THE UNIVERSITY OF NEW SOUTH WALES
SCHOOL OF MATHEMATICS
DEPARTMENT OF APPLIED MATHEMATICS

Separation of Variables Techniques in Particle Simulations and Integral Equations

by

PER-GUNNAR MARTINSSON

Supervisor
Dr William McLean
University of New South Wales

November 1995

Aknowledgements

First of all I would like to express my sincere gratitude to my supervisor, Dr McLean, who has put up with a stubborn exchange student for almost a whole year. The idea that it might be worthwhile to try to develop a general algorithm, is his. During the year he has given me a lot of time, has helped me to dig up unpublished articles and has always been willing to share his profound mathematical knowledge.

I will always be indebted to Professor Vidar Thomée at Chalmers University of Technology who initiated this exchange and gave me the chance to spend a fantastic year in Sydney. Dr Stig Larsson (also at Chalmers University of Technology) made it possible for me to write this thesis at UNSW and thus get my degree a lot faster; for this I give him my sincere thanks.

For his willingness to share his expertise in C programming I must thank Patrik Jansson. I can only regret that I had neither the time nor the programming skills to implement all his suggestions.

Then a lot of thanks go to this year's honours and post graduate students who made university life so much more enjoyable. Finally thanks to all the people (two in particular) I have met who have no idea what this thesis is about just for making this last year one of my best ever.

Abstract

With direct computation, solving the N-body problem involves evaluating all N^2 pairwise interactions. Similarly, the $\mathcal{O}(N^2)$ cost for multiplying an $N \times N$-matrix by a vector is the most expensive part of an iterative solver for integral equations. Here N denotes the number of boundary elements in the discretisation. Recently, methods have been developed that cut down these costs to $\mathcal{O}(N)$ or $\mathcal{O}(N \log N)$, among these are the Fast Multipole Method by Greengard, the Multipole Method for integral equations by Rokhlin and the Panel Clustering method by Hackbusch. It is shown that these methods are all based on an approximate separation of variables in the particle potential or the integration kernel. A general, $\mathcal{O}(N \log N)$ separation of variables algorithm to solve integral equations and evaluate particle potentials has been developed. The algorithm allows parameters to be chosen so that any desired accuracy can be obtained.

A program that simulates the dynamics of a two dimensional particle system has been developed. The program uses multipole expansions to evaluate the forces on all particles and this part of the program is shown to have computational cost $\mathcal{O}(N)$. The breakeven point where the program is faster than direct computation is less than 500 particles if the desired relative accuracy is 0.001.

Contents

Chapter 0

Introduction

In this thesis, we will study two different areas: particle simulations and integral equations. In chapter 1 we study numerical methods to solve the classical N-body problem; given N particles with some specified pairwise interaction we want to determine all internal interactions. Using direct computation we have to evaluate N^2 pairwise interactions. Techniques for cutting down the cost for large scale simulations have been known for some time. Recently the Fast Multipole Method by Greengard (see [14, 15]) has been widely used, primarily because it has an asymptotic cost of $\mathcal{O}(N)$ and because accurate error bounds exist. The method is also known as the Greengard-Rokhlin Method.

In chapter 2 we study different methods to discretise and solve integral equations. The discretisation schemes we study start by partitioning the domain for the integral into N boundary elements. Classical methods for solving the equation then have the asymptotic cost $\mathcal{O}(N^2)$. In 1983 Rokhlin presented a method using multipole expansions that cut down the cost to $\mathcal{O}(N)$. This method is based on a Nyström discretisation and only works for "nice" integration domains in two dimensions. A more generally applicable method known as "Panel Clustering" is presently being developed by Hackbusch and co-workers. The computational cost for this method is $\mathcal{O}(N \log N)$. See Hackbusch [16, 17] or Sauter [31].

We will show that these methods for evaluating potential fields and for solving integral equations are based on the same ideas. The central part is an approximate separation of the variables in the potential and the integration kernel respectively. In a region τ with some specified interior point y_τ we make the approximation

$$\Phi(x,y) \approx \sum_{\iota \in I_p} \varphi_\iota(x, y_\tau)\kappa_\iota(y_\tau, y) \qquad \text{for} \qquad y \in \tau.$$

The reason why this cannot be valid globally is that Φ typically is singular when $x = y$. The simplest example of such an expansion is a Taylor expansion around y_τ,

$$\Phi(x,y) \approx \sum_{k=0}^{p} F_{k,\tau}(x)(y-y_\tau)^k$$

where k is a multi index (see section 2.5). Since the expansions we use have a limited region of validity we have to partition the domain holding the particles (or the integration domain) into some regions and use different expansions in different regions.

The aim of the thesis was to develop a general separation of variables algorithm — one for which the previously mentioned methods are special cases. We succeeded in doing this and the general algorithm is presented in chapters 1 and 2. We make a special point of using the same notation in the algorithm for evaluating potentials and in the one for solving integral equations. It is hoped that this will simplify comparative studies of the two areas. In particular it should simplify the flow of ideas from one area to another.

Recently we learned of an article by Draghicescu, [12], where an algorithm which seems to fit into our frame work is presented for the vortex blob method in fluid dynamics.

In chapter 3 we specialise the general algorithm so we can use it to evaluate the forces on the molecules in a two dimensional fluid consisting of polar molecules. The algorithm was included in a program to simulate the molecular dynamics of the fluid. We present data that support a claim that the algorithm is $\mathcal{O}(N)$.

Chapter 1

Particle Simulations

1.1 The N-body Problem

We will study a general class of problems involving N interacting particles, where N is a large number. These problems occur in many areas of classical and modern physics; a few examples are given below. With a naïve approach one simply computes all pairwise interactions, giving rise to an $\mathcal{O}(N^2)$ cost for the computation. Clearly the cost quickly gets prohibitive using this straightforward method.

A simple way of reducing the cost is to truncate the potential, i.e., set the interaction to zero outside a sphere of fixed radius around each particle. If one assumes that the particles are reasonably evenly spaced this gives an $\mathcal{O}(N)$ algorithm but the technique is grossly unphysical unless the potential tends to zero very quickly with distance away from the particle (see Petersen *et al* [26] or Allen and Tildesley [2]). If the potential does tend to zero quickly this simple technique might give an accurate result. One potential that is commonly treated this way is the Lennard-Jones potential for molecular interactions, which decays as $\mathcal{O}(\frac{1}{r^6})$. However, if the molecules have any other long-range interaction (e.g. nonzero electrical dipole moment) we have to use a more sophisticated technique.

A more refined method for cutting down the computational cost is the so called "particle/mesh"-method. This algorithm has been used when the particles interact via gravitational or Coulombic interactions (actually the only essential requirement is that the potential is harmonic). What is usually done is that one imposes a grid on the computational domain and interpolates the point-charges (particles) to charges at the grid-nodes. Then a fast Poisson solver (usually the Fast Fourier Transform technique) is used to evaluate the potential. The potential in turn gives the force at each gridpoint via a numerical differentiation and finally the force at each charge is interpolated from the force at neighbouring gridpoints.

The described method is reasonably fast but unfortunately not very accurate. Apart from the errors in the potential one introduces additional errors in the numerical differentiation. The accuracy can be increased by refining the algorithm so that short-range interactions are computed directly while long-range interactions are taken care of by the grid. The total cost for this algorithm is $\mathcal{O}(N + M \log M)$ where M is the number of gridpoints. One usually has $M \sim N$ but with a small coefficient of proportionality so the algorithm is observed to be $\mathcal{O}(N)$ in most cases. The method is called P^3M (Particle/Particle - Particle/Mesh) and is described in detail by Hockney [20, 21]. However, the error bounds are very rough and if the distribution of charges is highly nonuniform then the error quickly becomes prohibitive.

Another method that is often mentioned in the literature is the optimised Ewald method. This is an $\mathcal{O}(N^{3/2})$ method that is used in molecular dynamics when one has to compute the forces within an infinite lattice of particles constructed from the computational box by repeating it infinitely in every direction. The technique is outlined below in section 1.1.1.

A lack of useful error estimates is a serious drawback with most of the older algorithms. If one sets up a model and the numerical outcome of the model does not correspond to what is observed in a physical experiment then one needs to know if this is due to faulty modelling or numerical errors.

The Fast Multipole method provides an algorithm which is $\mathcal{O}(N)$ and that has a guaranteed error bound. Furthermore, it allows short-range interactions to be computed directly. It should prove to be a valuable tool for performing particle simulations. However, the breakeven point where it becomes faster than other methods is quite high and since it is rather complicated to implement, especially in higher dimensions, it may not always be the best alternative.

Before we proceed to present the separation of variables algorithm we will present a few basic facts about molecular dynamics, the main field where particle simulations are performed. We will also mention some other areas that use particle simulations and point out a few special considerations that have to be made in the different cases.

1.1.1 Molecular Dynamics

The basic justification for studying molecular dynamics (MD) is the simple observation that many macroscopic properties of different substances can be deduced from interactions at a molecular level. For instance the dielectricity coefficient of a material is determined by the joint tendency of molecular dipoles to align parallel to each other. In some cases one can use statistical physics to model how all the molecules collectively give rise to some macroscopic property. Sometimes the models allow exact analytical results — the equation of state for an ideal gas or the properties of the Einstein crystal are a couple of famous examples (see Baxter [5]). If this is not feasible it is often possible to perform a computer simulation to investigate the properties of the system.

One might ask why we go to the trouble of doing complicated computer simulations when we can measure the macroscopic properties directly in laboratory experiments. The most common reason is that the properties of substances under extreme conditions such as high temperature or pressure — while easy to simulate in a computer — might be difficult or expensive to measure directly. We can also use computer simulations to gain knowledge of the microscopic structure of a material. Given that we know the macroscopic properties we can investigate what molecular interactions would give rise to those properties.

The first step in an MD simulation is to make a model of how the molecules interact. For small molecules this is fairly easy. Long-range electrical interactions between the electrons and the nuclei in different molecules can be approximated either with a low order multipole (dipole, quadrupole *etc*) placed in the middle of the molecule or by placing point charges at adequate positions in the molecules. Usually the molecule as a whole will be electrically neutral but this need not be the case. To account for short-range interactions it is common to use the Lennard-Jones potential

$$V(r) = \epsilon\Big(\Big(\frac{\sigma}{r}\Big)^{12} - \Big(\frac{\sigma}{r}\Big)^{6}\Big), \tag{1.1.1}$$

where r denotes distance from the centre of the molecule. This potential has empirically proved to be a good model for the many different parts of the short-range interaction, repulsion between the electron clouds being the strongest. At close distances it is strongly repulsive but as r increases it gets slightly attractive. Large molecules are far more complicated. It is not always possible to model the pairwise interactions as the result of some potential around each molecule. Instead one might have to make an approximation where we compute all pairwise interactions between the different parts or atoms making up the molecule and impose some constraint on the motion so the different parts always have the same relative positions. If it is not possible to model the interaction as a sum of pairwise interactions with all the other molecules or parts of molecules it gets very difficult to make a large scale computer simulation.

After having chosen a model for the pairwise interactions we have to choose an initial setup for the positions and velocities of all the molecules. If we model a fluid the initial setup is not very critical since the system will change a lot as we timestep it. However, we might reach an equilibrium state faster with a well chosen initial setup. A standard implementation consists of two different parts: one part computes the forces given the particle positions, the other part uses the forces to timestep the positions and velocities. In a typical simulation we have to do some 10^3 — 10^4 timesteps in order to reach thermodynamical equilibrium, see [6].

One problem we face is that any simulation has to use a finite number of molecules. A difficult question is to determine how many molecules we have to include in the model to get an accurate result. Another problem is how to make the number of particles in the computational cell constant. When one molecule approaches a boundary we cannot just reflect it since then we would model the boundary layer of the material which might have very different properties from the bulk material. What is usually done is that we reintroduce the particle on the opposite face of the cell with the same velocity. In other words our computational cell is a d-torus where d is the dimension of the space we are working in; usually $d = 3$. This approach introduces some error in the model. It has been shown that even for short-range potentials the result of the modelling might have anisotropies that should not be there (see Mandell [24]). Pratt and Haan [28], have investigated how periodic boundary conditions effect the equilibrium properties. If any of the particles have nonzero electric charges the sum of all pairwise interactions is not absolutely convergent and so depends on the order of summation. A natural order of summation is to sum first over all the boxes at a certain distance and then add the contributions from all these "shells", starting at the innermost. (See Allen and Tildesley [2, pp. 156 - 157])

Unless we want to compute all pairwise interactions we next have to choose some numerical technique to quickly evaluate the forces on all the molecules given their positions. Direct truncation of the potential as mentioned in the previous section is not possible unless the potential decreases very quickly. One commonly used method is the optimised Ewald algorithm. Originally developed 80 years ago as a tool for studying ionic crystals, the method is today one of the fastest for large-scale particle simulations with long-range interactions. The idea is that around every point charge we place a virtual, Gaussian (bell-shaped) charge distribution of the same magnitude but with the opposite sign to the original charge. This makes the net charge zero and since the long-range interactions vanish we can use a simple cut-off to evaluate the new potential. From this potential we must subtract the contribution from the virtual charge distributions. This is done by taking the Fourier transform to represent the virtual charges in frequency space, then we take an approximation to the sum of all the charges in frequency space and transform it back. This rather involved process is described in detail by de Leeuw *et al* [23] and Heyes [19]; they also show how to adjust the algorithm to handle dipoles and quadrupoles. An overview is given by Allen and Tildesley [2]. Optimised Ewald is $\mathcal{O}(N^{3/2})$ but with a smaller coefficient of proportionality than the $\mathcal{O}(N)$ Fast Multipole Algorithm, to be described in detail later. Different authors report different values for the break-even point. To make a just comparison the methods have to be implemented with "equal efficiency" and with the same relative accuracy. Ding *et al* [11] have reported the breakeven point to be about $N = 400$ whereas Soelvasen *et all* [32] claims that a rigorous comparison gives a breakeven of about 30000. As a comparison Board *et al* [7] reports that breakeven between the Fast Multipole Method and direct computation is 4000 for an isolated cell. With the test program we present in chapter 3 the breakeven point is less than 500 in two dimensions, it should be considerably higher in three dimensions.

Simulations in molecular dynamics have been carried out for extremely large systems. Recently it was reported that by using a simple truncation technique and powerful parallel machines, simulations with more than 500 million particles have been performed (see Beazley [6]). This might seem more than sufficient for any reasonable physical model but consider that the tiniest piece of matter contains at least billions of atoms. The above mentioned simulation contained less than a thousand timesteps but still required about a hundred hours CPU time on state of the art machines. In other words, we still have a long way to go. Furthermore we cannot use the truncation technique if we have any long-range interactions, for instance if the atoms have a nonzero electric moment. Note that if we are just interested in thermodynamical properties then smaller simulations are fully adequate.

In this section we have only described a few basic concepts to give a background before we give detailed descriptions of different algorithms. There are many questions we have not considered. For example it is not at all evident that it is possible to use classical physics for modelling a phenomenon on the molecular level. We must always estimate the magnitude of the energies involved and determine if we have to make allowances for quantum effects. Another problem we have not mentioned is that when we talk about "position" we must for a non-spherically symmetric molecule refer to both position in space and rotation angle. Likewise "velocity" must include angular velocity and "force" must include torques. For more complicated molecules we must even keep track of internal motions within the molecule (if they are affected by interactions with other molecules). A good overview is given by the very reliable and readable work by Allen and Tildesley [2].

1.1.2 Other Applications of Particle Simulations

Within astrophysics there are many different phenomena that are adequately modelled with the N-body gravitational problem. The number of particles involved ranges from 10^{11} in galaxies to just a handful in solar systems and small star clusters. The nature of the interaction in the different systems is very different. In a small system a small change in one particle's trajectory will greatly effect the forces on the other particles. To accurately model the dynamics of the system we have to follow the trajectory of each particle individually. A small change in initial conditions greatly affect later states and we have a chaotic behaviour. In a very large system on the other hand, it may be the case that relative positions of nearby particles do not change very rapidly and it might be sufficient to consider the particles as a more or less continuous mass distribution. In order to determine which type of behaviour dominates, astrophysicists calculate a parameter called the *binary collision time* which gives a measure of how much the dynamics is affected by pairwise interactions between individual particles (see Chandrasekhar [8]).

For a galaxy the binary collision time is very long so one does not have to follow each particle individually. Instead we group them together in clusters of about 10^6 particles and can model the collective dynamics of these "super stars". Using this technique, very accurate models have been constructed that give the well known spiral and elliptic shapes. Examples of systems where one has to follow all the different trajectories are small star clusters and local galaxy clusters ($N = 10^2 — 10^4$). The most difficult objects to model are the ones that fall in between. The so called globular clusters that can be found between neighbouring galaxies contain some $10^4 — 10^6$ stars. These clusters have very irregular dynamics and most older methods for particle simulations have failed to give accurate results.

The first simulations that were carried out used direct computation, thus needing $N(N-1)/2$ computations at each time step (see Aarseth [1]). More recently, $\mathrm{P}^3\mathrm{M}$ methods and different clustering methods have allowed far greater numbers of particles to be included in the simulations — at the expense of accuracy (see Appel [3] or Hockney [20]). More important still is that for these simulations one does not know how large the errors are and thus one cannot tell if a peculiarity in the result is a true part of the model or just a numerical error. The Fast Multipole Algorithm provides a reliable error estimate, is $\mathcal{O}(N)$, and should prove a valuable tool for performing large scale simulations.

The hierarchy of different stellar clusters is described in a classic work by Hoyle [22]. A description of methods to model their behaviour is found in Hockney [20].

In fluid dynamics the "vortex blob method" of Chorin is a way to determine the dynamics of a fluid without having to solve the Navier-Stokes equations explicitly. This is done by partitioning the fluid into N "vortex blobs" and equip these with a particular interaction. The long-range part of the interactions falls off as $1/r$ and we cannot use direct truncation. The short-range interaction is quite involved and we need a method that allows us to compute this part directly. The Fast Multipole Algorithm or the $\mathrm{P}^3\mathrm{M}$ method should prove valuable in these simulations. A detailed description of the vortex blob method is given in Chorin [9]. Recently, Hamilton [18] and Draghicescu, [12] have presented implementations of separation of variables algorithms for the vortex blob method.

Other examples of applications are given by Hockney in [20]. A table that gives a rough idea of the number of particles needed for different applications and what time-steps are usually used is given in appendix A.

1.2 The General Separation of Variables Algorithm

We will assume that the particles interact via a potential that can be separated into two parts, $\Phi = \Phi_{\text{near}} + \Phi_{\text{far}}$ (sometimes the potential also includes an exterior part but we will not be concerned with that here). The near field potential, Φ_{near}, is assumed to decrease rapidly enough to allow a simple truncation of the potential (i.e. we compute this part of the interaction directly between neighbouring particles and ignore it for distant particles). Henceforth we will look exclusively at the long-range interaction and so when we refer to Φ we will mean the long-range potential.

1.2.1 Approximate Separation of Variables in the Potential

Let $\Phi = \Phi(x, y)$ represent the potential at x, due to a particle at y. For a fixed pivot point y_0 we approximate Φ with

$$\tilde{\Phi}_p(x, y) = \sum_{\iota \in I_p} \varphi_\iota(x, y_0)\kappa_\iota(y_0, y), \tag{1.2.2}$$

where I_p is an index set of cardinality less than $C_I p^m$ for constant C_I and some integer m, usually $m \le d-1$. We will call p the *order* of the expansion. To be able to get an error bound we impose the following condition on $\tilde{\Phi}_p$:

There exist constants $\eta_0 < 1, C_1$ and C_2 such that for all $\eta \in (0, \eta_0]$ and for all x, y, y_0 we have

$$|\tilde{\Phi}_p(x, y) - \Phi(x, y)| \le C_1(C_2\eta)^p|\Phi(x, y)| \quad \text{if} \quad |y - y_0| \le \eta|x - y_0| \tag{1.2.3}$$

From now on we will suppress the subscript p when we write $\tilde{\Phi}$.

By a *cluster* we will refer to a set $\tau \subseteq R^d$ with an associated interior point y_τ, which we will call the *centre* of the cluster. Strictly speaking we should reserve the word cluster for the particles with positions in τ but we will not cause any confusion by using the word cluster for both the subset of R^d and the subset of the set of particles. We will call the cluster *η-admissible* with respect to a point $x \in R^d$ if

$$\forall y \in \tau : |y - y_\tau| \le \eta|x - y_\tau|$$

In this case the error bound for $\tilde{\Phi}$ will hold for all points $y \in \tau$ if we use y_τ as the pivot point.

In section 1.3 we show how multipole expansions can be used to generate the desired expansions. In section 2.5 we show how to use Taylor expansions for similar expansion of integration kernels. An example of how to use them for particle simulations is given in Draghicescu [12].

We can with a simple example illustrate the idea of the algorithm. Let S and T be two sets in R^d with centres x_0 and y_0 respectively. Assume that T is η-admissible with respect to every point in S for some $\eta < 1$. Assume further that there are K particles in S and L particles in T, at positions x_k and y_l respectively. Now if we want to evaluate the exact force at each particle in S due to the interaction with the particles in T we need to do KL calculations. However, if we use $\tilde{\Phi}$ instead of Φ we get

$$\tilde{\Phi}(x_k) = \sum_{l=1}^{L} \tilde{\Phi}(x_k, y_l) = \sum_{l=1}^{L} \sum_{\iota \in I_p} \varphi_\iota(x_k, y_0) \kappa_\iota(y_0, y_l) = \sum_{\iota \in I_p} \varphi_\iota(x_k, y_0) \sum_{l=1}^{L} \kappa_i(y_0, y_l)$$

The inner sum (the one over l) is seen as the i'th coefficient in a potential expansion due to *all* particles in T. Since they do not depend on x, these coefficients can be precomputed with $C_1 L$ operations (where $C_1 \sim \#I_p$, the cardinality of I_p). Then we can use this expansion to compute the force at each position x_k with $C_2 K$ operations (again $C_2 \sim \#I_p$). The total cost is $\mathcal{O}(K + L)$ instead of $\mathcal{O}(KL)$. Note that the coefficients C_1 and C_2 can be quite big if high accuracy is needed (in which case we need a big p which in turn gives a big index set I_p) so it may be that we need quite a large L and K before we gain any reduction in computational cost.

The idea of the full algorithm is to group particles together in clusters. When we want to compute the potential at a particle site we use expansions around the centres of the clusters to represent the potential due to all the particles contained in the cluster. The coefficients for all these expansions can be computed in a set-up phase. Due to the error bound (1.2.3) we can use big clusters for distant particles but have to use smaller ones for close particles; the cluster always has to be η-admissible with respect to the particle site. The interactions with the very nearest particles are computed directly.

1.2.2 The Tree of Clusters

We start with a computational domain Ω that contains N particles, and group these together in clusters, τ. These are assumed to form a partially ordered set with inclusion as the relation. (As before we make no distinction between a set of particles and the set in R^d that contains them.) The clusters that do not contain any other clusters (i.e. the terminal nodes of the tree) will be called *panels* and will sometimes be denoted π. It is further assumed that a cluster that is not a panel will have at least two children (clusters that are subsets on the next finer meshlevel of the original cluster). We will call the tree T, the root of the tree T_0, the children of the root T_1 and so on. We assume that all panels will be on the same level T_n for some n. Usually $\#T_n \sim N$. Since $\#T_{i-1} \leq \#T_i/2$ we know that $n \leq \log_2(\#T_n)$. Similarly we can find an upper bound for the number of clusters in T with a geometric series and find that $\#T \leq 2\#T_n$.

Now for each x in the computational domain we need a subset $\mathcal{C}(x)$ of T, called an *η-admissible covering*, such that

1. $\Omega \subseteq \cup_{\tau\in\mathcal{C}(x)} \tau$, i.e. $\mathcal{C}$ is a covering of Ω.
2. The intersection between the interiors of any two clusters in $\mathcal{C}(x)$ is empty.
3. Each τ in $\mathcal{C}(x)$ is either a panel or it is η-admissible with respect to x.
4. $\#\mathcal{C}(x) \leq C_\sigma \left(\frac{1}{\eta}\right)^{d-1} \log(2+\eta^{d-1}\#T_n)$ for some constant C_σ.
5. There is no other covering with fewer elements that complies with the previous conditions.

Condition 4 looks a bit odd but this is indeed the bound for the cardinality one finds for most "natural" trees. We will just use it to conclude that $\#\mathcal{C}(x)$ is $\mathcal{O}(\log \#T_n)$. A careful analysis also shows that the covering is unique. The interested reader is referred to Hackbusch [16].

We partition $\mathcal{C}(x)$ into two parts, $\Pi(x)$, consisting of all the non-η-admissible panels in $\mathcal{C}(x)$ and $\mathrm{T}(x)$ consisting of all other panels and clusters in $\mathcal{C}(x)$.

To find $\mathcal{C}(x)$ is easier than one might imagine. It is given by the recursive procedure *Divide* which takes two arguments, a cluster τ, and a subset $\mathcal{C}$ of T.

procedure *Divide*$(\tau,\mathcal{C})$
begin
  if [(τ is $\eta-$admissible w.r.t. x) **or** ($\tau$is a panel)] **then**
    $\mathcal{C} := \mathcal{C} \cup \{\tau\}$
  else
    for (τ_i child of τ) **do** *Divide*$(\tau_i,\mathcal{C})$
  end if
end

Once we have this subroutine, we set $\mathcal{C} = \emptyset$. Then the admissible covering, $\mathcal{C}(x) = \mathcal{C}$, is given by calling *Divide*$(T_0,\mathcal{C})$.

Performing this calculation clearly takes $\mathcal{O}(\#\mathcal{C}(x))$ operations.

The number of children at each node is often taken to be constant throughout the tree. If a small η is desired it is usually a good idea to choose a tree with many children at each node.

1.2.3 The Algorithm

Let x be an arbitrary point with η-admissible covering $\mathcal{C}(x)$. Every point (charge) in Ω is contained in exactly one cluster in the covering. Since the conditions in the error estimate for $\tilde{\Phi}$ above are satisfied (expansion around the centre of each cluster) we get (recall that the centre of cluster τ is denoted y_τ)

$$\begin{aligned}\tilde{\Phi}(x) &= \sum_{\tau \in \Pi(x)} \sum_{y_l \in \tau} \Phi(x, y_l) + \sum_{\tau \in \mathrm{T}(x)} \sum_{y_l \in \tau} \tilde{\Phi}(x, y_l) \\ &= \sum_{\tau \in \Pi(x)} \sum_{y_l \in \tau} \Phi(x, y_l) + \sum_{\tau \in \mathrm{T}(x)} \sum_{\iota \in I_p} \varphi_\iota(x, y_\tau) \Big[\sum_{y_l \in \tau} \kappa_\iota(y_\tau, y_l) \Big]\end{aligned}$$

Here we do a slight abuse of notation since we call the total approximation to the potential $\tilde{\Phi}$ even though we use the exact potential in the near field.

The number of elements in the sum over the panels depends on how many particles are close to x; if the distribution of particles is reasonably even then this number will be approximately constant. Computing the first part of the sum is thus $\mathcal{O}(N)$. In the second part of the sum we note that the sum inside the brackets *does not depend on x* ! This means that we can pre-calculate it for each τ and then use the pre-calculated values for evaluating the second part. Set

$$\kappa_\iota^\tau = \sum_{y \in \tau} \kappa_\iota(y_\tau, y).$$

To calculate these we start by setting them all to zero. Then for each particleposition and each ι we go through all the τ's in levels T_n to T_0 that contain x and add $\kappa_\iota(y_\tau, x)$ to each κ_ι^τ. The computational cost is $\mathcal{O}(\#\text{particles} \times \#\text{levels} \times \#I_p) = \mathcal{O}(N \times \log \#T_n \times p^d)$ and since $\#T_n \sim N$ we get $\mathcal{O}(N \log N)$.

In the next step we go through all the particleposition x_k. At each x_k we find $\mathcal{C}(x_k)$ (cost $\mathcal{O}(\log N)$) and evaluate the sum above using the exact potential for interactions in the nearfield, and the κ_ι^τ's for the long-range interactions. The second part of the sum contains $\#\mathcal{C}(x_k) \times \#I_p$ terms so the total cost for this part of the algorithm is $\mathcal{O}(N + N \times (\log N + \#\mathcal{C}(x_k) \times \#I_p)) = \mathcal{O}(N \log N)$. If we want we can include a short-range interaction (one for which spherical cut-off is acceptable) when we compute the exact potential in the nearfield.

Thus we have found an algorithm that is $\mathcal{O}(N \log N)$. For a specific case it is simple to find an error bound from equation (1.2.3). An example is given in the presentation of the Fast Multipole Algorithm.

1.2.4 How to Speed up the Algorithm

Even though the algorithm we described in the previous section is $\mathcal{O}(N \log N)$ the coefficient involved is typically very large and the way it stands the algorithm is not very useful. However, one can often increase the efficiency by organising the tree of clusters in such a way that we do not have to use *Divide* to find $\mathcal{C}(x)$ at each particle. Another way to speed it up is if we can use the values of κ_ι^τ from one level to find the values at the next coarser level. This is possible if we have a translation property for κ that tells us how to get $\kappa_\iota(y_{\tau'}, y)$ from $\kappa_\iota(y_\tau, y)$. A special case where we have a very simple translation property is if κ does not depend on y_0 at all, $\kappa_\iota(y_0, y) = \kappa_\iota(y)$. In this case we simply get $\kappa_\iota^\tau = \sum_{\tau' \subset \tau} \kappa_\iota^{\tau'}$. It is easy to find expansions like this but most of them give numerical problems and they are not much used for particle simulations, in section 2.5 we show how they are used for integral equations.

We will see in the presentation of the multipole algorithm how using an expansion system with good translation properties and efficient bookkeeping of the coefficients can reduce the computational cost to $\mathcal{O}(N)$.

1.3 The Fast Multipole Algorithm

1.3.1 Facts About Multipole Expansions

Multipole expansions have been used widely by physicists for a long time. The idea is that the harmonic potential that arises from some source distribution in a region can be approximated by a superposition of fields from some simple types of point-sources. A general expansion around the origin in R^d looks like

$$\Psi(x) = \sum_{k=0}^{\infty} \frac{1}{|x|^{k+d-2}} \sum_{j \in J_k} c_k^j Y_k^j(x/|x|). \tag{1.3.4}$$

The cardinality of I_p in (1.2.2) is equal to $\#(\cup_{k=0}^{p} J_k)$. In two dimensions the term "$\frac{1}{|x|^0}$" should be replaced by $\log(\frac{1}{|x|})$.

We see that for each k in the first sum we have a group of terms that have the same decay-rate when $x \to \infty$. The Y_k^j's are functions on S^{d-1} only, they span $\mathcal{L}^2(S^{d-1})$, and are called the *d-dimensional spherical harmonics*. These are defined as the eigenfunctions of the angular part of the Laplacian in polar coordinates (plane polar, spherical *etc* depending on the dimension). By matching these terms with suitable eigenfunctions of the radial part of the Laplacian (these are just positive and negative powers of r) we arrive at an expansion system of the form above. This system spans the kernel of the Laplacian in $R^3 \backslash \{0\}$ (we have implicitly imposed the condition that the functions vanish at infinity which makes all positive powers of r vanish). This is a classical separation of variables procedure.

The first term in the expansion corresponds to a point charge and there is no dependence on S^{d-1}, i.e. $J_0 = \{0\}$ and Y_0^0 is a constant in all dimensions (the constant is determined by a normalisation condition). The next terms, corresponding to $\frac{1}{|x|^{d-1}}$, is what physicists call the *dipole moment*, the next again is called *quadrupole moment* and so on. The reason for these names is that the k'th group of terms describe the potential from a limiting charge of 2^k particles whose positions tend to zero, whose moments of order lower that k are zero and whose individual charges tend to infinity in such a way that we get a finite potential in the limit.

A two-dimensional multipole expansion is most easily expressed in plane polar coordinates $x = (r\cos\theta, r\sin\theta)$,

$$\Psi(x) = c_0 \log r + \sum_{k=1}^{\infty}(c_k^{-1}e^{-ik\theta} + c_k^1 e^{ik\theta})/r^k$$

We have $J_k = \{-1, 1\}$ and $Y_k^j(\theta) = e^{ijk\theta}$ for all $k \geq 1$. (Note that the superscripts 1 and -1 do not denote a power, they are just indices.) With complex notation ($x = re^{i\theta}$) this is neatly written as

$$\Psi(x) = c_0 \log x + \sum_{k=1}^{\infty} \frac{c_k}{x^k}$$

In three dimensions it gets a little bit more complicated, here $J_k = \{-k, -k+1, ..., k-1, k\}$ and $Y_k^j(\theta, \varphi) = L_k^{|j|}(\cos\theta)e^{ij\varphi}$ where (r, θ, ϕ) are the usual spherical coordinates and $L_k^{|j|}$ are associated Legendre polynomials. The Y_k^j's are simply called spherical harmonics. We have $\#I_p = \sum_{k=0}^{p} 1 + 2k = (p+1)^2$.

The reason why these expansions are used so extensively is that for most naturally occuring charge-distributions one gets very fast convergence, i.e. one needs only a few terms in the expansion to accurately describe the potential. Physicists have for a long time used multipole expansions to approximate potentials but usually use only the first two or three terms (up to quadrupole-moments). This is often done without referring to them as the first terms in an exact Fourier-expansion and without any error estimates.

If we view equation (1.3.4) as the an expansion around the origin for the field from a particle at y, then we can interpret the coefficients c_k^j as functions of y. The expansion is known to converge when $|x| > |y|$. To get an expansion of form (1.2.2) around a point y_τ we identify

$$\kappa_\iota(y_\tau, y) = c_k^j(y - y_\tau) \qquad \text{and} \qquad \varphi_\iota(x, y_\tau) = \frac{Y_k^j(x - y_\tau)}{|x - y_\tau|^{k+d-2}}$$

where $\iota = \{k, j\}$.

1.3.2 The 2-dimensional Case

We will describe in detail how the algorithm works for a distribution of point charges in R^2. The reason why we study this when most physical problems are set in three dimensions is that in R^2 one of the miracles of mathematics works; namely that by using complex notation one can handle both magnitude and direction of a vector with a simple multiplication operator. This makes the mathematical analysis very transparent. The idea and the mathematics are essentially the same in any dimension, but in dimensions higher than two the expansions get far messier. More about this in a later section. The same thing goes for our choice of potential. To be able to give explicit error bounds we study specifically Coulombic interaction (or, equivalently, gravitational interaction). Generalising to other harmonic potentials is straightforward.

Explicitly what we mean with complex notation is that we identify R^2 with C. It is well known that for every harmonic function $u : R^2 \to R$ there exists a unique (up to a constant) analytic function $w : C \to C$ such that

$$u(x) = \Re(w(x_1 + ix_2))$$

Furthermore, the force F is given by

$$F = -\nabla u = (-\Re(w'), \Im(w')) \tag{1.3.5}$$

where prime denotes the (complex) derivative.

From now on we will follow the convention of calling the original analytic function the potential, and the derivative the force. However, it should be kept in mind that the physical quantities involved are u and F, given by the expressions above.

The (complex) potential from a unit strength point charge is given by

$$\Phi(x, y) = -\log(x - y)$$

Taylorexpanding $\log(1 - \alpha)$ with $\alpha = y/x$ gives the multipole expansion directly

$$\Phi(x, y) = -\log(x) + \sum_{k=1}^{\infty} \frac{1}{k}(\frac{y}{x})^k \qquad \text{for } |x| > |y|$$

Before we describe the algorithm in any detail we will present a few error bounds and translation properties.

Theorem 1 (Multipole Expansion) *Suppose that L charges, each of strength q_l and located at point y_l $(l = 1, 2, ..., L)$, are distributed in a sphere of radius R with centre at the origin. Then for any $x \in C$ with $|x| > R$ the potential $\Phi(x)$ from all these charges is given by*

$$\Phi(x) = Q\log(x) + \sum_{k=1}^{\infty} \frac{a_k}{x^k}$$

where $Q = \sum_{l=1}^{L} q_l$ and $a_k = -\frac{1}{k}\sum_{l=1}^{L} q_l y_l^k$

If the series is truncated after $p \geq 1$ terms the error is bounded by

$$\Big|\Phi(x) - Q\log(x) - \sum_{k=1}^{p} \frac{a_k}{x^k}\Big| \leq \frac{A}{1-\eta}\eta^{p+1} \tag{1.3.6}$$

where $\eta = |\frac{R}{x}|$ and $A = \sum_{l=1}^{m} |q_l|$.

Proof We start by looking at the potential $\Phi_l(x)$ from particle l. We have for $|y_l| < |x|$

$$\Phi_l(x) = q_l \log(x-y_l) = q_l \log(x) + q_l \log(1-y_l/x) = q_l \log(x) + q_l \sum_{k=1}^{\infty} -\frac{1}{k}\Big(\frac{y_l}{x}\Big)^k$$

Summation over l immediately gives the expressions for Q and a_k.

To prove the error bound we use the triangle inequality for the truncated part of the sum.

$$\Big|\Phi(x) - Q\log(x) - \sum_{k=1}^{p} \frac{a_k}{x^k}\Big| = \Big|\sum_{k=p+1}^{\infty} \frac{a_k}{x^k}\Big| = \Big|\sum_{k=p+1}^{\infty}\sum_{l=1}^{L} \frac{-q_l y_l^k}{k x^k}\Big| \leq$$

$$\leq \sum_{k=p+1}^{\infty}\sum_{l=1}^{L} \frac{|q_l| R^k}{k|x^k|} \leq A \sum_{k=p+1}^{\infty} \eta^k = \frac{A}{1-\eta}\eta^{p+1} \qquad \text{Q.E.D.}$$

Note that equation (1.3.6) tells us that multipole expansions fulfil the criterion given in (1.2.3).

The next lemma tells us that if we know the multipole expansions on one level in the cluster tree we can easily (with a linear transformation) get the expansions around the centres on the coarser levels. Let $S_R(x)$ denote a ball of radius R, centred at x.

Lemma 2 *Suppose that*

$$\Phi(x) = a_0 \log(x-y_0) + \sum_{k=1}^{\infty} \frac{a_k}{(x-y_0)^k} \tag{1.3.7}$$

is the multipole expansion from a set of L charges contained in $S_R(y_0)$. Then this series expansion around y_0 can be transformed to an expansion around a point z_0 in two ways.

1) To a new multipole expansion that converges outside $S_{R+|y_0-z_0|}(z_0)$,

$$\Phi(x) = a_0 \log(x-z_0) + \sum_{j=1}^{\infty} \frac{b_j}{(x-z_0)^j}$$

where

$$b_j = -\frac{a_0(y_0-z_0)^j}{j} + \sum_{k=1}^{j} a_k (y_0-z_0)^{j-k}\binom{j-1}{k-1}$$

with $\binom{j-1}{k-1}$ *a binomial coefficient. If the series is truncated after* p *terms we have the following error bound*

$$\Big|\Phi(x) - a_0\log(x-z_0) - \sum_{j=1}^{p}\frac{b_j}{(x-z_0)^j}\Big| \;\le \Big(\frac{A}{1-\frac{|y_0-z_0|+R}{|x-z_0|}}\Big)\Big|\frac{|y_0-z_0|+R}{x-z_0}\Big|^{p+1}$$

with A *defined in Theorem 1.*

2) To a power series that converges inside $S_R(z_0)$ *if* $|z_0-y_0| > (1+c)R$ *for some* $c > 1$,

$$\Phi(x) = \sum_{j=0}^{\infty} c_j(x-z_0)^j$$

where

$$c_0 = a_0\log(-(y_0-z_0)) + \sum_{k=1}^{\infty}\frac{a_k}{(y_0-z_0)^k}(-1)^k$$

and

$$c_j = -\frac{a_0}{j(y_0-z_0)^j} + \frac{1}{(y_0-z_0)^j}\sum_{k=1}^{\infty}\frac{a_k}{(y_0-z_0)^k}\binom{j+k-1}{k-1}(-1)^k \quad \mathit{for} \quad j \ge 1$$

If the series is truncated after p *terms we have the following error bound*

$$\Big|\Phi(x) - \sum_{j=0}^{p} b_j(x-z_0)^j\Big| < \frac{A(4e(p+c)(c+1)+c^2)}{c(c-1)}\Big(\frac{1}{c}\Big)^{p+1}$$

with A *defined as in theorem 1 and* e *the base of the natural logarithm.*

Proof In the proof we will assume (without loss of generality) that $z_0 = 0$. To prove part one we will transfer each of the terms in (1.3.7) separately. For the log-term we proceed as before.

$$a_0\log(x-y_0) = a_0\log(x) + a_0\log(1-y_0/x) = a_0\log(x) - a_0\sum_{i=1}^{\infty}\frac{1}{k}\frac{y_0^k}{x^k}$$

For the inverse powers of $x - y_0$ we use a MacLaurin expansion with respect to y_0,

$$\begin{aligned}\frac{a_k}{(x-y_0)^k} &= a_k\sum_{i=0}^{\infty}\frac{y_0^i}{i!}\frac{\partial^i}{\partial y_0^i}\frac{1}{(x-y_0)^k}\Big|_{y_0=0} \\ &= a_k\sum_{i=0}^{\infty}\frac{(k+i-1)!}{(k-1)!i!}\frac{y_0^i}{x^{k+i}} = a_k\sum_{i=0}^{\infty}\binom{k+i-1}{k-1}y_0^i\frac{1}{x^{k+i}}\end{aligned}$$

To get the expression for b_j we put all the terms with the same power, j, of $1/x$ together using $k + i = j$.

We have not considered where (if anywhere) the MacLaurin series above converges. However, we know from the theory of power-expansions that if a convergent series exists for any given point then it must have the form of the MacLaurin expansion. But we know from Theorem (1) that an expansion exists and we also know the domain of convergence. Thus we know that the coefficients we found are the correct ones. Similarly we get the error bound by just making the relevant replacements in the error bound in the theorem. Q.E.D.

The proof of part 2 of the lemma is very technical; we omit it here but it can be found in Greengard [14].

Note that the proof uses the seemingly superfluous assumption that the multipole expansion actually comes from a charge distribution. If we did not know this, we could not use the uniqueness of the expansion unless we gave some technical criteria that tell us for which sequences $\{a_k\}$ we can do the translation. Another important observation is that given the set $\{a_k\}_{k=0}^{p}$ we can actually compute $\{b_j\}_{j=0}^{p}$ exactly. This means that we can do the translation to a new multipole expansion *without any loss of precision* (given that round off errors are negligible).

Finally we need to know how to translate the power series given by part 2 of lemma 2. All we need to do is to expand a binomial and change the order of summation.

Lemma 3 *For any $x_0, x \in C$, $\{a_k\}_{k=0}^{n} \subseteq C$ we have*

$$\sum_{k=0}^{n} a_k(x - x_0)^k = \sum_{l=0}^{n}\Big(\sum_{k=l}^{n} a_k \binom{k}{l}(-x_0)^{k-l}\Big)x^l$$

Definition of the Problem

We will study the problem of evaluating the potential due to a set of N point charges in the plane. With a given relative accuracy, ϵ, we want to find the value of the potential at each of the particlesites. We assume that the particles are contained inside a box (square) with unit sides, centred at the origin.

Description of the Algorithm

The root of the tree of clusters, T_0, is the full computational box. This cluster is divided into four similar clusters which form the next finer meshlevel, T_1. Then we continue to subdivide recursively so that meshlevel T_m consists of 4^m similar clusters. We need to decide how far we should carry the subdivisions and it will turn out that a suitable choice is to go to meshlevel n where $n \sim \log_4(N)$. With this choice each box will on average contain one particle. We define the *neighbours* of a cluster τ as the clusters on the same meshlevel that share at least one boundary point with τ. The *parent* of a cluster is the cluster on the next coarser level that contains that cluster. Similarly a *child* is a cluster on the next finer meshlevel that is contained in the original cluster.

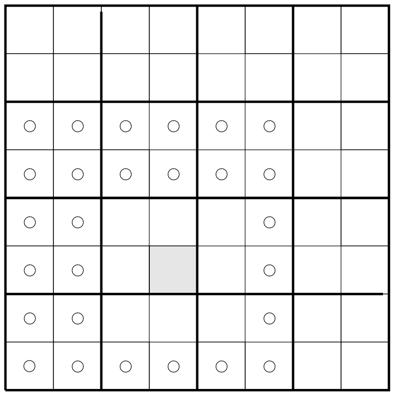

Figure 1.1: Interactionlist for a cluster on level T_3. The original cluster is shaded and the clusters in the interactionlist are marked with a ring.

I_τ Denotes the *interaction list* of a cluster τ, i.e. the set of clusters that are children of τ's parent's neighbours but are not neighbours of τ itself. For all intermediate meshlevels I_τ will contain 27 clusters.

Ψ_τ Denotes the approximation to the field from all particles in cluster τ given by a p-term multipole expansion around y_τ.

Υ_τ Denotes an analogous Taylor expansion that approximates the field from the particles in all clusters that are not contained in τ or its neighbours.

There are two main stages in the algorithm. In the first one we start by computing the multipole expansions, Ψ, around the centres of all the panels. Then we loop over all the meshlevels from finest to coarsest, computing expansions around each cluster on a level using the expansions on the next finer level. In the second stage we loop over all the levels from coarsest to finest. At each level we determine the Taylor expansions, Υ, around the centres of all clusters on that level (these account for interactions with all "distant" particles). This is done by combining the Taylor expansions from the next coarser level with the multipole expansions from all the clusters in the interaction list. Finally we loop over all the clusters. At each particle site in a cluster we evaluate the Taylor expansion around the centre of the panel and add the contributions from the nearby particles by direct computation.

Using a pseudo code we can be more precise in our description of the algorithm.

Step 1

for $\pi \in T_n$ **do**	Loop over all panels.
$\quad \{q_j,\ y_j\}_{j: y_j \in \pi} \mapsto \Psi_\pi$	The particles in panel π give the first multipole expansion.
end do	

Step 2

for $m = n-1, n-3, \ldots, 2$ **do**	Loop over all meshlevels.
$\quad$ **for** $\tau \in T_m$ **do**	Loop over all clusters at level m.
$\qquad \{\Psi_{\tau'}\}_{\tau' \in \tau} \mapsto \Psi_\tau$	Transfer multipole expansions from children.
$\quad$ **end do**	
end do	

Step 3

$\Upsilon_\tau = 0 \quad \forall \tau \in T_0 \cup T_1$	
for $m = 2, 3, \ldots, n$ **do**	Loop over all meshlevels.
$\quad$ **for** $\tau \in T_m$ **do**	Loop over all clusters at level m.
$\qquad \Upsilon_{\mathrm{P}(\tau)}, \{\Psi_{\tau'}\}_{\tau' \in \mathrm{I}_\tau} \mapsto \Upsilon^i_m$	$\mathrm{P}(\tau)$ denotes the parent of τ. Transfer Υ from the parent cluster and add the contributions from the particles in the interaction-list via their multipole expansions.
$\quad$ **end do**	
end do	

Step 4

for $\pi \in T_n$ **do**	Loop over all panels.
$\quad$ **for** $j : y_j \in \pi$ **do**	Loop over all particles in π.
$\qquad \tilde{\Phi}(y_j) = \Upsilon_\pi(y_j - y_\pi)$	Evaluate the far field part of $\tilde{\Phi}$ at y_j
$\qquad$ **for** $i : y_i \in \Pi(y_\pi),\ i \neq j$ **do**	Loop over all close particles.
$\qquad\quad \tilde{\Phi}(y_j) = \tilde{\Phi}(y_j) + \Phi(y_j, y_i)$	Compute interaction with neighbours exactly.
$\qquad$ **end do**	
$\quad$ **end do**	
end do	

There are two sources for errors in the algorithm. The first one is due to the fact that we use truncated multipole expansions. This error is controlled by part 1 of lemma 2, and all we need to do is to determine an η_0 such that every time we use a multipole expansion we expand around an η-admissible cluster τ, for some $\eta \leq \eta_0$. One can convince oneself that whenever we use a multipole expansion Ψ_τ it is evaluated at a point which is not contained in any of τ's neighbours. If the side of τ has length $2a$ then this means that the point where we evaluate the potential cannot be closer to the centre of τ than $3a$. See figure 1.2. In the lemma we need a sphere of radius $R = \sqrt{2}a$ to hold all the particles in τ. Thus we get that the smallest η_0 we can use is $\eta_0 = \sqrt{2}/3 = 0.471\ldots$ The fact that we might have arrived at the expansion around the centre of τ by transferring expansions from finer clusters is not relevant since the transfer formula is exact. Calling the relative error introduced by this approximation ϵ_1 we have $\epsilon_1 \sim (\frac{\sqrt{2}}{3})^p$.

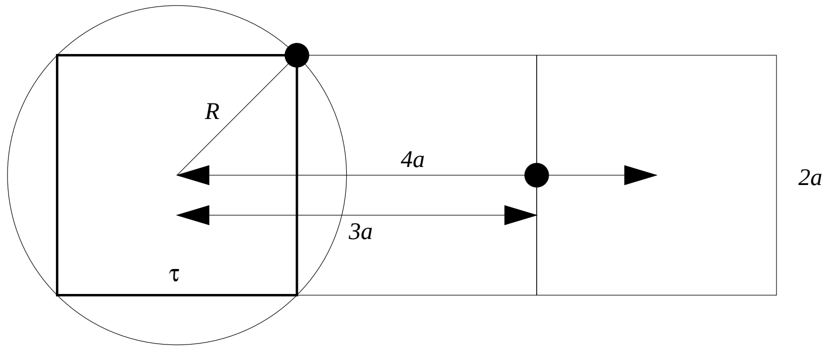

Figure 1.2: Worst case configuration.

The other error that we introduce is when we transfer a multipole expansion to a Taylor expansion. The worst case is given by the same geometric configuration as the previously discussed error. In part 2 of lemma 2 we get $R = \sqrt{2}a$ and the separation between the centres of the clusters is $4a$. Thus we get c_0 from $4a = (1 + c_0)\sqrt{2}a$ which implies a geometric factor of $\frac{1}{c_0} = \frac{\sqrt{2}}{4-\sqrt{2}} = 0.547\ldots$ Calling the relative error from this approximation ϵ_2 we get $\epsilon_2 \sim (\frac{\sqrt{2}}{4-\sqrt{2}})^p$.

Given a desired relative accuracy ϵ we see that we should choose p so that $\epsilon \sim (\frac{\sqrt{2}}{4-\sqrt{2}})^p$. However, a careful error analysis shows that the largest geometric factor we can get is $\frac{\sqrt{2}}{3} < \frac{1}{2}$ (see the next subsection) so in fact we can choose $p \sim -\log_2 \epsilon$. There has been a lot of confusion about the geometric factor in the Fast Multipole Method, with different papers giving different values. In appendix B we clarify a few misunderstandings in different articles, this might be helpful to the reader who wants to study the original references.

The choice of n depends on the implementation. Increasing n will make Step 4 faster and Steps 1, 2 and 3 slower. The slowest of these three is Step 3, to get a new Taylor expansion we have to transfer p coefficients for each one of 27 clusters in the interaction list. One will find that if we set $n \approx \log_4 N$ (so that every panel contains on average one particle) then Step 3 will take considerably more time than Step 4. Decreasing n by two or three will usually give the most efficient algorithm. In chapter 3 we will see how the choice of n affects the running time for a particular code. Note that the choice of n will not effect the accuracy of the method (unless we need a certain minimum cluster size for a short-range interaction).

Using $n \sim \log_4 N$ it is straightforward to check that the algorithm is indeed $\mathcal{O}(N)$. However, the coefficient is large so the method requires a sizeable N to be more efficient than other methods, such as the optimised Ewald method.

A Refinement of the Error Estimate

In the previous subsection we used an error estimate where we assumed that the error was as big as the norm of the truncated part of the series. We also used a couple of rather crude estimates such as $\frac{R^k}{k|x|^k} \leq \frac{R^k}{|x|^k}$. If we do a more precise error analysis it turns out that the geometric factor is never larger than $\frac{\sqrt{2}}{3}$ and that the coefficient is just some 10% of the value we found previously. In real simulations one will often find that the error is smaller still. The following discussion is based on an article by Petersen *et al.* [26]. Large parts of the calculations in that article are omitted; the only parts that are included are included since they give an idea of how to perform a more precise error analysis.

For a single pair of unit charges (with the log-potential), we will see how we can analytically express the value the fast multipole method gives of their interaction, whereupon we can investigate the error. For convenience we will study the force itself, not the potential which is what we usually compute. The reason for this is that the force falls off faster when the distance tends to infinity and we thus avoid the difficulties encountered when handling the singular log-potential. (Note that in the 2-dimensional case we can use the multipole method directly to compute the force F since it is given by a complex potential $\psi = -F_x + iF_y = \varphi'$.) Say that the particles are located in boxes in the finest partition whose centres are located at $\alpha_{1,n}$ and $\alpha_{2,n}$ respectively. Within those boxes, we call the local coordinates (relative to the centres of the boxes) z_1 and z_2. The forces these particles exert on each other are given by the complex force ψ. Set $\alpha = \alpha_{2,n} - \alpha_{1,n}$ to get

$$\psi(z_1, z_2, \alpha) = \frac{1}{\alpha + z_2 - z_1}$$

When we apply the multipole method to find the potential at the site of the second particle due to the first one we start by expanding the force in a multipole-expansion about $\alpha_{1,n} + z_1$,

$$\frac{1}{z - \alpha_{1,n} - z_1} = \sum_{i=0}^{p} \frac{z_1^i}{(z - \alpha_{1,n})^{i+1}} + \frac{1}{z - \alpha_{1,n} - z_1} \Delta(z_1, z, \alpha_{1,n}, p)$$

where from the formula for a geometrical series,

$$\Delta(a,z,b,p) = \left(\frac{a}{z-b}\right)^{p+1}$$

If the box that contains the second charge is in the interaction list of the first box then this expression gives us the error immediately. If this is not the case then as we have seen the multipole algorithm proceeds by transferring the expansion to coarser levels until we reach a level where the boxes are big enough to be in each others interaction lists. As we know the new coefficients are given by a linear combination of the old ones but since the multipole-expansion is unique it is immaterial how we got it. We can write down directly that the expansion looks like

$$\sum_{i=0}^{p} \frac{(z_1 + \alpha_{1,n} - \alpha_{1,n-1})^i}{(z - \alpha_{1,n-1})^{i+1}}$$

and if we assume that we have to go k steps until the box ends up in the interaction list an obvious induction gives

$$\frac{1}{z - \alpha_{1,n} - z_1} = \sum_{i=0}^{p} \frac{(z_1 + \alpha_{1,n} - \alpha_{1,n-k})^i}{(z - \alpha_{1,n-k})^{i+1}} + \frac{1}{z - \alpha_{1,n} - z_1}\Delta(z_1 + \alpha_{1,n} - \alpha_{1,n-k}, z, \alpha_{1,n-k}, p)$$

We see that this is exactly the same form as our expression for the finest mesh level (before we had started shifting the expansions). Having drawn this expected conclusion we can change our notation in order to make the calculations more transparent. We will place the origin at $\alpha_{1,n-k}$ and set $\xi = \alpha_{2,n-k} - \alpha_{1,n-k}$. To get $\psi(z_1, z_2, \xi)$ we simply replace z by z_2 in the truncated series above and with our new notation we get the following expansion of the potential about the centre of the box that contains the first particle

$$\sum_{i=0}^{p} \frac{z_1^i}{(\xi + z_2)^{i+1}}$$

Now recall that before the potential is evaluated the series above is transferred to a Taylor series in z_2 about $\alpha_{2,n-k}$ or, in our new notation, ξ. The new series expansion looks like

$$\sum_{i=0}^{p} \frac{z_1^i}{(\xi + z_2)^{i+1}} = \sum_{i=0}^{p} z_1^i \sum_{j=0}^{\infty} \frac{z_2^j}{j!} \frac{d^j}{d\xi^j} \xi^{-(j+1)}$$

The second series is also truncated after p terms to get the final expression for the force given by the fast multipole algorithm

$$\psi_p(z_1, z_2, \xi) = \sum_{i=0}^{p} \sum_{j=0}^{p} \frac{(i+j)!}{i!j!} \frac{z_1^i (-z_2)^j}{\xi^{i+j+1}}$$

Note that in the algorithm this Taylor expansion at meshlevel $n-k$ is transferred to an expansion at level n but by the same argument as before this does not change the value of the force at $\xi + z_2$. We define the relative error Δ_{rel} by

$$\psi(z_1, z_2, \xi) = \psi_p(z_1, z_2, \xi) + \psi(z_1, z_2, \xi)\Delta_{rel}(z_1, z_2, \xi, p)$$

In order to smooth the ground for an error analysis we will rewrite the error. By expanding $\Delta(z_1, z_2, \xi, p)$ (with Δ defined as above) in z_1 and z_2 one can convince oneself that

$$\Delta_{rel}(z_1, z_2, \xi, p) = \Delta(z_1, z_2, -\xi, p) + \Delta(z_2, z_1, \xi, p) - \Delta^*(z_1, z_2, \xi, p)$$

where

$$\Delta^*(z_1, z_2, \xi, p) = \frac{1}{\psi(z_1, z_2, \xi)} \sum_{i=p+1}^{\infty} \sum_{j=p+1}^{\infty} \frac{(i+j)!}{i!j!} \frac{z_1^i(-z_2)^j}{\xi^{i+j+1}}$$

is the quantity that remains to be estimated. This is done by Petersen *et al.* [26, chapter 3], through some clever complex analysis. The result is

$$\Delta^*(z_1, z_2, \xi, p) = \Big(-\frac{4z_1z_2}{\xi^2}\Big)^{p+1} \sqrt{\frac{1}{\pi(p+1)}} \frac{\xi(\xi + z_2 - z_1)}{(\xi + 2z_2)(\xi - 2z_1)} \Big(1 + O(\frac{1}{p})\Big)$$

Our task now is to do a worst case analysis; we want to find the values for z_1, z_2 and ξ that maximise the total error which is given by

$$\Big(\frac{z_1}{\xi + z_2}\Big)^{p+1} + \Big(\frac{z_2}{z_1 - \xi}\Big)^{p+1} - \Big(-\frac{4z_1z_2}{\xi^2}\Big)^{p+1} \sqrt{\frac{1}{\pi(p+1)}} \frac{\xi(\xi + z_2 - z_1)}{(\xi + 2z_2)(\xi - 2z_1)} \Big(1 + O(\frac{1}{p})\Big)$$

We see directly that of the possible configurations of the boxes the worst case is when they share an edge (since in this case $|\xi|$ is a minimum), i.e. when $\xi = \pm 2$ or $\pm 2i$ (in units of the side length of the box). Due to symmetry (there is no reason why any of these four configurations should cause a larger error than any other) we need only consider one of these cases. Next thing we have to study what combination of z_1 and z_2 gives the largest error. There are seven possible candidates; the analysis of each will not be presented here, the interested reader is referred to the above mentioned article by Petersen *et al.* The result of that analysis is that we get an upper bound for the geometric factor of $\frac{\sqrt{2}}{3}$, a slightly smaller value than our crude estimate gave. The coefficient in front of the factor is about 10 times smaller than our previous coefficient. Another reason for going through the tedious work of deducing these estimates is that the result is given in a simple analytical form that can be used for tuning the algorithm towards maximum efficiency in any particular case.

Other Boundary Conditions

In this section we will show how to adjust the algorithm to take care of boundary conditions other than the one previously studied (free particles). We will start by showing how to handle the case with periodic boundary conditions and then we will show how we can reduce the case with homogeneous Dirichlet condition to that case. Finally we will briefly sketch how to take care of other cases.

The problem we consider first is to calculate the potential from a distribution of charges given that the potential and the normal derivative of the potential should be the same on the two horizontal and the two vertical edges. Before we attack the problem we rephrase it slightly; if we compute the potential due to an infinite number of boxes, all identitical images of the first box, just translated to form a grid over the whole plane, then due to symmetry we infer that the above mentioned periodic boundary conditions must be satisfied. Due to the uniqueness of solutions to Poisson's equation we can use this theoretical construction to solve the original problem. In fact the periodic boundary condition often arises from a physical set-up just like the one with an infinite number of identical boxes. This is for instance the common model for a lattice in solid state physics. There the computational box corresponds to an irreducible cell of symmetry. In order to do this in two dimensions we have to assume that the net charge of the computational box is zero. Otherwise the leading term of the multipole expansion will be the log-term and we will not get convergence. This is a disturbing restriction that is not satisfied in many natural applications, astrophysics for instance. It basically tells us that two-dimensional models have a limited applicability. However, we can bypass the problem by working with the force instead of the potential. We still do not get absolute convergence for the infinite sum but by physical reasoning we can still assign a specific value to it. From equation (1.3.5) we know that the force itself can be represented by an analytic function and thus we can use the fast multipole algorithm directly as it stands.

To solve the problem one starts just as before in Steps 1 and 2 but we perform the loop over the meshlevels in Step 2 over *all* meshlevels, even the two coarsest. After Step 2 we will have multipole expansions around the centre of each cluster accounting for all the charges inside that cluster. In particular we will have an expansion $\Phi_{T_0}(z) = \sum_{k=1}^{p} \frac{a_k}{z^k}$ around the origin (the centre of the biggest cluster) accounting for all the charges in the whole computational box. This same expansion can be used for the charges in any one of the images of the original box. If the image-box is η-admissible with respect to all the points in the original box we can translate the multipole-expansion around the centre of the image to a Taylor-expansion around the origin that accounts for the charges in the image. We use part two of lemma 2 and if the image has centre z_0 (in which case z_0 will have integer real and imaginary parts) we get a local expansion around the origin

$$\Upsilon_{z_0}(z) = \sum_{j=1}^{p} z^j \frac{1}{(-z_0)^j} \sum_{k=1}^{p} \frac{a_k}{(-z_0)^k} \binom{j+k-1}{k-1} (-1)^k$$

If we denote by S the set of centres of image-boxes that are not neighbours of the original box then every box in S is η-admissible with respect to every point in the original box. Thus we can take care of all the images apart from the eight neighbours by evaluating the sum

$$\Upsilon(z) = \sum_{z_0 \in S} \Phi_{z_0}(z) = \sum_{j=1}^{p} b_j z^j$$

where

$$b_j = \sum_{k=1}^{p} a_k \binom{j+k-1}{k-1} (-1)^k \sum_{z_0 \in S} \frac{1}{(-z_0)^{j+k}}$$

If $j+k > 2$ then the last sum is absolutely convergent and the computation is straightforward. For the remaining cases $j+k=2$ and $j+k=1$ we have to resort to physical reasoning to choose a suitable value for the sum.

The sum $\sum_{z_0 \in S} \frac{1}{z_0}$ corresponds to the force at the origin exerted by a field of unit charges, one located at the centre of each box in S, that is, one at each complex number whose real and imaginary parts are integers, one of which is larger than one. Due to symmetry this force must be zero so we set $\sum_{z_0 \in S} \frac{1}{z_0} = 0$.

To evaluate $\sum_{z_0 \in S} \frac{1}{z_0^2}$ we consider a field of unit strength dipoles directed along the x-axis, one located at each complex number that has integer real and imaginary parts. Due to symmetry we can this time say that the potential must be the same at the two equivalent points $z = 1/2$ and $z = -1/2$. Thus $\delta\Phi \equiv \Phi(1/2) - \Phi(-1/2) = 0$. The contribution to $\delta\Phi$ from a dipole located at z_0 is

$$\frac{1}{z_0 - 1/2} - \frac{1}{z_0 + 1/2} = \frac{1}{z_0^2 - 1/4}$$

Call the set of all dipoles but the one at the origin S'. We get

$$0 = \delta\Phi = \sum_{\text{all dipoles}} \frac{1}{z_0^2 - 1/4} = -4 + \sum_{z_0 \in S'} \Big(\frac{1}{z_0^2} + \frac{1}{4z_0^4 - z_0^2} \Big) =$$

$$= -4 + \sum_{z_0 \in S} \frac{1}{z_0^2} + \sum_{z_0 \in S' \setminus S} \frac{1}{z_0^2} + \sum_{z_0 \in S'} \frac{1}{4z_0^4 - z_0^2}$$

The reason why we treat the dipole at the origin as a special case is that the expansion we do in the third equality is not valid unless $|z_0| \geq 1$. Of the sums in the last line the first one is the one we want to compute. The second one is finite, we sum over eight z_0's grouped together in four pairs symmetrically placed around the origin, thus that sum equals zero. The third sum takes some skill to compute but it is absolutely convergent so there is no ambiguity about the answer, $4-\pi$. We can conclude

$$\sum_{z_0 \in S} \frac{1}{z_0^2} = \pi$$

The conversion of the multipole expansion for the full box, Ψ_{T_0}, to a Taylor expansion around the origin, Υ_{T_0}, is a linear transformation of the coefficients which we will call L. The entries of the matrix L are constant and can be precomputed once and for all using the formula

$$L_{j,k} = \binom{j+k-1}{k-1}(-1)^k \sum_{z_0 \in S} \frac{1}{z_0^{j+k}} \tag{1.3.8}$$

Once this is done one simply starts Step 3 by doing this transformation. Then the only difference between the case with periodic boundary conditions and the case with free particles is that when one computes the local Taylor expansions in Step 3 one has to add to the interaction list the relevant boxes in the nearest image boxes.

It should be mentioned that there are other conventions for assigning the non-absolutely convergent sum a value. The approach described above is the one given by Greengard in [14]. A method commonly used when implementing the Ewald algorithm is to sum over all images at the same distance from the origin first. Then the contributions from all the different layers are summed up, starting with the innermost shell. This is the approach we use in the test-program in chapter 3. In this case the lowest powers vanish. When computing $\sum_{z_0 \in S} 1/z_0$ we pair z_0 with $-z_0$ and when we compute $\sum_{z_0 \in S} 1/z_0^2$ we pair z_0 with iz_0. Thus both these sums equal zero. It is easily seen that all terms with odd powers in the denominator vanish.

To handle the case with homogeneous Dirichlet boundary conditions we use the technique of images. The idea is simple; if we can construct a distribution of free charges by adding charges outside the computational domain in such a way that the boundary conditions are satisfied, then by again invoking uniqueness of solutions to Poisson's equation we know that we have found the desired solution.

To construct the new distribution we start by reflecting the whole set of charges in the upper boundary, a particle with charge q is reflected to a particle with charge $-q$. Then the potential from this pair is zero along the whole upper boundary and the same goes for all other particles. We still have not satisfied the Dirichlet condition on any other boundary. To satisfy it on the lower boundary we reflect *all* our charges, the original ones and the new image charges in the lower boundary. Now the condition is satisfied on the lower boundary but not on the upper one. If we continue this process and add in two new image boxes at each step we will at every second step satisfy the boundary condition on one edge but at the next step we will disturb it again. However, with each step this disturbance gets smaller and it is easily seen that in the limit, where we have added infinitely many boxes, every second one turned upside down the potential will be zero on both boundaries. The whole process is illustrated in figure 1.3.

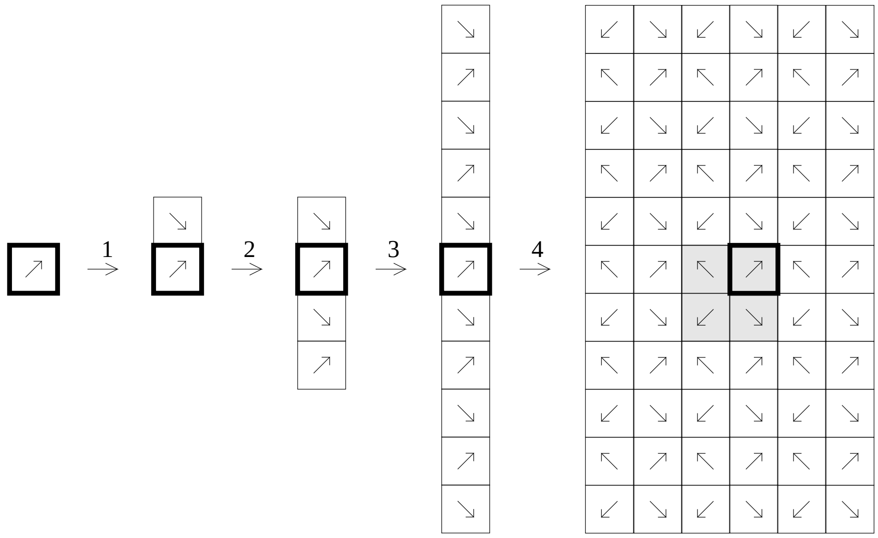

Figure 1.3: In step 1 we reflect the box in the upper edge. In step 2 we reflect both boxes in the lower edge. Step 3 represents an infinite number of horisontal reflections. In Step 4 we do all the vertical reflections as well. The "superbox" is shaded.

It still remains to satisfy the boundary conditions on the vertical sides, this is done in exactly the same way, just that now we reflect the whole column of charges in the left and right boundaries. This process does not disturb the potential on the horizontal boundaries since we preserve symmetry in each of these lines.

After having placed all the image charges we find that our charge distribution consists of an infinite number of images of the original box, each one reflected in one of four possible ways. See figure 1.3. We find that we end up with a charge distribution that consists of a "superbox" with centre at $z = 1/2 + i/2$, periodically repeated over the entire plane. We have thus reduced the problem to the one we solved before.

To handle homogeneous Neumann conditions we proceed in a similar manner. The only difference is that this time we reflect a particle to a particle with the same charge (we do not change the sign as we did before). Likewise we can handle mixed Neumann/Dirichlet conditions by using different reflection rules depending on which side we reflect in.

Time-stepping

When we choose the time-step in a dynamical simulation of a particle system we have to choose it small enough so that the force on each particle does not change too much between the time-steps. If it did then we would get a different trajectory and the simulation would not give an accurate result. The particles near a given particle interact strongly with that particle and they also change their relative positions rapidly. Thus it is the dynamics in the immediate neighbourhood of the particle that puts a limit on how long a time-step we can use. One idea that has been used in many algorithms is that we can use different time-steps for interactions in the near field and in the far field. See for instance Allen and Tildesley [2, pp. 152 - 168] or Streett *et al* [34].

Implementing this technique with the multipole algorithm would be quite simple. All long-range interactions are handled by the multipole expansions around the larger clusters. A modified algorithm could, after having performed a "normal" force computation, do a few timesteps where it stops after having looped over the two or three finest meshlevels in Step 2 and then start going downwards from that level in Step 3. That is, we would not recompute the expansions around the larger clusters in every timestep. It is even quite simple to estimate the errors we would introduce. By differentiating the expression for a multipole coefficient we can get an estimate of how rapidly it changes. Using coordinates relative to the centre of the cluster we get from theorem 1 for the k'th coefficient in the multipole expansion around τ the expression $\sum_{y_l \in \tau} -\frac{q_l(y_l-y_\tau)^k}{k}$. Note that it does not matter if the coefficient has been arrived at by transferring coefficients from a finer mesh since the translation formula is exact.

$$\frac{d}{dt}\sum_{l:y_l\in\tau} -\frac{q_l(y_l-y_\tau)^k}{k} = -\sum_{l:y_l\in\tau} q_l(y_l-y_\tau)^{k-1}\dot{y}_l = -\sum_{l:y_l\in\tau} q_l(y_l-y_\tau)^{k-1}v_l$$

where v_l denotes the velocity of particle l. We can get the relative change of the contribution from particle l by dividing by $-\frac{q_l(y_l-y_\tau)^k}{k}$ which gives $\frac{kv_l}{y_l-y_\tau}$. If $y_l - y_\tau$ is very small then it can be argued that the whole contribution from particle l is small so it does not matter that the relative change is large. An extra error is introduced by the fact that the members of the sum change as particles leave and enter the cluster but for a large cluster these effects are not important.

Having shown that it is perfectly feasible to use different time-steps within the multipole algorithm, we next state that there is very little to gain from doing so. The reason is that since the number of clusters on level m is given by 4^m, the vast majority of clusters are found on the two or three finest meshlevels ($\frac{3}{4}$ of all the clusters are found on the finest mesh-level, of the remaining $\frac{1}{4}$, we find $\frac{3}{16}$ on the second finest and so on). Thus the gain in computational time in skipping the coarser meshlevels is negligible.

An Adaptive Algorithm

If the distribution of particles is highly non-uniform the algorithm we have presented will not be very efficient. We have to refine the mesh until no cluster contains more than a handful of particles. If the distribution is non-uniform this means that most clusters will be empty. To handle this case Greengard presents an adaptive algorithm in [14]. The idea is that we subdivide only those clusters that contain more than a certain number of particles (this number has to be chosen beforehand). We will still get a few empty clusters but these will be marked as empty and will be subsequently ignored by the algorithm. The coding can be done with recursive functions and if it is well done the adaptive algorithm is almost as efficient as the original algorithm for uniform distributions and far more efficient for non-uniform ones.

It might seem that one should always use the adaptive algorithm but the coding does get more complicated so if the distribution of particles is expected to be reasonably uniform one might not find it worth the extra work. If the code is used in a commercial code for the N-body problem the adaptive algorithm would probably be the better choice.

1.3.3 The Fast Multipole Method in Three Dimensions

The big problem when we go to three dimensions is that the housekeeping gets far more complicated. We can no longer use complex notation but have to use proper vectors represented in some coordinate system. Multipole expansions in three dimensions are given in spherical coordinates and so it is very convenient to use them for a theoretical analysis. However, getting the local spherical coordinates for one point with respect to another one given the two points' individual spherical coordinates is a nasty numerical problem and it might be more efficient to use Cartesian coordinates when implementing the algorithm.

The theoretical analysis of the problem gets a bit more involved; proving translation properties for spherical multipole expansions takes some work and proving error estimates is even more complicated. However, one ends up with quite simple formulæ so these problems should not make a user concerned.

As we have already mentioned, in three dimensions we get $\#I_p = (p+1)^2$ so the computational time increases faster in this case when higher accuracy is needed.

If one uses the same interaction list in three dimensions as in two the geometric factors in the error bounds grow. The worst case occurs in the same geometrical configuration as in two dimensions but now we need a sphere of radius $\sqrt{3}a$ instead of $\sqrt{2}a$ to hold the cluster. This means that $\eta_0 \sim \frac{\sqrt{3}}{3} = 0.577...$ and $\frac{1}{c_0} \sim \frac{\sqrt{3}}{4-\sqrt{3}} = 0.764....$ Petersen *et al* [27] have with a worst case analysis shown that in fact the geometric factor is 3/4. This is still a very large geometric factor and it gives slow convergence. To get a smaller geometric factor one can define a new interaction list. For a cluster τ we define it to be the set consisting of *those children of the neighbours or the second neighbours of* τ*'s parents that are not neighbours of second neighbours of* τ. This gives $\frac{1}{c_0} = \frac{\sqrt{3}}{6-\sqrt{3}} = 0.406...$ but on the other hand this interaction list contains 875 elements instead of 189 for the smaller list. Greengard [14] claims that the larger list is to be preferred.

Chapter 2

Integral Equations

2.1 Introduction

2.1.1 Background

In this chapter we will look at different ways to discretise and solve the equation

$$\lambda u + Ku = r \tag{2.1.1}$$

where λ is a constant, u and r are functions from a smooth k-manifold Γ to R, and K is an integral operator on a function space on Γ,

$$[Ku](x) = \int_\Gamma k(x,y)u(y)\,ds_y$$

where ds_y is a k-form. In this paper ds is usually a $(d-1)$-form on a $(d-1)$-manifold embedded in R^d. This equation is a *Fredholm equation* of the first $(\lambda \equiv 0)$ or the second $(\lambda \neq 0)$ kind.

The main reason why integral equations have attracted much interest is that one can often use classical potential theory to rewrite an elliptic PDE as a Fredholm equation. The PDE itself can arise in any field of physics or engineering; electro-magnetic field theory, fluid mechanics or thermodynamics are but a few examples. The reason one does this transformation is that sometimes — especially if the boundary is complicated or if the domain is unbounded — it is easier to solve the integral equation than to solve the PDE directly. The next section has been devoted to showing a couple of examples of how the transformation can be done. These examples give an idea of what kernels one can expect to encounter.

Most classical methods to solve (2.1.1) involve triangulating the manifold Γ into, say, N smooth simplices (i.e. smooth images of line-segments, triangles etc). Then the equation is transformed into a matrix equation with a square matrix, at least of size $N \times N$, whereupon an iterative solver is used to solve the matrix equation (the exact dimension is approximately proportional to N, and determined by the finite-dimensional function space chosen to house the approximate solution). Thus the amount of work required is at least $\mathcal{O}(N^2)$. Even if the computing time is not a problem the cost of storing more than $N \times N$ matrix elements might be prohibitive. However, it turns out that we do not in fact need the matrix itself, because all the iterative algorithm uses is the result of the multiplication of the matrix with a vector. This is where the separation of variables method comes in. This algorithm provides a way to do that computation in $\mathcal{O}(N \log N)$ operations using an amount of storage that grows in the same manner. For some special methods we can even reduce the cost to $\mathcal{O}(N)$.

Doing the matrix multiplication is essentially the discrete equivalent of evaluating the result of the operator K. Sometimes that is the primary goal; for instance in numerical complex analysis the Cauchy integral,

$$f(z) = \frac{1}{2\pi i} \int_\Gamma \frac{h(t)}{t - z} dt,$$

is encountered when one needs the numerical value of an analytic function, when one wants to investigate conformal mappings *et cetera*.

2.1.2 Transformation of Laplace's Equation to Fredholm Equations

We will see how the exterior Dirichlet problem can be transformed in two different ways. Let Γ be a smooth closed surface in R^3 and denote by Ω the open region of R^3 that is not contained in Γ. The exterior Dirichlet problem can now be formulated: Find u such that

$$\begin{aligned} \triangle u &= 0 && \text{in } \Omega \\ u(x) &= g(x) && \text{for x} \in \Gamma \\ u(x) &= \mathcal{O}(|x|^{-1}) \end{aligned} \tag{2.1.2}$$

Under certain conditions on g, this problem has a unique solution. (If $g \in H^{1/2}(\Gamma)$ then there exists a unique $u \in H^1_{\text{loc}}(\overline{\Omega})$ that fulfils the criteria.)

Consider the following equation for v,

$$\int_\Gamma \frac{1}{4\pi|x-y|} v(y)\, ds_y = g(x) \qquad \text{for } x \in \Gamma. \tag{2.1.3}$$

If this equation has a "nice" solution then we can define a function u by

$$u : \Omega \to R : x \mapsto \int_\Gamma \frac{1}{4\pi|x-y|} v(y)\, ds_y$$

A formal differentiation with respect to x under the integral sign shows that the Laplacian vanishes, and by the defining equation for v it looks like $u|_\Gamma = g$ so u seems to be a solution of the PDE.

To do this properly one has to investigate under what conditions on g does equation (2.1.3) have a solution that is nice enough to allow differentiation under the integral sign, and also nice enough to let u be continuous as x approaches Γ. This analysis is technically quite involved and is omitted from this thesis.

Another way of rewriting the equation is to use the following corollary to the Gauss Divergence Theorem.

Lemma 4 *Let $\partial\omega$ be the smooth boundary of a bounded set ω. Let n be the outwards pointing unit normal to ω. Then if u and v are harmonic functions on ω and $\mathcal{C}^1$ on $\overline{\omega}$,*

$$\int_{\partial\omega}\left(u\frac{\partial v}{\partial n} - v\frac{\partial u}{\partial n}\right) ds = 0 \tag{2.1.4}$$

Let E denote the standard fundamental solution to Laplace's equation, so that $\triangle E = -\delta$, the Dirac's delta function. In three dimensions we get

$$E(x) = \frac{1}{4\pi|x|} \qquad \text{for } x \in R^3\backslash\{0\}$$

We apply the lemma taking u to be the solution of (2.1.2) and v to be the function defined by $y \mapsto E(y-x)$. This makes v harmonic when $y \neq x$. Let x be any point in Γ, and S_R a sphere around the origin large enough to contain the whole set Γ. For $\epsilon > 0$, Γ_ϵ is the part of Γ that is more than ϵ away from x, and S_ϵ is the part of a small sphere centred on x needed to cover the bit of Γ we cut away. Figure 2.1 should clarify the definitions.

Chose ω in the lemma to be the part of Ω that is contained inside S_R and that has boundary $S_R + S_\epsilon + \Gamma_\epsilon$ to get (n_y is an outwards pointing unit normal)

$$\int\limits_{S_R+S_\epsilon+\Gamma_\epsilon}\Big(u(y)\frac{\partial E(y-x)}{\partial n_y} - E(y-x)\frac{\partial u(y)}{\partial n_y}\Big)\, ds_y = 0 \tag{2.1.5}$$

We investigate the six parts of the integral separately. Changing variables to $z = y - x$ we get

$$\begin{aligned}\int_{S_\epsilon(x)} u(y)\frac{\partial E(y-x)}{\partial n_y}\, ds_y &= \int_{S_\epsilon(0)} u(z+x)\frac{1}{4\pi|z|^2}\, ds_z \\ &= u(x+\xi_\epsilon)\frac{1}{4\pi\epsilon^2}\int_{S_\epsilon(0)} ds_z\end{aligned}$$

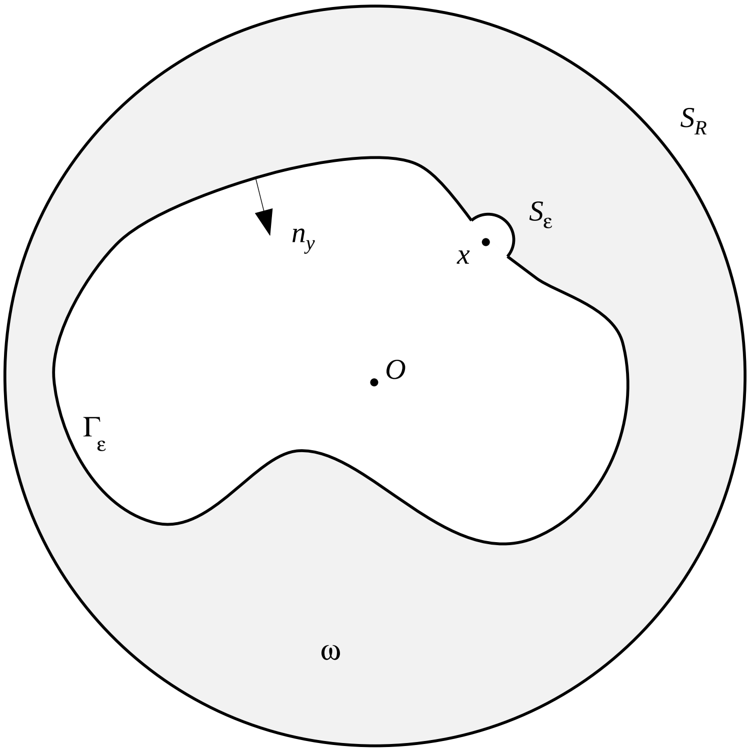

Figure 2.1: Integration domain for conversion of exterior Dirichlet problem (shaded in picture). We show a 2D cut of a 3D body.

where ξ_ϵ is a vector of length ϵ that can be introduced by virtue of the mean value theorem for integrals. As $\epsilon \to 0$, the area of S_ϵ tends to $2\pi\epsilon^2$ and this part of the integral is seen to tend to $u(x)/2$. Next,

$$\int_{S_\epsilon} \frac{\partial u(y)}{\partial n_y} E(y-x)\, ds_y = \int_{S_\epsilon} \frac{\partial u(y)}{\partial n_y} \frac{1}{4\pi\epsilon}\, ds_y = \frac{\partial u}{\partial n_y}\Big|_{x+\xi_\epsilon} \frac{1}{4\pi\epsilon} \int_{S_\epsilon} ds_y$$

and since the integral is $\mathcal{O}(\epsilon^2)$ this term tends to 0 as ϵ tends to 0.

$$|\int_{S_R} u(y) \frac{\partial E(y-x)}{\partial n_y}\, ds_y| \leq \int_{S_R} |u(y)| \Big| \frac{\hat{r}_y \cdot (y-x)}{4\pi |y-x|^3} \Big|\, ds_y$$

where $\hat{r}_y = y/|y| = y/R$. We can easily find constants R_0 and C such that $\frac{\hat{r}_y \cdot (y-x)}{4\pi|y-x|^3} \leq C/4\pi R^2$ whenever $R > R_0$. Thus if $R > R_0$ we find the following bound for the integral

$$\Big|\int_{S_R} u(y) \frac{\partial E(y-x)}{\partial n_y}\, ds_y \Big| \leq \int_{S_R} |u(y)| \frac{C}{4\pi R^2}\, ds_y \leq C ||u||_{\mathcal{L}^\infty(S_R)}$$

By the defining boundary value problem for u, (2.1.2), we know that u tends to zero as $|y|$ tends to infinity. Since R does not enter any other terms in

(2.1.5) the integral has to be independent of R. By our previous conclusions this constant value has to be zero.

If u is $\mathcal{O}(1/|x|)$ then (since u is harmonic) $\frac{\partial u}{\partial n}$ is $\mathcal{O}(1/|x|^2)$. Using this we can show (using the same technique as in the previous paragraph) that the second term in the integral over S_R also vanishes.

Next we consider the two terms

$$\int_{\Gamma_\epsilon} E(y-x)\frac{\partial u(y)}{\partial n_y}\, ds_y \quad \text{and} \quad \int_{\Gamma_\epsilon} u(y)\frac{\partial E(y-x)}{\partial n_y}\, ds_y$$

The first clearly has a weakly singular kernel and so its value is unambiguous as ϵ tends to 0. In the second term the kernel is

$$\frac{1}{4\pi}\frac{n_y \cdot (x-y)}{|y-x|^3}.$$

This appears to be strongly singular but since n_y is perpendicular to the surface the numerator is $\mathcal{O}(|y-x|^2)$ for a smooth surface. Thus the kernel is in fact weakly singular. This is not true for a non smooth surface or for other PDE's; in the general case one has to *define* the integral of a strongly singular integrand as the above limit value, this is sometimes called the *part finit* or *Cauchy principal value* of the integral.

Assembling the different parts we get

$$\frac{1}{2}u(x) - \int_\Gamma E(y-x)\frac{\partial u(y)}{\partial n_y}\, ds_y = -\int_\Gamma u(y)\frac{\partial E(y-x)}{\partial n_y}\, ds_y.$$

We know that $u|_\Gamma = g$ so we can substitute u and rewrite the equation to get

$$\int_\Gamma E(y-x)\frac{\partial u(y)}{\partial n_y}\, ds_y = \int_\Gamma g(y)\frac{\partial E(y-x)}{\partial n_y}\, ds_y + \frac{1}{2}g(x) \equiv r(x). \qquad (2.1.6)$$

Thus, $\frac{\partial u}{\partial n}_\Gamma$ is a solution of an integral equation with a weakly singular kernel.

Upon solving this integralequation, both $u|_\Gamma$ and $\frac{\partial u}{\partial n}|_\Gamma$ will be known. From this data we can find an explicit expression for $u(x)$ in all of Ω as follows. For a fixed x we take a small sphere around x with radius ϵ, integrate $u\frac{\partial E}{\partial n} - E\frac{\partial u}{\partial n}$ over a domain analogous to the one shown in figure 2.1 and do the same limiting procedure as we did when we found the integral equation. The result is

$$u : \Omega \to R : x \mapsto \int_\Gamma \Big(E(y-x)\frac{\partial u}{\partial n_y} - u(y)\frac{\partial E(y-x)}{\partial n_y} \Big)\, ds_y$$

All the terms in the right hand side are known.

For further details about these transformations we refer to McLean [25].

In a similar manner one can transform boundary value problems for Laplace's equation with other boundary conditions. Using volume integrals, it is even possible to transform a general, inhomogeneous elliptic PDE of form

$$\triangle u + \alpha \cdot \nabla u + \gamma u = \varphi$$

to an integral equation. Here γ and φ are scalar valued functions and α is a vector valued function. (In practise α and γ have to be constant since otherwise the kernel gets very complicated.) See Rokhlin [29].

2.2 Discretising Integral Equations

We will look at three different methods for discretising the Fredholm equation (2.1.1): the Collocation, the Nyström and the Galerkin methods. These will transform the integral equation to a matrix equation. Since the Fredholm equation itself can be shown to be elliptic for many commonly occurring kernels, the arising matrix will often be positive definite for a good discretisation technique. For equations of the second kind ($\lambda \neq 0$) the condition number of the matrix does not increase as the resolution of the dicretisation is increased. For the first kind equation 2.1.6, the condition number is typically $\mathcal{O}(N)$.

Of the three methods the Galerkin method is the most tractable from a theoretical viewpoint. There exist proofs of stability and convergence for a large class of problems involving both non-smooth domains and singular kernels. On the other hand it turns out to be very hard to prove anything for the collocation or Nyström methods. In fact they sometimes do not converge for certain interesting problems. However, if the problem does not involve any complicated boundaries or singular kernels, the collocation and Nyström methods give the same accuracy as the Galerkin method for equations of the second kind for considerably smaller computational time. See Sauter [31].

2.2.1 Boundary Elements

Often it is the case that Γ is a curved domain that might be difficult to represent and integrate over numerically. In this case one has to approximate Γ with some union of simpler elements, known as *panels*. These are usually simplices but can theoretically be of any shape. Denote a panel by π and the union of all π's by Γ_N, where N is taken to be the number of panels.

A question that arises is how to integrate the functions that are defined on Γ over Γ_N ? If the kernel has an obvious extension to the space surrounding Γ then one can simply use this. This is not very accurate nor is it always possible. A more precise but also more complicated way is to regard the π's as a parameter set for Γ. That is we use an atlas of the manifold to construct bijective maps

$$F_\pi : \gamma_\pi \subset \Gamma \to \pi : y \mapsto y'$$

Then we can use the exact formula for change of parameters

$$\int_{\gamma_\pi} k(x,y)u(y)\,ds_y = \int_\pi k(F_\pi^{-1}(x), F_\pi^{-1}(y'))u(F_\pi^{-1}(y'))J_{F_\pi}(y')\,ds_{y'}$$

where J_{F_π} is the Jacobian of F_π. If we use this technique then we work with the kernel

$$\hat{k}(x,y) = \sum_\pi k(F_\pi^{-1}(x), F_\pi^{-1}(y))J_{F_\pi}(y)\mathbf{1}_\pi(y)$$

defined on Γ_N. The drawback with this approach is that we can get a very complicated kernel. A slightly different description of this technique is given by Hackbusch in [16].

An analysis of the error introduced in this part of the discretisation process can be performed and it should be ascertained that the error is not considerably larger than the errors introduced in the following steps. This error is not directly connected to the errors introduced later and we will not try to analyse it here. Instead we shall henceforth assume that we can write $\Gamma = \cup_i \pi_i$ for some simple elements π_i.

2.2.2 The Tree of Clusters

The panels making up Γ are grouped together in what we call *clusters*. This is done in exactly the same way as we grouped the panels together in chapter 1. We recall that the conditions imposed on the clusters imply that we end up with a tree structure where all the panels are on the same level. We will call the root of the tree (i.e. Γ) level 0 and the panels level n. The number of levels is related to the number of panels by $n \leq \log_2 N$, and the number of clusters that are not panels is smaller than N.

2.2.3 Discretisation with the Collocation Method

We will look for an approximate solution u_q, living in some function space $\mathcal{S}^{N,k}$ where N is the number of panels in Γ, and k denotes some order of the function system used; often we will work with the usual finite element space of functions that are piecewise (over each panel) polynomials of degree k. We denote the dimension of $\mathcal{S}^{N,k}$ by q and we will always have $q \sim Nk$. For fixed k this means that $\mathcal{O}(N) = \mathcal{O}(q)$.

Assume further that u_q is uniquely determined by its value at q collocation points $\{x_i\}_{i=1}^q$.

We demand that equation (2.1.1) should be satisfied at the collocation points,

$$\lambda u_q(x_i) + K[u_q](x_i) = r(x_i) \qquad \forall i.$$

Then we expand u_q in a basis $\{v_i\}_{i=1}^q$ for $\mathcal{S}^{N,k}$, $u_q = \sum_{j=1}^q \alpha_j v_j$, to get

$$\sum_{j=1}^{q} [\lambda v_j(x_i) + K[v_j](x_i)]\alpha_j = r(x_i) \qquad \forall i.$$

This is clearly a matrix equation where the $q \times q$-matrix has i, j-entry

$$\lambda v_j(x_i) + K[v_j](x_i).$$

2.2.4 Discretisation with the Nyström Method

This method is essentially a specialisation of the collocation method, where we use a quadrature rule to approximate the integral operator K. A quadrature rule for an integral over some manifold Γ is a set $\{x_i, \omega_i\}_{i=1}^q$ where x_i are points in the manifold and ω_i are some weights such that

$$\int_\Gamma f\, ds \approx \sum_{i=1}^{q} f(x_i)\omega_i$$

If the error is $\mathcal{O}(1/q^s)$ then the quadrature is said to be of order s. The simplest possible quadrature is to partition Γ into some finite elements γ_i and set ω_i = area of γ_i (area is replaced by length/volume/... according to the dimension of the manifold). To get an estimate of the rate of convergence, let h denotes the size of the largest γ_i, and assume that $h \sim 1/q$. This gives for a smooth integrand a first order quadrature. If Γ is an interval $(a, b) \subset R$ then other well known quadratures exist. The trapezoid formula is given by

$$\begin{aligned} x_i &= a + i\frac{(b-a)}{q} \qquad i = 0, 1, 2, \ldots, q \\ \omega_i &= \frac{b-a}{q} \qquad i = 1, 2, \ldots, q-1 \\ \omega_0 &= \omega_q = \frac{b-a}{2q} \end{aligned}$$

The error term is of order 2 and even higher accuracy is attained by the fourth order Simpson's formula which uses the same partition but different weights. However, if the boundary is closed then the functions on it are periodic and in this case the following lemma says that the trapezoidal formula has excellent convergence.

Lemma 5 *Suppose that $\varphi : [0, L] \to R$ has $k \geq 1$ continuous derivatives. Suppose further that φ and its k derivatives are periodic with period L. Then the order of convergence for the trapezoidal rule is equal to $k + 1$.*

The proof is found for instance in [33].

Note that these quadrature formulæ can be used when Γ is a 1-dimensional connected manifold in R^d by simply mapping the integral down to an interval in R.

The points that are used in the quadrature are also used as collocation points (which explains our apparently ambiguous choice to call them x_i). Our discrete equivalent of (2.1.1) is now

$$\lambda(x_i)u_q(x_i) + \sum_{j=1}^{q} \omega_j k(x_i, x_j)u_q(x_j) = r(x_i)$$

This is a matrix equation for the vector $\{u_q(x_i)\}_{i=1}^{q}$ which by assumption determines u_q. We could equally well work with the expansion of u_q in a basis $\{v_j\}_j$. This is simply a different way of representing the solution. Getting from one representation to another involves solving a $q \times q$ matrix equation but since the base functions have local support we deal with a very sparse matrix and the work involved is $\mathcal{O}(N)$.

The Nyström method is most commonly used with piecewise linear functions so $N = q$ and we can take $\gamma_i = \pi_i$. Classically, the method has only been used in two dimensions and for smooth boundaries; in other cases the kernel will be singular when $x = y$. In the Separation of Variables approach though, we can if we want use Nyström discretisation in the far field only.

It should be mentioned that the classical Nyström method involves an object called the *Nyström interpolant* given by

$$u_q(x) = \frac{r(x) - \sum_{j=1}^{q} \omega_j k(x, x_j)u_q(x_j)}{\lambda}.$$

However, calculating this interpolant explicitly is very costly and since we never use it in the solution process we will not be referring to it henceforth.

2.2.5 Discretisation with the Galerkin Method

Let $\mathcal{S}^{N,k}$ and v_i have the same meaning as above.

In the Galerkin method we demand that our approximate solution u_q have the same projection with respect to the energy inner product on $\mathcal{S}^{N,k}$ as r. Mathematically this is formulated

$$\langle v_i, \lambda u_q + K u_q \rangle_{\mathcal{L}^2(\Gamma)} = \langle v_i, r \rangle_{\mathcal{L}^2(\Gamma)} \qquad \forall i$$

Henceforth we will drop the subscript on the inner products; if nothing else is said, we refer to the $\mathcal{L}^2(\Gamma)$ inner product. If we expand u_q as $\sum_{j=1}^{q} \alpha_j v_j$ then we can rewrite the equation as a matrix equation

$$\sum_{j=1}^{q} \Big[\langle v_i, \lambda v_j \rangle + \langle v_i, K v_j \rangle \Big] \alpha_j = \langle v_i, r \rangle \qquad \forall i$$

the matrix element i, j is

$$\langle v_i, \lambda v_j \rangle + \langle v_i, K v_j \rangle = \lambda \int_\Gamma v_i(x) v_j(x)\, ds_x + \int_\Gamma \int_\Gamma v_i(x) k(x,y) v_j(y)\, ds_y\, ds_x \tag{2.2.7}$$

Unless we have some very peculiar kernel k the matrix is dense even when the base functions v_i have local support. This is in sharp contrast to the usual Galerkin-FEM for PDE's which gives a sparse matrix. Furthermore, unless k is symmetric in its arguments then the matrix is non-symmetric. It means (just as for Nyström and Collocation discretisation) that unless we use some special techniques we would have to compute and store q^2 matrix elements, each of which requires the evaluation of a double integral over $\text{supp}(v_i) \times \text{supp}(v_j)$. Then we would have to solve the matrix equation for a full matrix. Clearly this will be expensive for large N.

2.2.6 Iterative Solution Techniques for the Matrix Equation

As we have seen, it is not very hard to discretise the Fredholm Equation to get a matrix equation. However, computing and storing the arising matrix are $\mathcal{O}(N^2)$ procedures and the cost for solving the matrix equation exactly by Gaussian elimination grows as $\mathcal{O}(N^3)$. There exist many approximate iterative techniques for solving matrix equations. Some techniques have the specific property that they never refer to the matrix itself, only to the result of the matrix applied to different vectors. This is very important to us since we will show that there exist methods to perform this vector-matrix multiplication with $\mathcal{O}(N \log N)$ operations — without referring to the matrix elements explicitly.

One such iterative technique is the General Conjugate Residual Method (GCR), described for instance in Eisenstadt [13]. A slightly different but equivalent algorithm is given by Axelsson in [4]. For GCR to work the matrix does not have to be symmetric but it is required that the symmetric part of the matrix is positive definite. This will usually be the case for our discretisation schemes, for Galerkin discretisation it can be shown that the number of negative eigenvalues is fixed as q is increased and by some manipulations we can avoid any problems these would give. The algorithm will converge to the exact solution in q iterations, and in each step the residual is reduced at least by a factor $\frac{\kappa(A)-1}{\kappa(A)+1}$ (where $\kappa(A)$ denotes the condition number of A). Another advantage of the fact that GCR never refers to the matrix itself is that we do not have to store q^2 entries in the matrix, just $\mathcal{O}(N \log N)$ auxiliary quantities. We should say that GCR usually converges far faster than the guaranteed error estimate.

2.3 Separation of Variables

2.3.1 Preliminaries

This technique is used to reduce the most critical cost in the process we outlined in the previous section; evaluating the result of the multiplication of a vector by a full $q \times q$-matrix.

The general idea is the same as for the particle simulations. Instead of expanding a potential between particles we now expand the kernel in the integration. The gain of this is threefold. First of all we get a reduction of computational cost and storage in the set-up phase since we do not have to compute and store q^2 matrix entries. Then each iteration in the process to solve the matrix equation is much faster. Finally when we compute the far-field components we can replace the possibly complicated kernel with a simple function that is easy to integrate.

The following definitions are all the same as for the particle case and are stated here just for completeness. The one thing that is different is that we define our approximation of the kernel to be the exact kernel when x and y are close. This is done for the sole purpose of simplifying the notation.

The approximate kernel $\tilde{k}_{y_\tau}$ has the form

$$\tilde{k}_{y_\tau}(x, y) = \sum_{\iota \in I_p} \varphi_\iota(x, y_\tau) \kappa_\iota(y_\tau, y) \tag{2.3.8}$$

where p denotes the order of the expansion and we demand that $\# I_p \leq C_I p^m$ for some constant C_I and integer m. Later on we can define y_τ as a function of x and y and remove the subscript. We impose the error bound:

There exist constants $\eta_0 < 1, C_1$ and C_2 such that for all $\eta \in (0, \eta_0]$ and x, y, y_τ we have

$$|\tilde{k}_{y_\tau}(x, y) - k(x, y)| \leq C_1 (C_2 \eta)^p |k(x, y)| \quad \textit{if} \quad |y - y_\tau| \leq \eta |x - y_\tau| \tag{2.3.9}$$

To each cluster τ we associate a *centre* y_τ and then we say that the cluster is η-admissible with respect to a point x if

$$\forall y \in \tau : |y - y_\tau| \leq \eta |x - y_\tau|$$

Let an *η-admissible covering* $\mathcal{C}(x)$ of Γ be a collection of panels and clusters for which the following is true (τ can be both a panel and a cluster).

1. $\Gamma \subseteq \cup_{\tau \in \mathcal{C}(x)} \tau$, i.e. $\mathcal{C}(x)$ is a covering of Γ.

2. The intersection between the interiors of any two clusters in $\mathcal{C}(x)$ is empty.

3. Each τ in $\mathcal{C}(x)$ is either a panel or it is η-admissible with respect to x.

4. $\#\mathcal{C}(x) \leq C\left(\frac{1}{\eta}\right)^{d-1} \log(2 + \eta^{d-1}N)$ for some constant C.

5. There is no other covering with fewer elements that satisfies the previous conditions.

The unique covering with the smallest number of elements is given by the recursive procedure *Divide* presented in section 1.2.2. The execution time for this procedure is proportional to $\#\mathcal{C}(x)$. We partition $\mathcal{C}(x)$ into two parts: $\Pi(x)$ consisting of all non-η-admissible panels in $\mathcal{C}(x)$, and $\mathrm{T}(x)$ consisting of all the other clusters and panels in $\mathcal{C}(x)$.

Now we can define $\tilde{k}$ so that in the near-field we use the exact kernel

$$\tilde{k}(x,y) = \begin{cases} k(x,y) & y \in \tau \in \Pi(x) \\ \tilde{k}_{y_\tau}(x,y) & y \in \tau \in \mathrm{T}(x) \end{cases} \tag{2.3.10}$$

2.3.2 Separation of Variables and Collocation

Computing the result of the matrix vector multiplication is in this case equivalent to the approximate evaluation of $K[u_q](x_i)$ for $i = 1, \ldots, q$. Using the original kernel this would mean evaluating

$$K[u_q](x_i) = \int_\Gamma k(x_i, y)u_q(y)\, ds_y$$

for all q different x_i's.

Using $\tilde{k}$ instead of k we get

$$\tilde{K}[u_q](x_i) = \sum_{\pi \in \Pi(x_i)} \int_\pi k(x_i,y)u_q(y)\, ds_y + \sum_{\tau \in \mathrm{T}(x_i)} \int_\tau \tilde{k}(x_i,y)u_q(y)\, ds_y$$

$$= \sum_{\pi \in \Pi(x_i)} \int_\pi k(x_i,y)u_q(y)\, ds_y + \sum_{\iota \in I_p} \sum_{\tau \in \mathrm{T}(x_i)} \varphi_\iota(x_i, y_\tau) \int_\tau \kappa_\iota(y_\tau, y)u_q(y)\, ds_y \tag{2.3.11}$$

For a given u_q we pre-compute and store the terms

$$J_\tau^\iota(u_q) = \int_\tau \kappa_\iota(y_\tau, y)u_q(y)\, ds_y.$$

If we want to evaluate Ku_q for many different u_q's (as when we use an iterative method for solving integral equations) then we will save computational cost by computing and storing $J_\tau^\iota(v_j)$ for all base functions v_j. Then each time we need to evaluate Ku_q for a new u_q we simply get $J_\tau^\iota(u_q) = \sum_{j=1}^q \alpha_j J_\tau^\iota(v_j)$. Similarly we can pre-pre-compute $\int_\pi k(x_i,y)v_j(y)\, ds_y$ for every panel and base function and then get the near field part from a sparse matrix multiplication. Finally we can in this case find all admissible coverings once and for all before the first iteration.

We estimate the computational cost for evaluating $\tilde{K}[u_q]$ once, if we do it repeatedly then we can get a further reduction of the cost by the pre-pre-computing we mentioned above. We set Ξ to be the cost for integrating over one panel; usually this cost is fixed when the grid is refined (if we use a finer grid then we want to use higher precision in the integration).

- **Compute all $J^{\iota}_{\tau}(u_q)$.** The cost for this is the cost for integrating once over every cluster. Since we need to integrate with the same precision we should measure over how big an 'area' (/length/volume ...) we have to integrate. All the clusters at one level in the tree form a partition of Γ, thus this step costs as much as integrating over the area of Γ n times. The cost for integrating over Γ is $N \times \Xi$ and recalling that $n \sim \log N$ we get the cost $\mathcal{O}(N \log N)$.

- **Integrating over the near field.** The number of panels in a covering will be bounded by some low number l. Thus we have to integrate over — at most — $l \times N$ panels. The total cost for this step is seen to be $\mathcal{O}(N)$.

- **Evaluating the φ's and summing up.** For each x_i we have to evaluate $\#I_p \times \#\mathrm{T}(x)$ different φ's. This is clearly $\mathcal{O}(\log N)$. Finding $\mathcal{C}(x_i)$ is $\mathcal{O}(\log N)$ and since the two sums have a constant number of terms, summing it up is $\mathcal{O}(N \log N)$.

Thus the evaluation of $\tilde{K}[u_q]$ is $\mathcal{O}(N \log N)$. This evaluation has to be done repeatedly until the iterative solver converges. If the number of iterations that have to be done is independent of the number of panels then the order of the growth in computational time for the whole process is $\mathcal{O}(N \log N)$.

Note that every time we use the approximate kernel we use clusters that are η-admissible with respect to the point where the potential is evaluated. By (2.3.9) this means that we will get a relative error $\epsilon \sim \eta^p$. If we want to achieve a given relative accuracy of ϵ we see that we should use an expansion of order $p \sim \log_\eta \epsilon$.

2.3.3 Separation of Variables and the Nyström Algorithm

Here we need to evaluate

$$\sum_{j=1}^{q} \omega_j k(x_i, x_j) u_q(x_j) \quad \forall i.$$

The analogy with the particle case is obvious. Evaluating this sum can be interpreted as evaluating the potential at x_i from q point charges of strength $\omega_j u(x_j)$ interacting via the potential k.

If we replace k by $\tilde{k}$ we get

$$\sum_{\tau\in\mathcal{C}(x_i)}\sum_{j:x_j\in\tau}\omega_j\tilde{k}(x_i,x_j)u_q(x_j)=$$

$$\sum_{\tau\in\Pi(x_i)}\sum_{j:x_j\in\tau}\omega_j k(x_i,x_j)u_q(x_j)+\sum_{\tau\in\mathrm{T}(x_i)}\sum_{j:x_j\in\tau}\sum_{\iota\in I_p}\omega_j\varphi_\iota(x_i,y_\tau)\kappa_\iota(y_\tau,x_j)u_q(x_j)=$$

$$\sum_{\tau\in\Pi(x_i)}\sum_{j:x_j\in\tau}\omega_j k(x_j,x_i)u_q(x_j)+\sum_{\tau\in\mathrm{T}(x_i)}\sum_{\iota\in I_p}\varphi_\iota(x_i,y_\tau)\Big[\sum_{j:x_j\in\tau}\omega_j\kappa_\iota(y_\tau,x_j)u_q(x_j)\Big]$$

This is all very similar to the collocation method; we simply approximate the integrals in the collocation method with a quadrature. For the general case the algorithm is still $\mathcal{O}(N\log N)$.

2.3.4 Separation of Variables and Galerkin

For the Galerkin method the matrix element we need to evaluate is given by (2.2.7). To get an $\mathcal{O}(N\log N)$ algorithm we have to assume that the base functions have local support, i.e. they are non-zero only in a few adjacent panels. If this is the case, then most of the terms $\langle v_i,\lambda v_j\rangle$ vanish and this part of the matrix element corresponds to multiplication with a sparse matrix. Techniques for handling sparse matrices are well known and we will concentrate on the second term,

$$\langle v_i,K[u_q]\rangle_{\mathcal{L}^2(\Gamma)}=\int_\Gamma\int_\Gamma v_i(x)k(x,y)u_q(y)\,ds_x\,ds_y.$$

We construct a new η-admissible covering that depends on v_i, not x_i as before. $\mathcal{C}(v_i)$ is the collection of panels and clusters such that every member of the covering is either a panel or a cluster that is η-admissible with respect to *every point in the support of* v_i. If that support is sufficiently localised then there will not be a big difference between this new covering and the old covering given by any point in the support. We define $\tilde{k}$ as before but with the new covering in (2.3.10). This gives us

$$\begin{aligned}
\int_{\Gamma\times}\int_{\Gamma} v_i(x)\tilde{k}(x,y)u_q(y)\,ds_y\,ds_x &= \int_{\mathrm{supp}(v_i)}\int_{\times\mathcal{C}(v_i)} v_i(x)\tilde{k}(x,y)u_q(y)\,ds_y\,ds_x\\
&= \sum_{\pi\in\Pi(v_i)}\int_{\mathrm{supp}(v_i)}\int_{\times\pi} v_i(x)k(x,y)u_q(y)\,ds_y\,ds_x\\
&+ \sum_{\tau\in\mathrm{T}(v_i)}\sum_{\iota\in I_p}\int_{\mathrm{supp}(v_i)}\int_{\times\tau} v_i(x)\varphi_\iota(x,y_\tau)\kappa_\iota(y_\tau,y)u_q(y)\,ds_y\,ds_x\\
&= \sum_{\pi\in\Pi(v_i)}\int_{\mathrm{supp(v_i)}}\int_{\times\pi} v_i(x)k(x,y)u_q(y)\,ds_y\,ds_x
\end{aligned}$$

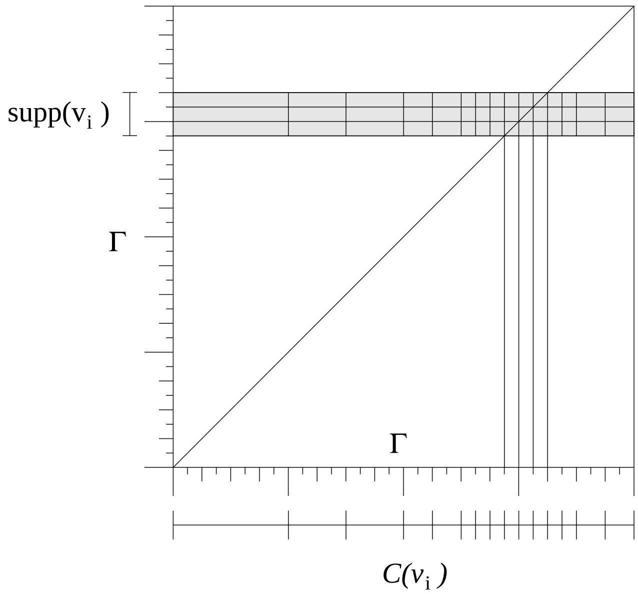

Figure 2.2: Integrationdomain for $\langle v_i, K[u_q]\rangle$ (shaded).

$$+ \sum_{\tau\in\mathrm{T}(v_i)} \sum_{\iota\in I_p} \int_{\mathrm{supp}(v_i)} v_i(x)\varphi_\iota(x, y_\tau)\, ds_x \int_\tau \kappa_\iota(y_\tau, y)u_q(y)\, ds_y$$

Unless the expansion is symmetric (κ and φ have the same form) we must compute two sets of coefficients.

$$J^\iota_\tau(u_q) = \int_\tau \kappa_\iota(y_\tau, y)u_q(y)\, ds_y \quad \text{and} \quad I^\iota_\tau(v_i) = \int_\tau \varphi_\iota(x, y_\tau)v_i(x)\, ds_x$$

We can perform the same analysis of the cost of computing the J^ι_τ's as we did in the collocation case and find that the cost is $\#I_p \times n \times N \times \Xi$ (recall that Ξ is the cost to integrate over a panel). Computing the I's is the same cost, and integrating exactly over the near-field is clearly an $\mathcal{O}(N)$ procedure. Stating that finding $\mathcal{C}(v_i)$ is $\mathcal{O}(\log N)$ (the same cost as finding $\mathcal{C}(x_i)$) we see that this technique gives an $\mathcal{O}(N \log N)$ cost for a Galerkin discretisation.

If we want to evaluate Ku_q for different u_q's, then we can use the technique we mentioned in connection to the collocation method; we pre-pre-compute $J^\iota_\tau(v_j)$ for all v_j's and then we get $J^\iota_\tau(u_q)$ by a sparse matrix multiplication. We can also pre-pre-compute all $\mathcal{C}(v_i)$'s.

2.3.5 How to Speed Up the Algorithm

Usually we expand in a specific function system that allows us to compute the coefficients more efficiently. For some expansions we have a map that gives the coefficients at one meshlevel from the coefficients at a finer level, formally:

$$L : \{J^{\iota}_{\tau'}\}_{\iota \in I_p, \tau' \subseteq \tau} \mapsto J^{\iota}_{\tau}$$

If this is the case then we need to do the numerical integration on the finest mesh level only. Then we get all the other coefficients by algebraic means. One especially simple case is if φ_ι does not depend on y_τ. This leads to an algorithm known as *panel clustering*, see [16, 31]. An example involving a more complicated algebraic transformation is given by the multipole algorithm for Nyström Discretisation, see [30]. Both these algorithms are presented in detail below.

We can also save much work if we use a well-structured cluster tree. The way we have presented the collocation method we have to find $\mathcal{C}(x_i)$ explicitly for every x_i and then compute the sum (2.3.11) using the J^{ι}_{τ}-coefficients. If x_i and x_j are close then most of the elements in $\mathcal{C}(x_i)$ and $\mathcal{C}(x_j)$ are the same. If we have a translation property for κ then we can exploit this and compute the sums for all x_i's jointly with $\mathcal{O}(N)$ work. An example of how this can be done is given in the presentation of the Fast Multipole Algorithm.

2.4 The Fast Multipole Algorithm for Nyström Discretisation

We will start by presenting the algorithm as it is given by Rokhlin in [30]. This article is considered the definition of the algorithm so we should present it. However, after having done this we will demonstrate that for some boundaries it does not work at all and for others we get poor convergence. This can be helped by modifying the algorithm and we will propose a suggestion how this can be done.

For a general description of multipole expansions we refer to subsection 1.3.1. We will study a special case but generalisation to a case with any bounded harmonic kernel is straightforward.

2.4.1 Definition of the Problem

We will use the multipole algorithm to evaluate the integral

$$\Phi(x) = \int_\Gamma \frac{n_y \cdot (x-y)}{2\pi |x-y|^2} u(y)\, ds_y,$$

where Γ is a smooth closed curve in R^2 with outwards pointing unit normal n_y and x can be a point either on or off Γ. This integral is called the double layer potential of u and it arises when discretising a Dirichlet boundary value problem in two dimensions. Using Nyström discretisation with N collocation points evenly spaced along the curve we get an approximation

$$\tilde{\Phi}(x) = \sum_{j=1}^{N} \omega_j \frac{n_{y_j} \cdot (x - y_j)}{2\pi |x - y_j|^2} u(y_j)$$

This is equivalent to evaluating the field from N dipoles, each with strength $u(y_j)\omega_j$ and direction n_{y_j}. Using complex notation as in the particle case we combine these three quantities into a complex dipole moment m_j.

Recall that evaluation of this integral is the time-consuming step in the iterative procedure to solve an integral equation with a double layer kernel.

2.4.2 Definitions

We group the collocation points together in clusters that form a binary tree. The clusters are named τ_m^i where m denotes meshlevel and i denotes a specific cluster at that level. Setting n to be the number of levels we get

$$\tau_m^i = \cup\{\tau_{m+1}^{2i-1}, \tau_{m+1}^{2i}\} = \cup\{\tau_{m+2}^{4i-3}, \tau_{m+2}^{4i-2}, \tau_{m+2}^{4i-1}, \tau_{m+2}^{4i}\} = \ldots = \cup\{\tau_n^j\}_{j=2^{n-m}(i-1)+1}^{2^{n-m}i}$$

To each cluster we associate a centre $y_{\tau_m^i} \in C$ which is defined to be the average position of the two particles in the middle of τ_m^i

$$y_{\tau_m^i} = \frac{y_{2^{n-m}(k-1)+2^{n-m-1}} + y_{2^{n-m}(k-1)+2^{n-m-1}+1}}{2}.$$

We also introduce the following definitions:

Ψ_m^i Denotes the approximation to the field from all dipoles in cluster τ_m^i given by a p-term multipole expansion around $y_{\tau_m^i}$.

Υ_m^i Denotes an analogous approximation to the field from the particles in all clusters that are not contained in τ_m^i or its neighbours.

I_m^i Denotes the *interaction list* of a cluster τ_m^i, i.e. the set of clusters that are children of τ_m^i's parents or its parent's neighbours but are not neighbours to τ_m^i. For all intermediate meshlevels I_m^i will contain three clusters.

2.4.3 The Algorithm

We can divide the algorithm into four steps, just as in the particle case. In Step 1 we compute the multipole expansions around the centres of the clusters on the finest meshlevel. In Step 2 we loop over all the levels, from finest to coarsest, and use the multipole expansions at one level to get the expansions at the next coarser one. In Step 3 we compute all the Υ's by transferring suitable Ψ's. We will end up with Υ^i_{n-1}'s that account for all long-range interactions, and then in Step 4 we transfer these expansions to the particle sites and use direct computation to account for the field from the closest particles.

Step 1

for $i = 1, \ldots, 2^{n-1}$ **do**	Loop over all clusters at level $n-1$.
$\quad\{m_j,\ y_j\}_{j=2i-1}^{2i} \mapsto \Psi^i_{n-1}$	The two dipoles in cluster i give the first multipole expansion.
end do	

Step 2

for $m = n-2, n-3, \ldots, 1$ **do**	Loop over all meshlevels.
$\quad$**for** $i = 1, 2, ..., 2^m$ **do**	Loop over all clusters at level m.
$\qquad\Psi^{2i-1}_{m+1}, \Psi^{2i}_{m+1} \mapsto \Psi^i_m$	Transfer multipole expansions from children.
$\quad$**end do**	
end do	

Step 3

$\Upsilon^i_1 = 0 \quad \forall i$	
for $m = 2, 3, ..., n$ **do**	Loop over all meshlevels.
$\quad$**for** $i = 1, 2, ..., 2^m$ **do**	Loop over all clusters at level m.
$\qquad\Upsilon^{\lfloor i/2 \rfloor}_{m-1},\ \{\Psi^j_m\}_{j:\ \tau^j_m \in \mathrm{I}^i_m} \mapsto \Upsilon^i_m$	Transfer Υ from the parent cluster and add the contributions from the particles in the interactionlist via their multipole expansions.
$\quad$**end do**	
end do	

Step 4

for $i = 1, 2, ..., N$ **do**	Loop over all dipoles.
$\quad\tilde{\Phi}(y_i) = \Upsilon^{\lfloor i/2 \rfloor}_{n-1}(y_i - y_{\tau^i_m})$	Evaluate the far field part of $\tilde{\Phi}$ at y_i.
$\quad$**for** $y_j \in \Pi(y_i)$ **do**	Loop over all close particles.
$\qquad\tilde{\Phi}(y_i) = \tilde{\Phi}(y_i) + \frac{m_j(y_i - y_j)}{2\pi\lvert y_i - y_j\rvert^2}$	Compute interaction with neighbours exactly.
$\quad$**end do**	
end do	

It is easily verified that the algorithm is $\mathcal{O}(N)$.

The way it stands, the algorithm looks very much like the algorithm for particle simulations. We have just specialised it to a case where the particles are evenly spaced along a line, using periodic boundary conditions. The problem arises because we organise the tree of clusters based on *the distance along the curve*. This is not sufficient to invoke lemma (2.3.9). Rokhlin does not mention this problem and for most "nice" domains we do get convergence. The reason is that even though distance is measured along the curve the clusters are usually sufficiently spaced to make them η-admissible for some η. However, it can happen that they are not η-admissible at all and certainly they are not all η-admissible for $\eta = 1/2$. Since we have different η's for different clusters we cannot guarantee any bound on the error. Recall that as η approaches 1 the coefficient in front of the geometric factor in (2.3.9) gets very big. Thus we have two different effects that give poor convergence.

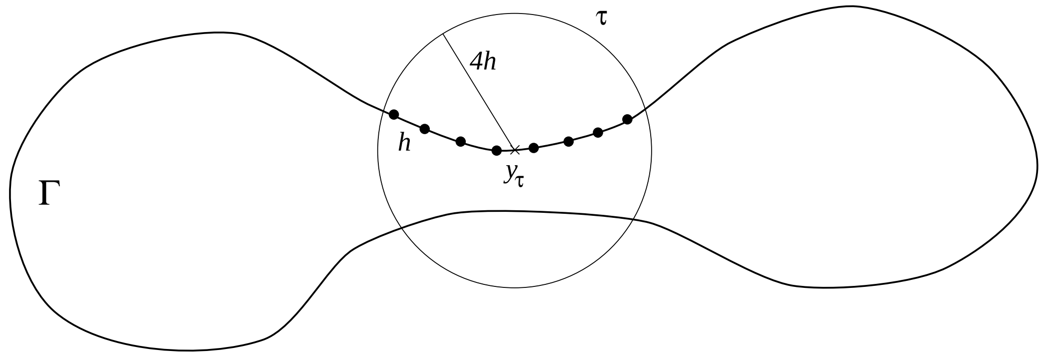

Figure 2.3: Boundary for which the Rokhlin Multipole method does not converge.

An example of a domain where we would get problems is given in figure 2.3. Clearly when we evaluate the interactions between boundary elements in the "waist" of the domain we should use a fine partition. But measuring their distance along the curve, the Rokhlin Fast Multipole Algorithm thinks they are far apart and uses large clusters to account for these interactions. If the cluster marked in the picture is used this clearly gives problems since the expansion around y_τ *does not converge* where it is evaluated.

2.4.4 A Modified Algorithm

To be able to use the error bounds we must know that the clusters we are using when we evaluate the potential at a point are all η-admissible with respect to that point for some η's bounded above by a figure smaller than 1. Similarly we must check that every time we transfer a multipole expansion to a Taylor expansion we obey the criteria in lemma 2 with c's that are bounded below by some $c_0 > 1$. We can do this as in the panel clustering method where we have to compute a whole new covering for every boundary element. This is rather complicated and it gives an $\mathcal{O}(N \log N)$ algorithm. If we want to preserve one of the strengths of the multipole algorithm — that the tree is well structured so we can compute all coefficients in one upward and one downward pass over all the clusters — we have to assume that the domain satisfies some regularity criterion. We suggest the following.

Assume that there exist constants R_0 and α such that every circle $S_R(y)$, centred on the curve and with radius $R < R_0$, contains no other part of the curve than parts of the segments of length αR extending from the centre of the circle.

Given these constants R_0 and α we pick a meshlevel m_0 to be the coarsest level where the length of the clusters (measured along the curve) is smaller than $2\frac{\alpha}{4}R_0$. The size of the clusters at meshlevel m_0 is given by $h2^{n-m_0}$ (where h denotes the spacing between collocation points) so we should pick the smallest m_0 such that

$$m_0 \geq n - \log_2 \frac{2\alpha R_0}{4h}$$

The modification we have to do in the algorithm is that we have to stop the upward pass in Step 2 at level m_0. Then we begin Step 3 by using *all* the clusters on level m_0 to compute the Υ_{m_0+1}'s and then continue the downward pass from that level.

If we pick any cluster with radius R on level m_0 or a finer level, we know that the points where this expansion is used are at least $3R$ away from the cluster's centre, measured along the curve. The smallest circle that can host this line-segment has radius $\frac{3}{\alpha}R$. Thus the cluster will be η-admissible for some $\eta \leq \frac{\alpha}{3} = \eta_0$.

Similarly we check what the smallest c is. When we transfer a multipole expansion to a Taylor expansion we do so between clusters of radii R whose centres are separated $4R$, all measured along the curve. This means that with the usual metric they are placed at least $\frac{4}{\alpha}R$ apart. The extreme case would give $\frac{4}{\alpha}R = (1+c)R$ which gives $\frac{1}{c_0} = \frac{\alpha}{4-\alpha}$. Unless α is smaller than 2 we see that the algorithm becomes useless.

The computational cost for the modified algorithm is higher than for the original one. At level m_0 we have to compute all pairwise interactions so the asymptotic complexity is given by $\mathcal{O}(m_0^2 + N) = \mathcal{O}((\frac{h2^n3}{2\alpha R_0})^2 + N) = \mathcal{O}((\frac{L}{\alpha R_0})^2 + N)$ where L is the total length of the curve. $\frac{L}{R_0}$ is twice the number of circles of radius R_0 we need to cover the curve. We see that the extra cost is proportional to this number squared. For most simple curves it should be sufficient to use a few dozen circles and in this case the extra cost is negligible.

If we cannot find constants R_0 and α there might be other ways to modify the algorithm. Before an investigation about that is started we strongly recommend to consider using the adaptive algorithm mentioned in section 1.3.2. No matter what the boundary looks like this will always give an $\mathcal{O}(N)$ algorithm with a guaranteed convergence rate. It should be possible to use a standard code so what small amount of time might be saved in execution time with a more customised algorithm is probably negligible compared to the cost of developing the new code. Considering how efficient the adaptive algorithm is it might even be cost effective to use it for a “nice” boundary.

2.5 Expanding the Kernel in Panel Clustering

The basic steps in the panel clustering algorithm are the same as in the general separation of variables algorithm. What signifies a panel clustering algorithm are some special properties in the functions we use in expansion 2.3.8.

We want to find an expansion of the form

$$\tilde{k}(x,y) = \sum_{\iota \in I_p} \varphi(x, y_\tau)\kappa_\iota(y). \tag{2.5.12}$$

We will assume that Γ is a surface in R^3 but generalisation to other cases is possible. For vectors $x, y, z, \ldots \in R^3$ and multi-indices $\alpha, \mu, \nu, \ldots \in N^3$ we adopt the following conventions

$$\begin{aligned}
|\alpha| &= \alpha_1 + \alpha_2 + \alpha_3 \\
z^\alpha &= z_1^{\alpha_1} + z_2^{\alpha_2} + z_3^{\alpha_3} \\
\alpha! &= \alpha_1!\alpha_2!\alpha_3! \\
\binom{\nu}{\mu} &= \frac{\nu!}{(\nu-\mu)!\mu!} \\
\sum_{|\nu|=0}^{p} c_\nu &= \sum_{s=0}^{p} \sum_{\nu_1=0}^{s} \sum_{\nu_2=0}^{s-\nu_1} c_{\nu_1,\nu_2,s-\nu_1-\nu_2} \\
\sum_{\nu=0}^{\mu} c_\nu &= \sum_{\nu_1=0}^{\mu_1} \sum_{\nu_2=0}^{\mu_2} \sum_{\nu_3=0}^{\mu_3} c_{\nu_1,\nu_2,\nu_3}
\end{aligned}$$

When we discretised the exterior Dirichlet problem in section 2.1.2 we saw that the Laplace operator $-\triangle$ in three dimensions has the fundamental solution $E(z) = \frac{1}{4\pi||z||}$. In general an elliptic PDE of order $2m$ will have a fundamental solution, or, equivalently, *singularity function* s of form

$$s(z) = \sum_{|\alpha|=t}^{\infty} c_\alpha \frac{z^\alpha}{||y-x||^{s+t}} \tag{2.5.13}$$

where t is even and $s = 3 - 2m$ is the *singularity*. When transforming the PDE to an integral equation we get a kernel of form

$$k(x,y) = \Big(\frac{\partial}{\partial n_x}\Big)^{\beta_1}\Big(\frac{\partial}{\partial n_y}\Big)^{\beta_2} s(x-y) \tag{2.5.14}$$

where β_1 and β_2 are integers such that $\beta_1 + \beta_2 \leq 2m$.

The difference between x and y in $k(x,y)$ is called z, i.e. $z = y - x$. We assume that we can expand $s(y-x)$ around a point z_0 in the following manner

$$s(z) = \sum_{|\nu|=0}^{p} \sum_{\mu=0}^{\nu} \kappa_{\nu,\mu-\nu}(z_0)\gamma_\nu(x)\gamma_{\mu-\nu}(y) + R_{p+1}(z_0, z) \tag{2.5.15}$$

where $\kappa_{\nu,\mu-\nu}$ are the expansion coefficients and γ_ν is the expansion system, usually polynomials. Combining equations (2.5.13), (2.5.14) and (2.5.15) gives us the p-order expansion for the kernel.

Example We again look at the example in section 2.1.2. Recall that (normalised to avoid the factor 4π)

$$s(x-y) = \frac{1}{||x-y||} \quad \text{and} \quad k(x,y) = \frac{\partial}{\partial n_y}\frac{1}{||x-y||}.$$

We set $z = y - x$, Taylor expand around z_0 and then reorder the terms to get

$$\begin{aligned} s(y-x) &= \sum_{|\nu|=0}^{p} \frac{1}{\nu!}\frac{\partial^{|\nu|}}{\partial z^\nu}\frac{1}{||z||}\Big|_{z=z_0}(y-x-z_0)^\nu + R_{p+1}(z_0,z) \\ &= \sum_{|\nu|=0}^{p}\sum_{\mu=0}^{\nu} \kappa_{\nu,\mu-\nu}(z_0) y^\mu x^{\nu-\mu} + R_{p+1}(z_0,z) \end{aligned}$$

Differentiating y^μ and truncating the error term gives the approximate kernel.

The number of terms in the expansion is $(p+1)^6/720 + \mathcal{O}(p^5)$ which means that we cannot use a large p. For $p = 3$ we have 28 terms and for $p = 4$ we have 84. From then on the number of terms grows rapidly. However, in collocation and Nyström discretisation we can group the terms that depend on z_0 and x together so that the number of terms is reduced to $\mathcal{O}(m^3)$.

For a fuller description of expansions like these we refer to Sauter [31].

2.6 About Error Estimates

Using the error bound (2.3.9) and Hölders inequality we trivially get for collocation discretisation

$$
\begin{aligned}
|[Ku_q](x) - [\tilde{K}u_q](x)| &= \Big| \int_{\mathrm{T}(x)} (k(x,y) - \tilde{k}(x,y)) u_q(y)\, ds_y \Big| \\
&\leq \begin{cases} ||u_q||_\infty \; C_1 (C_2\eta)^p \int_{\mathrm{T}(x)} |k(x,y)|\, ds_y \\[2ex] ||u_q||_2 \; C_1 (C_2\eta)^p \left(\int_{\mathrm{T}(x)} |k(x,y)|^2\, ds_y \right)^{1/2} \end{cases}
\end{aligned}
$$

In order to get estimates for the remaining integrals we have to know what kernel, what panels and what expansion system we are using. Usually we are given a specific expansion system and just have to choose certain parameters. An error estimate typically find optimal choices for the parameters and then give the result as a function of h, the coarseness of the boundary elements. Error estimates for panel clustering are given for instance by Hackbusch [16] and Sauter [31].

Chapter 3

An Implementation

3.1 Introduction

As an example of how to implement a separation of variables algorithm we show how to use the Fast Multipole Algorithm to simulate the dynamics of N dipoles in two dimensions. A requirement of the Fast Multipole Algorithm in two dimensions is that we can represent the potential as an analytic function. Thus we have to use the two dimensional dipole potential which limits the applicability of the model. Two dimensional dipoles are not commonly used so we have derived a few facts about them that we present in section 3.2.2.

The general structure of the program is the one presented in section 1.3.2. That algorithm is based on the one suggested by Greengard [14]. A few details in the implementation were inspired by Allen and Tildesley [2] but since this is a new algorithm and the published research articles do not go into detail most of the implementation had to be worked out from scratch. This means that other codes might be more efficient and that our results can possibly be improved. Indeed, we suggest below a few improvements that could have been implemented if time had permitted.

3.2 The Model

3.2.1 Definition of the Problem

We will place N particles in a box with unit side length. The box is considered a part of a rectangular lattice consisting of an infinite number of identical copies of the same box spread over the whole plane. The particles have a long-range interaction governed by a two dimensional dipole potential. The short-range interaction is governed by a Lennard-Jones 6-12 potential, (1.1.1).

3.2.2 Facts About Two-Dimensional Dipoles

The two-dimensional dipole is obtained as the limiting charge from two oppositely charged point charges whose positions tend to the same point and whose charges tend to infinity in such a manner that the potential tends to a finite value. For a dipole at the origin with dipole moment $\mathbf{m}$ we get

$$V(\mathbf{r}) = \lim_{\substack{\mathbf{d}\to 0\\ q\mathbf{d}\to\mathbf{m}}} \left[\frac{q}{2\pi\epsilon_0}\log\frac{1}{||\mathbf{r}-\mathbf{d}||} - \frac{q}{2\pi\epsilon_0}\log\frac{1}{||\mathbf{r}-(-\mathbf{d})||}\right] = \frac{\mathbf{m}\cdot\mathbf{r}}{\pi\epsilon_0 r^2}$$

where $r = ||\mathbf{r}||$. This is the real part of the analytic potential

$$W(z) = \frac{m}{\pi\epsilon_0 z}$$

where $m = m_x + im_y$.

The electric field is obtained by differentiation

$$\mathbf{E}(\mathbf{r}) = -\nabla V = -\frac{\mathbf{m}}{\pi\epsilon_0 r^2} + \frac{2(\mathbf{r}\cdot\mathbf{m})\mathbf{r}}{\pi\epsilon_0 r^4}$$

or, with complex notation

$$E = E_x + iE_y = -\overline{W'} = \frac{\overline{m}}{\pi\epsilon_0\overline{z}^2}.$$

Since E is not an analytic function we will work with $\overline{E}$ instead.

The torque on a dipole with moment $\mathbf{m}$ in an electric field $\mathbf{E}$ is given by $\tau = \mathbf{m}\times\mathbf{E}$. (We consider the vectors as lying in a plane embedded in R^3.) With this formula we can compute the torque exerted on a dipole α by a dipole β. If $\mathbf{r}$ denotes the vector from β to α we get

$$\tau_\alpha = \mathbf{m}_\alpha \times \mathbf{E}_\beta = -\frac{\mathbf{m}_\alpha\times\mathbf{m}_\beta}{\pi\epsilon_0 r^2} + \frac{2(\mathbf{m}_\beta\cdot\mathbf{r})\ \mathbf{m}_\alpha\times\mathbf{r}}{\pi\epsilon_0 r^4}$$

Remember that in two dimensions the torque is a scalar; it is given by the component of the vector torque that is orthogonal to the plane that contains the dipoles. Using complex notation we get the torque as the negative of the imaginary part of $m\overline{E}$. This gives

$$\tau = -\Im(m_\alpha\overline{E}_\beta) = \Im(\frac{\overline{m}_\alpha\overline{m}_\beta}{\pi\epsilon_0\overline{z}^2}) \tag{3.2.1}$$

Finally the force on a dipole in an electric field is given by $\mathbf{F} = [D\mathbf{E}]\mathbf{m}$ where $D\mathbf{E}$ denotes the Jacobian derivative matrix of $\mathbf{E}$. This expression gets very complicated when converted to vector notation. In complex notation on the other hand we simply get

$$F_\alpha = \overline{m_\alpha\overline{E}'_\beta} = -\frac{2\overline{m}_\alpha\overline{m}_\beta}{\pi\epsilon_0\overline{z}^3} \tag{3.2.2}$$

3.3 Translation Formulæ

We need to perform four different translations. We can get the formulæ for these by substituting in the lemmas in section 1.3.2. We work with $\overline{E}$ instead of of the potential, this gives us the force and torque directly without a numerical differentiation.

Individual Particle to Multipole

The electric field from a dipole with moment m, located at y is given by $\overline{E(z)} = \frac{m}{\pi\epsilon_0(z-y)^2}$. We can transfer this to a multipole expansion around y_0 given by

$$\overline{E(z)} = \sum_{k=2}^{\infty} a_k \frac{1}{(z-y_0)^k} \qquad \text{where} \qquad a_k = \frac{m}{\pi\epsilon_0}(k-1)(y-y_0)^{k-2} \tag{3.3.3}$$

Multipole to Multipole

We can transfer expansion (3.3.3) to an expansion around y_1

$$\overline{E(z)} = \sum_{i=2}^{\infty} \tilde{a}_i \frac{1}{(z-y_1)^i} \qquad \text{where} \qquad \tilde{a}_i = \sum_{k=2}^{i} a_k \binom{i-1}{k-1}(y_0-y_1)^{i-k} \tag{3.3.4}$$

This is easily verified. Set $\zeta = z - y_1$, $\eta = y_0 - y_1$ and $\gamma = \eta/\zeta$. Then we write E as a function of γ and if $|\zeta| > |\eta|$ we can make a MacLaurin expansion.

$$\begin{aligned}
\overline{E(z)} &= \sum_{k=2}^{\infty} a_k \frac{1}{(z-y_0)^k} = \sum_{k=2}^{\infty} a_k \frac{1}{((z-y_1)-(y_0-y_1))^k} \\
&= \sum_{k=2}^{\infty} a_k \frac{1}{\zeta^k}\frac{1}{(1-\gamma)^k} = \sum_{k=2}^{\infty} a_k \frac{1}{\zeta^k}\sum_{j=0}^{\infty}\frac{\gamma^j}{j!}\frac{\partial^j}{\partial\gamma^j}\left[\frac{1}{(1-\gamma)^k}\right]\Bigg|_{\gamma=0} \\
&= \sum_{k=2}^{\infty} a_k \frac{1}{\zeta^k}\sum_{j=0}^{\infty}\frac{\gamma^j}{j!}\frac{(k+j-1)!}{(k-1)!} = \sum_{k=2}^{\infty}\sum_{j=0}^{\infty}\frac{a_k}{\zeta^k}\frac{\eta^j}{\zeta^j}\binom{k+j-1}{k-1} \\
&= \sum_{i=2}^{\infty}\sum_{k=2}^{i} a_k \frac{\eta^{i-k}}{\zeta^i}\binom{i-1}{k-1} = \sum_{i=2}^{\infty}\frac{1}{(z-y_1)^i}\sum_{k=2}^{i}\binom{i-1}{k-1}(y_0-y_1)^{i-k}
\end{aligned}$$

For error bounds we refer to Lemma 2. Note that equation (3.3.3) is just a special case of (3.3.4).

Multipole to Taylor

Equation (3.3.3) can also be transferred to a Taylor expansion around y_1.

$$\overline{E(z)} = \sum_{i=0}^{\infty} b_i (z-y_1)^i \qquad \text{where} \qquad b_i = (-1)^i \sum_{k=2}^{\infty} a_k \binom{k+i-1}{k-1}\frac{1}{(y_0-y_1)^{k+i}} \tag{3.3.5}$$

This can be verified in a manner similar to equation (3.3.4). With the same η and ζ as before we get

$$\overline{E(z)} = \sum_{k=2}^{\infty} a_k \frac{1}{(\zeta-\eta)^k} = \sum_{k=2}^{\infty} a_k \sum_{j=0}^{\infty} \frac{\zeta^j}{j!} \frac{\partial^j}{\partial \zeta^j} \left[\frac{1}{(\zeta-\eta)^k} \right] \Bigg|_{\zeta=0} = \cdots$$

Taylor to Taylor

Finally we show how to transfer equation (3.3.5) to a Taylor expansion around y_2

$$\overline{E(z)} = \sum_{j=0}^{\infty} \tilde{b}_j (z-y_2)^j \qquad \text{where} \qquad \tilde{b}_j = \sum_{i=j}^{\infty} b_i \binom{i}{j} (y_2-y_1)^{i-j} \tag{3.3.6}$$

This follows directly from the binomial theorem. Now we set $\zeta = z - y_2$ and $\eta = y_2 - y_1$ to get

$$\overline{E(z)} = \sum_{i=0}^{\infty} b_i (\zeta+\eta)^i = \sum_{i=0}^{\infty} b_i \sum_{j=0}^{i} \binom{i}{j} \zeta^j \eta^{i-j} = \sum_{j=0}^{\infty} (z-y_2)^j \sum_{i=j}^{\infty} b_i \binom{i}{j} (y_2-y_1)^{i-j}$$

Note that the sign on η is the opposite compared to the previous examples !

3.4 Detailed Description of the Program

The program that we describe in this section has been implemented in GNU C++. We will not describe it in such detail that the language matters. However, we do need some rather complicated data structures so we strongly recommend that a language which has a natural way to support pointers and fields is used. For the simple program we are considering there is no need for recursive functions and we have not used them. In an implementation of the adaptive multipole algorithm recursive routines are needed so it does make sense to write everything in a language which supports recursive programming should one ever want to extend the program.

3.4.1 Data Structures

Apart from a long list of various scalars and arrays we need a few more complicated structures as well. We use the divisioning of the computational box into clusters that was presented in section 1.3.2. We let n denote the number of mesh-levels and use m as an index denoting one of the levels. On each level we number the clusters with two integers marking their x and y coordinates. We will use the letters i and j for these indices.

We will pre-compute the coordinates for the centres of all the clusters. To mark a centre c_m^{ij} we need three coordinates. The subscript marks the mesh-level and we have $0 \leq m \leq n$. The range of the superscripts depends on the mesh-level, we have $0 \leq i, j \leq 2^m - 1$. This means that we cannot use the standard multidimensional fields provided by most languages (unless we have so much memory that we can accept that most of the entries vanish). One possible solution is to define c as an $(n+1)$-dimensional array of pointers, where the m'th entry points to a $2^m \times 2^m$ matrix containing the coordinates for the cluster centres. We need to store $(n+1)+\sum_{m=0}^{n} 2^m = (n+1)+2^{n+1}-1$ pointers. We also need to store the $\sum_{m=0}^{n} 4^m = (4^{n+1}-1)/3$ complex numbers that mark the centres.

A similar construction can be used to represent the local multipole expansion (corresponding to Ψ in section 1.3.2). To each cluster we associate a $(p+1)$-term multipole expansion. We call the k'th coefficient $a^{ij}_{m,k}$ where m, i and j have the same ranges as they did for c^{ij}_m and the range for k is $0 \leq k \leq p$. This can be represented in the same way as c^{ij}_m, except that the entries in the $2^m \times 2^m$-matrices are pointers, pointing to the 0'th element in the $p+1$-multipole expansion for the cluster τ^{ij}_m. We need to store roughly $4^{n+1}/3$ pointers and $(p+1)4^{n+1}/3$ complex numbers. Analogously we store the coefficients $b^{ij}_{m,k}$ as the local Taylor expansions, corresponding to Υ in section 1.3.2.

In Steps 1 and 4 we need to access the particles that are contained in a given panel. Thus we need to use some data structure that keeps track of what particles are contained in which panel. This is not entirely straightforward since the number of particles contained in any one panel may change drastically during the simulation. We have chosen to use a doubly linked list to keep track of the particles. We will not present a detailed description of how we implemented this; the interested reader is referred to the source code in appendix D.

Before we enter the main loop of the program we pre-compute the last sum in equation (1.3.8). Define $\sigma(q) \equiv \sum_{z \in S} \frac{1}{z^q}$. We use the summation order we described in connection with the Ewald sum so $\sigma(2) = 0$.

We use the letters α and β as integer indices for the N particles. These are used to access N-arrays containing information about the positions, velocities, forces, angular positions, angular velocities and torques associated with each particle.

All the variables we use are summarised in the following table.

N	Number of particles.
p	Number of terms in the asymptotic expansion.
n	Number of meshlevels.
m	Index that marks a meshlevel.
i, j	Integers that mark x and y coordinates for a cluster.
c^{ij}_m	Complex coordinate for the centre of the cluster τ^{ij}_m
$a^{ij}_{m,k}$	The complex k'th multipole coefficient in the expansion for $\tau^{i,j}_m$

$b^{ij}_{m,k}$	The complex k'th Taylor coefficient in the expansion for $\tau^{i,j}_m$
α, β	Integers that mark particles.
m_α	Complex dipole moment of particle α.
r_α, v_α	The complex position and velocity of particle α.
$\phi_\alpha, \omega_\alpha$	The real angular position and velocity of particle α.
F_α, τ_α	The complex force and torque on particle α.
$\sigma(q)$	$= \sum_{z \in S} \frac{1}{z^q}$ with S as in equation (1.3.8), complex numbers.

3.4.2 The Program

The program has the following general structure.

Initialise the pointers.
Compute all binomial coefficients.
Compute all $\sigma(q)$'s.
Set all the initial velocities and positions. (Usually read them from a file.)
for $[t = 0, dt, 2dt, \ldots, \text{nstep} \times dt]$ **do**
  MULTIPOLE: $\{r_\alpha, \phi_\alpha\}_{\alpha=0}^{N-1} \mapsto \{F_\alpha, \tau_\alpha\}_{\alpha=0}^{N-1}$
  VERLETPOSITION: $\{r_\alpha|_t, v_\alpha|_{t-dt/2}, F_\alpha|_t\} \quad \mapsto \{r_\alpha|_{t+dt}, v_\alpha|_{t+dt/2}\} \quad \forall\alpha$
  VERLETANGLES: $\{\phi_\alpha|_t, \omega_\alpha|_{t-dt/2}, \tau_\alpha|_t\} \mapsto \{\phi_\alpha|_{t+dt}, \omega_\alpha|_{t+dt/2}\} \quad \forall\alpha$
end do
Save data.

Where 'VERLETPOSITION' refers to time-stepping of the particle positions and velocities with the Verlet algorithm. This very simple time-stepping algorithm is presented in [2] but the source code in appendix D should be self-explanatory. We let v_α represent the velocity between the two time-steps which makes the discretisation error $\mathcal{O}(dt^2)$. Similarly 'VERLETANGLES' refers to time-stepping of angular positions and velocities.

The Multipole routine we are using is the one outlined in section 1.3.2. The source code should be quite readable if we explain a few technical tricks.

Finding the Neighbours of a Panel

The reason why we choose to enumerate the panels with their x- and y-coordinates is that with this notation it is very simple to find the neighbours. Cluster τ^{ij}_m has the following neighbours

$$\tau_m^{(i-1)(j-1)}, \quad \tau_m^{i(j-1)}, \quad \tau_m^{(i+1)(j-1)}, \quad \tau_m^{(i-1)j}, \ldots$$

The computer program stores the pairs

$$\{(\mu_l, \nu_l)\}_{l=0}^{7} = (-1,-1), \quad (0,-1), \ldots, (1,1)$$

and then the neighbours are given by $\{\tau_m^{(i+\mu_l)(j+\nu_l)}\}_{l=0}^7$. If the cluster is close to the boundary then we need to find the relevant cluster in an image of the whole box. Remembering that we are essentially working on a torus with cyclic coordinates we realise that the proper cluster is found by just performing the additions modulo 2^m.

In a similar manner we find the clusters in the interaction list of τ_m^{ij}. This time we have to use a set of 27 relative coordinates. These coordinates depend on the location of the cluster in its parent, thus we have to keep track of four lists of 27 pairs.

Evaluating Powers of Complex Numbers

Look at the translation formula (3.3.4). When evaluating this we have to compute the term $(y_0 - y_1)^{i-k}$ for $k = 2, 3, ..., i$. In order to avoid computing high powers in a situation like this we compute the term containing the lowest power first, then we loop over the terms towards higher powers and at each step we just multiply the result from the previous calculation with the base.

Evaluating the Forces and Torques

We use formulæ (3.2.1) and (3.2.2) to evaluate all pairwise interactions between particles in the same or adjacent panels. In practise we loop over β for a fixed α. In this loop we just add up the contributions $\frac{m_\beta}{(r_\beta - r_\alpha)^2}$ and $\frac{m_\beta}{(r_\beta - r_\alpha)^3}$ respectively, then we multiply by m_α and the constants, and then conjugate after the loop. It would be even more efficient to include π and ϵ_0 in the moments (i.e. work with dimensionless quantities).

As we have mentioned before the program keeps track of which particles belong to which panel via a linked list. The machinery is kept hidden to the force routine — in Step 4 we just ask the linked list handler to give us, one by one, the particles that are contained in a given panel. We can also ask for all the neighbouring particles to a given particle.

A further improvement would be achieved if we used Newton's second law and computed all pairwise interactions just once. This could be achieved by looping over just four of the neighbours around each panel. Unfortunately time did not allow us to implement this improvement to the code.

3.5 Checking the Program

Testing if the program works correctly is not trivial. Just computing the force between two given particles in the computational box involves computing the interactions between one of the particles and an infinite number of images of the other particle (it is easily seen that the interaction between the particle and the images of itself vanishes). However, we can place the particles very close to each other and make the interaction with the images negligible. In order to be able to check that the force between any two particles is correctly computed we wrote a separate program in Matlab that gives an estimate to the infinite sum. The force from the program proved to converge to the value given by Matlab as p was increased.

To test that the program gave correct results for the force from many particles we compared the result from a specific set-up with the result from a direct computation of all N^2 interactions. The program that did the direct computation did not account for interactions with the image boxes which resulted in a small deviation between the two results. By clustering all the particles close together, the interaction with the images was minimised and the deviation was reduced to a minimum of about 1 %.

In a full scale MD simulation one should check that the program preserves physical quantities such as total energy (potential + translational kinetic + rotational kinetic). One can also check that various statistics (velocity distributions for instance) settle down to a value predicted by statistical physics as we reach the equilibrium state. See Allen and Tildesley [2]. Time has not allowed such tests to be performed.

3.6 Efficiency Tests

Two series of test runs have been performed. In the first we investigated how the CPU-time is divided between the different steps in the Fast Multipole Algorithm. We also did tests to see how the value of p affects the computational time. In the second series we compared the computational times between the Fast Multipole Algorithm and a program that computes all $\frac{N(N-1)}{2}$ pairwise interactions directly.

In Step 4 we compute all pairwise interactions between particles in adjacent panels. Thus we expect

$$T_4 \sim \sum_{\pi} N_{\pi}^2 \tag{3.6.7}$$

where T_4 is the time spent in Step 4 and N_π is the number of particles in panel π. This clearly shows that T_4 depends on the distribution of the particles, and is minimised for a uniform distribution. To get realistic results we have to run the tests on distributions that resemble the equilibrium state. It turned out that after some 50 timesteps fluctuations in the computation time for Step 4 had settled down. In all the tests we quote, we let the simulation run for 50 timesteps before we measured the times. (For the 16000 particle simulation we used just 20 timesteps.)

The program was run on a SUN SPARC 2000 and the quoted CPU times refer to time in user mode.

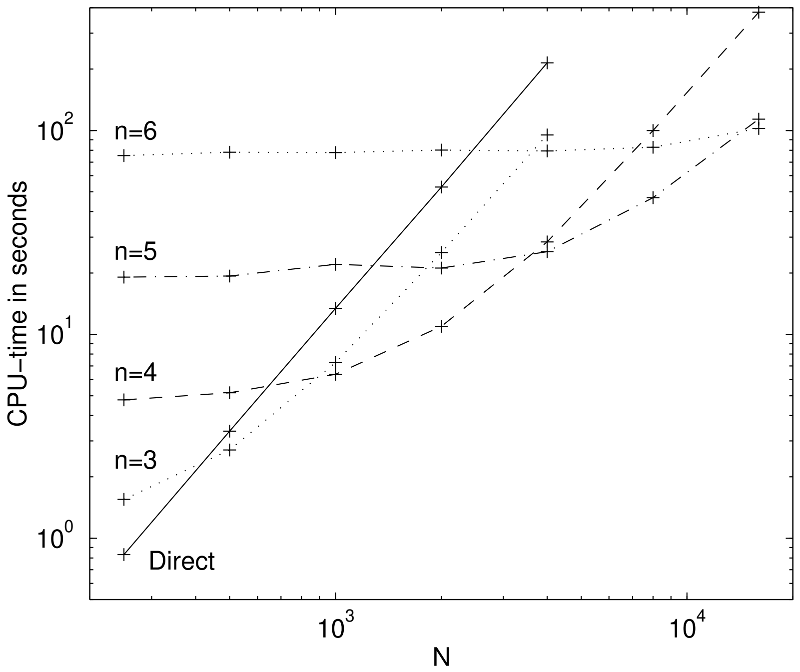

Figure 3.1: Computational time for the Fast Multipole Algorithm for different n and N. Note that we use a log-log scale.

3.6.1 Measurement of Time Spent in Different Steps

Since the aim here was to investigate how the time is distributed between the different steps in the algorithm, we did not use an optimised version of the program to save memory.

The first thing we did was to measure how long time the program spent in the different steps for different n and N. We did not need a high accuracy in these computations so we set $p = 7$. A table with the full data set is given in appendix C. Figure 3.1 shows how the computational time for the whole program depends on n and N.

The data in appendix C shows that the time spent in Steps 2 and 3 does not depend at all on N, as was expected since in these steps we just transfer the expansions between the different mesh-levels. Furthermore we found that the time spent in Steps 1 and 2 is completely negligible. Now we can explain the curves in figure 3.1. For small N, almost all time is spent in Step 3. The fact that the lines corresponding to different n are evenly spaced for small N in the log-log diagram reflects the fact that $T_3 \sim 4^n$. As N is increased (for fixed n), T_4 starts growing. For large N, the total computational time is dominated by T_4 and is seen to grow quadratically (note how the $n = 3$ and $n = 4$ lines asymptotically have the same slope as the direct computation line).

Our task is to find a function $n(N)$ that minimises $T_3 + T_4 \approx T_{\text{Total}}$. If we use this n then the total time for the multipole algorithm with N particles will be proportional to N, as can be seen in the graph. Note that this function depends on p; if a larger p is used then T_3 will increase rapidly while T_4 is virtually unchanged. Using data from the test run with $p = 10$ in the next section (we usually want higher precision than $p = 7$ can offer) we find the following numerical values for $n(N)$.

N	n	$\log_4 N$
250	2	3.98
500	3	4.48
1000	3	4.98
2000	4	5.48
3000	4	5.78
4000	4	5.98
8000	5	6.48
16000	5	6.98

Empirically we find that we should choose n so that $(\log_4 N) - 2 \leq n \leq (\log_4 N) - 1$ for $p = 10$. This means that on average we have between 4 and 16 particles in each panel. Remember that we can reduce the cost of Step 4 by some 30% with a more efficient implementation, which would suggest a slightly smaller value for n, but the difference is negligible.

To check the p dependence we look at the case when $n = 4$ and $N = 2000$. The values for T_3, T_4 and T_{Total} for different p are given in appendix C. As expected, we have that $T_3 \sim p^2$ for large p (the dependence seems linear for small p). It is clear from the results that if we want $p \geq 20$ then we should use $n = 3$ instead of $n = 4$ in this case.

3.6.2 Comparison between the Multipole Algorithm and Direct Computation

It is hard to make a fair comparison between calculating all pairwise interactions directly and using some approximate method. When we do the direct computation the result is accurate up to machine error and if we want to achieve the same accuracy with an approximate method it would be very expensive. In our comparison we have set $p = 10$ which gives a relative accuracy of about 0.001. Another problem is that we cannot account for all interactions with image charges with the direct computation. However, the cost for doing this with the multipole algorithm is very small so we can say that we compare the algorithm for the free particle case. Finally we have to choose if we should compute the Lennard-Jones potential when we compute all the pairwise interactions. In a real simulation one would use a spherical

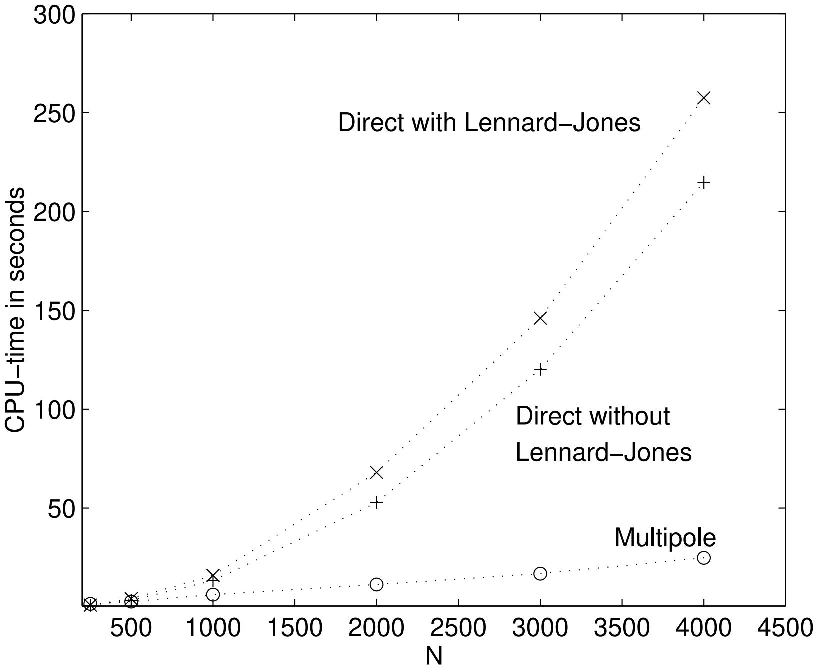

Figure 3.2: Computational time for direct computation vs the Fast Multipole Algorithm with $p = 10$.

cut-off for this interaction and this cost would not grow as N^2. We present computational times for two cases; in one we compute both parts of the potential between all particles; in the other we do not compute the Lennard-Jonas potential at all. The running time for a "real" program should lie somewhere between the running times for these programs.

Tests were run where the number of particles ranged from 250 to 4000 (up to 16000 for the multipole algorithm). All the CPU times are presented in appendix C. Figures 3.2 and 3.3 summarise the data. By measuring the slopes in the log-log diagram we verify that the growth is $\mathcal{O}(N)$ for the fast multipole method and $\mathcal{O}(N^2)$ for direct computation. The break-even point is about 300 when the direct computation is made with the Lennard-Jones potential and about 450 when it is done without it.

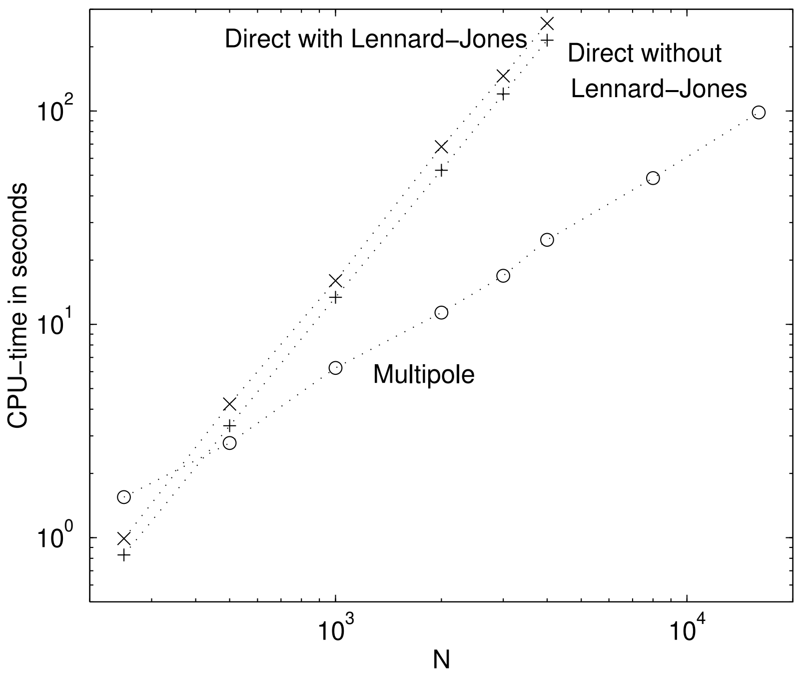

Figure 3.3: Computational time for direct computation vs the Fast Multipole Algorithm with $p = 10$.

Appendix A

Examples of Particle Simulations

The following table is adapted from Hockney, [20].

The length is given in meters, the time-step in seconds. The number of particles refers to the number of particles in a box with side length L.

Example	Computer particles	Particle attributes	"Real" N	N in model	L	dt
Correlated Systems						
Covalent liquids	Atoms or molecules	interaction parameters, mass, force, position, velocity	10^5	10^3	10^{-8}	10^{-12}
Ionic liquids	Charged ions	Charge, mass, force, position, velocity, radius	10^5	10^3	10^{-8}	10^{-12}
Stellar clusters	Stars	Mass, position, velocity, force	10^2 — 10^3	10^2 — 10^3	10^{17}	10^{15}
Galaxy clusters	Galaxies	Mass, position, velocity, force	10^4 — 10^5	10^4 — 10^5	10^{23}	10^{17}
Systems with long "binary collision time"						
Collisionless plasmas	"Superparticle" $\sim 10^8$ electrons or ions	Charge, mass, position, velocity, radius	10^9	$< 10^5$	10^{-5} — 10^{-2}	10^{-9} — 10^{-12}
Galaxies	"Superstar" $\sim 10^6$ stars	Mass, position, velocity, radius	10^{10} — 10^{11}	$< 10^5$	10^{21}	10^{13}
Other examples						
Semiconductor devices (diffusion model)	"Superparticle" $\sim 10^4$ holes or electrons	Charge, position	10^{19}	$< 10^5$	10^{-6}	10^{-9}
Inviscid, incompressible fluids	Vortex elements	Vorticity, position	∞	$< 10^5$	10^{-3} — 10^6	10^{-3} — 10^5

Appendix B

The Geometric Factor in the Fast Multipole Algorithm

In 1983, Rokhlin published an article, [30], where he introduced multipole expansions for fast evaluation of the matrix/vector multiplication in GCR. He claimed that the geometric factor in the error estimate is $1/2$ but as demonstrated in section 2.4.3 that is not correct; sometimes the solution does not even converge.

Then in 1987, Greengard published an article where he presented a method similar to Rokhlin's but applied to evaluation of potential fields [15]. Just looking at the error estimate for the rate of convergence he repeated the claim that the geometric factor is $1/2$. In his PhD thesis [14] from 1988 this was corrected; here he took the error introduced when transferring a multipole expansion to a Taylor expansion into account and arrived at the factor $\sqrt{2}/3$. Realising that this error became unacceptable for the three dimensional case Greengard also modified the algorithm in three dimensions so that the geometric factor was reduced from $\frac{\sqrt{3}}{4-\sqrt{3}}$ to $\frac{\sqrt{3}}{6-\sqrt{3}}$.

In 1994, Petersen *et al* [26] verified with a careful error estimate the factor given for the two dimensional case by Greengard in 1988. In the following paper from Petersen *et al* where they studied the three dimensional algorithm [27] they used the original version of the algorithm and showed that the error was $3/4$ instead of $\frac{\sqrt{3}}{4-\sqrt{3}}$. This analysis is to common knowledge correct but it is done for another version of the algorithm than the one Greengard recommends which explains why they report different geometric factors.

Ding *et al* [10] report that numerical calculations indicate a numerical factor of about $1/2$ which is consistent with the reported theoretical values.

Appendix C

Data From Test Runs

The program was run on a SUN SPARC 2000. All times are given in seconds and refer to time spent in user more.

C.1 Internal Time Budget

The following data were obtained while running a non-optimised multipole algorithm with $p = 7$.

$N = 250$	$n = 3$	$n = 4$	$n = 5$	$n = 6$
T_1	0.015	0.027	0.017	0.065
T_2	0.007	0.026	0.111	0.423
T_3	1.156	4.631	19.9	74.8
T_4	0.375	0.096	0.025	0.024
T_{Total}	1.55	4.77	19.1	75.3

$N = 500$	$n = 3$	$n = 4$	$n = 5$	$n = 6$
T_1	0.033	0.035	0.043	0.083
T_2	0.008	0.027	0.110	0.458
T_3	1.17	4.73	19.04	77.7
T_4	1.501	0.374	0.11	0.036
T_{Total}	2.71	5.17	19.32	78.2

$N = 1000$	$n = 3$	$n = 4$	$n = 5$	$n = 6$
T_1	0.060	0.063	0.017	0.108
T_2	0.007	0.026	0.108	0.465
T_3	1.157	4.78	21.45	77.3
T_4	6.03	1.49	0.395	0.104
T_{Total}	7.27	6.37	22.05	78

$N = 2000$	$n = 3$	$n = 4$	$n = 5$	$n = 6$
T_1	0.122	0.125	0.132	0.172
T_2	0.007	0.027	0.106	0.428
T_3	1.145	4.76	19.36	79.07
T_4	23.41	6.06	1.56	0.445
T_{Total}	25.2	10.95	21.15	80.1

$N = 4000$	$n = 3$	$n = 4$	$n = 5$	$n = 6$
T_1	0.233	0.237	0.248	0.295
T_2	0.007	0.027	0.110	0.44
T_3	1.17	4.83	19.05	77.1
T_4	94	23.35	6.02	1.54
T_{Total}	95	28.41	25.44	79.42

$N = 8000$	$n = 3$	$n = 4$	$n = 5$	$n = 6$
T_1	-	0.50	0.491	0.526
T_2	-	0.026	0.106	0.44
T_3	-	4.81	19.46	76.0
T_4	-	94.5	26.8	5.91
T_{Total}	-	100	46.8	82.8

$N = 16000$	$n = 3$	$n = 4$	$n = 5$	$n = 6$
T_1	-	0.90	0.905	0.99
T_2	-	0.026	0.110	0.45
T_3	-	4.68	18.91	76.66
T_4	-	374.5	93.64	24.28
T_{Total}	-	380.2	113.6	102.4

C.2 Comparison between the Fast Multipole Algorithm and Direct Computation

The multipole program was optimised and used $p = 10$.

N	Direct with Lennard-Jones	Direct without Lennard-Jones	Multipole	
			T	n
250	0.99	0.83	1.55	2
500	4.23	3.35	2.78	3
1000	16.0	13.4	6.25	3
2000	68.0	52.8	11.35	3
3000	146.1	120.2	16.9	4
4000	257.5	214.8	24.9	4
8000	-	-	48.5	5
16000	-	-	98.4	5

Appendix D

The C Source Code

```
#include <stdio.h>
#include <Complex.h>            // from GNU C++ Library
#include <math.h>               // from GNU C++ Library

const int NPARTICLES = 7;
const int n = 4;                // n = No of levels - 1
const int sqrtnpanels = (1<<n); // sqrtnpanels = 2^n, often used quantity
const int p = 7;                // p is the order of the multipoleexpansion
const int nstep = 1000          // nstep = number of timesteps
const float dt = 0.0002;        // time-step size
const float pi = 3.141527;
const float eps0 = 1;           // $\epsilon_{0}$
const float eps = 1;            // Lennard-Jones parameter
const float sigma = 0.007;      // Lennard-Jonas parameter
const float sigma6 = sigma*sigma*sigma*sigma*sigma*sigma;
Complex*** a[n+1];              // multipolecoeff for conj of E-field
Complex*** b[n+1];              // taylorexp for conj of E-field
Complex**  c[n+1];              // position for centre of box
// The following arrays are just auxillaries to store the expansions.
Complex** aRows[(1<<(n+1))-1];
Complex** bRows[(1<<(n+1))-1];
Complex*  cRows[(1<<(n+1))-1];
Complex* aColumns[( ( 1<<( (n+1)<<1 ) ) - 1)/3];
Complex* bColumns[( ( 1<<( (n+1)<<1 ) ) - 1)/3];
Complex  cColumns[( ( 1<<( (n+1)<<1 ) ) - 1)/3];
Complex mpols[(p+1)*( ( 1<<( (n+1)<<1 ) )-1)/3];
Complex taylors[(p+1)*( ( 1<<( (n+1)<<1 ) )-1)/3];

void setpointers(void);

void placeinpanels(Complex r[]);

void setcentres(void);

void print_rvF(Complex r[], Complex v[], Complex F[]);

void print_phiomegatau(float phi[], float omega[], float tau[]);

void setbcoeff(int bcoeff[2*p+1][2*p+1]);

void setsigma(Complex Sigma[2*p+1]);

void printcomplex(Complex z);

void placeinpanels(Complex r[]);
```

```
void stats(void);

void write_float_to_file(char *name, float f[], int nelements);
void write_complex_to_file(char *name, Complex f[], int nelements);
void read_float_from_file(char *name, float f[], int nelements);
void read_complex_from_file(char *name, Complex f[], int nelements);

int firstalfa(int i, int j);
int nextalfa(void);
int firstbeta(void);
int nextbeta(void);
int firstneighbourpart(void);
int nextneighbourpart(void);

main()
{
  int m;            // denotes a meshlevel
  int i,j,ii,jj;    // denotes boxes
  int alfa, beta;   // denotes particles
  int k, q;         // denotes term in expansion or summation index
  int sign;
  // r - position, v - velocity, F - force
  Complex r[NPARTICLES],v[NPARTICLES],F[NPARTICLES];
  // phi - angle, omega - angular velocity, tau - torque
  float phi[NPARTICLES],omega[NPARTICLES],tau[NPARTICLES];
  const float mom = 0.05, mass = 1, I = 1; // particle properties
  float t;                      // time
  int bcoeff[2*p+1][2*p+1];     // binomial coefficients
  Complex Sigma[2*p+1];         // explained below ...
  Complex delta, delta00, delta01, delta10, delta11; // relative pos. vectors
  Complex power, power00, power01, power10, power11; // powers of deltas
  Complex taucsum, fcsum, LJsum; // just auxiliary vars to compute forces
  float re, im, delta2, delta6; //             -"-
  // Rel. coordinates for particles in interactionlist (see section 3.3.2)
  const int neighbourlisti[4][27] = {{-2, -1, 0, 1, 2, 3, -2, 2, 3, -2, 2, 3,
                                      -2, 2, 3, -2, -1, 0, 1, 2, 3, -2, -1,
                                      0, 1, 2, 3},
                                     {-2, -1, 0, 1, 2, 3, -2, -1, 0, 1, 2, 3,
                                      -2, 2, 3, -2, 2, 3, -2, 2, 3,
                                      -2, -1, 0, 1, 2, 3},
                                     {-3, -2, -1, 0, 1, 2,
                                      -3, -2, 2, -3, -2, 2, -3, -2, 2,
                                     -3, -2, -1, 0, 1, 2, -3, -2, -1, 0, 1, 2},
                                     {-3, -2, -1, 0, 1, 2, -3, -2, -1, 0, 1, 2,
                                      -3, -2, 2, -3, -2, 2, -3, -2, 2,
                                      -3, -2, -1, 0, 1, 2}};
  const int neighbourlistj[4][27] = {{-2, -2, -2, -2, -2, -2, -1, -1, -1,
                                      0, 0, 0, 1, 1, 1,
                                      2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3},
                                     {-3,-3,-3,-3,-3,-3,-2,-2,-2,-2,-2,-2,
                                      -1, -1, -1, 0, 0, 0, 1, 1, 1,
                                      2, 2, 2, 2, 2, 2},
                                     {-2, -2, -2, -2, -2, -2, -1, -1, -1,
                                      0, 0, 0, 1, 1, 1,
                                      2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 3, 3},
                                     {-3,-3,-3,-3,-3,-3,-2,-2,-2,-2,-2,-2,
                                      -1, -1, -1, 0, 0, 0, 1, 1, 1,
                                      2, 2, 2, 2, 2, 2}};

  // Set the pointers for the "4-dim" field of the multipole-expansion
  setpointers();

  // Set the sigma sum,
  // Sigma[n] = sum 1/z^n where the sum is over all z = m + in where
  // m and n are integers whose absolute values are >=2
```

```
setsigma(Sigma);

// Set binomial coefficients
setbcoeff(bcoeff);

// Initialize particle positions, velocities, angles and ang. vel.
read_float_from_file("/1/Applied/pgm/c/data/phi.dat", phi, NPARTICLES);
read_float_from_file("/1/Applied/pgm/c/data/omega.dat", omega, NPARTICLES);
read_complex_from_file("/1/Applied/pgm/c/data/r.dat", r, NPARTICLES);
read_complex_from_file("/1/Applied/pgm/c/data/v.dat", v, NPARTICLES);

// Get the positions for all the centres, i.e. initialize c
setcentres();

for (t=0 ; t <= (nstep-1)*dt ; t += dt ) {
  printf("\nOUTERMOST LOOP             Time = %f\n", t);

  // place particles in boxes
  placeinpanels(r);

  // MULTIPOLE FORCE ALGORITHM

  // STEP ONE
  // Get multipoleexpansions around the centres of all panels
  for (i=0 ; i < sqrtnpanels ; ++i) {
    for (j=0 ; j < sqrtnpanels ; ++j) {
      for (k=0 ; k<=p ; ++k)
        a[n][i][j][k] = 0;
      // The following loop goes over all particles in box i,j
      for (alfa = firstalfa(i,j) ; alfa != -1 ; alfa = nextalfa() ){
        delta = r[alfa] - c[n][i][j];
        power = 1;
        for (k=2 ; k<=p ; ++k){
          a[n][i][j][k] += (k-1)*power*mom*polar(1.0,phi[alfa])/pi;//eps !
          power *= delta;
        }
      } // end loop over particles
    } // end j-loop
  } // end i-loop

  // STEP TWO
  // Transfer multipoleexp's from one level to the next higher one
  for (m=n-1 ; m>=0 ; --m){
    for (i=0 ; i < (1<<m) ; ++i) {
      for (j=0 ; j < (1<<m) ; ++j) {
        a[m][i][j][0] = a[m][i][j][1] = 0;
        // 00, 01 etc indicates the four children of each box
        delta00 = c[m+1][i<<1][j<<1]         - c[m][i][j];
        delta01 = c[m+1][i<<1][(j<<1)+1]     - c[m][i][j];
        delta10 = c[m+1][(i<<1)+1][j<<1]     - c[m][i][j];
        delta11 = c[m+1][(i<<1)+1][(j<<1)+1] - c[m][i][j];
        for (k=2 ; k<=p ; ++k){
          a[m][i][j][k] = 0;
          power00 = 1;
          power01 = 1;
          power10 = 1;
          power11 = 1;
          for (q=k ; q>=2 ; --q){     // q is just a summationindex
            a[m][i][j][k] +=
              a[m+1][i<<1][j<<1][q]*(float) bcoeff[k-1][q-1]*power00;
            a[m][i][j][k] +=
              a[m+1][i<<1][(j<<1)+1][q]*(float) bcoeff[k-1][q-1]*power01;
            a[m][i][j][k] +=
              a[m+1][(i<<1)+1][j<<1][q]*(float) bcoeff[k-1][q-1]*power10;
            a[m][i][j][k] +=
```

```
            a[m+1][(i<<1)+1][(j<<1)+1][q]*(float) bcoeff[k-1][q-1]*power11;
          power00 *= delta00;
          power01 *= delta01;
          power10 *= delta10;
          power11 *= delta11;
        } // end q-loop
      } // end k-loop
    } // end j-loop
  } // end i-loop
} // end m-loop

// STEP THREE
// Get Taylorexp's around all boxcentres from the parent box and
// all the boxes in the interactionlist
// As a preliminary we take care of all copies of the whole box
for (k=0 ; k<=p ; ++k){
  b[0][0][0][k] = 0;
  for (q=2 ; q<=p ; ++q)
    b[0][0][0][k] += (float) bcoeff[q+k-1][k]*
      Sigma[k+q]* a[0][0][0][q];
  b[0][0][0][k] *= 1-((k&1)<<1);
}
for (m=1 ; m<=n ; ++m){
  for (i=0 ; i < (1<<m) ; ++i) {
    for (j=0 ; j < (1<<m) ; ++j) {
      // First we transfer taylorexp from the parent box
      delta = c[m][i][j] - c[m-1][i>>1][j>>1];
      for (k=0 ; k<=p ; ++k){
        b[m][i][j][k] = 0;
        power = 1;
        for (q=k ; q<=p ; ++q){ // q is just summationindex
          b[m][i][j][k] +=
            b[m-1][i>>1][j>>1][q]*(float) bcoeff[q][k]*power;
          power *= delta;
        } // end q-loop
      } //end k-loop
      // When we transfer the expan's from the interaction list we
      // have four different lists, depending on which child we're
      // presently transfering to. 'dummy' marks which child
      // 1: ij=01 <upper left>     3: ij=11 <upper right>
      // 0: ij=00 <lower left>     2: ij=10 <lower right>
      {
        int child, ni, nj, index;
        child = ((i&1)<<1) + (j&1);
        for (index=0 ; index<=26 ; ++index){ // loop over interaction list
          ni = i + neighbourlisti[child][index];
          nj = j + neighbourlistj[child][index];
          ii = (ni + (1<<(n+1))) & ((1<<m)-1);
          jj = (nj + (1<<(n+1))) & ((1<<m)-1);
          delta = c[m][i][j] - c[m][ii][jj];
          if (ni < 0)
            delta += 1;
          else if (ni >= (1<<m))
            delta += -1;
          if (nj < 0)
            delta += Complex(0.0,1.0);
          else if (nj >= (1<<m))
            delta += Complex(0.0,-1.0);
          if (delta == 0) {
            printf("DELTA = 0 in STEP THREE");
            break;
          }
          for (k=0 ; k<=p ; ++k){
            sign = 1 - 2*(k & 1);
            power = 1;
```

```
            for (q=1 ; q <= k+2 ; ++q) // sets power = 1/delta^(k+2)
              power = power/delta;
            for (q=2 ; q<=p ; ++q){ // q is a summation index
              b[m][i][j][k] += sign*power*(float) bcoeff[q+k-1][q-1]*
                a[m][ii][jj][q];
              power = power/delta;
            } // end q-loop
          } // end loop over k
        } // end loop over interaction list
      } // end of block
    } // end j-loop
  } // end i-loop
} // end m-loop

// STEP FOUR
for (i=0 ; i < sqrtnpanels ; ++i) {
  for (j=0 ; j < sqrtnpanels ; ++j) {
    // The following loop goes over all the particles in box ij
    for (alfa = firstalfa(i,j) ; alfa != -1 ; alfa = nextalfa() ){
      taucsum = 0;// taucsum represents the conj of the local E-field
      fcsum = 0;  // fcsum repr the conj of the derivative of the E-field
      LJsum = 0;  // LJsum adds up the Lenard-Jonas force
      // First we calculate force from taylorexpansion for the panel
      delta = r[alfa] - c[n][i][j];
      power = 1;
      for (k=0 ; k<p ; ++k) { // k is order of expansion
        taucsum += b[n][i][j][k]*power;
        fcsum += b[n][i][j][k+1]*(k+1)*power;
        power *=delta;
      } // end q-loop
      taucsum += b[n][i][j][p]*power;
      // Next we calculate pairwise interactions within the panel
      for (beta = firstbeta() ; beta != -1 ; beta = nextbeta()) {
        delta = r[alfa] - r[beta];
        fcsum -= 2*mom*polar(1.0, phi[beta])/pi/delta/delta/delta;
        taucsum += mom*polar(1.0, phi[beta])/pi/delta/delta;
        re = real(delta); im = imag(delta);
        delta2 = re*re+im*im;
        delta6 = delta2*delta2*delta2;
        LJsum += 12*sigma6*delta/delta2/delta6/delta6 -
                 6*delta/delta6/delta2;
      } // end loop over particles in panel
      // Now we calculate pairwise interactions w/ particles in neighbours
      for (beta = firstneighbourpart() ; beta != -1 ;
           beta = nextneighbourpart()){
        delta = r[alfa] - r[beta];
        fcsum -= 2*mom*polar(1.0, phi[beta])
          /pi/eps0/delta/delta/delta;
        taucsum += mom*polar(1.0, phi[beta])/pi/delta/delta;
        re = real(delta); im = imag(delta);
        delta2 = re*re+im*im;
        delta6 = delta2*delta2*delta2;
        LJsum += 12*sigma6*delta/delta2/delta6/delta6 -
                  6*delta/delta6/delta2;
      }
      tau[alfa] = -mom*imag(polar(1.0,phi[alfa])*taucsum);
      F[alfa] = mom*conj(polar(1.0,phi[alfa])*fcsum) +
                eps*sigma6*LJsum;
    } // end loop over particles in panel ij
  } // end j=loop
} // end i-loop

// VERLET TIMESTEPPING

print_rvF(r,v,F);
```

```
    print_phiomegatau(phi, omega, tau);
    for (alfa = 0 ; alfa < NPARTICLES ; ++alfa){
      v[alfa] += dt*F[alfa]/mass;
      r[alfa] += dt*v[alfa];
      r[alfa] += Complex(10,10);
      r[alfa] = r[alfa] - Complex((int) real(r[alfa]), (int) imag(r[alfa]));
      omega[alfa] += dt*tau[alfa]/I;
      phi[alfa] += dt*omega[alfa];
    }

  } // end of time-loop

  write_float_to_file("/1/Applied/pgm/c/data/phi.dat", phi, NPARTICLES);
  write_float_to_file("/1/Applied/pgm/c/data/omega.dat", omega, NPARTICLES);
  write_complex_to_file("/1/Applied/pgm/c/data/r.dat", r, NPARTICLES);
  write_complex_to_file("/1/Applied/pgm/c/data/v.dat", v, NPARTICLES);
  printf("Wrote data to files.\n");

} // end of main body

// This routine just initialises the a,b and c fields' pointers
void setpointers(void)
{
  int i1, i2, i3;
  int c1, c2, c3;
  c1 = c2 = c3 = 0;
  for (i1=0 ; i1<=n ; ++i1){
    a[c1] = &aRows[c2];
    b[c1] = &bRows[c2];
    c[c1] = &cRows[c2];
    ++c1;
    for (i2 = 0 ; i2 < (1<<i1) ; ++i2){
      aRows[c2] = &aColumns[c3];
      bRows[c2] = &bColumns[c3];
      cRows[c2] = &cColumns[c3];
      ++c2;
      for (i3 = 0 ; i3 < (1<<i1) ; ++i3){
        aColumns[c3] = &mpols[(p+1)*c3];
        bColumns[c3] = &taylors[(p+1)*c3];
        ++c3;
      }
    }
  }
}

// This function sets c[m][i][j] = pos of centre of box ij on level m
void setcentres(void)
{
  int m, i, j, power;
  for (m=n ; m>=0 ; --m){
    power = 1<<m;
    for (i=0 ; i<power ; ++i)
      for (j=0 ; j<power ; ++j)
        c[m][i][j] = Complex(((float) i + 0.5)/power,
                      ((float) j + 0.5)/power);
  }
}

// Just computing the binomial coefficients
void setbcoeff(int bcoeff[2*p+1][2*p+1])
{
  int i,j;
  for (i=0 ; i<= 2*p ; ++i){
    for (j=i+1 ; j<=2*p ; ++j)
      bcoeff[i][j] = 0;
```

```
    bcoeff[i][0] = bcoeff[i][i] = 1;
    for (j=1 ; j<=i-1 ; ++j)
      bcoeff[i][j] = bcoeff[i-1][j-1] + bcoeff[i-1][j];
  }
}

// Sigma is explained above
void setsigma(Complex Sigma[2*p+1]){
  int N,level,i,j,k;
  const float eps=1e-6;
  Complex delta, sum, levelsum, power;
  Sigma[0] = Sigma[1] = 0;
  Sigma[2] = 0; // According to the litterature, Sigma[2] = pi ...
  for (N = 4 ; N <= 2*p ; N+=2){
    Sigma[N-1] = 0;
    sum = 0;
    levelsum=1.0;
    for ( level = 2 ; abs(levelsum)>eps ; ++level){
      levelsum = 0;
      i = -level;
      for (j=0 ; j<=level ; ++j){
        delta = Complex(i,j);
        power = Complex(1,0);
        for (k=1 ; k<=N ; ++k) power *= delta;
        levelsum += 1.0/power;
      }
      j = level;
      for ( i = 1-level ; i<=level ; ++i){
        delta = Complex(i,j);
        power = Complex(1,0);
        for (k=1 ; k<=N ; ++k) power *= delta;
        levelsum += 1.0/power;
      }
      i = level;
      for ( j = level-1 ; j>=1 ; --j){
        delta = Complex(i,j);
        power = Complex(1,0);
        for (k=1 ; k<=N ; ++k) power *= delta;
        levelsum += 1.0/power;
      }
      sum += levelsum;
    }
    Sigma[N] = 2*sum;
  }
}


void print_rvF(Complex r[], Complex v[], Complex F[])
{
  int i;
  for (i=0 ; i<NPARTICLES ; ++i)
    printf("Particle %d : r = (%2.3f,%2.3f), v = (%2.3f,%2.3f), F = (%2.3f,%2.3f)\n",
           i, real(r[i]), imag(r[i]), real(v[i]), imag(v[i]), real(F[i]), imag(F[i]));
}

void print_phiomegatau(float phi[], float omega[], float tau[])
{
  int i;
  for (i=0 ; i<NPARTICLES ; ++i)
    printf("Particle %d : phi = %2.4f (deg) omega = %2.4f (deg/s) tau = %2.4f\n",
           i, phi[i]*180.0/pi,omega[i]*180.0/pi,tau[i]);
}

void printcomplex(Complex z)
{
```

```
  printf("(%1.4f, %1.4f)",real(z),imag(z));
}

void write_float_to_file(char *name, float f[], int nelements)
{
  int i;
  FILE *FF;

  FF = fopen(name, "w");
  for (i=0 ; i<nelements ; ++i)
    fprintf(FF, "%f ", f[i]);
  fprintf(FF," \n=== DATA ================================== \n");
  fprintf(FF,"NPARTICLES = %d   n = %d   p = %d   nstep = %d  \n",
          NPARTICLES, n, p, nstep);
  fclose(FF);
}

void write_complex_to_file(char *name, Complex f[], int nelements)
{
  int i;
  float re, im;
  FILE *FF;

  FF = fopen(name, "w");
  for (i=0 ; i<nelements ; ++i){
    re = real(f[i]);
    im = imag(f[i]);
    fprintf(FF, "%f %f ", re, im);
  }
  fprintf(FF," \n=== DATA ================================== \n");
  fprintf(FF,"NPARTICLES = %d   n = %d   p = %d   nstep = %d  \n",
          NPARTICLES, n, p, nstep);
  fclose(FF);
}

void read_float_from_file(char *name, float f[], int nelements)
{
  int i;
  FILE *FF;
  float trash;

  FF = fopen(name, "w");
  for (i=0 ; i<nelements ; ++i){
    fscanf(FF, "%f ", &trash);
    f[i] = trash;
  }
  fclose(FF);
}

void read_complex_from_file(char *name, Complex f[], int nelements)
{
  int i;
  float re,im;
  FILE *FF;

  FF = fopen(name, "r");
  for (i=0 ; i<nelements ; ++i){
    fscanf(FF, "%f %f ", &re, &im);
    f[i] = Complex(re, im);
  }
  fclose(FF);
}

// The following variables are used to keep track of in which panel
// the particles are. To highlight the variables shared by the
```

```
// functions they are typed in uppercase letters. An entry -1 in
// the lists denotes there are no more particles in that box.

static int startingpart[sqrtnpanels][sqrtnpanels];
typedef struct qwerty {
  int next, prev;
} particlelisttype;
static particlelisttype particlelist[NPARTICLES];
static int I, J, II, JJ, ALFA, BETA, NEIGHBOUR, NEIGHBOURPART;

void placeinpanels(Complex r[])
{
  int i,j, alfa, beta;
  for (i = 0 ; i < sqrtnpanels ; ++i)
    for (j = 0 ; j < sqrtnpanels ; ++j)
      startingpart[i][j] = -1;
  for (alfa = 0 ; alfa < NPARTICLES ; ++alfa)
    particlelist[alfa].prev = particlelist[alfa].next = -1;

  for (alfa = 0 ; alfa < NPARTICLES ; ++alfa){
    i = (int) (sqrtnpanels * real(r[alfa]));
    j = (int) (sqrtnpanels * imag(r[alfa]));
    if (startingpart[i][j] == -1)
      startingpart[i][j] = alfa;
    else {
      beta = startingpart[i][j];
      particlelist[beta].prev = alfa;
      particlelist[alfa].next = beta;
      startingpart[i][j] = alfa;
    }
  }
}

void stats(void)
{
  int i,j, dummy, alfa;
  for (j=sqrtnpanels-1 ; j>=0 ; --j){
    for (i=0 ; i<sqrtnpanels ; ++i){
      dummy = 0;
      if ((alfa = firstalfa(i,j)) == -1){
        printf(".");
        ++dummy;
      }
      else {
        putchar((alfa%26) + 'A');
        ++dummy ;
        for ((alfa = nextalfa()) ;
             (alfa != -1) && (dummy < 2) ; alfa = nextalfa()){
          putchar((alfa%26) + 'A');
          ++dummy;
        }
      }
      while (dummy < 2){
        printf(" ");
        ++dummy;
      }
    }
    printf("\n");
  }
}

int firstalfa(int i, int j)
{
  I = i;
  J = j;
```

```
  ALFA = startingpart[I][J];
  return ALFA;
}

int nextalfa(void)
{
  ALFA = particlelist[ALFA].next;
  return ALFA;
}

int firstbeta(void)
{
  BETA = startingpart[I][J];
  if (BETA == ALFA)
    BETA = particlelist[BETA].next;
  return BETA;
}

int nextbeta(void)
{
  BETA = particlelist[BETA].next;
  if ( BETA == ALFA )
    BETA = particlelist[BETA].next;
  return BETA;
}

int firstneighbourpart(void)
{
  const int neighboursi[8] = {-1, 0, 1, -1, 1, -1, 0, 1};
  const int neighboursj[8] = {-1, -1, -1, 0, 0, 1, 1, 1};
  NEIGHBOUR = 0;
  II = ( (I + neighboursi[NEIGHBOUR]) + (1<<(n+1))) & ((1<<n)-1);
  JJ = ( (J + neighboursj[NEIGHBOUR]) + (1<<(n+1))) & ((1<<n)-1);
  NEIGHBOURPART = startingpart[II][JJ];
  while( ( NEIGHBOURPART == -1) && ( NEIGHBOUR != 7 ) ){
    ++ NEIGHBOUR;
    II = ( (I + neighboursi[NEIGHBOUR]) + (1<<(n+1))) & ((1<<n)-1);
    JJ = ( (J + neighboursj[NEIGHBOUR]) + (1<<(n+1))) & ((1<<n)-1);
    NEIGHBOURPART = startingpart[II][JJ];
  }
  return NEIGHBOURPART;
}

int nextneighbourpart(void)
{
  const int neighboursi[8] = {-1, 0, 1, -1, 1, -1, 0, 1};
  const int neighboursj[8] = {-1, -1, -1, 0, 0, 1, 1, 1};
  NEIGHBOURPART = particlelist[NEIGHBOURPART].next;
  while( ( NEIGHBOURPART == -1) && ( NEIGHBOUR != 7 ) ){
    ++ NEIGHBOUR;
    II = ( (I + neighboursi[NEIGHBOUR]) + (1<<(n+1))) & ((1<<n)-1);
    JJ = ( (J + neighboursj[NEIGHBOUR]) + (1<<(n+1))) & ((1<<n)-1);
    NEIGHBOURPART = startingpart[II][JJ];
  }
  return NEIGHBOURPART;
}
```

Bibliography

[1] S.J. Aarseth *Monthly Notices of the Royal Astronomical Society 126 (1963), pp. 223-225.* Dynamical Evolution of Clusters of Galaxies - I

[2] M.P. Allen and D.J. Tildesley *Oxford University Press, New Yord, 1987* Computer Simulation of Liquids

[3] A.W. Appel *SIAM Journal on Scientific and Statistical Computation 6 (1985) pp. 85 - 103* An Efficient Program for Many-body Simulation

[4] O. Axelsson *Linear Algebra Applications 29, (1980) pp. 1-16* Conjugate Gradient Type Methods for Unsymmetric and Inconsistent Systems of Linear Equations

[5] R.J. Baxter *Academic Press, London 1982* Exactly Solved Models in Statistical Mechanics

[6] D.M. Beazley and P.S. Lomdahl *SIAM News (Newsletter of the Society for Industrial and Applied Mathematics), Volume 28, (Number 2,1995)* Large-Scale Molecular Dynamics on MPPs

[7] J.A. Board, J.W. Causey, J.F. Leathrum, A. Windemuth, K. Shulten *Chemical Physics Letters 198, (1992), pp. 89 - 94* Accelerated Molecular Dynamics Simulation with the Parallell Fast Multipole Algorithm

[8] S. Chandrasekhar *Dover Publications Inc., New York, 1942* Principles of Stellar Dynamics

[9] A.J. Chorin *Journal of Fluid Mechanics 57 (1973) pp. 785 - 796* Numerical Study of Slightly Viscous Flow

[10] H-Q Ding, N. Karasawa, W.A. Goddard *Journal of Chemical Physics 97, pp. 4309 - 4315* Atomic Level Simulations of a million particles: The cell multipole method for Coulomb and London non-bond interactions

[11] H-Q Ding, N. Karasawa, W.A. Goddard *Chemical Physics Letters 196, (1992), pp. 6-14* The Reduced Cell Multipole Method for Coulomb Interactions in Periodic Systems with Million atom unit cells

[12] C.I. Draghicescu *Journal of Computational Physics, 116, (1995), pp. 69 - 78* A Fast Algorithm for Vortex Blob Interactions

[13] S.C. Eisenstadt, H.C. Elman, M.H. Schultz *SIAM Journal of Numerical Analysis 20, (1983), pp. 345 - 357* Variational, Iterative Methods for Non-symmetric Systems of Linear Equations

[14] L.F. Greengard *MIT Press, Cambridge 1988* The Rapid Evaluation of Potential Fields in Particle Systems

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

[16] W. Hackbusch and Z.P. Nowak *Numerische Mathematik 54, (1989), pp. 463 - 491* On the Fast Matrix Multiplication in the Boundary Element Method by Panel Clustering

[17] W. Hackbusch and S.A. Sauter *Bericht Nr. 92-18, Univarsität Kiel* On the Efficient Use of the Galerkin Method to Solve Fredholm Integral Equations

[18] J.T. Hamilton, G. Majda *Journal of Computational Physics, 121, pp. 29 - 50 (1995)* On the Rokhlin Greengard Method with Vortex Blobs for Problems Posed in all Space or Periodic in one direction

[19] D.M. Heyes *Journal of Chemical Physics 74, pp. 1924 - 1929 (1981)* Electrostatic Potentials and Fields in Infinite Point Charge Lattices

[20] R.W. Hockney and J.W. Eastwood *McGraw-Hill, New York, 1981* Computer Simulation Using Particles

[21] R.W. Hockney, J.W. Eastwood and D. Lawrence *Computational Physics Communication 19 (1980) pp. 215 - 261* P3M3DP — the three dimensional periodic particle-particle/particle-mesh program

[22] F. Hoyle *Doubleday and Co. Inc. New York 1982* Astronomy

[23] S.W. de Leeuw, J.W. Perram, E.R. Smith *Proceedings of the Royal Society A373, pp. 27 - 56, (1980)* Simulation of Electrostatic Systems in Periodic Boundary Conditions. I Lattice Sums and Dielectric Constant.

[24] M.J. Mandell *Journal of Statistical Physics 15 (1976) pp. 299 - 305* On the properties of a periodic fluid

[25] W. McLean *Workshop on Theoretical and Numerical Aspects of Geometric Variational Problems, Proceedings of the Centre for Mathematics and its Applications 26, Australian National University, 1991* Variational Properties of some Simple Boundary Integral Equations

[26] H.G. Petersen, D. Soelvason, J.W. Perram and E.R. Smith *Proceedings of the Royal Society, London, 448, (1995), pp. 389 - 400* Error Estimates for the Fast Multipole Method I, The 2-dimensional case

[27] H.G. Petersen, D. Soelvason and E.R. Smith *Proceedings of the Royal Society, London, 448, (1995), pp. 401 - 418* Error Estimates for the Fast Multipole Method II, The 3-dimensional case

[28] L.R. Pratt and S.W. Haan *Journal of Chemical Physics 74 pp. 1864 - 1872* Effects of Periodic Boundary Conditions on Equilibrium Properties of Computer Simulated Fluids, I, Theory

[29] V. Rokhlin *Computers & Mathematics with Applications, 11 (1985) pp. 667 - 679* Application of Volume Integrals to the Solution of Partial Differential Equations

[30] V. Rokhlin *Journal of Computational Physics 60, (1983) pp. 187-207* Rapid Solution of Integral Equations of Classical Potential Theory

[31] S.A. Sauter *To appear in "The IMA Volumes in Mathematics and its Applications": Wavelets, Multigrid and other Fast Methods and Their use in Wave Propagation, Springer-Verlag* The Panel Clustering Method in 3-d BEM

[32] D. Soelvasen, J Kolafa, H.G. Petersen, J.W. Perram *Computational Physics Communications, 87, (1995), pp. 307 - 318* A Rigorous Comparison of the Ewald method and the Greengard-Rokhlin Method

[33] J. Stoer, R. Burlich *Springer-Verlag, New York, 1993* Introduction to Numerical Analysis, second edition

[34] W.B. Streett, D.J. Tildesley, G. Saville *Molecular Physics 35, (1978), pp. 639 - 648* Multiple Time Step Methods in Molecular Dynamics

[35] A. Wouk *Wiley, New York, 1978* A course of Applied Functional Analysis