A MULTIPOLE ALGORITHM FOR CALCULATING POTENTIAL FIELDS

A thesis submitted in partial fulfillment of the requirement for the degree of Bachelor of Science with Honors in Physics from the College of William and Mary in Virginia,

by

Theodore Vecchione

Accepted for ____

(Not Even Honors, Honors, High Honors or Highest Honors)

Advisor: ____

Dr. Henry Krakauer

Williamsburg, Virginia
December 2000

Abstract

The exact calculation of a potential field of paired interactions involves a double summation over each of the interacting particles. The computational scaling is of order N^2, where N is the number of particles. However, with the appropriate implementation of a faster multipole algorithm the number of operations can be reduced to order N. This algorithm is therefore beneficial to all fields of science where potential fields are used in simulations of large systems. Examples of applications range from solid state physics, to fluid dynamics and astrophysics. The project under taken consists of implementing a multipole algorithm in both two and three dimensions which could eventually be used in Monte Carlo simulations of crystal growth.

Contents

List of Figures

1 Introduction

In several fields of science today it has become important to be able to calculate the interactions of many bodies. This is called the N body problem, where N is the number of bodies studied. It is believed that the sum of these interactions can be used to predict future evolution of a system. Although this inherently implies a dynamical system (one that is allowed to change), it can also be used to study equilibrium conditions of static (that do not change) systems. This paper assumes that the systems being studied are static. In such a context it is necessary only that the spatial locations and characteristic interaction strengths of all members are given. Calculating the sum of the interactions then provides a means for determining the intrinsic stability of the system.

To achieve such ends, the term "potential field" is introduced, which represent the interactions of all the members. Mathematically the force per charge strength experienced by a particle is the negative vector gradient of a scaler potential field: $\vec{F}/q = -\vec{\nabla}\Phi$. In general, to calculate the potential field of interactions of a system of N particles requires a double summation over each individual interaction. Otherwise stated, each member interacts with each other member. Consequently the interaction is counted twice, once for each member. This is why the term "double summation" is used. A double summation method can calculate a potential field without error and independent of the member distribution. This contrasts other approximation methods discussed later which use assumptions appropriate to certain distributions. Unfortunately, this direct method is somewhat costly in terms of calculations as it scales as $O(N^2)$ in calculations. For some simulations this becomes the limiting factor on the size of the systems considered.

One solution to this problem is to perform an Ewald summation which assumes regular periodic boundary conditions. Therefore the complexity of computations is reduced by assuming that within a system there are regular patterns of distribution which repeat them-self periodically along boundries. This type of approach has a scaling behavior of $O(N \log N)$. Other solutions and algorithms are introduced in the next section. The solution chosen for this paper will be the fast multipole algorithm. This algorithm has a computational scaling of order $O(N)$ and if modified to accommodate either periodic or terminal boundary conditions is well suited for possible applications in Monte Carlo simulations of crystal growth and structure.

2 Other Methods

There are no shortage of N body methods which attempt to improve upon the short comings of the other methods. The following subsections are a listing of some of these methods. In the end, each algorithm is best applicable to only certain simulations. Some are faster, while others are more accurate. Some handle close inter-particle distances and some handle nonuniform particle distributions better than others. In choosing an algorithm one must consider the simulation of interest and select accordingly.

2.1 Particle-Particle (PP)

The Particle-Particle method is the first method mentioned in the introduction. It is the direct evaluation of an interaction law as it is applied to a system of particles. This method calculates for each particle the interaction with each other particle and then sums the individual results. The order of operations is thus $N(N-1)$. Again, one drawback to this method is that the computational scaling can become the limiting factor in the size of system studied. Another drawback is that if a system's time evolution is studied then as the inter-particle distances go to zero the interactions inversely go to infinity and the time intervals must be shortened to avoid overflow. Thus it adds a level of complexity to the simulations.

2.2 Particle-Mesh (PM)

The Particle-Mesh method does not calculate the direct interactions of the particles within a system. Instead, a grid-like mesh is created which divides the system into cells. Along the mesh the characteristics of nearby particles are through various schemes acquired as a density. For each scheme it is important that the fluctuations in interactions of nearby particles be "smoothed", a term in reference to the the continuity of the derivatives of the function used in assigning charges to the mesh. One such scheme, called "Nearest-Grid-Point" (NGP), adds each particle into the nearest mesh point and then divides by the cell volume, but it is rarely used as it creates discontinuous forces. Another such scheme, called "Cloud-in-a-Cell" (CIC), involves each mesh point with $2^{Number of dimensions}$ of its nearest neighbors. This scheme gives continuous forces but discontinuous derivatives. Once a scheme has been implemented the Poisson equation $\nabla^2(\phi) = 4\pi G\rho$ is solved along the mesh. Any forces are then calculated and interpolated to the particles. The total method is of order of operations $O(N + Ng \log N_g)$ where N_g is the number of grid points. One application of the PM method is in situations where it is important to suppress the effects of an inverse separation interactions as this method does not treat particles as being individually localized. One disadvantage to using a mesh-based method is that it has difficulties handling nonuniform particle distributions or large dynamical changes (such as shock waves). To compensate for this there are PM algorithms which employ meshes of finer gridding in selected subregions of the system and which allow for moving or adaptive grids.

2.3 Particle-Particle/Particle-Mesh (P3M)

The P3M method improves upon the PM method by applying it to calculate slowly-varying long-range interactions and then supplementing it with the PP method of direct summation of all rapidly-varying short-range interactions.

Thus the P3M method can be used to calculate interactions of closely located particles. Two meshes are used in this method. One is a "charge-potential" mesh, and the other is a "chaining" (coarser) mesh. The charge-potential mesh is used at different stages of the PM calculation to store charge density values, charge harmonics, potential harmonics, and potential values. The chaining mesh is an array of cells to locate pairs of neighboring particles in the short-range calculation. The number of operations is of order $O(N + N_g)$. The disadvantage of the P3M algorithm is that it too easily becomes dominated by the PM method.

2.4 Particle Multiple-Mesh (PM2)

Nested-grid particle methods try to increase the ratio between the size of the system and the smallest resolved scale. In a plain Particle-Mesh method, this is the number of grid zones. In a Particle-Particle/Particle-Mesh, it is the ratio between the box size and the particle softening or the inter-particle separation, whichever is larger. In the nested-grid methods it's the ratio between the box size and the spacing of the finest sub-grid. The PM2 method describes a nested particle mesh in which there is only one set of particles which exist independently of any sub-grids and therefore the grid structure can be redefined at every step. This method uses an initial mass granularity that already matches the desired resolution on the sub-grid. First assign a mesh density to the top grid using a Cloud-in-Cell algorithm. Next, solve for the potential on the grid. Then create boundary conditions on the parent grid and introduce sub-grids to compute inter-particle forces. Generate the density array on the sub-grid and again solve the field potential equation. Finally impose boundary conditions on the sub-grid and sum all results. Each step is of computational order $O(N_g \log N_g)$ for the potential solver and $O(N)$ for the charge assignment.

2.5 Nested Grid Particle-Mesh (NGPM)

The Nested Grid Particle-Mesh is a scheme which uses a series of nested lattices in each of which a new set of less massive particles is introduced. Thus both the force and mass resolution increase for the more finely spaced sub-lattices as opposed to other methods in which only the force resolution increases. These "particles" are constructs that sample the phase space distribution of the much smaller, more numerous, real particles of the physical system. For the method, first assign a density to grid mesh of parent grid. Then define a sub-grid anywhere within the parent grid consisting of n^3 sub-grid particles of mass m_p/n_s^3 where p = the parent grid and s = the sub-grid. A buffer zone is included around the sub-grid to help smooth any transients. Calculate the interactions of the parents from the parent-grid mesh-defined potential. Calculate the interactions of the sub-grid particles. Set the density over the sub-grid region equal to the mean over the sub-grid. Solve the field potential equation on the sub-grid. This represents the contribution to the sub-grid potential whose parent particles have not entered the sub-grid. Interpolate this potential using a "Cloud-in-a-Cell" (CIC) scheme to compute the background forces on the sub-grid particles. Calculate the forces on the sub-grid particles due to themselves using quasi-isolated boundary conditions. When integrating the forces, asynchronous time-steps which is an improvement to other nested grid methods. The order of computation is between $O(\log N_g)$ and $O(N_g \log N_g)$.

2.6 Tree-Code (TC) Top Down

The tree code divides interactions into external interactions which are of order $O(N)$ and can be computed for each particle independently, nearest neighbor interactions which are also of order $O(N)$ and far field interactions which are of order $O(N^2)$. This method's "divide-and-conquer" way is to build a hierarchy of boxes which refine the computational domain into smaller and smaller regions. Then for each subregion, compute the center of mass and total mass for all the particles it contains. Finally, for each particle, traverse the tree to compute the force on it. The tree structures thus split the mass distribution into a hierarchy of regions, such that the region near any particle in question is explored in detail, and the more distant regions are explored only coarsely as single massive pseudo-particles. Because Tree Codes are grid-less they waste no time simulating regions devoid of matter. They are of computational complexity $O(N \log N)$ operations but require a large amount of storage.

2.7 Tree-Code (TC) Bottom Up

In this scheme the tree is built from the bottom up, rather than the top down, by taking the two particles nearest each other, setting them as leaf nodes and tying them together. Then recursively continuine to doing so until there is only one node left. Next do the same thing, except do it with the centers of mass (CM) of each of these particle pairs. One feature of this algorithm is that each of the particles and each of the CM's all the way up the tree can be given individual time steps but such that every time-step is some power-of-two multiple of the smallest time step. This allows for easily synchronize the time-steps of all the particles, while simultaneously allowing for small time-steps in subsections of the system where inter-particle distances are small thus handling them correctly. This algorithm can also use perturbation techniques and the computational order is $O(N \log N)$. The drawback is that the method is not as simple as other algorithms.

2.8 Tree-Code Particle Mesh (TPM)

This method uses a particle-mesh (PM) algorithm to handle long range interactions in all areas and a multiple tree-code (TC) approach to handle short range interactions in regions where the mass-density is high. The TPM method solves the Poisson equation. The equations of motion are integrated from the potential and the tree-forces $F = \sum_{ijk} w_i w_j w_k \nabla(phi)) + F(tree)$ using a leap-frog scheme, where the w are "Cloud-In-Cell" weights for the PM. Multiple time scale integrations are implemented in this scheme as the particles in high density regions have different dynamic time scales from the particles in low density regions. The order of computation is between $O(\log N)$ and $O(N \log N)$. One drawback to this method is that it has more free variables than the other methods.

2.9 Self-Consistent Field (SCF)

In the Self-Consistent Field (SCF) the particles do not interact directly with each-other but rather through their contribution to the combined gravitational field of the system. The orbits in a time evolving potential are computed to find the coevolving density distribution. The result is then cyclicly used to recalculate itself with greater precision provided that there are no collisions. Effectively, N one-body problems are iteratively solved, rather than one N-body problem. The SCF method is an $O(N)$ method. It retains tightly-bound particles, and may be capable of resolving steeper density gradients than direct N-body techniques. One drawback is that it depends greatly on specific expansions to the interactions being studied and is inflexible towards other interactions.

2.10 Symplectic Method

This method is based on Hamiltonian mechanics which models conservative natural phenomena such as the motion of the heavenly bodies. Hamiltonian systems preserve energy as well as symplectic invariants that characterize the long term dynamics of nearby points in phase space. Since the flow of a Hamiltonian system is symplectic, it seems not unreasonable to ask the same of a discrete (numerical) approximation. Standard numerical integration schemes do not respect the global geometry present from the Hamiltonian nature of the dynamics. Even if energy conservation is built into the integrations, numerical integration techniques for solar system dynamics is limited to the investigation of short-time quantitative phenomena only, and not to long-time qualitative phenomena investigations. Symplecticness is one of the few principles we have for evaluating the behavior of schemes for conservative nonlinear evolutionary phenomena. Numerical methods have been devised to approximate the time-Δtime map of the exact dynamics to any desired order in the time-step, and are symplectic. These algorithms give the exact evolution of the Hamiltonian system and preserve the Poincare invariants, namely the integrals: $\int\int\sum_i dq_i dp_i$ and $\int\int\int\int\sum_{i\neq k} dq_i dp_i dq_k dp_k$.

2.11 Fast-Multipole-Method (FMM)

The Fast Multipole Method (FMM) is a tree code that uses a far field multipole expansion and a short field direct calculation to represent the potential in stead of the force. The essence of the FMM is a kind of Taylor expansion of the potential worked into a hierachercl divide-and-conquer paradigm. Given a distribution of particles, called the sample space, the algorithm divides and subdivides this space into increasingly finer regions. Each region that is divided into smaller regions is called the parent region and each of the smaller subdivided regions are called the child regions. The level of the finest child regions is predetermined by an accuracy parameter, set by the user. Once the finest level has been reached, the algorithm starts by using a multipole expansion to calculate the potential due to each child region in any other region of the sample space. As will be shown, the multipole expansion of a potential is a summation of terms, grouped in orders of the the vector separation of the distribution of particles divided by the distance at which the distribution is being considered. Therefore the calculation can be truncated at a predetermined order by ignoring all higher order terms. This is again a measure of the accuracy and depends greatly on the separation distances of each particle being considered. Conversely, the order of truncation determines the distances from this child region at which the multipole expansion is valid. Thus in this method there is a trade off between accuracy and speed which is determined by the order of expansion used. The decision of whether the mass and center of mass of a box can be used, or whether it must be decomposed into its children, depends only on the position and size of the box, not on the distribution of particles within the box.

After the initial hieracharcle decent, the algorithm then ascends the parent-child levels, summing the contributions of the potential due to each child subdivision in order to solve for the multipole expansion of the potential in the parent division. The potential in each parent division is therefore due to all the particles contained therein. Once the algorithm has reached the coarsest division, the original sample space, and has summed the multipole contributions from all constituent particles, it then reverses the order and works its way back to the finest level. For the potential at any point in question, the algorithm determines the regions for which the multipole expansion of the potential is valid. In the regions of interaction where in the potential is not valid, the algorithm explicitly calculates the interactions by the use of adouble sum and then adds the two results. This algorithm not as suitable for studies involving close inter-particle distances because it is slower than other methods and because it calculates the interactions of an arbitrary number of particles in essentially constant time. Thus it is better suited for applications which have uniform time steps. The FMM is capable of achieving full machine precision and is arguably an $O(N)$ algorithm.

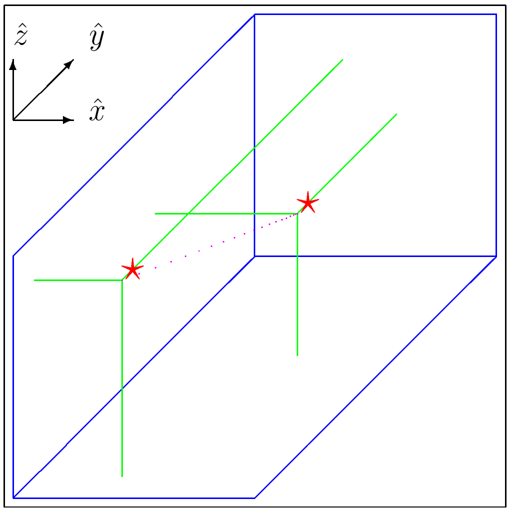


Figure 1: Two Particles In A Three Dimensional System

3 Depection Of Multipole Method

This section is a little more graphical recap of the multipole algorithm presented in the previous section. First concider two particles whose interaction is depicted in figure (1).

The idea is that a system such as figure (1) is easy to calculate using a direct approach, but as the system becomes more like figure (2) this type of approach becomes more and more difficult for a computer to handle.

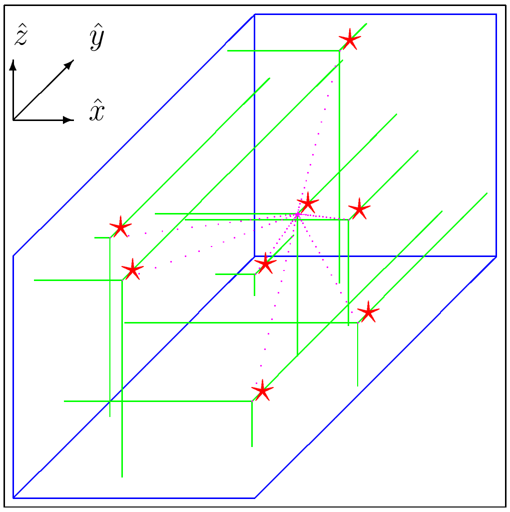


Figure 2: A More Complex Three Dimensional System

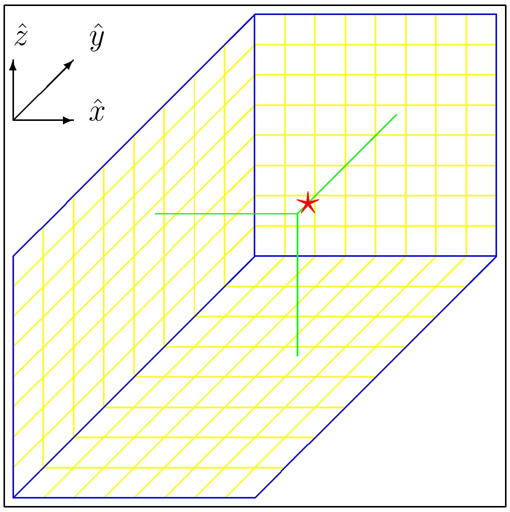


Figure 3: Any Particle In A Three Dimensional System

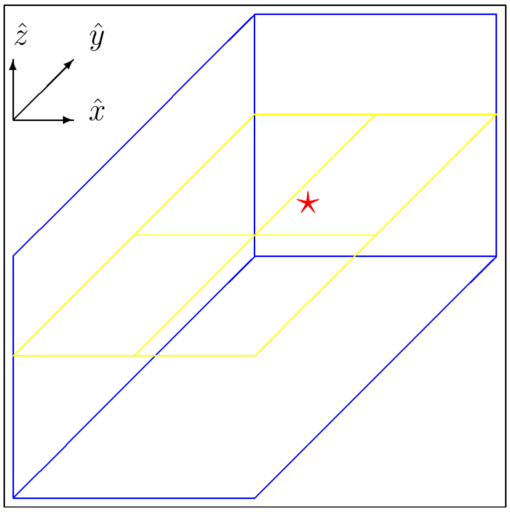


Figure 4: Any Particle in Level 1 Of A Multipole Algorithm

The solution is to start of with any particle in the largest region of a system as depicted in figure (3). The algorithm then begins to devide the region into smaller regions as in figure (4). As the algorithm continues to devide the regions into even smaller regions, if a minimum distance of seperation is met then the algorithm evaluates the multipole expansion in that region.

If this condition isn't met, the algorithm continues to subdivide untill it reaches a certain lowest level; see figures (5) and (6).

At this level all remaining regions are evaluated, either directly or through a multipole expansion and then they are summed with all previously calculated regions. This is a standard divide and conquer strategy.

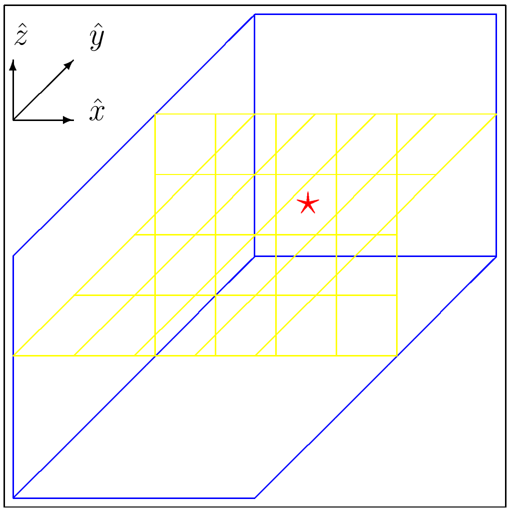


Figure 5: Level 2 Of A Three Dimensional System

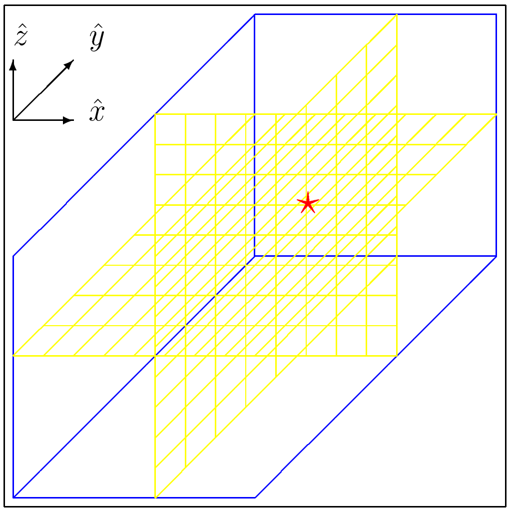


Figure 6: Lowest Level Of A Three Dimensional System

4 Multipole Theory

4.1 General Multipole Expansion In 3 Dimensions

The electrostatic potential due to a collection of point charges is:

$$V(\vec{d}) = \frac{1}{4\cdot\pi\cdot\varepsilon_0}\cdot\sum_{i=1}^{N}\frac{q_i}{\left|\vec{\Re}_i\right|}$$

where the vector $\vec{\Re}$ is given by

$$\vec{\Re}_i = \vec{r}_i - \vec{d}$$

squaring both sides,

$$\Re_i^2 = d^2 + r_i^2 - 2\cdot d\cdot r_i\cdot\cos(\theta_i)$$

$$\Re_i^2 = d^2\cdot\left[1 + \left(\frac{r_i}{d}\right)^2 - 2\cdot\frac{r_i}{d}\cdot\cos(\theta_i)\right]$$

and solving for $\Re$ yields:

$$\Re_i = d\cdot\sqrt{1+\xi_i} \quad \text{where} \quad \xi_i = \frac{r_i}{d}\cdot\left(\frac{r_i}{d} - 2\cdot\cos(\theta_i)\right)$$

therefore, expanding the inverse of $\Re$ in powers of ξ:

$$\frac{1}{\Re_i} = \frac{1}{d}\cdot(1+\xi_i)^{\frac{-1}{2}} = \frac{1}{d}\cdot\left(1 - \frac{1}{2}\cdot\xi_i + \frac{3}{8}\cdot\xi_i^2 - \frac{5}{16}\cdot\xi_i^3 + O(\xi_i)\right)$$

and substituting back in the value of ξ:

$$\frac{1}{\Re_i} = \frac{1}{d}\cdot\left[\begin{matrix} 1 + \frac{r_i}{d}\cdot\cos(\theta_i) + \left(\frac{r_i}{d}\right)^2\cdot\frac{3\cdot\cos(\theta_i)^2 - 1}{2} \ldots \\ + \left(\frac{r_i}{d}\right)^3\cdot\frac{5\cdot\cos(\theta_i)^3 - 3\cdot\cos(\theta_i)}{2} + O\left(\frac{r_i}{d}\right)\end{matrix}\right]$$

such that the inverse of $\Re$, in terms of the Legendre polynomials, becomes

$$\frac{1}{\Re_i} = \frac{1}{d}\cdot\sum_{n=0}^{\infty}\left(\frac{r_i}{d}\right)^n\cdot P_n(\cos(\theta_i))$$

The potential can therefore be written as:

$$V(\vec{d}) = \frac{1}{4\cdot\pi\cdot\varepsilon_0}\cdot\sum_{n=0}^{\infty}\sum_{i=1}^{N}\frac{q_i}{d}\cdot\left(\frac{r_i}{d}\right)^n\cdot P_n(\cos(\theta_i))$$

by convention, this is usually rewritten as:

$$V(\vec{d}) := \frac{1}{4\cdot\pi\cdot\varepsilon_0}\cdot\left[\frac{Q}{d} + \frac{\vec{p}\cdot\hat{d}}{d^2} + \frac{\frac{1}{2}\cdot\sum_{(i,j)}\hat{r}_i\cdot\hat{r}_j\cdot Q_{ij}}{d^3} \text{.. "Other Terms"}\right]$$

where

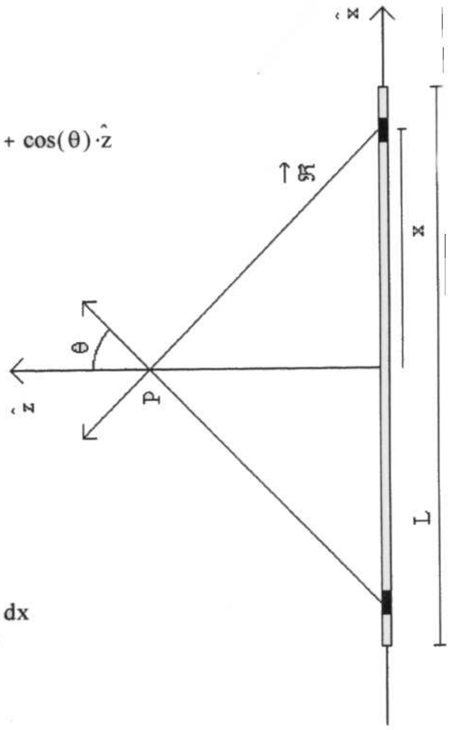
$$Q = \int \rho(r')\,d\tau' \qquad \vec{p} = \int r'\cdot\rho(r')\,d\tau' \qquad Q_{ij} = \int \left(3\cdot r'_i\cdot r'_j - r'^2\cdot\delta_{ij}\right)\cdot\rho(r')\,d\tau'$$

To make the transition from two to three dimensions in the calculation the electrostatic potential, the inverse dependence on the distance to each point is replaced by a log. This concept is well illustrated by solving for the electric field in three dimensions due to a line charge.

4.2 The Potential Due To A Line Charge

The electric field do to a line charge is given by:

$$E\left(\vec{r}\right) = \frac{1}{4\cdot\pi\cdot\varepsilon_0}\cdot\int \frac{\lambda\left(\vec{r'}\right)}{\Re^2}\cdot\hat{\Re}\; dl'$$



where the radial vector is given by:

$$\vec{\Re} = \Re\cdot\hat{\Re} \qquad \Re = \left(z^2 + x^2\right)^{\frac{1}{2}} \qquad \hat{\Re} = \sin(\theta)\cdot\hat{x} + \cos(\theta)\cdot\hat{z}$$

using standard trigonometry,

$$\sin(\theta) = \frac{x}{\Re} \qquad \cos(\theta) = \frac{z}{\Re}$$

The integrand can be expressed as:

$$\frac{\hat{\Re}}{\Re^2} = \frac{x}{\left(z^2 + x^2\right)^{\frac{3}{2}}}\cdot\hat{x} + \frac{z}{\left(z^2 + x^2\right)^{\frac{3}{2}}}\cdot\hat{z}$$

The electric field is therefore given by:

$$E\left(\vec{r}\right) = \frac{\lambda\cdot\hat{x}}{4\cdot\pi\cdot\varepsilon_0}\cdot\int_{\frac{-L}{2}}^{\frac{L}{2}} \frac{x}{\left(z^2 + x^2\right)^{\frac{3}{2}}}\,dx + \frac{\lambda\cdot z\cdot\hat{z}}{4\cdot\pi\cdot\varepsilon_0}\cdot\int_{\frac{-L}{2}}^{\frac{L}{2}} \frac{1}{\left(z^2 + x^2\right)^{\frac{3}{2}}}\,dx$$

Evaluating each of the integrals:

$$\int_{\frac{-L}{2}}^{\frac{L}{2}} \frac{x}{\left(z^2 + x^2\right)^{\frac{3}{2}}}\,dx = 0 \qquad \int_{\frac{-L}{2}}^{\frac{L}{2}} \frac{1}{\left(z^2 + x^2\right)^{\frac{3}{2}}}\,dx = \frac{2\cdot L}{z^2\cdot\sqrt{4\cdot z^2 + L^2}}$$

yields the following result:

$$E\left(\vec{r}\right) = \frac{2\cdot L\cdot\lambda\cdot\hat{z}}{4\cdot\pi\cdot\varepsilon_0\cdot z\cdot\sqrt{4\cdot z^2 + L^2}}$$

Taking the limit as the length of the line becomes infinite:

$$\lim_{L\to\infty} \frac{2\cdot L\cdot\lambda\cdot\hat{z}}{4\cdot\pi\cdot\varepsilon_0\cdot z\cdot\sqrt{4\cdot z^2 + L^2}} = \frac{\lambda\cdot\hat{z}}{2\cdot\pi\cdot\varepsilon_0\cdot z}$$

It was found that the field varies as the inverse of the distance to this line charge. This approach represents a two dimensional translational invariance. This is exactly the model of Coloumb's potential needed and therefore the electric field in two dimensions varies as the inverse of the distance and not the inverse of the distance squared.

4.3 General Multipole Expansion In 2 Dimensions

As shown above, the electrostatic potential in two dimensions, is:

$$V\left(\vec{d}\right) = \frac{-1}{4\cdot\pi\cdot\varepsilon_0}\cdot\sum_{i=0}^{N} q_i\cdot\log\left(\left|\vec{\Re}_i\right|\right)$$

where the vector $\vec{\Re}$ is given by

$$\vec{\Re}_i = \vec{r}_i - \vec{d}$$

squaring both sides,

$$\Re_i^2 = d^2 + r_i^2 - 2\cdot d\cdot r_i\cdot\cos\left(\theta_i\right)$$

$$\Re_i^2 = d^2\cdot\left[1 + \left(\frac{r_i}{d}\right)^2 - 2\cdot\frac{r_i}{d}\cdot\cos\left(\theta_i\right)\right]$$

and solving for $\Re$ yields:

$$\Re_i = d\cdot\sqrt{1-\xi_i} \quad \text{where } \xi_i = \frac{r_i}{d}\cdot\left(2\cdot\cos\left(\theta_i\right) - \frac{r_i}{d}\right)$$

The log then simplifies as:

$$\log\left(\Re_i\right) = \log\left(d\cdot\sqrt{1-\xi_i}\right) = \log(d) + \frac{1}{2}\cdot\log\left(1-\xi_i\right)$$

The second log can be expanded in a series:

$$\log\left(1-\xi_i\right) = \sum_{k=1}^{\infty} \frac{-1}{k}\cdot\xi_i^{\,k}$$

The potential can therefore be written as:

$$V\left(\vec{d}\right) = \frac{-1}{4\cdot\pi\cdot\varepsilon_0}\cdot\sum_{i=1}^{N} q_i\cdot\left(\log(d) - \sum_{k=1}^{\infty}\frac{1}{2\cdot k}\cdot\xi_i^{\,k}\right)$$

$$V\left(\vec{d}\right) = \frac{1}{4\cdot\pi\cdot\varepsilon_0}\cdot\sum_{k=1}^{\infty}\sum_{i=1}^{N}\frac{q_i}{2\cdot k}\cdot\xi_i^{\,k} - \frac{1}{4\cdot\pi\cdot\varepsilon_0}\cdot\sum_{i=1}^{N} q_i\cdot\log(d)$$

defining:

$$a_k = \sum_{i=1}^{N}\frac{q_i}{2\cdot k}\cdot\left(\frac{r_i}{d}\right)^k\cdot\left(2\cdot\cos\left(\theta_i\right) - \frac{r_i}{d}\right)^k$$

this simplifies to:

$$a_k = \sum_{n=0}^{k}\sum_{i=1}^{N}\frac{(-1)^n\cdot 2^{k-n-1}\cdot q_i\cdot(k-1)!}{n!\cdot(k-n)!}\cdot\left(\frac{r_i}{d}\right)^{k+n}\cdot\cos\left(\theta_i\right)^{k-n}$$

and finally the potential can be written as:

$$V\left(\vec{d}\right) = \frac{1}{4\cdot\pi\cdot\varepsilon_0}\cdot\sum_{k=1}^{\infty} a_k - \frac{1}{4\cdot\pi\cdot\varepsilon_0}\cdot Q\cdot\log(d) \quad \text{where } Q = \sum_{i=1}^{N} q_i$$

4.4 Generic Comparison Of Multipole Expansions

Alternativly, replace $\Re_i$ in the steps above with a complex scalar quantity:

$$z_j - z_0 = \Re_{jx} + i\cdot\Re_{jy}$$

The magnitude of the vector $\overrightarrow{\Re_i}$ is then given by

$$\left|\overrightarrow{\Re_i}\right| = \left|z_0 - z_i\right|$$

The log then simplifies as:

$$\log\left(z_0 - z_j\right) = \log\left[z_0\cdot\left(1 - \frac{z_i}{z_0}\right)\right] = \log\left(z_0\right) + \log\left(1 - \frac{z_i}{z_0}\right)$$

The second log can be expanded in a series:

$$\log\left(1 - \frac{z_i}{z_0}\right) = -\sum_{k=1}^{\infty} \frac{1}{k}\cdot\left(\frac{z_i}{z_0}\right)^k$$

The potential can therefore be written as:

$$V\left(\vec{d}\right) = \frac{-1}{4\cdot\pi\cdot\varepsilon_0}\cdot\sum_{i=1}^{N} q_i\cdot\left[\log\left(z_0\right) - \sum_{k=1}^{\infty}\frac{1}{k}\cdot\left(\frac{z_i}{z_0}\right)^k\right]$$

$$V\left(\vec{d}\right) = \frac{1}{4\cdot\pi\cdot\varepsilon_0}\cdot\sum_{k=1}^{\infty}\sum_{i=1}^{N}\frac{q_i}{k}\cdot\left(\frac{z_i}{z_0}\right)^k - \frac{1}{4\cdot\pi\cdot\varepsilon_0}\cdot\sum_{i=1}^{N} q_i\cdot\log\left(z_0\right)$$

and finally as:

$$V\left(\vec{d}\right) = \frac{1}{4\cdot\pi\cdot\varepsilon_0}\cdot\sum_{k=1}^{\infty}\frac{a_k}{z_0^{\,k}} - \frac{1}{4\cdot\pi\cdot\varepsilon_0}\cdot Q\cdot\log\left(z_0\right) \quad \text{where } a_k := \sum^{N}\frac{q_i\cdot z_i^{\,k}}{k} \text{ and } Q := \sum^{N} q_i$$

For a comparison:

The multipole expansion in three dimensions resulted in:

$$V\left(\vec{d}\right) = \frac{1}{4\cdot\pi\cdot\varepsilon_0}\cdot\sum_{n=1}^{\infty}\sum_{i=1}^{N}\frac{q_i}{d}\cdot\left(\frac{r_i}{d}\right)^n\cdot P_n\left(\cos\left(\theta_i\right)\right) + \frac{1}{4\cdot\pi\cdot\varepsilon_0}\cdot\sum_{i=1}^{N}\frac{q_i}{d}$$

while the multipole expansion in two dimensions resulted in:

$$V\left(\vec{d}\right) = \frac{1}{4\cdot\pi\cdot\varepsilon_0}\cdot\sum_{n=1}^{\infty}\sum_{i=1}^{N}\sum_{k=0}^{n}\frac{(-1)^k\cdot 2^{n-k-1}\cdot q_i\cdot(n-1)!}{k!\cdot(n-k)!}\cdot\left(\frac{r_i}{d}\right)^{n+k}\cdot\cos\left(\theta_i\right)^{n-k} \ldots$$
$$+\frac{-1}{4\cdot\pi\cdot\varepsilon_0}\cdot\sum_{i=1}^{N} q_i\cdot\log(d)$$

The differences between these two results can be seen by replacing the inverse distance dependence in the monopole term of the three dimensional expansion with: $\frac{1}{d} \leftarrow -\log(d)$

and replacing the Legendre polynomials with: $P_n\left(\cos\left(\theta_i\right)\right) \leftarrow \frac{d}{2\cdot n}\cdot\left(2\cdot\cos\left(\theta_i\right) - \frac{r_i}{d}\right)^k$

4.5 Validity Of The Multipole Expansion

Figure A

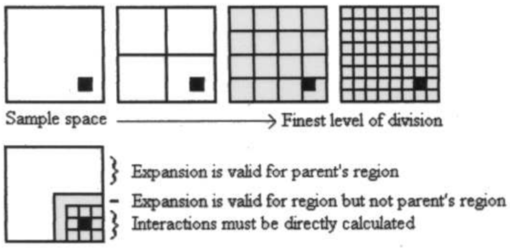


Figure B

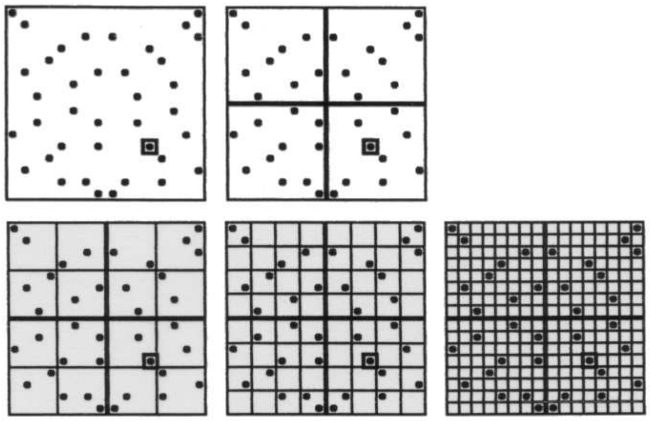
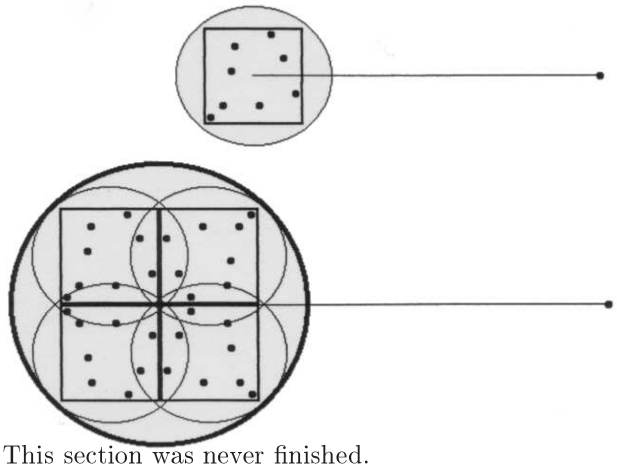
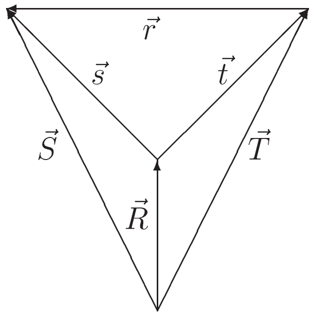


Figure C



This section was never finished.

4.6 Translational Operations in Two Dimensions



The following derivation relates the coefficients of multipole expansions of the same particles with respect to different origins.

$$\Phi(t) = -q\ln\left(|r|\right) \tag{1}$$

$$\Phi(t) = -q\ln\left(|s-t|\right) \tag{2}$$

$$\Phi(t) = -q\ln\left(t-s\right) \tag{3}$$

$$\Phi(t) = -q\ln\left(t\right) - q\ln\left(1-\frac{s}{t}\right) \tag{4}$$

$$\Phi(t) = -q\ln\left(t\right) - (-q)\sum_{m=1}^{\infty}\frac{1}{m}\left(\frac{s}{t}\right)^m \tag{5}$$

$$\Phi(t) = a_0\ln\left(t\right) - \sum_{m=1}^{\infty}\frac{a_m}{mt^m} \tag{6}$$

$$a_m = \sum_{n=1}^{N} -q_n s_n^m \tag{7}$$

$$\Phi(T) = b_0\ln\left(T\right) - \sum_{m=1}^{\infty}\frac{b_m}{mT^m} \tag{8}$$

$$b_m = \sum_{n=1}^{N} -q_n S_n^m \tag{9}$$

$$S_n = s_n + R \tag{10}$$

$$b_m = \sum_{n=1}^{N} -q_n\left(s_n+R\right)^m \tag{11}$$

$$b_m = \sum_{n=1}^{N} -q_n\sum_{k=0}^{m}\frac{m!}{(m-k)!k!}R^{m-k}s_n^k \tag{12}$$

$$b_m = \sum_{k=0}^{m}\frac{m!R^{m-k}}{(m-k)!k!}a_k \tag{13}$$

```
/*****************************************************************************
 *
 *
```

5 Algorithm

```
 *
 *****************************************************************************/

#include<stdlib.h>

#include<unistd.h>

#include<fcntl.h>

#include<stdio.h>

#include<math.h>

#include<time.h>

#define numberOfParticles 10000

#define numberOfTerms     10

#define numberOfLevels    6

#define _O_RDWR  002

#define _O_CREAT 100

#define _S_IRUSR 400

#define _S_IWUSR 200

/*****************************************************************************
 *
 *
```

5.1 Bitmap Output Routines

```
 *
 *****************************************************************************/

int sizeof_BITMAPINFOHEADER = 0x00000028;

int sizeof_BITMAPFILEHEADER = 0x0000000e;

int sizeof_BITMAPHEADER     = sizeof_BITMAPINFOHEADER+

                              sizeof_BITMAPFILEHEADER;

const int default_biWidth   = 0x021c;//540

const int default_biHeight  = 0x0288;//648
```

```
const int default_biBitCount= 0x0018;
int default_Datasize        = default_biWidth*
                              default_biHeight*
                              default_biBitCount/8;
int default_bfSize          = sizeof_BITMAPHEADER+
                              default_Datasize;
typedef unsigned short int  WORD  __attribute__ ((aligned(2)));
typedef unsigned int        DWORD __attribute__ ((aligned(2)));
typedef unsigned int        LONG  __attribute__ ((aligned(2)));
typedef struct
  WORD  bfType;             // file type, must be "BM"
  DWORD bfSize;             // size of file in bytes
  WORD  bfReserved1;        // must be zero
  WORD  bfReserved2;        // must be zero
  DWORD bfOffBits;          // offset to bitmap bits, in bytes
BITMAPFILEHEADER;
typedef struct
  DWORD biSize;             // number of bytes required by the structure
  LONG  biWidth;            // width of the bitmap, in pixels
  LONG  biHeight;           // height of the bitmap, in pixels
  WORD  biPlanes;           // number of planes of device, must be 1
  WORD  biBitCount;         // number of bits-per-pixel
  DWORD biCompression;      // type of compression for a bottom-up bitmap
  DWORD biSizeImage;        // size of image, in bytes
  LONG  biXPelsPerMeter;    // horizontal resolution of device, in pixels/meter
  LONG  biYPelsPerMeter;    // vertical resolution of device, in pixels/meter
  DWORD biClrUsed;          // number of color indexes used in the color table
  DWORD biClrImportant;     // number of color indexes required for displaying
BITMAPINFOHEADER;
BITMAPFILEHEADER fileHeader=
  0x4d42,
```

```
  default_bfSize,

  0x0000,

  0x0000,

  sizeof_BITMAPHEADER
;

BITMAPINFOHEADER infoHeader=

  sizeof_BITMAPINFOHEADER,

  default_biWidth,

  default_biHeight,

  0x0001,

  default_biBitCount,

  0x0000,

  0x00000000,

  0x00000000,

  0x00000000,

  0x00000000,

  0x00000000
;

char image[default_biHeight]

          [default_biWidth]

          [default_biBitCount/8];

void initializeBitmap()

  for(int i=default_biHeight-1;i>=0;--i)

    for(int j=0;j<default_biWidth;j++)

      for(int k=0;k<(default_biBitCount/8);k++)

image[i][j][k]=0xff;


void outputBitmap()

  int output=open("output.bmp",_O_RDWR|_O_CREAT,_S_IRUSR|_S_IWUSR);

  write(output,&fileHeader,sizeof_BITMAPFILEHEADER);

  write(output,&infoHeader,sizeof_BITMAPINFOHEADER);

  write(output,image,default_Datasize);
```

```
  close(output);

void initializeGlyphs();

unsigned long long int glyph[0x80][0x02];

void putGlyph(int index,int x,int y,char r,char g,char b)

  unsigned char bitIsolator;

  char *row;

  y=default_biHeight-y;

  for(int k=0;k<2;k++)

      row=(char *)&glyph[index][k];

      for(int yinc=8*(1-k);yinc<8*(2-k);yinc++)

  bitIsolator=0x80;

  for(int xinc=0;xinc!=8;xinc++)

      if(bitIsolator&*row)

  image[y+yinc][x+xinc][0]=b;

  image[y+yinc][x+xinc][1]=g;

  image[y+yinc][x+xinc][2]=r;

      bitIsolator/=2;

  row++;



void putString(char *string,int x,int y,char r,char g,char b)

  for(int i=0;i<strlen(string);i++)

      putGlyph(string[i],x+i*8,y,r,g,b);


void putLine(int x0,int y0,int dx,int dy,int length,char r,char g,char b)

  int y=default_biHeight-y0;

  for(int x=x0;(x<x0+length)&&(y<default_biHeight-y0+length);x+=dx,y+=dy)

      image[y][x][0]=b;

      image[y][x][1]=g;

      image[y][x][2]=r;
```

```
void initializeBox(int yOfset)
  putLine(0x0b8,0x030+yOfset,1,0,0x110,0x00,0x00,0xff);
  putLine(0x0b8,0x140+yOfset,1,0,0x110,0x00,0x00,0xff);
  putLine(0x0b8,0x140+yOfset,0,1,0x110,0x00,0x00,0xff);
  putLine(0x1c8,0x140+yOfset,0,1,0x110,0x00,0x00,0xff);

void initializeCoordinateBox(int yOfset)
  putLine(0x140,0x138+yOfset,1,1,0x080,0xff,0x00,0x00);
  putLine(0x0c0,0x138+yOfset,1,0,0x080,0xff,0x00,0x00);
  putLine(0x0c0,0x138+yOfset,0,1,0x080,0xff,0x00,0x00);
  putLine(0x0c0,0x138+yOfset,1,1,0x080,0xff,0x00,0x00);
  putLine(0x0c0,0x0b8+yOfset,1,1,0x080,0xff,0x00,0x00);
  putLine(0x140,0x0b8+yOfset,0,1,0x080,0xff,0x00,0x00);
  putLine(0x140,0x0b8+yOfset,1,0,0x080,0xff,0x00,0x00);
  putLine(0x140,0x038+yOfset,1,0,0x080,0xff,0x00,0x00);
  putLine(0x1c0,0x0b8+yOfset,0,1,0x080,0xff,0x00,0x00);

int asciiSizeOfTime(unsigned int time)
  if(time<10)return 1;
  return asciiSizeOfTime(time/10)+1;

char *convertToAscii(unsigned int time)
  int index,result;
  int asciiSize=asciiSizeOfTime(time);
  char *ascii=(char *)malloc(asciiSize);
  for(index=asciiSize-1;index>-1;index--)
      result=time%10;
      time/=10;
      ascii[index]=0x30+result;

  ascii[asciiSize]=0x00;
  return ascii;
```

```
char *NUMOfParticles=convertToAscii(numberOfParticles);

void initializeGraph()

  putString("For N =        Particles",0xc0,0x10,0x00,0x00,0x00);

  putString(NUMOfParticles,0x100,0x10,0x00,0x00,0x00);

  putString("The Brute ",0x060,0x050,0x00,0x00,0x00);

  putString("Force     ",0x060,0x060,0x00,0x00,0x00);

  putString("Algorithm ",0x060,0x070,0x00,0x00,0x00);

  putString("Produced  ",0x060,0x080,0x00,0x00,0x00);

  putString("This Graph",0x060,0x090,0x00,0x00,0x00);

  putString("In        ",0x060,0x0a0,0x00,0x00,0x00);

  putString("Second(s) ",0x060,0x0c0,0x00,0x00,0x00);

  initializeBox(0);

  initializeCoordinateBox(0);

  putString("And The   ",0x060,0x190,0x00,0x00,0x00);

  putString("Multipole ",0x060,0x1a0,0x00,0x00,0x00);

  putString("Algorithm ",0x060,0x1b0,0x00,0x00,0x00);

  putString("Produced  ",0x060,0x1c0,0x00,0x00,0x00);

  putString("This Graph",0x060,0x1d0,0x00,0x00,0x00);

  putString("In        ",0x060,0x1e0,0x00,0x00,0x00);

  putString("Second(s) ",0x060,0x200,0x00,0x00,0x00);

  initializeBox(0x140);

  initializeCoordinateBox(0x140);

/******************************************************************************
 *
 *
```

5.2 General N-Body Structures

```
 *
 ******************************************************************************/

typedef struct PARTICLE

  double x;

  double y;
```

```
  double q;
  double bp;
  double mp;
  double rms;
  struct PARTICLE *nextParticle;
 Particle;
Particle particle[numberOfParticles];
int particleInitialize()
  int i;
  double randomTemp;
  for(i=0;i<numberOfParticles;i++)
      particle[i].x=drand48();
      particle[i].y=drand48();
      randomTemp=drand48();
      if(randomTemp>0.5)
  particle[i].q=1.0;
      else
  particle[i].q=-1.0;

  return 1;

void calculateRootMeanSquare()
  int i;
  for(i=0;i<numberOfParticles;i++)
      particle[i].rms=
sqrt(pow(particle[i].bp-particle[i].mp,2))/numberOfParticles;


void colorBitmap()
  int bxOfset=0x0c0;
  int byOfset=0x140;
  int mxOfset=0x0c0;
  int myOfset=0x280;
```

```
  for(int i=0;i<numberOfParticles;i++)

      double bx=128.0*(particle[i].x+particle[i].y)+bxOfset;

      double by=byOfset-128.0*(particle[i].y+particle[i].bp);

      double br=256.0*(particle[i].bp);

      double bg=256.0*(particle[i].bp);

      double bb=256.0*(1.0-particle[i].bp);

      putGlyph('*',(int)bx,(int)by,

       (unsigned char)br,(unsigned char)bg,(unsigned char)bb);

      double mx=128.0*(particle[i].x+particle[i].y)+mxOfset;

      double my=myOfset-128.0*(particle[i].y+particle[i].mp);

      double mr=256.0*(particle[i].mp);

      double mg=256.0*(particle[i].mp);

      double mb=256.0*(1.0-particle[i].mp);

      putGlyph('*',(int)mx,(int)my,

       (unsigned char)mr,(unsigned char)mg,(unsigned char)mb);


typedef structdouble x,y;complex;

double real(complex a)return a.x;

double imaginary(complex a)return a.y;

complex conjugate(complex a)return (complex)a.x,-a.y;

complex negate(complex a)return (complex)(-1)*a.x,(-1)*a.y;

complex scaleradd(complex a,double b)return (complex)a.x+b,a.y;

complex scalersubtract(complex a,double b)return (complex)a.x-b,a.y;

complex scalermultiply(complex a,double b)return (complex)a.x*b,a.y*b;

complex scalerdevide(complex a,double b)return (complex)a.x/b,a.y/b;;

complex add(complex a,complex b)return (complex)a.x+b.x,a.y+b.y;

complex subtract(complex a,complex b)return (complex)a.x-b.x,a.y-b.y;

complex multiply(complex a,complex b)
  return (complex)a.x*b.x-a.y*b.y,a.y*b.x+b.y*a.x;;

complex inverse(complex a)
  return scalerdevide(conjugate(a),real(multiply(conjugate(a),a)));

complex devide(complex a,complex b)return multiply(a,inverse(b));
```

```
complex power(complex a,int b)
  if(b==0)return (complex)1,0;
  return multiply(a,power(a,b-1));

double magnitude(complex a)return sqrt(real(multiply(a,conjugate(a))));

complex logrithm(complex a)
  return (complex)log(magnitude(a)),asin(a.y/magnitude(a));

/*****************************************************************************
 *
 *
```

5.3 Order N Squared Brute Force Algorithm

```
 *
 ****************************************************************************/

void bruteForceAlgorithm()

  int i,j;

  double min,max;

  for(i=0;i<numberOfParticles;i++)

    for(j=0;j<numberOfParticles;j++)

      if(i!=j)

  complex t=particle[i].x,particle[i].y;

  complex s=particle[j].x,particle[j].y;

  particle[i].bp+=(-1)*particle[j].q*

    magnitude(logrithm(add(t,negate(s))));


  min=particle[0].bp;

  max=particle[0].bp;

  for(i=1;i<numberOfParticles;i++)

      if(particle[i].bp>max)max=particle[i].bp;

      if(particle[i].bp<min)min=particle[i].bp;

  for(i=0;i<numberOfParticles;i++)

      particle[i].bp=(particle[i].bp-min)/(max-min);


/*****************************************************************************
 *
 *
```

5.4 Multipole Algorithm

```
 *
 ******************************************************************************/

/******************************************************************************
 *
 *
```

5.5 L. Greengard and V. Rokhlin Algorithm

```
 *
 * The following algorithm was copied from L. Greengard and V. Rokhlin
 *
 * Initialization:
 * The following variables are defined:
 *   Theta[l,i] = multipole expansion due to particles within a region
 *   Psi[l,i]   = multipole expansion due to non-neighbor particles
 *   PsiBar[l,i]= expansion due to particles outside parent's neighbors
 *   Phi[l,i]   = the direct calculation due to all neighbor particles
 * Choose a level of refinement
 *   Setn=log[4](N)andp=log[2](precision)
 *
 * Step1:
 * Form an expansion about the center of each region on the finest level
 * do i box=1,.,4n
 *   Form a p-term multipole expansion Theta[n,ibox]
 *
 * Step2:
 * Form an expansion in larger regions by summing any subdivided regions.
 * Do this for all levels
 * do l=n-1,.,0
 *   do ibox=1,.,4l
 *     Form a multipole expansion Theta[l,ibox] by shifting the center of
 *     each subdivisions boxes to the center of the divisions box and adding
 *     them together
 *
 * Step3:
 * Form an expansion for each region on each level of the potential due
 *   to all non-neighbor regions except for the lowest level
 * Set PsiBar[1,ibox]=PsiBar[2,ibox]=PsiBar[3,ibox]=PsiBar[4,ibox]=(0,0,0,0)
 * do l=1,.,n-1
 *   do ibox=1,.,4l
 *     Form Psi[l,ibox] by converting the multipole expansion Theta[l,j] of
 *     each box j in the interaction list of box i box to a local expansion
 *     about the center of box ibox, adding the local expansions together and
 *     adding the result to PsiBar[l,ibox]
 *   do ibox=1,.,4l
 *     Form the expansion PsiBar[l+1,j] for each subdivision by expanding
 *     Psi[l,ibox] about the subdivisions centers
 *
 * Step4:
 * Compute the expansion on the lowest level.
 * do ibox=1,.,4n
 *   Form Psi[l,ibox] by converting the multipole expansion Theta[l,j] of each
 *   box j in the interaction list of box ibox to a local expansion about the
 *   center of box ibox, adding the local expansions together and adding
 *   the result to PsiBar[l,ibox]
```

```
 *
 * Step5:
 * Evaluate the local expansion at each particle position
 * do ibox=1,.,4n
 *   For all particles p[j] located at point r[j] evaluate Theta[n,ibox](r[j])
 *
 * Step6:
 * Explicitly compute the interactions of all neighbor particles
 * do ibox=1,.,4n
 *   For all particles p[j] calculate Phi[l,j].
 *
 * Step7:
 * Add terms from Step 5 and Step 6 to calculate the total potential
 * do ibox=1,.,4n
 *   For all particles p[j] add Psi[l,j] and Phi[l,j]
 *
 * The following calculations were performed by L.Greengard and V.Rokhlin
 * on the complexity level of each step in thier algorithm.
 *
 * Step Operation Count  Explanation
 * 1    N*p              One calculation per particle on finest level
 * 2    N*p^2            At each higher level l,4l shifts of coordinates
 * 3    <=28*N*p^2       For the entries in the interaction list at each level
 * 4    <=27*N*p^2       Again, the same interaction list
 * 5    <=27*N*p^2       One p-term expansion per particle
 * 6    (9/2)*N*k[n]     k[n] is upper bound on number of particles in finest
 *                       regions where interactions are directly calculated
 * 7    N                Combining the two terms for each particle
 *
 * These steps are all of order linear in N.  Thus the overall complexity of
 * the algorithm is arguably linear as well.  For the worst case scenario, the
 * number of operations goes as:
 * alpha*N
 * where alpha is given by:
 *  alpha=p+83p^2+(9/2)k_n+1
 *
 * In comparison, for the direct method which performs a double sum over all
 * interacting particles, the number of operations goes as:
 *
 *  beta*N^2
 * where beta is typically less that alpha
 *
 ****************************************************************************/

int particleInit=particleInitialize();

typedef struct REGION

  double x;

  double y;

  double a[numberOfTerms][2];

  int numberOfParticlesInRegion;

  struct PARTICLE *nextParticle;

 *Region;

int determineLevel(int r)
```

```
  int l=0;
  int count=0;
  while(r>=count)
      count+=(int)pow(4,l++);
  return l-1;

int geometricSum(int base,int initial,int final)
  if(initial==final)
      return (int)pow(base,final);
  return (int)pow(base,initial)+geometricSum(base,initial+1,final);

int getChildRegion(int r)
  int l=determineLevel(r);
  return (-3)*geometricSum(4,0,l)+4*(r+(int)pow(4,l));

int getParentRegion(int r)
  return (r-1-(r-1)%4)/4;

int initializeNumberOfRegions()
  int numberOfRegions=0,l;
  for(l=0;l<numberOfLevels;l++)
      numberOfRegions=getChildRegion(numberOfRegions);
  return numberOfRegions;

int numberOfRegions=initializeNumberOfRegions();
Region region=(Region)malloc(sizeof(REGION)*numberOfRegions);
void getRegionCoordinates(int r,double &x,double &y)
  double X,Y,Q;
  int P,c,l;
  if(r==0)
      x=0.5;
      y=0.5;
  else
```

```
      l=determineLevel(r);
      P=getParentRegion(r);
      c=getChildRegion(P);
      Q=1.0/pow(2,l+1);
      getRegionCoordinates(P,X,Y);
      if((r%2)==0)
  x=X+Q;
  if(r<c+2)
      y=Y+Q;
  else
      y=Y-Q;

      else
  x=X-Q;
  if(r<c+2)
      y=Y+Q;
  else
      y=Y-Q;



int firstRegionOnLowestLevel=numberOfRegions-(int)pow(4,numberOfLevels-1);
void placeParticleInRegion(int i,int r)
  int c,higherRegion;
  Particle *particlePointer;
  if(r<firstRegionOnLowestLevel)
      c=getChildRegion(r);
      if(particle[i].x<region[r].x)
  if(particle[i].y<region[r].y)
      placeParticleInRegion(i,c+2);
  else
      placeParticleInRegion(i,c);
```

```
    else
  if(particle[i].y<region[r].y)
    placeParticleInRegion(i,c+3);
  else
    placeParticleInRegion(i,c+1);

  else
    higherRegion=r;
    while(higherRegion!=0)
  region[higherRegion].numberOfParticlesInRegion++;
  higherRegion=getParentRegion(higherRegion);

    region[0].numberOfParticlesInRegion++;
    if(region[r].nextParticle==0)
  region[r].nextParticle=&particle[i];
    else
  particlePointer=region[r].nextParticle;
  while(particlePointer->nextParticle!=0)
    particlePointer=particlePointer->nextParticle;
  particlePointer->nextParticle=&particle[i];


void regionInitialize()
  int r,i,l,higherRegion,currNumberOfRegions,firstRegion;
  double x,y;
  Particle *particlePointer;
  for(r=0;r<numberOfRegions;r++)
    getRegionCoordinates(r,x,y);
    region[r].x=x;
    region[r].y=y;
  for(i=0;i<numberOfParticles;i++)
    placeParticleInRegion(i,0);
```

```
  for(r=firstRegionOnLowestLevel;r<numberOfRegions-1;r++)
      if(region[r].nextParticle==0)
  i=1;
  while(region[r+i].nextParticle==0&&(r+i)<(numberOfRegions-1))
      i++;
  region[r].nextParticle=region[r+i].nextParticle;
      else
  particlePointer=region[r].nextParticle;
  while(particlePointer->nextParticle!=0)
      particlePointer=particlePointer->nextParticle;
  i=1;
  while(region[r+i].nextParticle==0&&(r+i)<(numberOfRegions-1))
      i++;
  particlePointer->nextParticle=region[r+i].nextParticle;


  currNumberOfRegions=numberOfRegions;
  for(l=numberOfLevels;l>0;l--)
      firstRegion=currNumberOfRegions-(int)pow(4,l-1);
      r=firstRegion;
      while(r<currNumberOfRegions)
  higherRegion=getParentRegion(r);
  region[higherRegion].nextParticle=region[r].nextParticle;
  r+=4;

      currNumberOfRegions=firstRegion;


void directlyCalculatePotential(int i,int r)
  int n;
  Particle *particlePointer;
  particlePointer=region[r].nextParticle;
  for(n=0;n<region[r].numberOfParticlesInRegion;n++)
```

```
    if(particlePointer!=&particle[i])
  complex t=particle[i].x,particle[i].y;
  complex s=particlePointer->x,particlePointer->y;
  particle[i].mp+=(-1)*particlePointer->q*
   magnitude(logrithm(add(t,negate(s))));
    particlePointer=particlePointer->nextParticle;

void normalizeMultipolePotential()
  int i;
  double min,max;
  min=particle[0].mp;
  max=particle[0].mp;
  for(i=1;i<numberOfParticles;i++)
    if(particle[i].mp>max)max=particle[i].mp;
    if(particle[i].mp<min)min=particle[i].mp;
  for(i=0;i<numberOfParticles;i++)
    particle[i].mp=(particle[i].mp-min)/(max-min);

void calculateBaseMultipoleCoefficients(int r)
  int m,count;
  Particle *particlePointer;
  complex s=region[r].x,region[r].y;
  count=region[r].numberOfParticlesInRegion;
  for(particlePointer=region[r].nextParticle;
    count--!=0;particlePointer=particlePointer->nextParticle)
    region[r].a[0][0]+=(-1)*particlePointer->q;
  for(m=1;m<numberOfTerms;m++)
    count=region[r].numberOfParticlesInRegion;
    for(particlePointer=region[r].nextParticle;
  count--!=0;particlePointer=particlePointer->nextParticle)
  complex t=particlePointer->x,particlePointer->y;
```

```
  complex v=scalermultiply(power(add(t,negate(s)),m),
   (-1)*particlePointer->q);
  region[r].a[m][0]+=real(v);
  region[r].a[m][1]+=imaginary(v);


int factorial(int i)
  if(i==0)
      return 1;
  return i*factorial(i-1);

int Gamma[numberOfTerms][numberOfTerms];
void calculateHigherMultipoleCoefficients(int r)
  int m,k,c,C;
  C=getChildRegion(r);
  for(c=C;c<C+4;c++)
      for(m=0;m<numberOfTerms;m++)
  complex R=region[c].x-region[r].x,region[c].y-region[r].y;
  for(k=0;k<=m;k++)
      complex v=region[c].a[k][0],region[c].a[k][1];
      v=multiply(v,scalermultiply(power(R,m-k),Gamma[m][k]));
      region[r].a[m][0]+=real(v);
      region[r].a[m][1]+=imaginary(v);



void calculateAllMultipoleCoefficients()
  int r,m,k;
  for(m=0;m<numberOfTerms;m++)
      for(k=0;k<=m;k++)
  Gamma[m][k]=factorial(m)/(factorial(m-k)*factorial(k));
```

```
  for(r=firstRegionOnLowestLevel;r<numberOfRegions;r++)
      calculateBaseMultipoleCoefficients(r);

  for(r=firstRegionOnLowestLevel-1;r>=0;r--)
      calculateHigherMultipoleCoefficients(r);

void multipoleExpansionInRegion(int i,int r)

  int m;
  complex v;
  complex u=particle[i].x-region[r].x,particle[i].y-region[r].y;
  particle[i].mp+=region[r].a[0][0]*magnitude(logrithm(u));
  for(m=1;m<numberOfTerms;m++)
      v=(complex)region[r].a[m][0],region[r].a[m][1];
      particle[i].mp-=magnitude(devide(v,scalermultiply(power(u,m),m)));

  //directlyCalculatePotential(i,r);

int particleIsWellSeperatedFromRegion(int i,int r)
  int l;
  double Q;
  l=determineLevel(r);
  Q=1.0/pow(2,l-0.5);
  if(sqrt(pow(region[r].x-particle[i].x,2)+
  pow(region[r].y-particle[i].y,2))>Q)
      return 1;
  else
      return 0;

void calculatePotential(int i,int r);
void divideAndConquerRegion(int i,int r)
  int C=getChildRegion(r);
```

```
  for(int c=C;c<C+4;c++)

      calculatePotential(i,c);


void calculatePotential(int i,int r)

  int l=determineLevel(r);

  int s=particleIsWellSeperatedFromRegion(i,r);

  if(l==numberOfLevels-1)

      directlyCalculatePotential(i,r);

  else if(s==1)

      multipoleExpansionInRegion(i,r);

  else

      divideAndConquerRegion(i,r);


void multipoleAlgorithm()

  int i;

  regionInitialize();

  calculateAllMultipoleCoefficients();

  for(i=0;i<numberOfParticles;i++)

      calculatePotential(i,0);


  normalizeMultipolePotential();

/******************************************************************************
 *
 *
```

5.6 Main Function

```
 *
 ******************************************************************************/

int main()

  time_t initial_time,final_time;

  double timeIncrement;

  char *timeString;
```

```
 initializeGlyphs();
 initializeBitmap();
 initializeGraph();
 initial_time=time(&initial_time);
 bruteForceAlgorithm();
 final_time=time(&final_time);
 timeIncrement=difftime(final_time,initial_time);
 timeString=convertToAscii((unsigned int)timeIncrement);
 putString(timeString,0x060,0x0b0,0x00,0x00,0x00);
 initial_time=time(&initial_time);
 multipoleAlgorithm();
 final_time=time(&final_time);
 timeIncrement=difftime(final_time,initial_time);
 timeString=convertToAscii((unsigned int)timeIncrement);
 putString(timeString,0x060,0x1f0,0x00,0x00,0x00);
 colorBitmap();
 outputBitmap();
 calculateRootMeanSquare();
 char *line="--------";
 printf("i       TAB"
"x       TAB"
"y       TAB"
"q       TAB"
"bp      TAB"
"mp      TAB"
"rms     TAB"
"nextPartNEWLINE");
 printf("%sTAB%sTAB%sTAB%sTAB%sTAB%sTAB%sTAB%sNEWLINE",
line,line,line,line,line,line,line,line);
 for(int i=0;i<numberOfParticles;i++)
   printf("%iTABTAB%fTAB%fTAB%fTAB%fTAB%fTAB%fTAB%xNEWLINE",i,
```

```
   particle[i].x,particle[i].y,particle[i].q,
   particle[i].bp,particle[i].mp,particle[i].rms,
   particle[i].nextParticle);
  printf("NEWLINE"
 "i       TAB"
 "x       TAB"
 "y       TAB"
 "a[0][0] TAB"
 "a[0][1] TAB"
 "a[1][0] TAB"
 "a[1][1] TAB"
 "numPart TAB"
 "nextPartNEWLINE");
  printf("%sTAB%sTAB%sTAB%sTAB%sTAB%sTAB%sTAB%sTAB%sNEWLINE",
 line,line,line,line,line,line,line,line,line);
  for(int i=0;i<numberOfRegions;i++)
    printf("%iTABTAB%fTAB%fTAB%fTAB%fTAB%fTAB%fTAB%iTABTAB%xNEWLINE",i,
   region[i].x,region[i].y,
   region[i].a[0][0],region[i].a[0][1],
   region[i].a[1][0],region[i].a[1][1],
   region[i].numberOfParticlesInRegion,
   region[i].nextParticle);
  exit(0);

/******************************************************************************
 *
 *
```

5.7 Hexidecimal Glyphs

```
 *
 ******************************************************************************/
void initializeGlyphs()
```

```
glyph[0x20][0]=0x0000000000000000ULL;glyph[0x20][1]=0x0000000000000000ULL;
glyph[0x21][0]=0x0000000008080808ULL;glyph[0x21][1]=0x0808080008080000ULL;
glyph[0x22][0]=0x0000222222220000ULL;glyph[0x22][1]=0x0000000000000000ULL;
glyph[0x23][0]=0x000000001212127EULL;glyph[0x23][1]=0x24247E4848480000ULL;
glyph[0x24][0]=0x00000000083E4948ULL;glyph[0x24][1]=0x380E09493E080000ULL;
glyph[0x25][0]=0x00000000314A4A34ULL;glyph[0x25][1]=0x0808162929460000ULL;
glyph[0x26][0]=0x000000001C222222ULL;glyph[0x26][1]=0x1C39454246390000ULL;
glyph[0x27][0]=0x0000080808080000ULL;glyph[0x27][1]=0x0000000000000000ULL;
glyph[0x28][0]=0x0000000408081010ULL;glyph[0x28][1]=0x1010101008080400ULL;
glyph[0x29][0]=0x0000002010100808ULL;glyph[0x29][1]=0x0808080810102000ULL;
glyph[0x2A][0]=0x0000000000000849ULL;glyph[0x2A][1]=0x2A1C2A4908000000ULL;
glyph[0x2B][0]=0x0000000000000808ULL;glyph[0x2B][1]=0x087F080808000000ULL;
glyph[0x2C][0]=0x0000000000000000ULL;glyph[0x2C][1]=0x0000000018080810ULL;
glyph[0x2D][0]=0x0000000000000000ULL;glyph[0x2D][1]=0x007E000000000000ULL;
glyph[0x2E][0]=0x0000000000000000ULL;glyph[0x2E][1]=0x0000000018180000ULL;
glyph[0x2F][0]=0x0000000002020408ULL;glyph[0x2F][1]=0x0810102040400000ULL;
glyph[0x30][0]=0x0000000018244242ULL;glyph[0x30][1]=0x4242424224180000ULL;
glyph[0x31][0]=0x0000000008182808ULL;glyph[0x31][1]=0x08080808083E0000ULL;
glyph[0x32][0]=0x000000003C424202ULL;glyph[0x32][1]=0x0C102040407E0000ULL;
glyph[0x33][0]=0x000000003C424202ULL;glyph[0x33][1]=0x1C020242423C0000ULL;
glyph[0x34][0]=0x00000000040C1424ULL;glyph[0x34][1]=0x44447E0404040000ULL;
glyph[0x35][0]=0x000000007E404040ULL;glyph[0x35][1]=0x7C020202423C0000ULL;
glyph[0x36][0]=0x000000001C204040ULL;glyph[0x36][1]=0x7C424242423C0000ULL;
glyph[0x37][0]=0x000000007E020204ULL;glyph[0x37][1]=0x0404080808080000ULL;
glyph[0x38][0]=0x000000003C424242ULL;glyph[0x38][1]=0x3C424242423C0000ULL;
glyph[0x39][0]=0x000000003C424242ULL;glyph[0x39][1]=0x3E02020204380000ULL;
glyph[0x3A][0]=0x0000000000001818ULL;glyph[0x3A][1]=0x0000001818000000ULL;
glyph[0x3B][0]=0x0000000000001818ULL;glyph[0x3B][1]=0x0000001808081000ULL;
glyph[0x3C][0]=0x0000000000020408ULL;glyph[0x3C][1]=0x1020100804020000ULL;
glyph[0x3D][0]=0x000000000000007EULL;glyph[0x3D][1]=0x0000007E00000000ULL;
glyph[0x3E][0]=0x0000000000402010ULL;glyph[0x3E][1]=0x0804081020400000ULL;
```

```
glyph[0x3F][0]=0x000000003C424202ULL;glyph[0x3F][1]=0x0408080008080000ULL;
glyph[0x40][0]=0x000000001C224A56ULL;glyph[0x40][1]=0x5252524E201E0000ULL;
glyph[0x41][0]=0x0000000018242442ULL;glyph[0x41][1]=0x427E424242420000ULL;
glyph[0x42][0]=0x000000007C424242ULL;glyph[0x42][1]=0x7C424242427C0000ULL;
glyph[0x43][0]=0x000000003C424240ULL;glyph[0x43][1]=0x40404042423C0000ULL;
glyph[0x44][0]=0x0000000078444242ULL;glyph[0x44][1]=0x4242424244780000ULL;
glyph[0x45][0]=0x000000007E404040ULL;glyph[0x45][1]=0x7C404040407E0000ULL;
glyph[0x46][0]=0x000000007E404040ULL;glyph[0x46][1]=0x7C40404040400000ULL;
glyph[0x47][0]=0x000000003C424240ULL;glyph[0x47][1]=0x404E4242463A0000ULL;
glyph[0x48][0]=0x0000000042424242ULL;glyph[0x48][1]=0x7E42424242420000ULL;
glyph[0x49][0]=0x000000003E080808ULL;glyph[0x49][1]=0x08080808083E0000ULL;
glyph[0x4A][0]=0x000000001F040404ULL;glyph[0x4A][1]=0x0404044444380000ULL;
glyph[0x4B][0]=0x0000000042444850ULL;glyph[0x4B][1]=0x6060504844420000ULL;
glyph[0x4C][0]=0x0000000040404040ULL;glyph[0x4C][1]=0x40404040407E0000ULL;
glyph[0x4D][0]=0x0000000042426666ULL;glyph[0x4D][1]=0x5A5A424242420000ULL;
glyph[0x4E][0]=0x0000000042626252ULL;glyph[0x4E][1]=0x524A4A4646420000ULL;
glyph[0x4F][0]=0x000000003C424242ULL;glyph[0x4F][1]=0x42424242423C0000ULL;
glyph[0x50][0]=0x000000007C424242ULL;glyph[0x50][1]=0x7C40404040400000ULL;
glyph[0x51][0]=0x000000003C424242ULL;glyph[0x51][1]=0x4242425A663C0300ULL;
glyph[0x52][0]=0x000000007C424242ULL;glyph[0x52][1]=0x7C48444442420000ULL;
glyph[0x53][0]=0x000000003C424240ULL;glyph[0x53][1]=0x300C0242423C0000ULL;
glyph[0x54][0]=0x000000007F080808ULL;glyph[0x54][1]=0x0808080808080000ULL;
glyph[0x55][0]=0x0000000042424242ULL;glyph[0x55][1]=0x42424242423C0000ULL;
glyph[0x56][0]=0x0000000041414122ULL;glyph[0x56][1]=0x2222141408080000ULL;
glyph[0x57][0]=0x0000000042424242ULL;glyph[0x57][1]=0x5A5A666642420000ULL;
glyph[0x58][0]=0x0000000042422424ULL;glyph[0x58][1]=0x1818242442420000ULL;
glyph[0x59][0]=0x0000000041412222ULL;glyph[0x59][1]=0x1408080808080000ULL;
glyph[0x5A][0]=0x000000007E020204ULL;glyph[0x5A][1]=0x08102040407E0000ULL;
glyph[0x5B][0]=0x0000000E08080808ULL;glyph[0x5B][1]=0x0808080808080E00ULL;
glyph[0x5C][0]=0x0000000040402010ULL;glyph[0x5C][1]=0x1008080402020000ULL;
glyph[0x5D][0]=0x0000007010101010ULL;glyph[0x5D][1]=0x1010101010107000ULL;
```

```
glyph[0x5E][0]=0x0000182442000000ULL;glyph[0x5E][1]=0x0000000000000000ULL;
glyph[0x5F][0]=0x0000000000000000ULL;glyph[0x5F][1]=0x000000000007F00ULL;
glyph[0x60][0]=0x0020100800000000ULL;glyph[0x60][1]=0x0000000000000000ULL;
glyph[0x61][0]=0x0000000000003C42ULL;glyph[0x61][1]=0x023E4242463A0000ULL;
glyph[0x62][0]=0x0000004040405C62ULL;glyph[0x62][1]=0x42424242625C0000ULL;
glyph[0x63][0]=0x0000000000003C42ULL;glyph[0x63][1]=0x40404040423C0000ULL;
glyph[0x64][0]=0x0000000202023A46ULL;glyph[0x64][1]=0x42424242463A0000ULL;
glyph[0x65][0]=0x0000000000003C42ULL;glyph[0x65][1]=0x427E4040423C0000ULL;
glyph[0x66][0]=0x0000000C1010107CULL;glyph[0x66][1]=0x1010101010100000ULL;
glyph[0x67][0]=0x0000000000023A44ULL;glyph[0x67][1]=0x444438203C42423CULL;
glyph[0x68][0]=0x0000004040405C62ULL;glyph[0x68][1]=0x4242424242420000ULL;
glyph[0x69][0]=0x0000000808001808ULL;glyph[0x69][1]=0x08080808083E0000ULL;
glyph[0x6A][0]=0x0000000404000C04ULL;glyph[0x6A][1]=0x0404040404044830ULL;
glyph[0x6B][0]=0x0000000040404448ULL;glyph[0x6B][1]=0x5060504844420000ULL;
glyph[0x6C][0]=0x0000000018080808ULL;glyph[0x6C][1]=0x08080808083E0000ULL;
glyph[0x6D][0]=0x0000000000007649ULL;glyph[0x6D][1]=0x4949494949490000ULL;
glyph[0x6E][0]=0x0000000000005C62ULL;glyph[0x6E][1]=0x4242424242420000ULL;
glyph[0x6F][0]=0x0000000000003C42ULL;glyph[0x6F][1]=0x42424242423C0000ULL;
glyph[0x70][0]=0x0000000000005C62ULL;glyph[0x70][1]=0x42424242625C4040ULL;
glyph[0x71][0]=0x0000000000003A46ULL;glyph[0x71][1]=0x42424242463A0202ULL;
glyph[0x72][0]=0x0000000000005C62ULL;glyph[0x72][1]=0x4240404040400000ULL;
glyph[0x73][0]=0x0000000000003C42ULL;glyph[0x73][1]=0x40300C02423C0000ULL;
glyph[0x74][0]=0x0000000010107C10ULL;glyph[0x74][1]=0x10101010100C0000ULL;
glyph[0x75][0]=0x0000000000004242ULL;glyph[0x75][1]=0x42424242463A0000ULL;
glyph[0x76][0]=0x0000000000004242ULL;glyph[0x76][1]=0x4224242418180000ULL;
glyph[0x77][0]=0x0000000000004149ULL;glyph[0x77][1]=0x4949494949360000ULL;
glyph[0x78][0]=0x0000000000004242ULL;glyph[0x78][1]=0x2418182442420000ULL;
glyph[0x79][0]=0x0000000000004242ULL;glyph[0x79][1]=0x424242261A02023CULL;
glyph[0x7A][0]=0x0000000000007E02ULL;glyph[0x7A][1]=0x04081020407E0000ULL;
glyph[0x7B][0]=0x0000000C10100808ULL;glyph[0x7B][1]=0x1010080810100C00ULL;
glyph[0x7C][0]=0x0000080808080808ULL;glyph[0x7C][1]=0x0808080808080808ULL;
```

```
glyph[0x7D][0]=0x0000003008081010ULL;glyph[0x7D][1]=0x0808101008083000ULL;
glyph[0x7E][0]=0x0000003149460000ULL;glyph[0x7E][1]=0x0000000000000000ULL;
glyph[0x7F][0]=0x0000000000000000ULL;glyph[0x7F][1]=0x0000000000000000ULL;
```

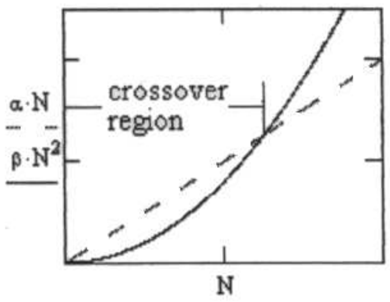
6 Results

The results of the attempted coding of a fast multipole algorithm in both two and three dimensions are varied. Each will now be discussed in turn.

In terms of the three dimensional algorithm the results were total failure. An overwhelming amount of time durring the course of the project was dedicated to the three dimensional algorithm. However, due to a failure in understanding the underlying mathmatical principles there were not even results which could have been discussed in this thesis. A lot of time was spent in vain here.

In terms of the two dimensional algorithm, the end result was a mixed success. A hierarchal algorithm was in fact implemented; the results of which constitute the next four subsections. In analyzing the output, one can quickly notice that the multipole algorithm does not produce an identical output. However, it does produce output which is noticably related to the output of the brute force method. For a more precise investigation, the section labled "Numeric Output for 100 Particles" is included. This is the standard output of the coded multipole program for N=100 particles. There are only three main collums of interest. They are the collum labeled bp, which is the potential calculated by the brute force method, the collum labeled mp, which is the potential calculated by the multipole algorithm, and the collum labeled rms, which is the root mean square of the two. A quick glance down these collums shows that the multipole algorithm produces results which are rarely off by more than a third of the value produced by the brute force algorithm.

The one main success of the implementation of the code in two dimensions is that the execution times outputed by the algorithm show clearly that the algorithm, althought not presently accurate, is much quicker than the brute force algorithm.

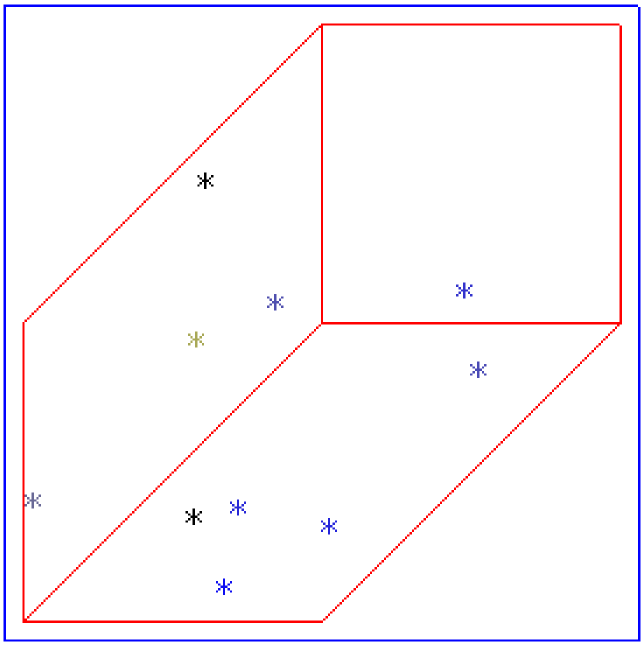


In addition, a plot (as above) of the number of calculations as a function of N shows that there is a cross over region corresponding to a certain number of particles for which the multipole expansion is more costly. As will be evident from the data presented, this crossover region occured somewhere between N=10 and N=1000 particles. For large numbers of particles, the multipole algorithm is much more efficient.

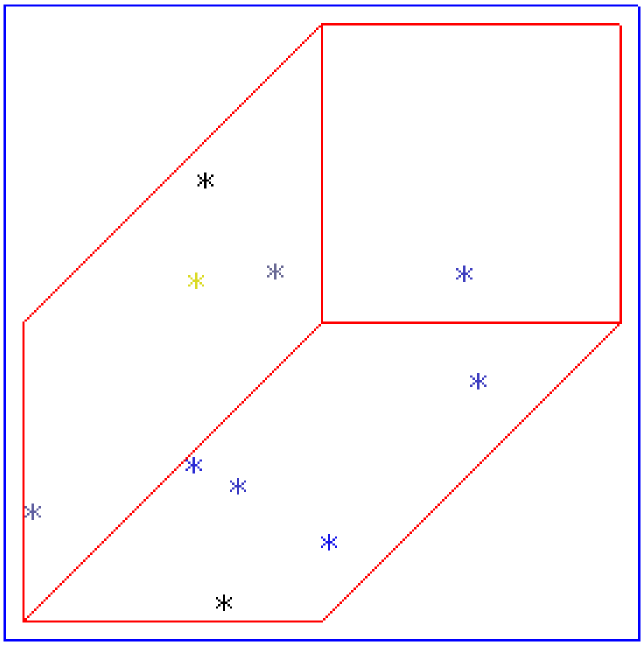
6.1 Output For 10 Particles

For N = 10 Particles

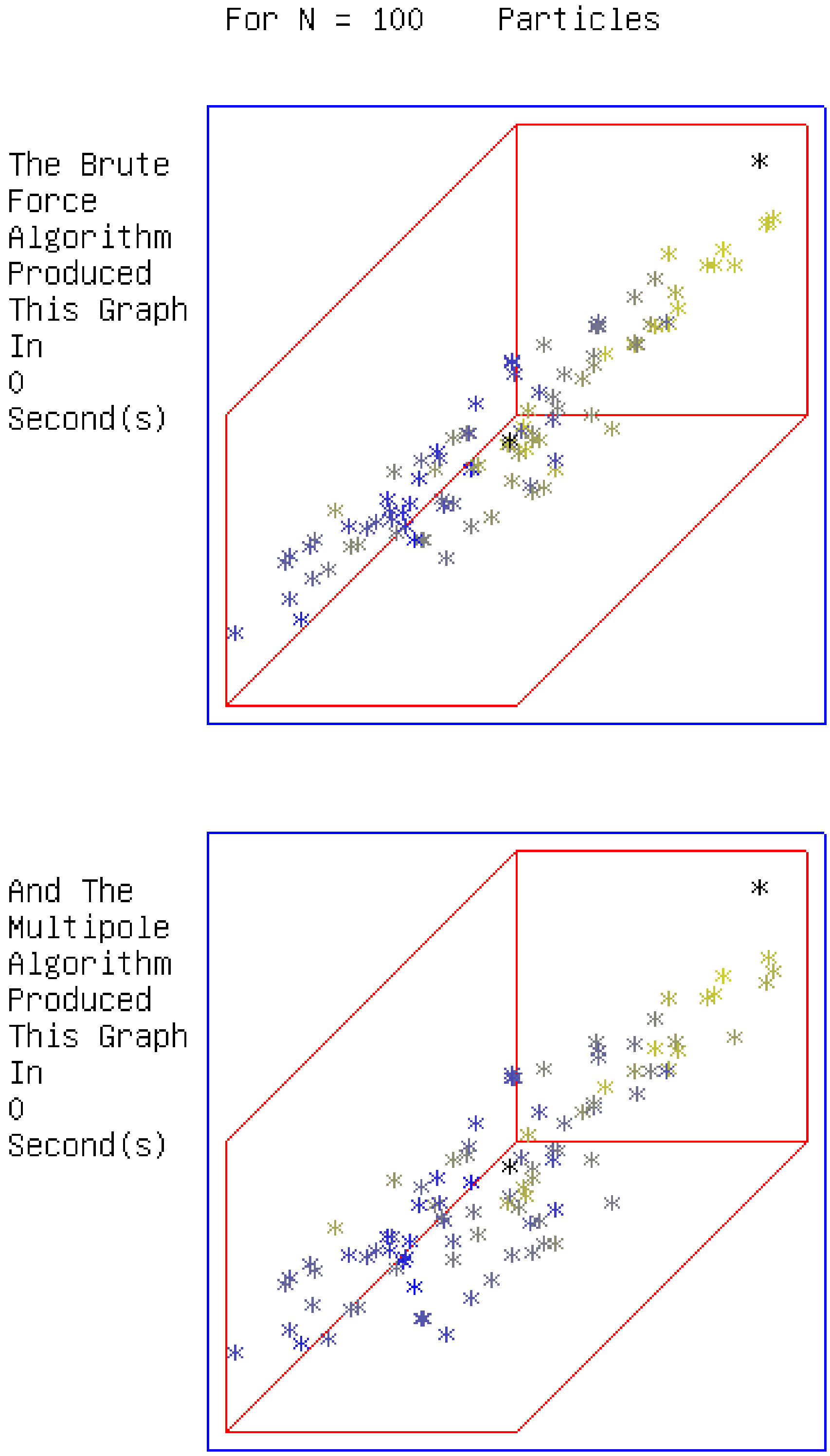
The Brute
Force
Algorithm
Produced
This Graph
In
0
Second(s)



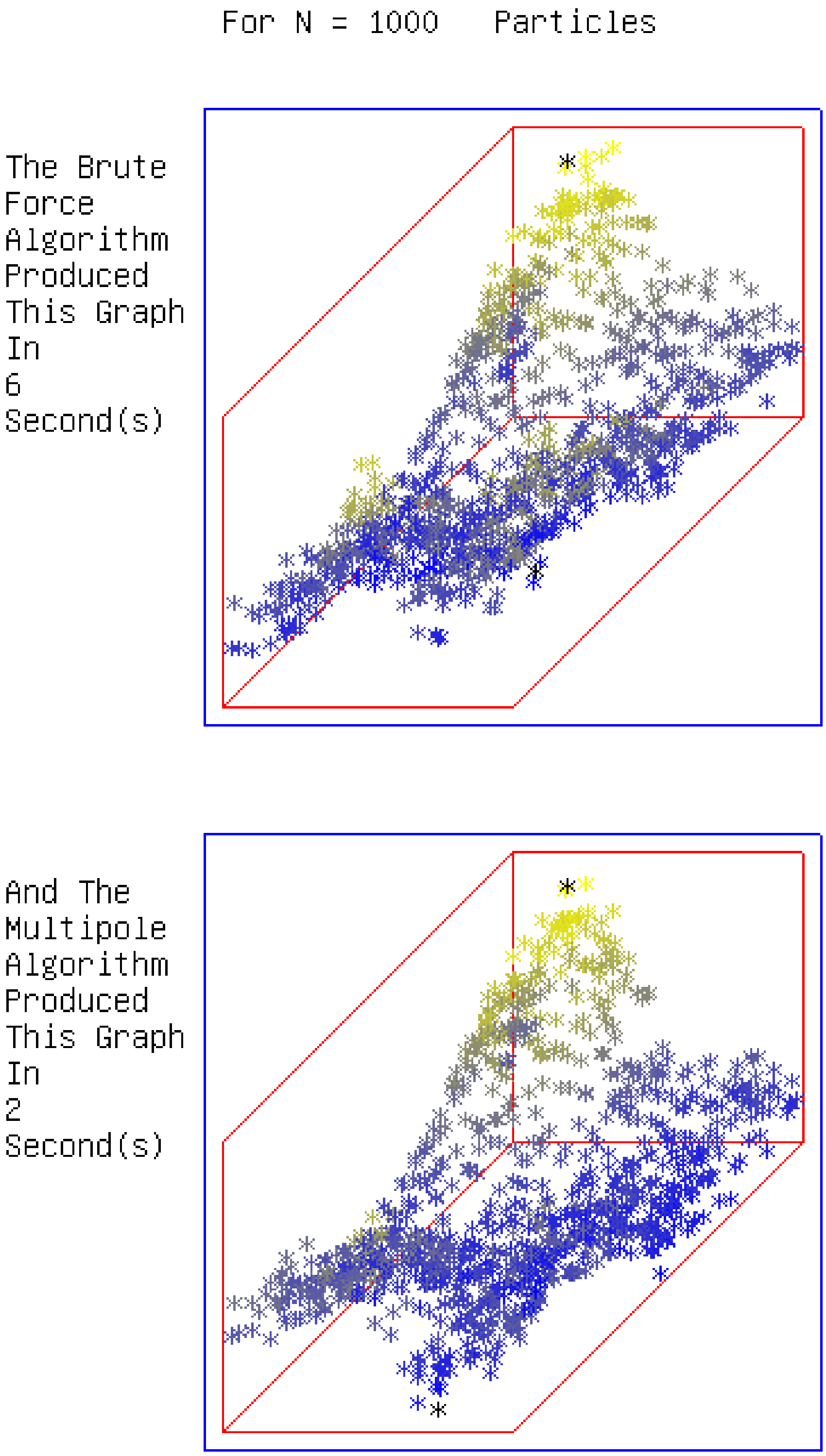
And The
Multipole
Algorithm
Produced
This Graph
In
0
Second(s)



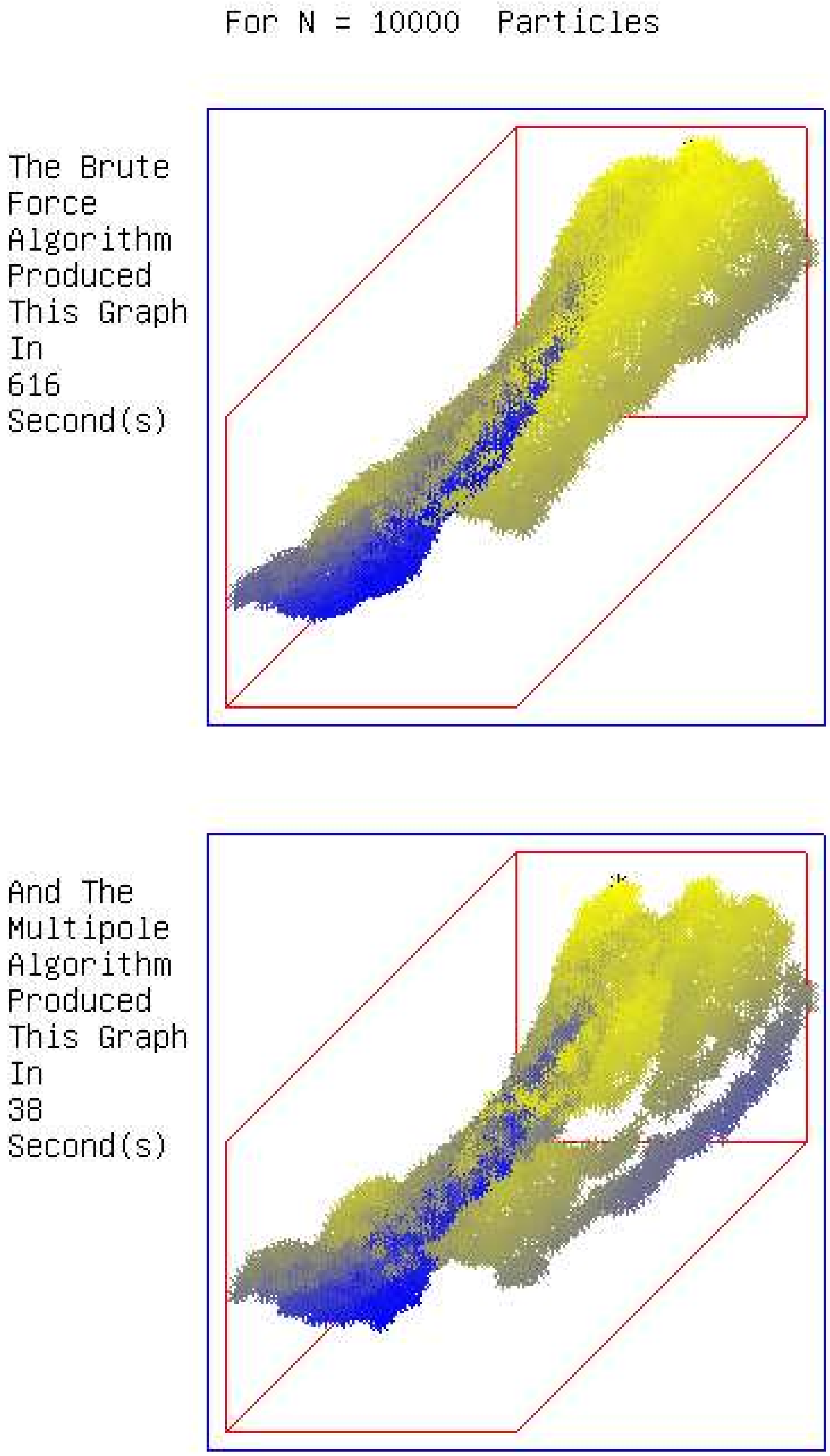
6.2 Output For 100 Particles



6.3 Output For 1000 Particles



6.4 Output For 10000 Particles



6.5 Numeric Output For 100 Particles

i	x	y	q	bp	mp	rms	nextPart
----	--------	--------	--------	--------	--------	--------	--------
0	0.000000	0.000985	-1.000000	0.263072	0.284019	0.000209	814f9dc
1	0.176643	0.364602	-1.000000	0.290560	0.321969	0.000314	814fc4c
2	0.092298	0.487217	1.000000	0.191771	0.107699	0.000841	814f120
3	0.454433	0.233178	1.000000	0.589815	0.560502	0.000293	814f90c
4	0.931731	0.568060	1.000000	0.755465	0.694984	0.000605	814fef0
5	0.050832	0.767051	-1.000000	0.059558	0.107589	0.000480	814f42c
6	0.252360	0.298197	1.000000	0.515522	0.581429	0.000659	814f8d8
7	0.531557	0.920261	1.000000	0.558600	0.516108	0.000425	814fe20
8	0.810429	0.188420	1.000000	0.781807	0.667223	0.001146	814f1bc
9	0.570614	0.076775	1.000000	0.503254	0.322726	0.001805	814fa44
10	0.984891	0.118352	1.000000	0.704279	0.539524	0.001648	0
11	0.784484	0.100916	-1.000000	0.559924	0.431143	0.001288	814f0b8
12	0.019842	0.378377	1.000000	0.249638	0.241858	0.000078	814febc
13	0.680923	0.752707	-1.000000	0.574332	0.502376	0.000720	814fb14
14	0.624407	0.126462	1.000000	0.577847	0.480172	0.000977	814f28c
15	0.770928	0.186536	-1.000000	0.599942	0.436454	0.001635	814f634
16	0.509500	0.315939	1.000000	0.517653	0.450758	0.000669	814fe54
17	0.367247	0.876312	-1.000000	0.443866	0.480378	0.000365	814f7d4
18	0.525252	0.427835	1.000000	0.478493	0.399971	0.000785	814fbb0
19	0.667979	0.171142	1.000000	0.667090	0.516743	0.001503	814f494
20	0.861905	0.769221	-1.000000	0.757160	0.738365	0.000188	814f870
21	0.966730	0.334284	-1.000000	0.630249	0.466149	0.001641	814f188
22	0.270315	0.074262	1.000000	0.607395	0.641743	0.000343	814f4fc
23	0.810159	0.464211	1.000000	0.758334	0.735183	0.000232	814f530
24	0.512203	0.586510	1.000000	0.485067	0.396097	0.000890	814fc18
25	0.871689	0.577967	1.000000	0.736589	0.751674	0.000151	814f564
26	0.333529	0.392828	-1.000000	0.308308	0.341685	0.000334	814fa10
27	0.111869	0.852681	1.000000	0.297365	0.374573	0.000772	815012c
28	0.954887	0.904283	-1.000000	0.789047	0.692515	0.000965	814f704
29	0.686187	0.126987	-1.000000	0.499014	0.348484	0.001505	814f084
30	0.000503	0.192952	1.000000	0.331809	0.350504	0.000187	814f974
31	0.374959	0.026786	-1.000000	0.531940	0.408511	0.001234	814f940
32	0.785343	0.442235	-1.000000	0.570053	0.508717	0.000613	814fbe4
33	0.853353	0.525972	1.000000	0.724915	0.724611	0.000003	814ef80
34	0.266314	0.783484	-1.000000	0.304866	0.326138	0.000213	814ff8c
35	0.196997	0.447177	1.000000	0.407084	0.411452	0.000044	814f738
36	0.966093	0.756841	-1.000000	0.771302	0.612880	0.001584	814f6d0
37	0.868966	0.134039	1.000000	0.757320	0.687529	0.000698	814f69c
38	0.018088	0.940855	1.000000	0.260207	0.301334	0.000411	814fd1c
39	0.799972	0.143217	1.000000	0.764412	0.658847	0.001056	814f8a4
40	0.908283	0.774415	-1.000000	0.804351	0.805050	0.000007	814f83c
41	0.944288	0.884822	-1.000000	0.785210	0.675725	0.001095	814f808
42	0.163028	0.424817	-1.000000	0.206950	0.187756	0.000192	81500c4
43	0.031128	0.608430	1.000000	0.186968	0.180816	0.000062	8150028
44	0.182497	0.525121	1.000000	0.334960	0.278263	0.000567	814fd50
45	0.308790	0.942626	-1.000000	0.386920	0.381394	0.000055	814f9a8
46	0.930512	0.911384	-1.000000	0.764121	0.736367	0.000278	814ffc0
47	0.532812	0.702077	1.000000	0.510746	0.429202	0.000815	814fb7c
48	0.768085	0.751941	-1.000000	0.684079	0.602375	0.000817	814f600
49	0.853967	0.174644	-1.000000	0.574329	0.456044	0.001183	814faac
50	0.419054	0.296370	-1.000000	0.427918	0.459679	0.000318	814fdb8
51	0.418716	0.138448	-1.000000	0.467782	0.439206	0.000286	814f328
52	0.299780	0.021012	-1.000000	0.457778	0.310993	0.001468	814fe88
53	0.037109	0.221059	1.000000	0.335091	0.367887	0.000328	814fae0
54	0.494525	0.882522	1.000000	0.538555	0.463956	0.000746	814f598
55	0.059002	0.119801	1.000000	0.385031	0.401312	0.000163	814fdec
56	0.341916	0.456812	1.000000	0.495506	0.508564	0.000131	8150230

57	0.585150	0.069927	1.000000	0.508544	0.336253	0.001723	814fff4
58	0.403161	0.585483	-1.000000	0.372528	0.370526	0.000020	814f01c
59	0.892277	0.176332	-1.000000	0.586294	0.480082	0.001062	814f0ec
60	0.115748	0.158915	1.000000	0.421924	0.411317	0.000106	8150194
61	0.681571	0.811796	1.000000	0.750958	0.695319	0.000556	814f460
62	0.329342	0.737553	1.000000	0.516359	0.521598	0.000052	81501c8
63	0.721261	0.657982	-1.000000	0.600817	0.594226	0.000066	814fc80
64	0.610365	0.415004	1.000000	0.534347	0.501055	0.000333	814f1f0
65	0.773965	0.274311	-1.000000	0.650934	0.460989	0.001899	814f2f4
66	0.601041	0.514091	1.000000	0.517302	0.468529	0.000488	814fcb4
67	0.180893	0.278501	-1.000000	0.341406	0.336502	0.000049	814fce8
68	0.642986	0.742832	-1.000000	0.513351	0.433823	0.000795	814f390
69	0.618052	0.618289	1.000000	0.566992	0.526222	0.000408	814fd84
70	0.181457	0.307928	-1.000000	0.338763	0.327874	0.000109	814f3f8
71	0.009182	0.951453	1.000000	0.241123	0.281912	0.000408	8150160
72	0.204940	0.604093	1.000000	0.345392	0.388325	0.000429	814fb48
73	0.559567	0.546866	-1.000000	0.310840	0.227454	0.000834	815005c
74	0.449752	0.306805	-1.000000	0.398167	0.363501	0.000347	814f4c8
75	0.215800	0.049896	-1.000000	0.396930	0.397156	0.000002	814ff58
76	0.574563	0.914820	-1.000000	0.413937	0.341200	0.000727	814f154
77	0.654159	0.355701	1.000000	0.671157	0.681144	0.000100	8150090
78	0.391946	0.033850	-1.000000	0.532957	0.405592	0.001274	814f258
79	0.091604	0.444385	1.000000	0.237743	0.189583	0.000482	814f5cc
80	0.921284	0.605927	1.000000	0.772778	0.719187	0.000536	814ef18
81	0.014717	0.686701	1.000000	0.202745	0.203382	0.000006	814f76c
82	0.129647	0.059656	-1.000000	0.319366	0.300401	0.000190	814ef4c
83	0.391159	0.865676	-1.000000	0.447128	0.432028	0.000151	814ff24
84	0.924289	0.886719	1.000000	1.000000	1.000000	0.000000	814f2c0
85	0.654911	0.076340	-1.000000	0.443342	0.271189	0.001722	814f050
86	0.003252	0.526670	1.000000	0.198473	0.159494	0.000390	81500f8
87	0.654019	0.545893	1.000000	0.592719	0.567476	0.000252	8150298
88	0.726753	0.253315	-1.000000	0.622272	0.530067	0.000922	8150264
89	0.138354	0.482949	-1.000000	0.102934	0.027628	0.000753	814eee4
90	0.100876	0.504214	1.000000	0.203200	0.161003	0.000422	814f7a0
91	0.010048	0.821570	1.000000	0.228304	0.255064	0.000268	814f224
92	0.019459	0.929619	-1.000000	0.000000	0.000000	0.000000	814efb4
93	0.037962	0.190257	-1.000000	0.120862	0.122872	0.000020	814eeb0
94	0.447314	0.649382	-1.000000	0.350279	0.296463	0.000538	81501fc
95	0.487155	0.649658	1.000000	0.505683	0.425390	0.000803	814fa78
96	0.356802	0.394116	1.000000	0.542836	0.554625	0.000118	814efe8
97	0.724702	0.306141	-1.000000	0.615940	0.583080	0.000329	81502cc
98	0.910780	0.740914	-1.000000	0.785786	0.776354	0.000094	814f3c4
99	0.651344	0.368411	-1.000000	0.398792	0.365359	0.000334	814f35c

i	x	y	a[0][0]	a[0][1]	a[1][0]	a[1][1]	numPart	nextPart
0	0.500000	0.500000	-2.000000	0.000000	4.919528	-0.363752	100	814f668
1	0.250000	0.750000	-5.000000	0.000000	1.506656	0.858111	21	814f668
2	0.750000	0.750000	1.000000	0.000000	1.040533	1.495465	25	814f01c
3	0.250000	0.250000	3.000000	0.000000	0.872592	-1.280758	29	814ef18
4	0.750000	0.250000	-1.000000	0.000000	0.999748	0.063429	25	814f258
5	0.125000	0.875000	-2.000000	0.000000	0.171105	-0.119889	6	814f668
6	0.375000	0.875000	3.000000	0.000000	-0.286016	-0.039424	5	814f224
7	0.125000	0.625000	-6.000000	0.000000	0.212590	0.294770	6	814ff24
8	0.375000	0.625000	0.000000	0.000000	0.033977	-0.152346	4	814fb48
9	0.625000	0.875000	0.000000	0.000000	0.042359	-0.064530	4	814f01c
10	0.875000	0.875000	6.000000	0.000000	0.159765	-0.383812	8	814f460
11	0.625000	0.625000	-2.000000	0.000000	0.255687	0.230820	8	814f83c
12	0.875000	0.625000	-3.000000	0.000000	-0.042278	0.337987	5	8150298
13	0.125000	0.375000	1.000000	0.000000	0.314635	-0.273359	9	814ef18
14	0.375000	0.375000	0.000000	0.000000	0.251257	-0.153122	6	814f3f8

15	0.125000	0.125000	0.000000	0.000000	0.171046	-0.391933	8	814f4c8
16	0.375000	0.125000	2.000000	0.000000	0.010653	-0.337344	6	814ef4c
17	0.625000	0.375000	-1.000000	0.000000	0.428522	-0.211611	7	814f258
18	0.875000	0.375000	2.000000	0.000000	-0.034120	-0.163382	4	814f35c
19	0.625000	0.125000	-2.000000	0.000000	0.142949	0.009021	6	814f188
20	0.875000	0.125000	0.000000	0.000000	-0.162603	0.054400	8	814f050
21	0.062500	0.937500	-1.000000	0.000000	0.054690	-0.025189	3	814f668
22	0.187500	0.937500	0.000000	0.000000	0.000000	0.000000	0	814efb4
23	0.062500	0.812500	-1.000000	0.000000	-0.008585	-0.094700	3	814efb4
24	0.187500	0.812500	0.000000	0.000000	0.000000	0.000000	0	814f224
25	0.312500	0.937500	2.000000	0.000000	0.051037	-0.056062	2	814f224
26	0.437500	0.937500	-1.000000	0.000000	-0.057025	0.054978	1	814f9a8
27	0.312500	0.812500	1.000000	0.000000	-0.046186	-0.029016	1	814f598
28	0.437500	0.812500	1.000000	0.000000	-0.046341	0.053176	1	814ff8c
29	0.062500	0.687500	-1.000000	0.000000	0.047783	0.000799	1	814ff24
30	0.187500	0.687500	0.000000	0.000000	0.000000	0.000000	0	814f76c
31	0.062500	0.562500	-3.000000	0.000000	0.052244	0.048185	3	814f76c
32	0.187500	0.562500	-2.000000	0.000000	-0.012437	-0.004214	2	814f7a0
33	0.312500	0.687500	-1.000000	0.000000	-0.016842	-0.050053	1	814fb48
34	0.437500	0.687500	0.000000	0.000000	-0.039841	-0.000276	2	81501c8
35	0.312500	0.562500	0.000000	0.000000	0.000000	0.000000	0	814fa78
36	0.437500	0.562500	1.000000	0.000000	-0.034339	0.022983	1	814fa78
37	0.562500	0.937500	0.000000	0.000000	0.043006	-0.005441	2	814f01c
38	0.687500	0.937500	0.000000	0.000000	0.000000	0.000000	0	814f154
39	0.562500	0.812500	0.000000	0.000000	0.000000	0.000000	0	814f154
40	0.687500	0.812500	0.000000	0.000000	-0.000648	-0.059089	2	814f154
41	0.812500	0.937500	0.000000	0.000000	0.000000	0.000000	0	814f460
42	0.937500	0.937500	2.000000	0.000000	0.030399	-0.061230	4	814f460
43	0.812500	0.812500	2.000000	0.000000	0.004991	-0.103838	2	814f2c0
44	0.937500	0.812500	2.000000	0.000000	-0.000624	-0.093745	2	814f600
45	0.562500	0.687500	-1.000000	0.000000	0.029688	-0.014577	1	814f83c
46	0.687500	0.687500	2.000000	0.000000	-0.010753	0.025814	2	814fb7c
47	0.562500	0.562500	-2.000000	0.000000	-0.046729	-0.047025	4	814f390
48	0.687500	0.562500	-1.000000	0.000000	0.033481	0.016607	1	815005c
49	0.812500	0.687500	0.000000	0.000000	0.000000	0.000000	0	8150298
50	0.937500	0.687500	1.000000	0.000000	-0.026720	0.053414	1	8150298
51	0.812500	0.562500	-2.000000	0.000000	-0.100042	0.021061	2	814f3c4
52	0.937500	0.562500	-2.000000	0.000000	0.021984	-0.048987	2	814ef80
53	0.062500	0.437500	-3.000000	0.000000	-0.016243	0.002521	3	814ef18
54	0.187500	0.437500	1.000000	0.000000	-0.083114	0.023090	3	814f5cc
55	0.062500	0.312500	0.000000	0.000000	0.000000	0.000000	0	814eee4
56	0.187500	0.312500	3.000000	0.000000	-0.023507	0.013531	3	814eee4
57	0.312500	0.437500	-1.000000	0.000000	-0.052689	-0.020600	3	814f3f8
58	0.437500	0.437500	0.000000	0.000000	0.000000	0.000000	0	814efe8
59	0.312500	0.312500	-1.000000	0.000000	0.060140	0.014303	1	814efe8
60	0.437500	0.312500	2.000000	0.000000	-0.006195	-0.021825	2	814f8d8
61	0.062500	0.187500	-2.000000	0.000000	0.009602	-0.007669	4	814f4c8
62	0.187500	0.187500	0.000000	0.000000	0.000000	0.000000	0	814eeb0
63	0.062500	0.062500	0.000000	0.000000	-0.059002	-0.118816	2	814eeb0
64	0.187500	0.062500	2.000000	0.000000	-0.029553	-0.015448	2	814fdec
65	0.312500	0.187500	0.000000	0.000000	0.000000	0.000000	0	814ef4c
66	0.437500	0.187500	0.000000	0.000000	-0.035717	-0.094730	2	814ef4c
67	0.312500	0.062500	1.000000	0.000000	0.091925	-0.088964	3	814f328
68	0.437500	0.062500	1.000000	0.000000	-0.045554	-0.028650	1	814fe88
69	0.562500	0.437500	-2.000000	0.000000	-0.010617	0.032162	2	814f258
70	0.687500	0.437500	0.000000	0.000000	0.000000	0.000000	0	814f1f0
71	0.562500	0.312500	-1.000000	0.000000	0.053000	-0.003439	1	814f1f0
72	0.687500	0.312500	2.000000	0.000000	0.073640	-0.052833	4	814fe54
73	0.812500	0.437500	0.000000	0.000000	-0.024816	-0.021976	2	814f35c
74	0.937500	0.437500	0.000000	0.000000	0.000000	0.000000	0	814fbe4
75	0.812500	0.312500	1.000000	0.000000	-0.038535	-0.038189	1	814fbe4

76	0.937500	0.312500	1.000000	0.000000	0.029230	0.021784	1	814f2f4
77	0.562500	0.187500	-1.000000	0.000000	-0.061907	0.061038	1	814f188
78	0.687500	0.187500	0.000000	0.000000	0.018209	-0.044154	2	814f28c
79	0.562500	0.062500	-2.000000	0.000000	-0.030764	-0.021702	2	814f084
80	0.687500	0.062500	1.000000	0.000000	-0.032589	0.013840	1	814fff4
81	0.812500	0.187500	-1.000000	0.000000	-0.041972	0.083004	5	814f050
82	0.937500	0.187500	1.000000	0.000000	-0.045223	-0.011168	1	814faac
83	0.812500	0.062500	1.000000	0.000000	-0.028016	0.038416	1	814f0ec
84	0.937500	0.062500	-1.000000	0.000000	-0.047391	-0.055852	1	814f0b8

7 Conclusions

First, obvioulsly the algorithm is much more efficient at calculating the potential due to paired interactions of large numbers of particles than the method of performing explicitly a double sum over each of the particles. However at smaller numbers of particles, this algorithm becomes less efficient, which is expected.

Second, at the present time, the algorithm is flawed, but the fact that the algorithm produces results which are not too far off from the brute force method can only be due to the correct evaluation of all short field interactions as well as the correct evaluation of the monopole term in the multipole expansion. It is then to be assumed that the program failes to accuratly calculate the other multipole coefficients. In the end, this is where further work needs to be directed. The shift of coefficients from one region to another is difficult and would require some more attention to be properly implemented.

Third, the one good development was in the successfull choice of a suitable numbering scheme for the calculation boxes on all levels of refinement. This was quite difficult and in the end it envolves an array of intricatly linked particles. So at the same time the particles are accessed as an array and as a linked list. Because each region has a number of particles contined within and an entry point into the linked list this becomes very efficient for implementing a divide and conquer strategy which is at the heart of this algorithm.

References

[1] L. Greengard and V. Rokhlin, "A Fast Algorithm for Particle Simulations" Journal of Computational Physics 73, 325-348 (1987)

[2] John Board and Klaus Schulten, "The Fast Multipole Algorithm" Computing in Science & Engineering, 76-79 January/February2000

[3] David J. Griffiths, "Introduction to Electrodynamics" Prentice Hall; 1997