[d]
      if maxcoord[d] < x[d]
        maxcoord[d] = x[d]

  Create a grid $G$ with $mincoord$ and $maxcoord$ as the corner points.
  Discretize $G$ into boxes whose maximum side length is $h$.
  Initialize storage for a Taylor series with $p^D$ terms with zeros.
  Assign each point $x \in X_Q \cup X_R$ into boxes in $G$.
end
```

Figure 1.18: Each reference point and query point is assigned to a grid.

```
FGTMidStep($N_F$, $N_L$)
  for each box $B$ in $G$
    $N_R$ = number of reference points in $B$
    Form the interaction list $B_I$ of $(2n + 1)^D$ query boxes $C$ within range of $B$.
    if $N_R \leq N_F$
      for each box $C$ in $B_I$
        $N_Q$ = number of query points in $C$
        if $N_Q \leq M_L$
          for each $x_q \in C$       // Naive algorithm
            for each $x_r \in B$
              $\hat{p}(x_q) += K_h(x_q, x_r)$
        else
          Convert each $x_r \in B$ into a Taylor series about the center $x_Q$ of $C$ and
          add to Taylor series for $C$.
    else
      Form Hermite expansion about center of $x_R$ of $B$ using $x_r \in B$.
      for each box $C$ in $B_I$
        $N_Q$ = number of query points in $C$
        if $N_Q \leq M_L$
          for each $x_q \in C$       // Evaluate Hermite expansion
            $\hat{p}(x_q) +=$ Hermite expansion of $B$ evaluated at $x_q$
        else
          Convert Hermite expansion into a Taylor series about the center $x_Q$ of $C$ and a
          to Taylor series for $C$.
end
```

Figure 1.19: The interaction list of each nonempty box is processed in a pair-wise manner by using one of the four approximation methods

```
FGTPostProcess($N_F,\ N_L$)
  for each box $B$ in $G$
    $N_Q\ =\ number\ of\ query\ point\ in\ B$
    if $N_Q\ >\ M_L$
      for each point $x_q\ \in\ X_Q$
        $\hat{p}(x_q)\ +=\ Taylor\ expansion\ of\ B\ evaluated\ at\ x_q$

  for each $x_q\ \in\ x_Q$
    $\hat{p}(x_q)\ /=\ (N\ \cdot\ V_{DH})$
end
```

Figure 1.20: For each box containing query points, the Taylor expansion is evaluated at each query point and normalization factor is applied.

1.5.2.10 Error Control

[4] came up with corrections to the incorrect error bound derivations mentioned in [12, 13]. It relies on the improved bound for Hermite functions [26], but I could not obtain the original literature to verify its correctness. Nevertheless, the error bounds derived in [4] offers only an indirect error control for our purpose.

1.5.3 Summary

In this section, we have discussed fast multipole methods and their potentials in tackling the N-body problems in low dimensional settings. In particular, I have analyzed the original fast multipole method using the Gaussian kernel [12, 13] in details. Due to the nature of the kernel function, this algorithm has been limited to using a simple grid scheme. In addition to having to maintain the number of Taylor and multipole constants of order $O(p^D)$, each grid box has to maintain an exponentially large number of boxes in its interaction list.

So how do we overcome the *curse of dimensionality* in fast Gauss transform? Our idea here is to advocate the usage of trees whenever possible. I have provided the necessary additional tools for moving the original grid-based algorithm to a full-fledged tree-based one. In the next section, we will take all of these tools and develop the first tree-based fast Gauss transform. We will see how trees can be useful in developing fast N-body algorithms when combined with powerful analytical tools provided by fast multipole methods.

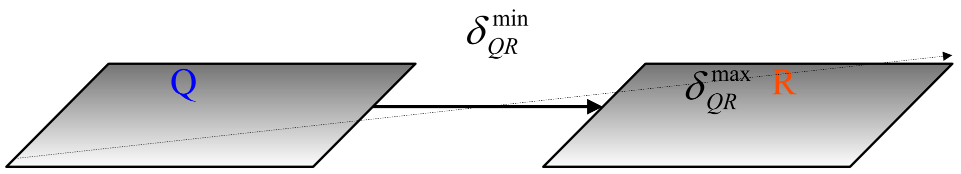


Figure 1.21: Recall that this is the canonical situation in which the query/reference node is considered.

1.6 Fast Gauss Transform using Dualtree Recursion

We are now ready to combine the techniques developed in Sections 1.5.2 and 1.4. Again, our goal is to achieve fast speed and full error control on the density estimates, by combining the best of two worlds: *computational geometry* (dual-tree) and *approximation theory* (fast multipole methods). The development in this section will be relatively shorter than the previous ones, as we already have the tools to construct our new method.

1.6.1 Data Structure

Hierarchial subdivions of the reference dataset and the query dataset are constructed using [18].

1.6.2 Algorithm

1.6.2.1 Data-adaptive Accuracy Control [11]

We can improve the accuracy of density estimates by replacing the finite-difference approximation with multipole approximation. However, one important issue must be addressed before applying the methodology of fast multipole methods in a tree-setting.

One drawback of fast-multipole methods (including fast Gauss transform) is that the algorithm overestimates the work needed to achieve the user-specified precision by fixing p (the truncation order of the multipole/Taylor expansions) in advance.

Here, we present how to choose p adaptively at each node-to-node comparison performed during dual-tree recursion. Of course, doing this eliminates the need of one of the operators we developed in the previous section (namely, the multipole-to-multipole operator) because multipole and Taylor coefficients on a given pair of query/reference nodes are computed on the fly. Nevertheless, two other operators are still useful in development of our algorithm.

1.6.2.2 Cached Sufficient Statistics

Our approach here is to store the coefficients of different orders of p (for both multipole and Taylor coefficients). For example, consider the node (which serves as both a query node and a reference node) with the following sets of multipole coefficients $mcoeffs$ and Taylor coefficients $lcoeffs$ (in the C array notation form) for a two-dimensional dataset:

$p^D = 0$	$mcoeffs[0] = NULL$	$lcoeffs[0] = NULL$
$p^D = 1$	$mcoeffs[1] = \{\mathbf{4}\}$	$lcoeffs[1] = \{3\}$
$p^D = 4$	$mcoeffs[2] = \{\mathbf{4}, \mathbf{3}, -\mathbf{5}, \mathbf{6}\}$	$lcoeffs[2] = NULL$
$p^D = 9$	$mcoeffs[3] = NULL$	$lcoeffs[3] =$ $\{1, -0.03, -0.1552, 0, -0.0005,$ $-0.1, 2, -0.5, -0.00333\}$
$p^D = 16$	$mcoeffs[4] = \{\mathbf{4}, \mathbf{3}, 2, 3, -\mathbf{5}, \mathbf{6}, \cdots\}$	$lcoeffs[4] = NULL$

Here we used $1-$ *indexed* arrays to store the coefficients. Therefore, the multipole coefficients and Taylor coefficients of order 0 are not defined. The arrays of multipole coefficients basically represent different truncation orders of the multipole expansion using Equation 1.25 formed from the reference points residing in the node. Notice that the mutipole coefficients of lower order p_{lower} appear as a subset of the ones of higher order p_{higher} because these multipole coefficients represent the *same information* but with different accuracy.

The Taylor (local) coefficients of order p are formed from two ways: either by translating from a multipole expansion of order p from another node, or by directly accumulating as a Taylor series. In contrast with the multipole coefficients stored on this node, these Taylor coefficients of different orders represent *distinct information*; each of them could account for contributions of different reference nodes. Hence, their information should be combined later. On the final post-processing staffs, all of the coefficients of different orders of p will be propagated downward to its children using *local-to-local translation operator*. Once it reaches a leaf node, each query point will be evaluated at all of the different orders of series that have accumulated above.

Most importantly, these coefficients serves as a form of cached sufficient statistics. For example, suppose our cost-model determines that the cheapest operation for the given query/reference node is translating the multipole coefficients of order p on the reference node to the Taylor coefficients of order p on the query node. Our algorithm will appropriately check whether the slot for the multipole coefficients of order p has been filled out already (by checking against $NULL$ pointer). If no multipole coefficients of order p have been formed on the reference node, they will be computed and stored, possibly serving as a "lookup-cache" for another query node and translated into Taylor coefficients. If the multipole coefficients have already been computed, there is no problem. Of course, whether we need to compute multipole coefficients or not is appropriately handled in the cost model we develop in the next part.

In practice, we place a bound on the order of the coefficients p_{limit} we store in each node so that there is a bound on the space complexity. We used $p_{limit} = 8$, above which we believe there is no improvement for $D \geq 2$.

1.6.2.3 Cost Model

Recall that node-to-node comparison in dualtree KDE is analogous to box-to-box comparison in fast Gauss transform. Recall also that the original fast Gauss transform uses one of the four evaluation strategies. We can develop the "cost-model" that simulates this on a tree-setting in a straightforward way with combined lower/upper density estimate given by the original dual-tree KDE.

Suppose that we are given a pair of a query node and a reference node. Recall that the error of the approximation using $N_R \bar{K_h}$ with respect to any query point $x_q \in X_Q$:

$$e_{QR} = \frac{N_R(K_h(\delta_{QR}^{min}) - K_h(\delta_{QR}^{max}))}{2} \tag{1.34}$$

Hence, we got the following local pruning criterion which ensures the global error tolerance ϵ:

$$|K_h(\delta_{QR}^{min}) - K_h(\delta_{QR}^{max})| \leq \frac{2\epsilon}{N_R}\phi_Q^{min} \tag{1.35}$$

If we want to replace the estimate $N_R \bar{K_h}$ with the multipole/Taylor approximation, the situation changes slightly. Namely, the total error of the approximation using the new approximation strategy is doubled.

$$e_{QR} = N_R(K_h(\delta_{QR}^{min}) - K_h(\delta_{QR}^{max})) \tag{1.36}$$

Then, we get the following local pruning criterion which ensures the global error tolerance ϵ for multipole/Taylor approximations:

$$\begin{aligned} |K_h(\delta_{QR}^{min}) - K_h(\delta_{QR}^{max})| &\leq \frac{\epsilon}{N_R}\phi_Q^{min} \\ N_R|K_h(\delta_{QR}^{min}) - K_h(\delta_{QR}^{max})| &\leq \epsilon\phi_Q^{min} \end{aligned} \tag{1.37}$$

In order to use the multipole/Taylor approximations, we simply choose enough p terms so that it is less than left side of the second inequality in Equation 1.37. We could use any of the error bounds developed in Section 1.5.2.4 or Section 1.5.2.5. Let us work with the ones in Section 1.5.2.4 for example, which places the size restriction on the query node and the reference node in consideration. Here is a simple method to determine the number of terms needed for performing a direct Hermite evaluation over all query points, doing a direct Taylor coefficient accumulation of all reference points, or invoking the *multipole-to-local translation operator*.

1. Computing the number of terms p_{DM} needed to perform a direct evaluation of Hermite series for each query point (the third algorithm in Section 1.5.2.6):
   - If the maximum side length of the reference node is at least $2h$, return MAXINT (no pruning possible)
   - Otherwise, choose the smallest $p_{DM} \geq 1$ such that: $\frac{N_R}{(1-r)^D} \sum\limits_{k=0}^{D-1} \binom{D}{k}(1 - r^{p_{DM}})^k \left(\frac{r^{p_{DM}}}{\sqrt{p_{DM}!}}\right)^{D-k} < N_R|K_h(\delta_{QR}^{min}) - K_h(\delta_{QR}^{max})|$
2. Computing the number of terms p_{DL} needed to accumulate each reference point as Taylor coefficients in the query node (the second algorithm in 1.5.2.6):
   - If the maximum side length of the query node is at least $2h$, return MAXINT (no pruning possible)
   - Otherwise, choose the smallest $p_{DL} \geq 1$ such that: $\frac{N_R}{(1-r)^D} \sum\limits_{k=0}^{D-1} \binom{D}{k}(1 - r^{p_{DL}})^k \left(\frac{r^{p_{DL}}}{\sqrt{p_{DL}!}}\right)^{D-k} < N_R|K_h(\delta_{QR}^{min}) - K_h(\delta_{QR}^{max})|$
3. Computing the number of terms p_{M2L} needed to translate from a multipole expansion to a local expansion (the fourth algorithm in 1.5.2.6):
   - If the maximum side length of the query node or the reference node is at least $2h$, return MAXINT (no pruning possible)

- Otherwise, choose the smallest $p_{M2L} \geq 1$ such that:
$\frac{N_R}{(1-2r)^{2D}} \sum_{k=0}^{D-1} \binom{D}{k} ((1-(2r)^{p_{M2L}})^2)^k \left(\frac{((2r)^{p_{M2L}})(2-(2r)^{p_{M2L}})}{\sqrt{p_{M2L}!}} \right)^{D-k} < N_R |K_h(\delta_{QR}^{min}) - K_h(\delta_{QR}^{max})|$

We are now ready to describe our cost model for choosing the "best" evaluation method. A high-level pseudocode is given in Figure 1.6.2.3. Basically, we compute the estimate cost for each of the four evaluation methods using its asymptotic complexity (ignoring the constant in front). Note that the multipole coefficient arrays on the reference node serve as a cache-lookup, and the costs of performing certain operations are appropriately adjusted if these need to be computed on the fly.

It is then straightforward to plug in this decision procedure into the original dual-tree KDE and develop our new method:

1.6.2.4 Summary

In this section, we have developed the first algorithm using multipole/Taylor approximations using dual-tree recursion. In the next chapter, we shall see how all of the methods we have discussed compare in practice.

```
procedure chooseEvaluationMethod($X_Q$, $X_R$)
  D := dimensionality of points
  $p_{DM}$ := minimum number of terms needed for direct Hermite evaluation.
  $p_{DL}$ := minimum number of terms needed for direct Taylor coefficient accumulation.
  $p_{M2L}$ := minimum number of terms needed for multipole-to-local translation operator.

  // The following four costs are minimum costs for performing
  // Hermite evaluation, direct Taylor accumulation, M2L translation,
  // and bruteforce evaluation.
  $costDM := p_{DM}^{D} \cdot N_Q$
  $costDL := p_{DL}^{D} \cdot N_R$
  $costM2L := D \cdot p_{M2L}^{D+1}$
  $costDirect := D \cdot N_Q \cdot N_R$

  If no multipole coefficient of order $p_{DM}$ has been formed on $X_R$,
    $costDM := costDM + p_{DM}^{D} \cdot N_R$

  If no multipole coefficient of order $p_{M2L}$ has been formed on $X_R$,
    $costM2L := costM2L + p_{M2L}^{D} \cdot N_R$

  If $costDM$ is the minimum of all four,
    report HermiteEvaluation
  If $costDL$ is the minimum of all four,
    report DirectTaylorAccumulation
  If $costM2L$ is the minimum of all four,
    report M2LTranslation
  Otherwise,
    report NaiveMethod
end procedure
```

Figure 1.22: Here is a simple cost model for determining which three of the four evaluation methods to use for a given query node and a reference node.

```
KDE($Q$, $R$)
  Do up/down mass propagation.
  method := chooseEvaluationMethod($Q$, $R$)

  if method == HermiteEvaluation
    Evaluate the Hermite expansion of order $p_{DM}$
    formed on $R$ at each query point in $Q$.
    return
  else if method == DirectTaylorAccumulation
    Accumulate each reference point in $R$ as the Taylor
    coefficient of order $p_{DL}$ on $Q$.
    return
  else if method == M2LTranslation
    Translate the multipole expansion of order $p_{M2L}$
    on $R$ to the Taylor expansion of the same order on $Q$.
    return

  if leaf(Q) and leaf(R)
    Run a naive quadratic algorithm on every pair of points
    in Q and R
  else
    KDE(Q.left, R.left)
    KDE(Q.left, R.right)
    KDE(Q.right, R.left)
    KDE(Q.right, R.right)
```

Figure 1.23: Straightforward way of using the decision procedure *chooseEvaluationMethod* as a subroutine in the original dual-tree pseudocode

Chapter 2

Experimental Results for KDE Algorithms

In this section, we finally get to compare the KDE algorithms discussed so far in practical settings.

Recall that the density estimate error ϵ for a query point x_q as the percentage deviation from the density estimate computed by the trivial naive algorithm. That is,

$$err = \frac{|\hat{p}_{alg}(x_q) - \hat{p}_{naive}(x_q)|}{\hat{p}_{naive}(x_q)} \tag{2.1}$$

No matter how beautiful the theoretical derivation of an algorithm is, it is considered useless if it offers no error control and performs poorly in practice. Our goal is to achieve fast speed and improved accuracy.

2.1 Implementation Languages and Machines

All the codes that we have written and obtained are written in C and C++, and was compiled under $-O6 - funroll - loops$ flags on Linux kernel 2.4.26. The experiments were run on a dual-processor AMD Opteron 242 machine with $1MB$ cache/CPU with $8GB$ of main memory. All of the experiments were memory-based with no disk-access.

2.2 Datasets used

In our experiments, we have used real-world datasets ranging from astronomy and biology.

textbfDescriptions

1. *bio5*: biology; measurements from high-throughput screening experiments.

2. *colors50k*: astronomy; colors (differences of magnitudes in different bandwidths).

3. *corel*: images; the features are the outputs of texture filters.

4. *covtype38d*: remote sensing, forest cover type data.

5. *edsgc-radec*: astronomy, RA and DEC positions of galaxies from the EDGSC survey.

6. *mockgalaxy_1M*: cosmological simulation, positions of galaxies from simulated data.

7. *sj2-50000-2*: astronomy, x and y positions of galaxies from the SDSS survey.

2.2.1 Statistics

The following table summarizes the statistics about the datasets we used. For kernel density estimation algorithms, the optimal bandwidth h^* for each dataset has been found for the Gaussian kernel function by likelihood crossvalidation.

Dataset	Number of data points	Dimensionality	h^* Gaussian
bio5	103010	5	0.079055
colors50k	50000	2	0.164755
corel	37749	32	0.219494
covtype38d	150000	38	0.0432638
edsgc-radec	1495877	2	0.00193855
mockgalaxy_D_1M	1000000	3	0.0038025
sj2-50000-2	50000	2	0.00317556

2.3 Implementation Details of Surveyed Algorithms

In summary, we have discussed the following algorithms for computing the kernel density estimate using the Gaussian kernel.

2.3.1 Multidimensional Fast Fourier Transform

I have consulted [22, 23, 27] for using fast Fourier transform in a kernel density estimation setting. The multidimensional fast Fourier transform and its inverse operation was adapted from [20, 10].

2.3.2 Dual-tree KDE

My thesis advisors have provided the code.

2.3.3 Improved Fast Gauss Transform

The authors of [29] have provided their C++ implementations of their new algorithm.

2.3.4 Fast Gauss Transform

The author made his Fortran code available online at [25], which is hard-coded for two-dimensional datasets whose points lie in the unit hypercube $[0,1]^D$. I have written my own efficient implementation with extension to higher dimensions in C and made corrections to the error bound in the author's original code.

2.3.5 Tree-based Fast Gauss Transform using Dual-tree Recursion

This is the algorithm I have developed for this thesis. I have developed several versions with weak error control and full error control. This section contains the measurements for two of the implementations with full error control.

2.4 Abbreviations for Algorithms

The following abbreviations are defined for the following 7 algorithms surveyed in this chapter.

1. *MFFT:* KDE using multidimensional fast Fourier transform
2. *IFGT:* The improved fast Gauss transform developed in [29]
3. *DTREE:* The dualtree KDE algorithm developed in [11].
4. *DFGT_AP:* tree-based fast Gauss transform algorithm using dual-recursion with auto-pruning capability. The error bound used here is derived in 1.5.2.4
5. *DFGT_AP2:* tree-based fast Gauss transform algorithm using dual-recursion with auto-pruning capability. By using another of Strain's idea in the error bound derivation 1.5.2.5, the node side constraint is lifted in this algorithm by using enough terms.
6. *FGT:* My own implementation of Greengard and Strain's original fast Gauss transform extended to handle higher dimensions. Note that the error bound used in the original implementation was wrong and was replaced with the correct derivation I have derived in 1.5.2.4.
7. *NAIVE:* A naive $O(N^2)$ algorithm. Note that the density estimates computed by this method has 0% deviation.

2.5 Kernel Density Estimation on Optimal Bandwidth

In this section, we empircally evaluate all of the algorithms when computing KDE on the optimal bandwidth.

We have bound the maximum error at three different levels: $\epsilon = 50\%, \epsilon = 10\%$, and $\epsilon = 1\%$, and required at least 95% of the data points to be within a given error level since the algorithms with weak or no error control had hard time satisfying the maximum percentage deviation ϵ for all of the query points.

2.5.1 Tweaking the Parameters

Ideally, any algorithms should be "intelligent" enough to choose the optimal bandwidths to do the required amount of work in the least amount of time possible. Unfortunately, some of the algorithms that we surveyed were overridden with "tweak parameters" that make this task impossible. This sections describes how I have devised the automated procedure of finding the optimal parameters for each dataset. Of course, finding the optimal parameters involves computing the true density estimates using the naive method and obtaining the maximum percentage deviation, but this cost is not included.

2.5.1.1 Multidimensional Fast Fourier Transform

This algorithm has put in a "tuning-mode," in which it runs until the desired precision is obtained. Both the search time (the total running time) and the running time (the elapsed time for the final run) are recorded.

The number of grid points starts at 16 (which is between 10 and 25, two numbers used in [27] for experiments). If 95% of the data points are within the desired ϵ, the algorithm terminates. Otherwise, it will double the number of grid points and repeat; but I have limited the number of grid points to be less than 10,000, above which there is no improvement in precision due to limited floating point precision.

The results on high dimension (dimension ≥ 5) are not available for IFGT and MultiFFT. There is an issue with allocating a multidimensional array in C for MultiFFT that limits the number of grid points along each dimension.

2.5.1.2 Improved Fast Gauss Transform

It is much harder to control error on high dimensions for IFGT; the relevant paper states that the experiments have been only done on uniformly distributed dataset.

The README file gives some hints to fixing the value of p, so I have used the following values of p for each dataset

- sj2-50000-2: $p = 8$
- colors50k: $p = 8$
- bio5-nomissing: $p = 5$
- corel: $p = 3$
- covtype38d: $p = 2$

I have tried the following methods to put IFGT in the tuning mode:

- Cluster the data with $K = 10\sqrt{(}n)$ clusters only once, which fixes ρ_x. Then, I started with $\rho_y = 16$ and keep doubling until the first computation that met the error criterion, using the initial clusters.
- Cluster the data with $K = \sqrt{(}n)$ clusters, and use $\rho_x = \rho_y$. If the error criterion is not met, double K and re-cluster again until the error criteron is met.

Unfortunately, both of these did not work. The few of the results I have is produced by artifically high values of K on sj2-50000-2 and colors50k. Therefore, the results listed below are just obtained by "trial-and-error" by human inputs; the search time is not reported.

2.5.1.3 Dual-tree KDE

These will run in two mode: "auto-mode" using $\tau = \epsilon$ in which every data points will have error less than ϵ, and "tuning-mode" in a slightly different way than one described above. Because the dual-tree algorithm guarantees max error bound by using τ, this modification will try to come up with the maximum value of τ such that at least 95% of the data points have error less than ϵ.

The algorithm will start with $\tau = \epsilon$ and keep doubling τ while the error criterion is met. The search time is the total elapsed time and the running time is the elapsed time for the last trial that satisfied the error criterion.

2.5.1.4 Fast Gauss Transform

Although this algorithm guarantees the absolute deviation of each density estimate (not percentage derivation) by some upper bound τ (a *tweak parameter*), I have decided to use τ as an estimate of ϵ, hence used $\tau = \epsilon$.

2.5.2 Measurements

Each entry of the table has the total elapsed timing. For algorithms with weak or no error control, "searching times" (enclosed in parentheses) and tweak parameters that worked for given epsilon are listed below the timings. For the dual-tree tuning modes, some search times are reported in form of $\geq$'s because a possibly higher value of τ works for the given error bound, but I had to terminate the experiments due to time constraints.

$sj2-50000-2$, $D=2$, $N=50000$, $h^*=0.00317556$							
Algorithm	$\epsilon = 0.5$ (95%)	$\epsilon = 0.5$ (100%)	$\epsilon = 0.1$ (95%)	$\epsilon = 0.1$ (100%)	$\epsilon = 0.01$ (95%)	$\epsilon = 0.01$ (100%)	$\epsilon = 0$
$MFFT$	3.054508 (3.877533) $M = 512$	–	–	–	–	–	–
$IFGT$	31.8816 (70.061) $K = 14272$ $\rho_y = 2.5$	–		–		–	–
$DTREE$	3.4396	3.4396	3.7191	3.7191	4.0817	4.0817	–
$DTREE$ $Tuning$	$\leq$ 2.9559 ($\geq$ 14.606644) $\tau \geq 8$	$\leq$ 2.9559 ($\geq$ 14.606644) $\tau \geq 8$	$\leq$ 3.2516 ($\geq$ 15.965513) $\tau \geq 1.6$	$\leq$ 3.2516 ($\geq$ 15.965513) $\tau \geq 1.6$	$\leq$ 3.6351 ($\geq$ 17.844) $\tau \geq 0.16$	$\leq$ 3.6351 ($\geq$ 17.844) $\tau \geq 0.16$	–
$NAIVE$	–	–	–	–	–	–	301.696

$colors50k$, $D=2$, $N=50000$, $h^*=0.164755$							
Algorithm	$\epsilon = 0.5$ (95%)	$\epsilon = 0.5$ (100%)	$\epsilon = 0.1$ (95%)	$\epsilon = 0.1$ (100%)	$\epsilon = 0.01$ (95%)	$\epsilon = 0.01$ (100%)	$\epsilon = 0$
$MFFT$	0.091313 (0.183268) $M = 128$	–	2.988081 (3.808575) $M = 512$	–	65.6791 (83.145188) $M = 2048$	–	–
$IFGT$	4.502 (8.156) $K = 3568$ $\rho_y = 2.5$	–	–	–	–	–	–
$DTREE$	65.5077	65.5077	90.2097	90.2097	117.6247	117.6247	–
$DTREE$ $Tuning$	$\leq$ 12.5769 ($\geq$ 188.988619) $\tau \geq 8$	52.5751 (117.238195) $\tau = 1$	$\leq$ 42.6866 ($\geq$ 337.861552) $\tau \geq 1.6$	90.2097 $\tau = 0.1$ $\tau \geq 0.16$	$\leq$ 83.6847 ($\geq$ 505.95)	117.6247 $\tau = 0.01$	
$NAIVE$	–	–	–	–	–	–	329.72493

bio5, $D = 5$, $N = 150000$, $h^* = 0.079055$							
Algorithm	$\epsilon = 0.5$ (95%)	$\epsilon = 0.5$ (100%)	$\epsilon = 0.1$ (95%)	$\epsilon = 0.1$ (100%)	$\epsilon = 0.01$ (95%)	$\epsilon = 0.01$ (100%)	$\epsilon = 0$
$MFFT$	–	–	–	–	–	–	–
$IFGT$	–	–	–	–	–	–	–
$DTREE$	100.117	100.117	113.15	113.15	130.062	130.062	–
$DTREE$ $Tuning$	$\leq$ 73.588 ($\geq$ 427.624991) $\tau \geq 8$	87.3244 (276.332975) $\tau = 2$	$\leq$ 89.1612 ($\geq$ 499.9740) $\tau \geq 1.6$	113.15 $\tau = 0.1$	$\leq$ 109.066 ($\geq$ 591.76) $\tau \geq 0.16$	130.066 $\tau = 0.01$	
$NAIVE$	–	–	–	–	–	–	1966.304

corel, $D = 32$, $N = 37749$, $h^* = 0.219494$							
Algorithm	$\epsilon = 0.5$ (95%)	$\epsilon = 0.5$ (100%)	$\epsilon = 0.1$ (95%)	$\epsilon = 0.1$ (100%)	$\epsilon = 0.01$ (95%)	$\epsilon = 0.01$ (100%)	$\epsilon = 0$
$MFFT$	–	–	–	–	–	–	–
$IFGT$	–	–	–	–	–	–	–
$DTREE$	161.302	161.302	164.942	164.942	169.167	169.167	–
$DTREE$ $Tuning$	$\leq$ 157.526 ($\geq$ 634.011928) $\tau \geq 8$	$\leq$ 157.526 ($\geq$ 634.011928) $\tau \geq 8$	$\leq$ 160.335 ($\geq$ 801.193) $\tau \geq 1.6$	164.943 $\tau = 0.1$	169.165 $\tau = 0.01$	$\leq$ 163.038 ($\geq$ 819.50) $\tau \geq 0.16$	–
$NAIVE$	–	–	–	–	–	–	710.224950

covtype38d, $D = 38$, $N = 150000$, $h^* = 0.0432638$							
Algorithm	$\epsilon = 0.5$ (95%)	$\epsilon = 0.5$ (100%)	$\epsilon = 0.1$ (95%)	$\epsilon = 0.1$ (100%)	$\epsilon = 0.01$ (95%)	$\epsilon = 0.01$ (100%)	$\epsilon = 0$
$MFFT$	–	–	–	–	–	–	
$IFGT$	–	–	–	–	–	–	
$DTREE$	156.737	156.737	158.903	158.903	161.745	161.745	–
$DTREE$ $Tuning$	$\leq$ 153.149 ($\geq$ 565.897478) $\tau \geq 8$	$\leq$ 153.149 ($\geq$ 565.897478) $\tau \geq 8$	158.903 $\tau = 0.1$	158.903 $\tau = 0.1$	$\leq$ 157.597 ($\geq$ 732.66) $\tau \geq 0.16$	161.741 $\tau = 0.001$	–
$NAIVE$	–	–	–	–	–	–	13157.1

2.6 Effect of Bandwidths on Timings

I have experimented on how varying multiples of optimal bandwidth affect the time required for kernel density estimation on low-dimensional dataset ($D \leq 3$).

The following four methods have been tested for comparison in this section. Three of these methods guarantee bounding the maximum percentage deviation by the user specified ϵ.

Each entry has four numbers associated in the following order: preprocessing time (s), total elapsed time (s), true maximum per-datum error, true average per-datum error for $\epsilon = 0.01$.

For fast Gauss transform, I used $\tau = 0.01$, and recorded how many of the density estimates exceeded the user-specified $\epsilon = 0.01$ as one additional number; 0 means every query density estimate were at most 1% away from its true density estimate. Note that some of the timing measurements for fast Gauss transform are missing since the code *crashed due to the explosive numbers of boxes in the grid and the interaction list for a small bandwidth h*.

Readers should also note that the true maximum per-datum error and true avarage per-datum error for larger datasets have not been computed, since the density estimates computed by the naive method could not been obtained in time. Dual-tree derivative methods ($DTREE, DFGT\_AP1, DFGT\_AP2$) should satisfy these requirements for a given user ϵ, but more verification of the density estimates computed by fast Gauss transform is needed at this moment. On a minor note, I should measure the preprocessing time for fast Gauss transform in the near future.

$sj2 - 50000 - 2$, $D = 2$, $N = 50000$, $h^* = 0.00317556$							
$Algorithm \backslash h^*$	0.001	0.01	0.1	1	10	100	1000
$DTREE$	0.575318	0.592949	0.291020	0.287356	0.293285	0.288630	0.290581
	1.674842	**2.088624**	**1.563841**	**4.094356**	**44.439172**	**133.956959**	**15.547222**
	0	0	0.488988	0.000108633	0.000573328	0.00134004	0.00237811
	0	0	0.000780278	1.39063e-05	0.00018876	0.000454112	0.000527571
$DFGT\_AP$	0.306066	0.300515	0.299120	0.299436	0.296600	0.298661	0.307647
	0.866876	**1.049845**	**2.940228**	**48.459328**	**24.983645**	**3.553662**	**2.322058**
	0	2.37292e-16	4.67675e-16	7.16959e-15	3.07906e-06	8.93728e-05	0.000264482
	0	4.74583e-21	2.39294e-17	7.12942e-16	7.78035e-08	5.99022e-06	4.91007e-05
$DFGT\_AP2$	0.308580	0.302881	0.305307	0.307005	0.298226	0.303194	0.301193
	0.866867	**1.066825**	**3.11714**	**51.247585**	**168.796198**	**3.597192**	**2.109576**
	0	2.37292e-16	4.67675e-16	7.16959e-15	1.12982e-05	0.000120159	0.000773699
	0	4.74583e-21	2.39294e-17	7.12942e-16	1.70852e-07	8.18484e-06	0.000107175
FGT	–	–	–	–	–	–	–
	–	–	–	**7.583130**	**5.825566**	**50.943135**	**164.021533**
	–	–	–	0.00048954	0.000137774	0.000179816	6.46716e-08
	–	–	–	2.75839e-05	1.16575e-05	2.0979e-05	2.07801e-08
	–	–	–	0	0	0	0

$colors50k$, $D = 2$, $N = 50000$, $h^* = 0.164755$							
$Algorithm \backslash h^*$	0.001	0.01	0.1	1	10	100	1000
$DTREE$	0.399409	0.394542	0.391424	0.388719	0.387904	0.399782	0.395366
	1.255746	**1.829179**	**6.551644**	**117.764408**	**363.783493**	**14.657439**	**0.618148**
	0	0	0.000132609	0.135383	0.00386445	0.00367039	0.0075942
	0	0	8.60675e-05	0.000657529	2.62139e-05	0.00246688	0.00629269
$DFGT\_AP$	0.409544	0.412928	0.410145	0.411002	0.412522	0.407743	0.409358
	1.282855	**4.924589**	**143.136479**	**56.392096**	**8.180041**	**3.773496**	**3.234854**
	2.39652e-16	9.08779e-16	1.24088e-14	4.44545e-06	0.000134087	0.00196874	0.00780571
	3.07215e-20	5.18636-e17	1.1124e-15	1.27604e-07	7.93838e-06	0.000132507	2.74952e-05
$DFGT\_AP2$	0.416339	0.413435	0.413226	0.416737	0.411788	0.422886	0.415152
	1.321605	**5.172735**	**152.775772**	**271.061628**	**20.973487**	**5.115325**	**3.290515**
	2.39652e-16	9.08779e-16	1.24088e-14	6.86244e-05	0.00573335	0.00594394	0.00780571
	3.07215e-20	5.18636e-17	1.1124e-15	1.21315e-06	2.13845e-05	0.00016351	2.35232e-05
FGT	–	–	–	–	–	–	–
	–	–	–	**10.761949**	**136.277286**	**293.987943**	**304.915158**
	–	–	–	1	0.459224	3.22875e-11	0
	–	–	–	0.00140568	2.94442e-05	1.02516e-13	0
	–	–	–	76	32	0	0

$edsgc - radec$, $D = 2$, $N = 1495877$, $h^* = 0.00193855$							
$Algorithm \backslash h^*$	0.001	0.01	0.1	1	10	100	1000
$DTREE$	9.561811	9.565492	9.658469	9.643481	9.682244	9.558405	9.651560
	25.551161	**31.566672**	**49.816636**	**173.392816**	**4163.09278**	**53467.267**	**31169.4011**
	–	–	–	–	–	–	–
	–	–	–	–	–	–	–
$DFGT\_AP$	10.047415	9.998338	9.989018	9.996815	10.125650	10.040910	10.004285
	26.11873	**32.672224**	**108.212968**	**3669.8047**	**15725.1429**	**412.186907**	**132.204131**
	–	–	–	–	–	–	–
	–	–	–	–	–	–	–
$DFGT\_AP2$	10.056729	10.101711	10.129689	10.082438	10.055522	10.161404	10.156005
	26.187449	**33.327844**	**113.654731**	**3904.6212**	**84913.20**	**1512.57751**	**108.309**
	–	–	–	–	–	–	–
	–	–	–	–	–	–	–
FGT	–	–	–				
	–	–	–	**71.533981**	**304.865168**	**5506.949202**	**71095.405003**
	–	–	–	–	–	–	–
	–	–	–	–	–	–	–
	–	–	–	–	–	–	–

$mockgalaxy\_D\_1M$, $D = 3$, $N = 1000000$, $h^* = 0.0038025$							
$Algorithm \backslash h^*$	0.001	0.01	0.1	1	10	100	1000
$DTREE$	7.181983 **29.363957** – –	7.191571 **36.265959** – –	7.166260 **77.572684** – –	7.163092 **156.963** – –	7.184758 **274.123** – –	7.209003 **26303.422** – –	7.240229 **39770.162** – –
$DFGT\_AP$	7.545416 **29.763887** – –	7.546534 **37.176529** – –	7.553372 **129.812306** – –	7.577777 **255.702** – –	7.550699 **44135.022** – –	7.503927 **2433.41** – –	7.338271 **206.793655** – –
$DFGT\_AP2$	7.549349 **29.994817** – –	7.607630 **38.296554** – –	7.125942 **138.694298**	7.221381 **286.617567**	7.124708 **42126.668949**	7.297186 **2604.225**	7.222901 **211.920986**
FGT	– – – – –	– – – – –	– – – – –	– – – – –	– **213.776268** – – –	– **1336.137933** – – –	– **654.964563** – – –

Chapter 3

Conclusion and Future Research Directions

In this thesis, we have developed new techniques to tackle two important N-body problems.

For kernel density estimation problem, we have successfully combined two powerful techniques from *computational geometry* and *approximation theory* to achieve fast speed and direct error control.

3.1 Fast Gauss Transform Using Dual Recursion

3.1.1 Pruning Rule

Due to the nature of the Gaussian kernel, pruning of a query node and a reference node is possible only if the node(s) in consideration meet certain size constraints or by using many terms of the series. We believe that by obtaining a tighter upper bound on Hermite functions, we can formulate the pruning criterion using the *well-separated-ness* of the query/reference node pair (as done with other kernel functions).

3.1.2 Handling Higher Dimensions

Although our new algorithms achieved success on low dimensions and a large number of data points, we have not had a chance to improve our multipole-based tree algorithm on higher dimensions ($D > 3$). Our ultimate goal is to design an algorithm that intelligently combines the multipole approximation with the simple finite-difference approach in the original dual-tree KDE.

Advocating the $O(D^p)$ expansion form developed in [29] might be the solution for overcoming the *curse of dimensionality* and will be explored in details.

3.2 Extension to Other Kernel Functions

We believe it is very straightforward to extend the techniques developed in this thesis to other useful kernel functions.

3.3 Kernel Independent Fast Multipole Methods

There are variants of fast multipole methods whose derivations do not depend on the kernel function. We hope to investigate the usability of these algorithms by referring to relevant literature.

3.4 Potential Applications

Given enough time and collaboration from experts in relevant fields, we hope to investigate potentials of the newly developed techniques in the following areas.

1. Computer graphics: photon mapping, surface reconstruction [8], radiosity [16]
2. Statistics: nonparametric density estimation.
3. Computer vision: mean-shift analysis.

Appendix A

Pseudocodes for KDE Algorithms

A.1 KDE using Multidimensional Fast Fourier Transform

A.1.1 Multidimensional FFT Routines

```
define struct complex
  double real
  double imag
end define

// procedure for 1 dimensional fft on complex data
procedure fftc1 :=
  input f        // put the complex data to be transformed here. result is
                 // returned here.
  input N        // the number of points
  input skip     // each point is separated by ``skip'' in f.
  input forward  // 1 means forward fft, -1 means inverse fft.

  pi2 := 4 * asin(1)
  c := (complex *) f  // cast f to a complex array.

  for index1 := 1, index2 := 0; index1 < N, index1 := index1 + 1
    for b := N / 2; index2 >= b; b := b / 2
      index2 := index2 - b
    index2 := index2 + b
    if index2 > index1
      temp1 := c[index2 * skip]
      c[index2 * skip] := c[index1 * skip]
      c[index1 * skip] := temp1

  for trans_size := 2; trans_size <= N; trans_size := trans_size * 2
    pi2n := forward * pi2 / trans_size
    cospi2n := cos(pi2n)
    sinpi2n := sin(pi2n)
    wb.real := 1
    wb.imag := 0
    for b:= 0 to trans_size / 2 - 1
      for trans := 0 to trans < N / trans_size - 1
        index1 := (trans * trans_size + b) * skip
        index2 := index1 + trans_size / 2 * skip
        temp1 := c[index1]
        temp2 := c[index2]
        c[index1].real = temp1.real + wb.real * temp2.real -
                         wb.imag * temp2.imag
        c[index1].imag = temp1.imag + wb.real * temp2.imag +
                         wb.imag * temp2.real
        c[index2].real = temp1.real - wb.real * temp2.real +
                         wb.imag * temp2.imag
        c[index2].imag = temp1.imag - wb.real * temp2.imag -
                         wb.imag * temp2.real
      temp1 := wb
      wb.real := cospi2n * temp1.real - sinpi2n * temp1.imag
      wb.imag := cospi2n * temp1.imag + sinpi2n * temp1.real

  if forward < 0
    for index1 := 0; index1 < skip * N; index1 := index1 + skip
      c[index1].real := c[index1].real / N
      c[index1].imag := c[index1].imag / N
end procedure

// procedure for multidimensional fft on complex data
procedure fftcn :=
  input f        // put the complex data in this array. result returned here.
  input ndims    // the dimensionality of the transform.
  input size     // size[i] tells the number of points in the i-th dimension
  input forward  // 1 means forward fft, -1 means inverse fft.

  planesize := 1
  skip := 1
  totalsize := 1

  for dim := 0 to ndims - 1
    totalsize := totalsize * size[dim]

  for dim := ndims - 1 to 0 step -1
    planesize := planesize * size[dim]
    for i := 0 to totalsize - 1 step planesize
      for j := 0 to skip - 1
        fftc1(f + 2 * (i + j), size[dim], skip, forward)
    skip := skip * size[dim]
end procedure

procedure fftrn :=
  input f        // put the real data in this array
  input fnyquist // nyquist frequencies are returned here.
  input ndims    // the dimensionality of the transform
  input size     // size[i] tells the number of points in the i-th dimension
  input forward  // 1 means forward fft, -1 means inverse fft.

  indexneg := 0
  N := size[ndims - 1]
  pi2n := 4 * asin(1) / N
  cospi2n := cos(pi2n)
  sinpi2n := sin(pi2n)

  // Cast f and fnyquist so that you can access those as arrays of complex
  // numbers.
  c := (struct complex *) f
  cnyquist := (struct complex *) fnyquist

  totalsize := 1
  indices := MALLOC_ARRAY(ndims, int)

  // Set the last dimension of the size in half because now we are using f as
  // an array of complex numbers.
  size[ndims - 1] := size[ndims - 1] / 2

  for i := 0 to ndims - 1
    totalsize := totalsize * size[i]
    indices[i] := 0
  // If forward fft,
  if forward == 1
    fftcn(f, ndims, size, 1)
    for i := 0 to (total_size / size[ndims - 1]) - 1
      cnyquist[i] := c[i * size[ndims - 1]]

  wb := undefined complex
  for index := 0 to totalsize - 1 step size[ndims - 1]
    wb.real := 1
    wb.imag := 0
    for b := 1 to N / 4 - 1
      temp1 := wb
      wb.real := cospi2n * temp1.real - sinpi2n * temp1.imag
      wb.imag := cospi2n * temp1.imag + sinpi2n * temp1.real
      temp1 := c[index + b]
      temp2 := c[indexneg + N / 2 - b]
      c[index+b].real = 0.5*(temp1.real + temp2.real +
                             forward * wb.real * (temp1.imag+temp2.imag) +
                             wb.imag * (temp1.real - temp2.real));
      c[index+b].imag = 0.5*(temp1.imag - temp2.imag -
                             forward * wb.real * (temp1.real-temp2.real) +
                             wb.imag*(temp1.imag+temp2.imag));
      c[indexneg+N/2-b].real = 0.5 * (temp1.real + temp2.real -
                                      forward *wb.real *
                                      (temp1.imag + temp2.imag) -
                                      wb.imag * (temp1.real - temp2.real));
      c[indexneg+N/2-b].imag = 0.5 * (-temp1.imag + temp2.imag -
                                      forward * wb.real *
                                      (temp1.real - temp2.real) +
                                      wb.imag * (temp1.imag + temp2.imag));

    temp1 := c[index]
    temp2 := cnyquist[indexneg / size[ndims - 1]]
    c[index].real = 0.5 * (temp1.real + temp2.real +
                           forward * (temp1.imag + temp2.imag))
    c[index].imag = 0.5 * (temp1.imag - temp2.imag -
                           forward * (temp1.real - temp2.real))

    cnyquist[indexneg / size[ndims-1]].real = 0.5 * (temp1.real + temp2.real -
                                              forward * (temp1.imag + temp2.imag))
    cnyquist[indexneg / size[ndims-1]].imag = 0.5 * (-temp1.imag + temp2.imag -
                                              forward * (temp1.real - temp2.real))
    stepsize=size[ndims-1]

    for j := ndims - 2, while indices[j] == size[j] - 1 and j >= 0, step -1
      indices[j] := 0
      indexneg := indexneg - stepsize
      stepsize := stepsize * size[j]

    if(indices[j] == 0)
      indexneg := indexneg + stepsize * (size[j] - 1)
    else
      indexneg := indexneg - stepsize
    if(j >= 0)
      indices[j] := indices[j] + 1

  if forward == -1
    fftcn(f, ndims, size, -1)

  size[ndims - 1] := size[ndims -1] * 2
end procedure
```

A.1.2 Multidimensional FFT KDE Subroutines

```
#define TAU       4.0          /* Wand (p435) */
int M;                         /* Wand (p441) */
#define PI        3.1415926535897932384626

procedure assignWeights :=
  input datapt, gridsizes, mincoords, indices, enlarged_dims, discretized,
        level, volume, pos skip

  if(level == -1)
    discretized[pos] := discretized[pos] + volume
  else

    /**
     * Recurse in the right direction
     */
    coord := datapt[level]
    leftgridcoord := mincoords[level] + indices[level] * gridsizes[level];
    rightgridcoord := leftgridcoord + gridsizes[level]
    leftvolume := volume * (rightgridcoord - coord)
    rightvolume := volume * (coord - leftgridcoord)
    nextskip := enlarged_dims[level] * skip
    nextleftpos := pos + skip * indices[level]

    if(leftvolume > 0.0)
      assignWeights(datapt, gridsizes, mincoords, indices,
                    enlarged_dims, discretized, level - 1, leftvolume,
                    nextleftpos, nextskip)

    if(rightvolume > 0.0)
      assignWeights(datapt, gridsizes, mincoords, indices,
                    enlarged_dims, discretized, level - 1, rightvolume,
                    nextleftpos + skip, nextskip)
end procedure

procedure retrieveWeights :=
  input dataptnum, datapt, num_dims, gridsizes, discretized, size,
        indices, mincoords, volume, densities, level, pos, skip, divfactor

  if(level == -1)
    densities[dataptnum] := densities[dataptnum] + discretized[pos] * volume
                            / divfactor
  else

    /**
     * Recurse in the right direction
     */
    coord := datapt[level];
    leftgridcoord := mincoords[level] + indices[level] * gridsizes[level]
    rightgridcoord := leftgridcoord + gridsizes[level]
    leftvolume := volume * (rightgridcoord - coord)
    rightvolume := volume * (coord - leftgridcoord)
    nextskip := size[level] * skip
    nextleftpos := pos + skip * indices[level]

    if(leftvolume > 0.0)
      retrieveWeights(dataptnum, datapt, num_dims, gridsizes, discretized,
                      size, indices, mincoords, leftvolume, densities,
                      level - 1, nextleftpos, nextskip, divfactor)

    if(rightvolume > 0.0)
      retrieveWeights(dataptnum, datapt, num_dims, gridsizes, discretized,
                      size, indices, mincoords, rightvolume, densities,
                      level - 1, nextleftpos + skip, nextskip, divfactor)
```

```
end procedure

procedure retrieveDensities :=
  input dataset, num_rows, dim, gridsizes, discretized, size, mincoords,
        gridbinvolume, bandwidsqd

  densities := (double *) malloc(num_rows * sizeof(double))
  normc := pow((2.0 * PI * bandwidsqd),((double)dim) / 2.0) * num_rows

  minindices := (int *) malloc(dim * sizeof(int))

  for(r = 0; r < num_rows; r++)
    densities[r] = 0.0

    for(d = 0; d < dim; d++)
      minindices[d] := floor((dataset[r * dim + d] - mincoords[d])/
                             gridsizes[d])

    retrieveWeights(r, dataset + r * dim, dim, gridsizes, discretized,
                    size, minindices, mincoords, 1.0, densities, dim - 1,
                    0.0, 1, gridbinvolume * normc)

  return densities;
end procedure

procedure discretize_dataset :=
  input dataset, num_rows, dim, bandwidth, gridsizes, mincoords, maxcoords,
        diffcoords, kernelweights_dims, enlarged_dims, numenlargedgridpts,
        gridbinvolume

  /**
   * Temporary used to count the number of elements in the enlarged matrices
   * for the kernel weights and bin counts. Also calculate the volume of each
   * grid bin.
   */
  numengridpts := 1
  gvolume := 1.0

  /**
   * Temporary index array to locate the bin for each data point.
   */
  minindices := (int *) malloc(dim * sizeof(int));

  /**
   * Find the min/max in each coordinate direction, and calculate the grid
   * size in each dimension.
   */
  for(d = 0; d < dim; d++)
    min := MAXDOUBLE
    max := MINDOUBLE

    for(r = 0; r < num_rows; r++)
      coord := dataset[r * dim + d]
      if(coord > max)
        max := coord
      if(coord < min)
        min := coord

    /**
     * Following Silverman's advice here
     */
    mincoords[d] := min
    maxcoords[d] := max
    diffcoords[d] := maxcoords[d] - mincoords[d]
    gridsizes[d] := diffcoords[d] / ((double) M - 1)
    gvolume := gvolume * gridsizes[d]

    /**
     * Determine how many kernel weight calculation to do for this dimension.
     */
    kernelweights_dims[d] := M - 1
    possiblesample := floor(TAU * bandwidth / gridsizes[d])

    if(kernelweights_dims[d] > possiblesample)
      if(possiblesample == 0)
        possiblesample := 1
      kernelweights_dims[d] = possiblesample

    /**
     * Wand p440: Need to calculate the actual dimension of the matrix
     * after the necessary 0 padding of the kernel weight matrix and the
     * bin count matrix.
     */
    enlarged_dims[d] := ceil(log(M + kernelweights_dims[d]) / log(2))
    enlarged_dims[d] := 1 << enlarged_dims[d];
    numengridpts := numengridpts * enlarged_dims[d]


  /**
   * Allocate the memory for discretized grid count matrix and initialize it.
   */
  discretized := (double *) malloc(numengridpts * sizeof(double))
  *numenlargedgridpts := numengridpts
  *gridbinvolume := gvolume
  gvolume := 1.0 / gvolume
  for(d = 0; d < numengridpts; d++)
    discretized[d] := 0.0;

  /**
   * Now loop over each data and calculate the weights at each grid point.
   */
  for(r = 0; r < num_rows; r++)

    /**
     * First locate the bin the data point falls into and identify it by
     * the lower grid coordinates.
     */
    for(d = 0; d < dim; d++)
      minindices[d] := floor((dataset[r * dim + d] - mincoords[d])/
                             gridsizes[d]);

    /**
     * Assign the weights around the neighboring grid points due to this
     * data point. This results in 2^num_dims number of recursion per data
     * point.
     */
    assignWeights(dataset + r * dim, gridsizes, mincoords, minindices,
                  enlarged_dims, discretized, dim - 1, gvolume, 0, 1)

  return discretized;
end procedure

procedure scale_data_by_meanstdev :=
  input m, num_dims, n

  int i, j;

  for (i=0; i < num_dims; i++)
```

```
    s := 0.0
    u := 0.0
    for(j = 0; j < n; j++)
      u := u + m[num_dims * j + i]

    u /= n
    for(j = 0; j < n; j++)
      diff := m[num_dims * j + i] - u
      s := s + diff * diff

    s := s / (n - 1)
    s := sqrt(s)

    for(j = 0; j < n; j++)
      m[num_dims * j + i] := m[num_dims * j + i] - u;
      m[num_dims * j + i] := m[num_dims * j + i] / s;
end proecure

procedure gaussify :=
  input gridsizes, enlarged_dims, kernelweights, kernelweights_dims, acc,
        precalc, level, pos, skip

  if(level == -1)
    kernelweights[pos] := exp(precalc * acc)
  else
    half := kernelweights_dims[level]

    for(g = 0; g <= half; g++)
      addThis := g * gridsizes[level]
      newacc := acc + addThis * addThis
      newskip := skip * enlarged_dims[level]

      gaussify(gridsizes, enlarged_dims, kernelweights, kernelweights_dims,
               newacc, precalc, level - 1, pos + skip * g, newskip)

      /**
       * If this is not the 0th frequency, then do the mirror image thingie.
       */
      if(g != 0)
        gaussify(gridsizes, enlarged_dims, kernelweights, kernelweights_dims,
                 newacc, precalc, level - 1,
                 pos + skip * (enlarged_dims[level] - g), newskip)
end procedure
```

A.1.3 Multidimensional FFT KDE Main Routine

```
begin procedure FFTKDEMain :=
  input argc, argv    // Standard C style arguments

  num_rows := atoi(argv[2])
  num_cols := atoi(argv[3])
  bandwidth := atof(argv[4])
  M := atoi(argv[5])
  epsilon := atof(argv[6])

  bandwidthsqd := bandwidth * bandwidth
  dataset := parseData(argv[1], num_rows, num_cols)

  scale_data_by_meanstdev(dataset, num_cols, num_rows)

  /**
   * Store the number of points along each dimension in the zero padded
   * matrix, min/max/diff for each dimension.
   */
```

```
size := (int *) malloc(num_cols * sizeof(int))
mincoords := (double *) malloc(num_cols * sizeof(double))
maxcoords := (double *) malloc(num_cols * sizeof(double))
diffcoords := (double *) malloc(num_cols * sizeof(double))
gridsizes := (double *) malloc(num_cols * sizeof(double))
kernelweights_dims := (int *) malloc(num_cols * sizeof(int))

start_time := get_time()
discretized := discretize_dataset(dataset, num_rows, num_cols,
                                  bandwidth, gridsizes, mincoords,
                                  maxcoords, diffcoords, kernelweights_dims,
                                  size, &numgridpts, &gridbinvolume)

nyquistnum := 2 * numgridpts / size[num_cols - 1]
d_fnyquist := (double *) malloc(nyquistnum * sizeof(double))
k_fnyquist := (double *) malloc(nyquistnum * sizeof(double))
kernelweights := (double *) malloc(numgridpts * sizeof(double))
precalc := -0.5 / bandwidthsqd;

for(d = 0; d < numgridpts; d++)
  kernelweights[d] := 0.0

/**
 * FFT the discretized bin count matrix.
 */
fftrn(discretized, d_fnyquist, num_cols, size, 1)

/**
 * Calculate the required kernel weights at each grid point. This matrix
 * will be convolved with fourier transformed data set.
 */
gaussify(gridsizes, size, kernelweights, kernelweights_dims, 0.0, precalc,
         num_cols - 1, 0, 1)

/**
 * FFT the kernel weight matrix.
 */
fftrn(kernelweights, k_fnyquist, num_cols, size, 1)

/**
 * We need to invoke the convolution theorem for FFT here. Take each
 * corresponding complex number in kernelweights and discretized and do
 * an element-wise multiplication. Later, pass it to inverse fft function,
 * and we have our answer!
 */
for(d = 0; d < numgridpts; d += 2)
  real1 := discretized[d]
  complex1 := discretized[d + 1]
  real2 := kernelweights[d]
  complex2 := kernelweights[d + 1]
  discretized[d] = real1 * real2 - complex1 * complex2
  discretized[d + 1] = real1 * complex2 + complex1 * real2

for(d = 0; d < nyquistnum; d += 2)
  real1 := d_fnyquist[d]
  complex1 := d_fnyquist[d + 1]
  real2 := k_fnyquist[d]
  complex2 := k_fnyquist[d + 1]
  d_fnyquist[d] := real1 * real2 - complex1 * complex2
  d_fnyquist[d + 1] := real1 * complex2 + complex1 * real2

/**
 * Inverse FFT the elementwise multiplied matrix.
 */
```

```
  fftrn(discretized, d_fnyquist, num_cols, size, -1)

  /**
   * Retrieve the densities of each data point.
   */
  densities := retrieveDensities(dataset, num_rows, num_cols, gridsizes,
                                 discretized, size, mincoords, gridbinvolume,
                                 bandwidthsqd)

  stop_time := get_time()
end procedure
```

Reference List

[1] L. Ahlfors, *Complex Analysis*, McGraw-Hill, 3rd ed., 1979.

[2] C. R. Anderson, *An implementation of the fast multipole method without multipoles*, SIAM Journal on Scientific and Statistical Computing, 13 (1992).

[3] J. Barnes and P. Hut, *A hierarchial $o(n \log n)$ force-calculation algorithm*, Nature, 324 (1986).

[4] B. Baxter and G. Roussos, *A new error estimate of the fast gauss transform*, SIAM Journal on Scientific Computing, 24 (2002), pp. 257–259.

[5] R. Beatson and L. Greengard, *A short course on fast multipole methods.* Lecture notes on fast multipole methods.

[6] J. L. Bentley, *Multidimensional binary search trees used for associative searching*, Communications of the ACM, 18 (1975).

[7] P. B. Callahan, *Dealing with Higher Dimensions: The Well-separated Pair Decomposition and Its Applications*, PhD thesis, Johns Hopkins University, 1995.

[8] J. C. Carr, R. K. Beatson, J. B. Cherrie, T. J. Mitchell, W. R. Fright, B. C. McCallum, and T. R. Evans, *Reconstruction and representation of 3D objects with radial basis functions*, in SIGGRAPH 2001, Computer Graphics Proceedings, E. Fiume, ed., ACM Press / ACM SIGGRAPH, 2001, pp. 67–76.

[9] J. Carrier, L. Greengard, and V. Rokhlin, *A fast adaptive multipole algorithm for particle simulations*, SIAM Journal on Scientific and Statistical Computing, 9 (1988), pp. 669–686.

[10] G. Felder. http://www.science.smith.edu/departments/Physics/fstaff/gfelder/ffteasy/ffteasy.c.

[11] A. G. Gray, *Bringing Tractability to Generalized N-Body Problems in Statistical and Scientific Computation*, PhD thesis, Carnegie Mellon University, 2003.

[12] L. Greengard and J. Strain, *The fast gauss transform*, SIAM Journal on Scientific and Statistical Computing, 12 (1991), pp. 79–94.

[13] L. Greengard and X. Sun, *A new version of the fast gauss transform*, Documenta Mathematica, Extra Volume ICM (1998), pp. 575–584.

[14] J. Huang and L. Greengard, *A new version of the fast multipole method for screened coulomb interactions in three dimensions.*, Journal of Computational Physics, 180 (2002), pp. 642–658.

[15] A. T. Ilher, *An overview of fast multipole methods.* http://ssg.mit.edu/~ihler/papers/ihler_area.pdf, 2004.

[16] V. Jain, *Implementing occlusion in fmm based global illumination*, tech. rep., Indian Institute of Technology, March 2004.

[17] A. Moore and M. S. Lee, *Cached sufficient statistics for efficient machine learning with large datasets*, Journal of Artificial Intelligence Research, 8 (1998), pp. 67–91.

[18] A. W. Moore, *The anchors hierarchy: Using the triangle inequality to survive high-dimensional data*, in Proceedings of the Twelfth Conference on Uncertainty in Artificial Intelligence, AAAI Press, 2000, pp. 397–405.

[19] S. M. Omohundro, *Five balltree construction algorithms*, tech. rep., International Computer Science Instititue, December 1989.

[20] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling, *Numerical Recipes in C: The Art of Scientific Computing*, Cambridge University Press, 1988.

[21] M. H. Protter and C. B. Morrey, *A First Course in Real Analysis*, Springer, 2nd ed., 1991.

[22] B. Silverman, *Kernel density estimation using the fast fourier transform*, Applied Statistics, 31 (1982), pp. 93–99.

[23] B. W. Silverman, *Density Estimation for Statistics and Data Analysis*, Chapman and Hall, 1986.

[24] J. Strain, *The fast gauss transform with variable scales*, SIAM Journal on Scientific and Statistical Computing, 12 (1991), pp. 1131–1139.

[25] J. A. Strain. http://math.berkeley.edu/∼strain/Codes/gauss.tar.gz.

[26] O. Szász, *On the relative extrema of the hermite orthogonal functions*, J. Indian Math. Soc., 15 (1951), pp. 129–134.

[27] M. P. Wand, *Fast computation of multivariate kernel estimators*, Journal of Computational and Graphical Statistics, 3 (1994), pp. 433–445.

[28] E. W. Weisstein. http://mathworld.wolfram.com/HermitePolynomial.html.

[29] C. Yang, R. Duraiswami, N. A. Gumerov, and L. Davis, *Improved fast gauss transform and efficient kernel density estimation*, International Conference on Computer Vision, (2003).