Atomistic Modeling of Biomolecular Interactions

New Mexico Adventures in
Supercomputing Challenge
Final Report
April 7, 2004

026
Eldorado High School

Team Members

Tom Dimiduk

Jeff Dimiduk

Daniel Appel

Ryan Shea

Teacher

David Dixon

Project Mentor

Susan R. Atlas

Executive Summary

Molecular motors are a group of biological proteins, which are critical to life because they are responsible for all transport within cells. They are the primary mechanism for converting chemical energy to mechanical work at very small scales. There are numerous subfamilies of different proteins, but they all basically use ATP (adenosine triphosphate) molecules as an energy source to move $\sim 8-30$ nanometers along a microtubule. Understanding how these proteins work can lead to new therapies for cancer and muscular disorders. A better understanding of how these proteins work could also make it possible to incorporate these proteins into nanoscale biomimetic devices.

The size of molecular motor proteins makes it impossible to directly observe their function, so scientists recently began to attempt to use atomistic simulations to model the protein structure and function. For atomistic modeling, each atom is represented as a particle in space with attributes such as position, velocity, and charge. The modeling is done by approximating forces between particles, and changing the position and velocity of each particle as a result of these forces, ($\boldsymbol{F}$ = m$\boldsymbol{a}$), for each unit of time. These calculations become very computationally challenging when modeling biophysical systems because proteins exist in aqueous solutions containing many thousands of atoms and ions which need to be individually modeled in addition to modeling the protein itself.

The particular part of the molecular motor dynamics simulation that we worked on was the calculation of long-range electrostatic interactions. In the specific case of long-range forces, the behavior of charged particle interactions requires a sum over a prohibitively large number of particles. We use a periodically repeated unit cell to simplify calculations without introducing artificial boundary effects. We further simplify the calculations by using Ewald summation, a technique in which a conditionally convergent sum is simplified to a real space sum, and a reciprocal (or Fourier) space sum. We then are able to increase the efficency of our calculations by using Particle Mesh Ewald,a method that allows us to use Fast-Fourier Transforms to calculating the Fourier space term.

Computational efficiency is of utmost importance because modeling molecular motor systems containing many thousands of atoms is computationally intensive. Our Fortran90 code successfully combined modules to calculate the complementary error function, Cardinal B-Splines, and mesh interpolation in order to implement the Particle Mesh Ewald method. With this, we calculated the real and Fourier of the Particle Mesh Ewald method, and could calculate long-range interactions for the many thousands of particles needed in molecular motor biophysical simulations.

Contents

1 Introduction

Motor proteins such as kinesin, mysosin and dynein are interesting because they occur in all living organisms, and are critical to life, but perhaps more directly they could be used to design nanoscale biomimetic devices if they are understood sufficiently. However the small size of motor proteins makes them difficult to study; the structure and function cannot be fully elucidated through experimental techniques. Even something as simple as how a kinesin protein walks is still in debate [2]. Direct observation of protein function cannot be performed using current methods of observation, such as coating the specimen with metals for the electron microscopes, as this prevents the protein from functioning [5]. Because of this problem, little is known about the conformational changes[1] of motor proteins function. Specifically we are working on modeling the electrostatic interactions of the thousands or even hundreds of thousands of atoms in a molecular motor proteins and the surrounding aqueous solution. The electrostatic charges derive from the polarity of the water molecules, the ions in solution, and the charged portions of the molecular motor. The interactions between these charges are important because they are the main long-range forces in molecular simulations. Our work primarily is concerned with helping make calculation of the interactions between individual pairs of atoms as efficient as possible. As each atom interacts with each other atom, the time required to perform the calculation goes up as the square of the number of atoms. In order to more efficiently calculate these interactions, we have used a method known as Particle Mesh Ewald (PME) to model the electrostatic particle interactions. Specifically, our goal has been to develop a way of modeling biological molecules[2] in a realistic environment, that is to say, one in which the molecule is surrounded by water molecules. PME, an improvement over the Ewald method [7], makes it possible to compute these interactions more quickly and thus permits the solution of large problems that would otherwise be inaccessible to modern machines in a reasonable timescale.

1.1 Biology Motivation

Myosin proteins are responsible for all muscle movement. Muscle cells contain parallel arrays of actin microfilaments and myosin proteins [5]. The myosin proteins have a lever arm with a pair of catalytic heads extending out towards a parallel actin filament. By hydrolyzing a molecule of ATP, the myosin arm binds to the actin filament and then rotates through an angle, in a motion similar to an oar stroke, pulling the actin filament by around 10 nm. After each pull, the myosin arm detaches from the actin filament so that it does not interfere with the pull of the other myosin proteins and then recocks for another stroke. The combination of small pulls by myosin arms along the length of the cell moves the actin and myosin filaments parallel to each other, contracting the muscle fiber. This contraction is responsible for all macroscopic motion of animals [20]. Kinesin proteins are the transportation system for the interior of a cell. They move a variety of cargos along microtubules. These cargos include organelles, proteins for secretion, RNA, and the mitotic spindle during mitosis. During cellular mitosis, Dynein proteins move chromosomes [12]. Dynein proteins are also responsible for the motion of cilia and flagella [5]. The exact method by which proteins produce motion is not well understood, but some theories have been proposed. It is known that all of the proteins go through a catalytic cycle involving the breakdown of ATP. The main components of this cycle involve: the protein binding to a molecule of ATP, binding to an actin filament or microtubule, splitting the ATP into an adenosine diphosphate molecule and an inorganic phosphate molecule, moving some portion of the protein through a distance, releasing the

[1]Changes in the shape of a protein, i.e. motion
[2]Molecules used by living organisms; generally containing carbon

two product molecules, and releasing the bond to the structural fiber. However, not all proteins proceed through this cycle in the same order [20]. The most prevalent theory holds that the cycle of ATP binding and hydrolysis create strain in the protein that produces motion when it is relieved. The strain appears to be produced in a spring-like coiled segment called an alpha helix. A lever arm called the neck linker amplifies this strain into the several nanometer motion of the protein [8]. An array of protein segments called switches is needed to coordinate binding and release of the microtubule or filament with motion and the ATP cycle [20]. Because so little is known about these proteins, they are an active topic of research. Understanding these proteins is important for a variety of reasons. Chromosomal nondisjunction diseases, like Down syndrome, could be caused by a malfunction in these proteins. Disruption of normal functioning of kinesin interferes with development which causes defects, and interferes with cardiovascular and neurological function [20]. Because cancer cells divide frequently and kinesin and dynein proteins are necessary for cellular division, drugs that interfered with kinesin or dynein function could be effective chemotherapeutic agents [14]. Motor proteins are the smallest known mechanism for converting chemical energy to mechanical work, and more efficient than most macroscale motors. Understanding how these proteins work could also allow the production of biomemetic, nanoscale, motile robots. Far from a simple process, the eight-nanometer step of a molecular motor protein relies on complex interactions between hundreds of thousands of atoms, and helps provide organization for the great complexity of life.

2 Specific problem

Our goal is to calculate the electrostatic forces on an atom so that we can simulate the manner in which it will move. The electrostatic component of the force between charged particles (atoms) is predicted by Coulomb's law[3]. Using this law, it is possible to calculate the electrostatic force at a given instant for all particles in the system. If this is done, it will produce an accurate representation of the electrostatic forces, but unfortunately the computations involved render the problem intractable for large systems. The goal of our work was to implement another way of describing the electrostatic interactions in the system, which is still relatively accurate, but is computationally feasible.

2.1 Solution Strategy

To model a solvated protein and avoid artifacts[4] due to boundry condiations, we create a periodic system by repeating a cell containing the system in which we are interested. This is useful because all the charge interactions one might expect to encounter in a real biological system are represented, but the setup is not so complex as to prohibit the computation.

2.2 Ewald

The first large problem we encounter in trying to calculate the electrostatic interactions is that to find the value we would have to evaluate an infinite conditionally convergent sum. However, the implementation of a periodic system also allows us to employ the Ewald method, which is specifically

[3]Coulomb's law:

$$F = \frac{q_0 q_1}{4\pi\epsilon_0 r^2} \tag{1}$$

relates the distance and charge of particles to the force they produce

[4]Behaviors introduced by simulation techniques that do not represent actual behavior

designed for computing electrostatic interactions in periodic systems. Ewald summation allows us to convert the conditionally convergent sum into two rapidly convergent sums [18] [10]. This splits the sum into two portions, one of which is evaluated in real (direct) space and one in reciprocal (Fourier) space[5]. The Ewald sum works by neutralizing long range forces with the introduction of a Gaussian cloud of opposite charge around the particle in question.[6] The portion of the charges not masked by the Gaussians comprises the real space term. The Fourier space reciprocal space term removes the contribution of the Gaussians, giving the same value as the original sum. Without the Ewald method, it would be impossible to model the system because even forces far outside the scope of the model would need to be included. The problem with The Ewald technique is that to compute the reciprocal space term, we have to compute a Fourier transform, which is very slow relative to the rest of our program, $\mathcal{O}(N^2)$

2.3 PME

An operation called Fast Fourier Transform exists that can perform the required Fourier Transform much faster, $\mathcal{O}(N\, logN)$. The problem is that a Fast Fourier Transform can only be preformed on a function that is defined on a regularly spaced grid. Because the position of the charges in our cell is not regular it is not initially feasable to use FFTs. Another technique, first proposed by Tom Darden in [7], called Particle Mesh Ewald (PME) solves this problem. In this technique, Darden used an interpolating function to assign pieces the charges to a regularly spaced mesh that could be operated on by a Fast Fourier Transform. This adds additional overhead, but all of the new operations are $\mathcal{O}(N)$ or less, the overall technique is $\mathcal{O}(N\, log(n))$ which is a significant improvement over the conventional Ewald. We used an adaptation of the PME method, suggested in [9]. This uses an alternative interpolating function, called Cardinal B-Splines which are more accurate and differentiable.

2.3.1 Cardinal B-Splines

The Cardinal B Spline is an interpolating function that we used to construct a charge mesh. It works in a similar manner to Lagrange interpolating polynomials[7], but has several important advantages. The B-Spline does a more accurate interpolation, the charge mesh constructed using a given order of B-Spline will more accurately represent the actual charge distribution than would an interpolation made using Lagrange polynomials of the same order. The other advantage is that they are differentiable. This is important because differentiating the potential function we construct will allow one to easily calculate forces on a particle. Because the kinetics equations used in a molecular dynamics simulation use forces not potentials, this is a very useful property. Lagrange polynomials are only defined at specific points, and thus are not differentiable. The B-Spline is recursively defined such that it is differentiable a number of times equal to the order of the function minus one.

2.4 Equations

Particle mesh Ewald breaks the problem up into three terms, the direct space term(2), the reciprocal space term (4), and the self-interaction term(5). The Ewald is the real term plus the reciprocal term and minus the self interaction term.

[5]A domain in which it is easier to process periodic functins, for more discussion see [16]

[6]A Gaussian is a function similar to a bell curve, concentrated in the middle and falling off towards both extremes.

[7]The technique used for interpolation in the original PME method put forth in [7], they are only defined at certain points and thus are not differentiable

2.4.1 Ewald Equations

$$\Phi_{\text{dir}}(\boldsymbol{r};\alpha) = \sum_{n} \frac{\text{erfc}(\alpha|\boldsymbol{r}+\boldsymbol{n}|)}{|\boldsymbol{r}+\boldsymbol{n}|} \tag{2}$$

erfc is the complementary error function:

$$\text{erfc}(x) = \frac{2}{\sqrt{\pi}} \int_{x}^{\infty} e^{-t^2} dt \tag{3}$$

(Computer implementation of erfc given in A.4.1: `derfc.F`)

$$\Phi_{\text{rec}}(\boldsymbol{r};\alpha) = \frac{1}{\pi V} \sum_{m\neq 0} \frac{\exp(-\pi^2 \boldsymbol{m}^2/\alpha^2)}{\boldsymbol{m}^2} \exp(2\pi i \boldsymbol{m}\cdot\boldsymbol{r}) \tag{4}$$

$$\mathcal{U}_{\text{self}} = \frac{\alpha}{\sqrt{\pi}} \sum_{i=1}^{N} q_i^2 \tag{5}$$

In these equations, $\boldsymbol{r}$ represents a distance between two particles[8], α is a constant which can be adjusted to tweak the output values, and V is the volume of the unit cell we are looking at. As mentioned under our description of the PME method, we replicate a particular region many times to create a realistic biological system. This is the volume of one such region. m and n are just unit vectors[9] normal faces of the unit cell.

The self interaction term, see Appendix A.3: `selfTerm.F` for code, is a relatively simple function that simply corrects for the fact that the other terms calculate a particle's potential relative to its own charge field, which must be accounted for.

The real space term, see Appendix A.4: `realTerm.F` for code, calculates short range interactions. It makes use of a cutoff radius to limit computational cost. Because this term is short range, the contributions to it from distant particles are very small, and neglecting them does not significantly impair accuracy. Because computing the interactions from all particles would make this term $\mathcal{O}(n^2)$ which would make it the slowest portion of the code, the cutoff radius is very important. To implement the cutoff radius we make use of integer rounding and division to rapidly compute which particles are within a certain radius of a given particle, and the calculate their effects on the given particle. Because the number of particles that must be calculated for each particle is $\mathcal{O}(1)$, this allows the real space term to be $\mathcal{O}(n)$.

$$\mathcal{U}_{\text{reciprocal}} = \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} q_i q_j \Phi_{rec}(\boldsymbol{r}_j - \boldsymbol{r}_i;\alpha) \tag{6}$$

[10].

In this q_i and q_j are charges of particles in the system and $\boldsymbol{r}_i$ and $\boldsymbol{r}_j$ refer to the respective positions of these charges. α and $\boldsymbol{n}$ are the same as referrered to in the previous equation.

[8]It also has direction; it is a vector

[9]a vector whose magnitude is one

[10]Ewald equations adapted from [10]

2.4.2 PME Equations

The PME method replaces the reciprocal term with another set of equations that are lower order. The direct replacement for $\mathcal{U}_{reciprocal}$ is:

$$\mathcal{U}_{\text{reciprocal}} \simeq \sum_{k\in C, k\neq 0} \frac{\exp\left(-\pi^2 \boldsymbol{k}^2/\alpha^2\right)}{\boldsymbol{k}^2} \boldsymbol{k} \left| \sum_{k,l,m=0}^{2^A-1} Q_H^{(p)} \exp\left(2\pi i \boldsymbol{k}\cdot \boldsymbol{s}_{klm}\right) \right|^2 \tag{7}$$

[11].

This depends on the charge mesh $Q_H^{(p)} \exp\left(2\pi i \boldsymbol{k}\cdot \boldsymbol{s}_{klm}\right)$, and the sum over C indexes over all mesh points. The mesh function is calculated calculated using 8

$$\begin{aligned} \boldsymbol{s}_{klm} = \quad & \sum_{i=1}^{N} \sum_{n_1,n_2,n_3} q_i M_n(u_{li} - k_l - n_l K_l) \\ & M_n(u_{2i} - k_2 - n_2 K_2) \\ & M_n(u_{3i} - k_3 - n_3 K_3) \end{aligned}$$

where M_n is the Cardinal B-Spline of order n given by the recursive function:

$$F = M_n(u) = \frac{u}{n-1} M_{n-1}(u) + \frac{n-u}{n-1} M_{n-1}(u-1) \tag{8}$$

and, for any real number, u, $M_2(u) = 1 - |u-1|$ for $0 \leq u \leq 2$ and $M_2(u) = 0$ for $u < 0$ $u > 2$[12]

The PME reciprocal space term, see Appendix A.5: `recpTerm.F` for code, calculates the reciprocal contribution using the Particle Mesh Ewald Method as discussed earlier in Section 2.3.

3 Background Research

Before we could begin writing our code for implementing equations 5 - 8 we had to learn several things: the necessary biology in order to understand the context of what we were coding and how our code would be used, the underlying mathematics and physics of electrostatic interactions, the computational techniques that we were going to use (Ewald, PME, Cardinal B-Splines, erfc), and the coding language, FORTRAN90. Because of the magnitude of this project we also had to learn how to use Makefiles in order to make repeated compilation feasable. Also because we are approaching the problem from the perspective of atomistic molecular dynamics rather than the protein we did not need as much biology as if we were working at the scale of protein interactions. The next step was to learn the mathematics necessary for the computation, especially Fourier transforms. Fourier transforms are a method of transforming normal functions into a periodic space, where the calculations we are trying to do over an infinitely repeating cell become much easier. Because all of us had already been exposed to calculus we were able to jump to Fourier transforms and the basics of Cardinal B-Spline. We also needed the basic physics that was required for the problem, especially Newton's Third Law and Couloumb's Law. Next we had to learn the necessary mathematical methods for the problem. First we learned about Ewald summations; then we learned about a more complicated but computationally more efficient, for large numbers of particles, method called Particle Mesh Ewald (PME) which breaks the charges up and assigns pieces of them to points on a mesh. This allows the use of Fast Fourier Transforms, which require

[11]PME equation adapted from [7]

[12]Cardinal B-Spline equations adapted from [9].

a regularly-spaced mesh. To assign the charges to the mesh we used an advanced mathematical technique called Cardinal B-Splines. Learning these mathematical techniques in sufficient detail to use them in our code proved to be the most difficult part of the project and most of the time before we actually began coding was spent on learning and understanding them. We also had to learn FORTRAN90 for this project, but did not have time for a full course so we did a warm up project programming the complimentary error function (erfc), a simple integral that we need for the real space term, see Equation 3 and code in Appendix A.4.1. Because of the relativly large amounts of code we worked with, we found it useful to learn and use CVS (Concurrent Versions System) to manage editing of files, and makefiles to handle compilation. To confirm the accuracy of our code for calculating several of the functions, we also learned `Matlab` which we used to produce graphs and run numerical checks. We also had to learn how to interface with the FFTW library, [11] was very helpful in this regard.

4 Results

4.1 Complementary Error Function

We have written code that computes the complementary error function (erfc) accurately to the limits of a double precision real variable. The error function is an integral that cannot be solved analytically, see equation 3, for all of the cases we need, so we have used a rational Chebyshev expansion[13] to rapidly approximate the erfc to the required accuracy. Our mentor has code that computed the error function, but it has several problems, one of them fairly severe. The first problem was that her code was not as accurate as is needed. The second, more troublesome problem was that her program that used the error function gave different results when ran on `IBM AIX` and `Linux` machines. To correct this we found and wrote several functions to approximate the error function and tested them on both machines. Eventually we found that the we were encountering a namespace conflict, the IBM complier had an intrinsic function to evaluate error functions and it was calling that function automatically. We then ran comparisons of the outputs of five different error function implementations: our mentor's original code, our corrected version of that code, the intrinsic function in the `IBM xlf90` compiler, code the naval surface warfare group make available, and a function we wrote. Our function, the IBM function, and the naval surface warfare codes all gave correct results out to the limit of accuracy we compared. We decided to use the naval surface warfare code (code in Appendix A.4.1) because the IBM code was only usable on that compiler, and we assume that their code has been more completely checked than ours. We also have reason to believe that the naval surface warfare code may be more accurate than our code. They stated that it was accurate to 14 decimal places [23], while we are only sure of ours to 12 decimal places.

4.2 Cardinal B-Splines

We have also written code that computes Cardinal B splines of order two through six, see Equation 8. Figure 2 demonstrates that the Cardinal B splines our code generates are symmetric about their center, as stated in [9]. We also checked, using `Matlab`, that they obey the sum property they are supposed to, namely, the sum of all integer values of the spline function is one. To confirm that our function calculated the spline properly, we made use of a number of mathematical properties of the spline function, as given in [9]. The most important ones are symmetry, limited range where the function is nonzero, and the fact that the sum of the function evaluated for all integers is one. The first two properties are readily evident from 2, made in `Matlab`. We originally created this graph as

[13]A series of precalculated terms that can be used to approximate a function

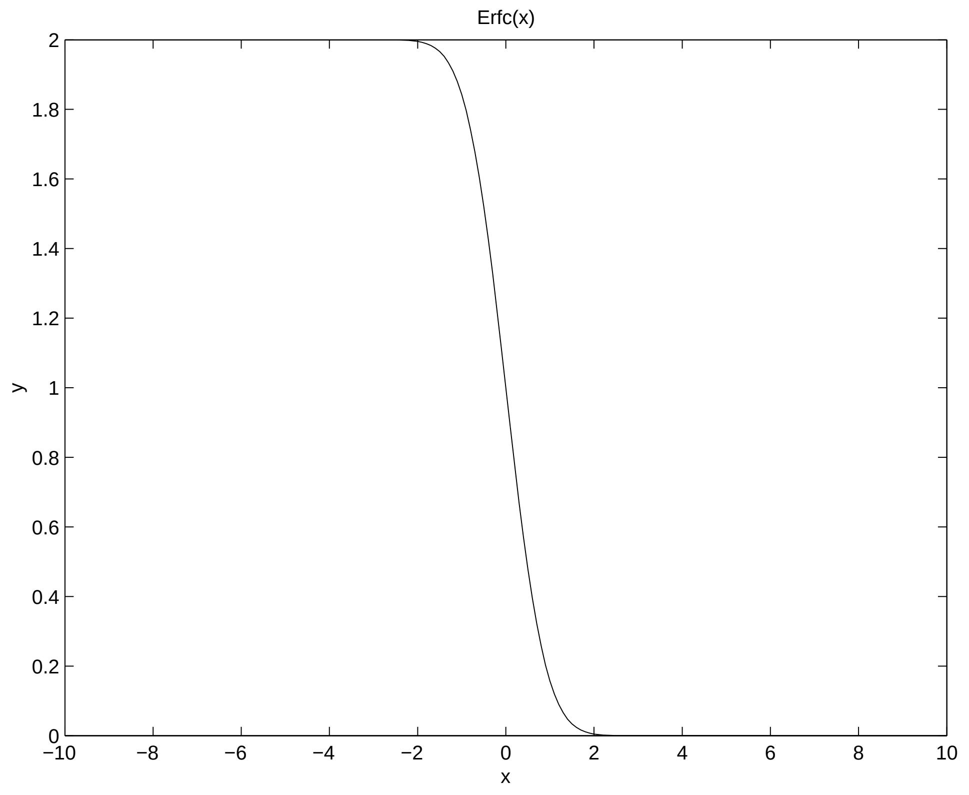

Figure 1: Output of derfc.F for $1 \leq x \leq 10$

a debugging tool. The spline function was not giving correct results, so this graph allowed us to see exactly where the function was incorrect. The varying asymmetries in the graphs of the different order functions pointed us directly to where the errors were in the code. Higher order splines have non zero values for a larger domain, allowing them to use more values in interpolation, though obviously at the cost of more computational time. We confirmed the sum property using output from our code and numerical tools in `matlab`. This property is important because we multiply the Cardinal B-Spline value by the charge of a particle and then make use of its values at integer valued mesh points. This sum property ensures that the charge field will not have its magnitude distorted by interpolation.

4.3 Computing Ewald Energy

Our most significant accomplishment has been writing a subroutine that calculates the Coulombic potential energy of a lattice of atoms using the Particle Mesh Ewald technique. As discussed earlier, this consists of three terms that add together to give the total electrostatic energy. Our code for the functions is in Appendices `selfTerm.F`, `realTerm.F` and `recpTerm.F`, computing, respectively, the self interaction correction, the real space interactions, and the reciprocal space interactions. The real space term uses the erfc approximation function to perform the most difficult portion of the math, and a cutoff radius to reduce the computation to $\mathcal{O}(N)$. The reciprocal space term uses the Cardinal B-Spline functions to construct a discrete mesh that approximates the charge distribution of the particle lattice. A series of loops and if statements allow us to interpolate a charge mesh from the particles with varying degrees of accuracy. We have currently implemented interpolation for Cardinal B-Splines of order two through six. We then make use of fast Fourier transforms from

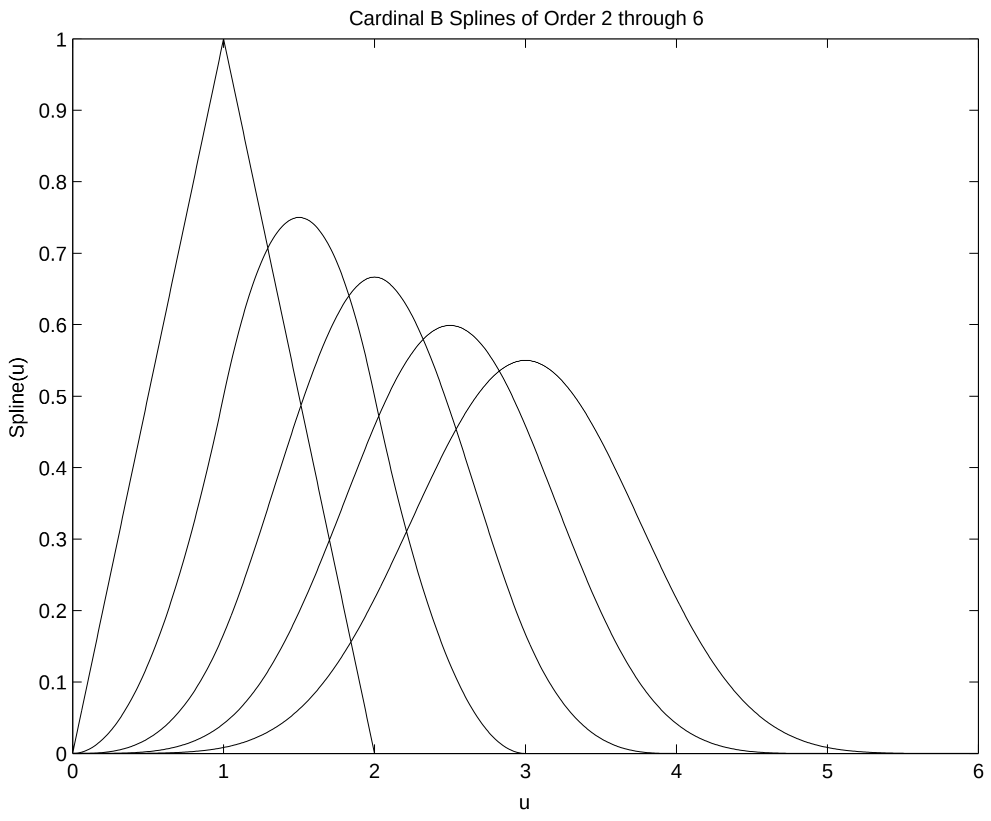

Figure 2: Cardinal B-Splines for order 2 through 6

the FFTW library [21] to perform the required transform to reciprocal space. Our code starts with a file, lattice.dat (see Appendix B), that describes the geometry of a system of particles. We have succeeded in producing results that compare well to the conventional Ewald code we used for comparison. As demonstrated in Appendices C.2 and C.1, our code produces results that agree with the standard, Conventional Ewald, to within three significant figures. We hope by the final presentation to test our code on more and larger lattices and run timing studies comparing our code to the conventional Ewald code.

4.4 Tools Used

We wrote our program in Fortran 90; used the Portland Group high performance `pgf90` compiler to compile our code, and makfiles to handle compilation. We made use of Concurrent Versions System (CVS) to manage editing between group members. We used the FFTW library for Fast Fourier transforms, and used erfc code from NERSC [23]. We wrote this report using the LaTeXDocument Preparation System.

5 Conclusions

We were able to calculate the complimentary error function, Cardinal B Splines, a charge mesh, the self-interaction and real space Ewald terms, and, using FFTW the library, compute the reciprocal space term. We have combined all these functions into an overreaching Particle Mesh Ewald Code that can be used to compute long-range interactions for a molecular dynamics simulation. The overall code adds the real and reciprocal space terms and subtracts the self-interaction term, yeilding the Coulombic potential of the lattice. We are currently working with a lattice of 32 atoms with varying charges, but our code is equally capable of addressing the thousands of atoms needed for large biomolecules and their systems. Because our algorithm is $O(n \log n)$ overall, it will run significantly faster for large numbers of particles than the conventional Ewald code our mentor is currently using. This should improve the feasibility of simulating biological systems with very large numbers of particles. The logical next step would be to combine our code with a subroutine that calculates short range quantum mechanical interactions to run a molecular dynamics simulation.

6 Acknowledgements

We would like to thank our Mentor Dr. Susan R. Atlas, our Sponsor David Dixon, and the UNM High Performance Computing Center.

References

[1] Abramowitz, Milton, and Stegun, Irene A. Ed. June 1964. *Handbook of Mathematical Functions with Formulas, Graphs, and Mathematical Tables.* Dover Publ. Inc. NY, NY. Pp. 297-311.

[2] Asbury, Charles L., Fehr, Adrian N. and Block, Steven M. 2003. "Kinesin Moves by an Asymmetric Hand-Over-Hand Mechanism". *Science.* 302:2130-2134.

[3] Allen, M.P. and Tildesley, D. J. 1991. *Computer Simulation of Liquids.* Clarendon Press, Oxford. 1991. Pp. 24-33 and 156-167.

[4] Berry, R. Stephen, Rice, Stuart A., and Ross, John. *Physical Chemistry.* John Wiley and Sons. NY, NY. Pp. 67-68.

[5] Campbell, Neil A. 1996. *Biology: Fourth Edition.* Addison-Wesly: Menlo Park. 1996. Pp. 129-133.

[6] Cheatham, Thomas E. III and Brooks, Bernard R. "Overview recent advances in molecular dynamics simulation towards the realistic representation of biomolecules in solution". *Theor. Chem. Acc.* 1998. 99:279-288.

[7] Darden, T. York, D. and Pederson, L. "Partlice mesh Ewald: An $N - \log N$ method for Ewald sums in large systems". *National Institute of Environmental Health Sciences.* 5 March 1993.

[8] Endow, Sharyn A. 2003. "Kinesin motors as molecular machines". *BioEssays.* 2003. 25:1212-1219

[9] Essmann, *et. al.* "A smooth particle mesh Ewald method". *J. Chem. Phys.* 1995. 103(19): 8577-8593.

[10] Frenkel, Daan and Smit, Berend. "Understanding Molecular Simulation From Algorithms to Applications". AP: San Diego. Appendix B Long-Range Inteactions. Pp. 347-355.

[11] Frigo, Matteo and Johnson, Steven G. *FFTW Users Manual.* 2003.

[12] Goldstein, Lawrence S. B. and Philip, Alistair Valentine. "The Road Less Traveled: Emerging Principles of Kinesin Motor Utilization". 1999. 15:141-83

[13] Lerlet, Loup. 1967. "Computer 'Experiments' on Classical Fluids. I. Thermodynamical Properties of Lennard-Jones Molecules". *Physical Review.* 159(1): 98-103.

[14] Mountain, Vicki and Compton, Duane A. "Dissecting the Role of Molecular Motors in the Mitotic Spindle". *The Anatomical Record.* 2000. 261:14-24.

[15] Peterson, Henrik G. "Accuracy and efficiency of the particle mesh Ewald method". *J. Chem. Phys*. 1995. 103(9): 3668-3679.

[16] Press, W.H. Teukolsky, S.A., Vetterling, W.T. and Flannery, B.P. *Numerical Recipes in Fortran, The Art of Scientific Computing, Second Edition.* Cambridge University Press: NY. 1992. Pp. 491-537.

[17] Rapaport, C. C. *The Art of Molecular Dynamics Simulation.* Cambridge University Press: Cambridge. 1995. Pp. 16-33 and 264-267.

[18] Schlick, Tamar. *Molecular Modeling and Simulation An Interdisciplinary Guide.* Springer. NY. 2002. Pp 270-285.

[19] Spanier, Jerome and Oldham, Keith B. *An Atlas of Functions.* Hemisphere Publishing Corp.: Washington. 1987. Pp. 385-388.

[20] Vale, Ronald D. and Milligan, Ronald A. "The Way Things Move: Looking Under the Hood of Molecular Motor Proteins". *Science.* 2000. 288:88-95.

[21] `http://www.fftw.org/`. 2004.

[22] http://Mathword.Wolfram.com. 2004.

[23] http://www.csit.fsu.edu/~burkardt/f_src/nswc/nswc.f90

[24] http://pages.prodigy.net/lofting/gauss.html

[25] http://www.ee.duke.edu/~ayt/ewaldpaper/node16.html

[26] http://www.ces.clrc.ac.uk/mis/software/DL_POLY/USRMAN3/node59.html

A Appendix A: Code

A.1 Main Program

```
c ********************************************************************
c     PME.F
c ********************************************************************
c
c  ProTeam's Particle Mesh Ewald (PME) code.
c
c ********************************************************************

      program PME

      use global_parameters
      use scalars
      use serial_arrays
      use pmeVar
      implicit none

c     define constants
      pi = two*asin(one)
      rtPi = sqrt(pi)

c     hardware whoami, whomai is used in parallel code, but at the present
c     we our running our code on a serial machine
      whoami = 0

      alpha = two

      maxAtoms = 100
      numt     = 20
      maxt     = 30

      allocate(    numa(0:maxt)    )
      allocate(   itype(maxAtoms)      )
      allocate(  iatnum(maxAtoms)      )
      allocate(    atom(3,maxAtoms)    )
      allocate(   qatom(maxAtoms)      )

      call ReadInput

c     Now that we know meshsize, allocate Qmesh

      allocate( Qmesh(0:meshSize,0:meshSize,0:meshSize) )

      call SetupFFT ! move outside timesteps in real MD simulation
      call CalcPME

      print *, "coulombic potential  = ", eCouloumb

      stop

      end program PME


c *********************************************************************
      subroutine CalcPME
c *********************************************************************

      use global_parameters
      use scalars
      use serial_arrays
      use pmeVar
      implicit none

      call CalcReal
      call CalcRecp
      call CalcSi


      eCouloumb = realTerm + recpTerm + selfTerm
      return
      end

c-----------------------------------------------------------------------------
      subroutine SetupFFT
c     setup for 3D complex-to-complex FFT.
c-----------------------------------------------------------------------------
c
      use scalars
      use serial_arrays
      use pmeVar
      implicit none
      include 'fftw_f77.i'

      call fftw3d_f77_create_plan(meshForward,meshSize,meshSize,
     .     meshSize,FFTW_FORWARD,FFTW_MEASURE + FFTW_IN_PLACE)

      call fftw3d_f77_create_plan(meshBackward,meshSize,meshSize,
     .     meshSize,FFTW_BACKWARD,FFTW_MEASURE + FFTW_IN_PLACE)

      return
      end
```

A.2 Read Input

```
c *********************************************************************
c     ReadInput.F
c *********************************************************************
c     ... adaptation of ReadLattice from Vernet
c *********************************************************************

      subroutine ReadInput

      use global_parameters
      use scalars
      use serial_arrays
      use pmeVar
      implicit none

      integer, dimension(48)   :: itext
      integer meshExp,sum
      integer itemp, isum, ir, ia, itrap
      character*11 fn
      real scaleFactor
      itrap = 0

c     hardwire filename

      fn   = 'lattice.dat'

c     ... initialize arrays

      a1     = zero
      a2     = zero
      a3     = zero
      b1     = zero
      b2     = zero
      b3     = zero
      iatnum = 0
      qatom  = zero
      atom   = zero
      numa   = 0
      itype  = 0

      if (whoami .eq. 0) then
         open(5,file=fn,status='old')
         read(5,'(48a1)') itext

c     get real box length (RboxL), box length is assumed to be 1 in the code
c     but this will allow numbers to be converted to actual length by
c     by multiplying by this number
```

```
         read(5,'(f16.10)') RboxL
         print *, "Box Length = ",RBoxL

c     Read rcut
         read(5,'(f16.10)') rcut
         print *, "cutoff radius = ",rcut

c     Read mesh density, convert to a power of two
          read(5,'(i2)') meshExp
          print *, 'mesh Exp =', meshExp
          meshSize = 2**meshExp
          print *, 'mesh size = ', meshSize

c     Read alpha
         read(5,'f15.10') alpha
         print *, "alpha = ",alpha

c     Read p, the order of the spline
         read(5,'i2') p
         print *, 'p ', p

c     ... read direct (a1, a2, and a3) and reciprocal (b1, b2, and b3) lattice
c         basis vectors, and total number of atoms (numa(0)).

c     get vectors defining suit direct and recpriprocal cells
         read(5,'(3f17.10)')
     &        (a1(ir),ir=1,3), (a2(ir),ir=1,3), (a3(ir),ir=1,3),
     &        (b1(ir),ir=1,3), (b2(ir),ir=1,3), (b3(ir),ir=1,3)
         scaleFactor = one/a1(1)

         read(5,'(i4)') numa(0)
         print *, "number of atoms = ",numa(0)
         if (numa(0) .gt. maxAtoms) then
           write(6,'(/,''==> Error:: numa(0) exceeds maxAtoms'')')
           write(6,'(4x,''numa(0) = '',i3)') numa(0)
           write(6,'(4x,''maxAtoms    = '',i3)') maxAtoms
           itrap = 1
         endif
      endif

c     ... each atomic position line in lattice.dat is of the following form
c         (example line from Al Sigma= 3 grain boundary file)
c
c  numt  at #   at. chg.      x                 y                 z
c   1    13      3.00     1.6662812455     8.6737885974     7.9316165296
c
c         numt is a label shared by atoms of the same type.  The value of this
c         label is stored in itype(atom #) and the total number of atoms of a
```

```
c         given type is stored in numa(<type label>).  The total number
c         of atoms overall (of any type) is stored in numa0).

      if (whoami .eq. 0) then
         qtot  = zero
         itemp = 1
         isum  = 0

         do ia = 1,numa(0)
           read(5,'(i4,i6,f10.2,3f17.10)')
     &          numt, iatnum(ia), qatom(ia), (atom(ir,ia),ir=1,3)
           do ir =1,3
              atom(ir,ia) = atom(ir,ia)*scaleFactor
           end do
! ir (1,2,3) is the (x, y, z) coordinates of an atom, ia is the atom number

c     ... the next 'if' block assumes that all atoms of a given type
c         are grouped together in the input file.   First atoms of type
c         '1' are processed, then type '2', etc.  isum is re-initialized
c         when the next type is encountered to re-start number-of-atoms-of-
c         this-type counter.

           if (numt .gt. itemp) then
             itemp = numt
             isum = 0
           end if
           isum = isum + 1
           numa(itemp) = isum
           qtot = qtot + qatom(ia)
           itype(ia) = numt
         end do

         close(5)

         do ia = 1,numa(0)
           if (itype(ia) .gt. maxt) then
             write(6,'(/,''==> Error:: numt exceeds maxt'')')
             write(6,'(4x,''numt = '',i3)') itype(ia)
             write(6,'(4x,''maxt = '',i3)') maxt
             itrap = 1
           endif
         enddo
         sum = 0
         do ia = 1,numa(0)
            sum = sum + qatom(ia)
         end do
         if (sum .ne. 0) then
            print *, "error lattice has net charge"
            stop
         end if
      endif

c     ... given vectors a1, a2, a3, volume= a1 dotted into (a2 x a3)

      unitcv = abs( a1(1)*(a2(2)*a3(3) - a2(3)*a3(2))
     &            + a1(2)*(a2(3)*a3(1) - a2(1)*a3(3))
     &            + a1(3)*(a2(1)*a3(2) - a2(2)*a3(1)) )

      if(whoami.eq.0) then
        write(6,'(/,'' Lattice input:  '',48a1)') itext
        write(6,'(/,2x,'' primitive lattice vectors:'')')
        write(6,'(/,4x,'' a1:'',3f17.10)') (a1(ir),ir=1,3)
        write(6,'(4x,'' a2:'',3f17.10)') (a2(ir),ir=1,3)
        write(6,'(4x,'' a3:'',3f17.10)') (a3(ir),ir=1,3)
        write(6,'(/,4x,'' b1:'',3f17.10)') (b1(ir),ir=1,3)
        write(6,'(4x,'' b2:'',3f17.10)') (b2(ir),ir=1,3)
        write(6,'(4x,'' b3:'',3f17.10)') (b3(ir),ir=1,3)
        write(6,'(/,2x,'' unit cell volume:'',f17.10)') unitcv
        write(6,'(/,2x,'' atoms:'')')
        write(6,'(/,5x,''number'',5x,''atomic number'',23x,''position'')
     &')
        do ia = 1,numa(0), 1
          write(6,'(/,6x,i3,11x,i3,5x,3f17.10)')
     &          ia, iatnum(ia), (atom(ir,ia),ir=1,3)
        end do
      endif

      if(itrap .eq. 1) then
         print *, 'error'
         stop
      end if
c     put in error handler to terminate program

      return
      end
```

A.3 Self Interaction Term

```
c ***********************************************************************
c     selfTerm.F
c ***********************************************************************
      subroutine CalcSi
c ***********************************************************************

      use global_parameters
      use serial_arrays
      use scalars
      use pmeVar

      implicit none

      integer j

      pi = two*asin(one)
      rtPi = sqrt(pi)   !init variables
      selfTerm = 0



      do j = 1, numa(0), 1
         selfTerm = selfTerm + qatom(j)*qatom(j)
      end do

      selfTerm = -1*alpha/rtPi * selfTerm

      return

      end
```

A.4 Real Space Term

```
c *********************************************************************
c     selfTerm.F
c *********************************************************************
      subroutine  CalcReal
c *********************************************************************
      use global_parameters
      use scalars
      use serial_arrays
      use pmeVar

      implicit none
      integer i, j, k
      real dX, dY, dZ, dsq, d
      real derfc

      pi = two*asin(one)
      rtPi = sqrt(pi)
      rcutsq = rcut*rcut
      realTerm = zero

c     calculate real sum using only terms within rcut
      do i = 1, numa(0)
         do j = 0, numa(0)
            if (i .ne. j) then
c     nested ifs are used to reduce the array iterations that must be computed
c     if the difference in x, y, or z coordinate alone is greater than rcut

c     minimum image is handled peacewise as each term is evaluated to be less
c     than rcut

         dX = atom(1, i) - atom(1, j)
         dX = dX - anint(dX)
         if (abs(dX) < rcut) then
            dY = atom(2, i) - atom(2, j)
            dY = dY - anint(dY)
            if (abs(dY) < rcut) then
               dZ = atom(3, i) - atom(3, j)
               dZ = dZ - anint(dZ)
               if (abs(dZ) < rcut) then
                  dsq = dX*dX + dY*dY + dZ*dZ
                  if (dsq < rcutsq) then
                     d = sqrt(dsq)
                     if (d .ne. 0) then
                        realTerm = realTerm + derfc(alpha*d)/d

                     end if
                  endif
               endif
            endif
         endif
      end if
   end do
end do

return

end
```

A.4.1 Complementary Error Function

```
c *********************************************************************
c     derfc.F used with permission from Naval Surface Warfare Group
!*****************************************************************************
      function derfc (x)
!
!*****************************************************************************
!
!! DERFC: the complementary error function
!
      double precision derfc
      double precision an, ax, c, eps, rpinv, t, x, w
      double precision a(21), b(44), e(44)
      double precision dpmpar, dcsevl
!
!     rpinv = 1/sqrt(pi)
!
  data rpinv /.56418958354775628694807945156077259d0/
!
  data a(1)  / .12837916709551257389615890312150d+00/, &
       a(2)  /-.37612638903183752463205296770700d+00/, &
       a(3)  / .11283791670955125738961589029310d+00/, &
       a(4)  /-.26866170645131251759432353725420d-01/, &
       a(5)  / .52239776254421878421118124478770d-02/, &
       a(6)  /-.85483270234508528325401640811870d-03/, &
       a(7)  / .12055332981789664250207171824980d-03/, &
       a(8)  /-.14925650358406250904307285268200d-04/, &
       a(9)  / .16462114365889242610807235781090d-05/, &
       a(10) /-.16365844691234687574089684296740d-06/
  data a(11) / .14807192815870217154008186278110d-07/, &
       a(12) /-.12290555301451201408005101553310d-08/, &
       a(13) / .94227590584371970173130550842120d-10/, &
       a(14) /-.67113667409693850858962572271590d-11/, &
       a(15) / .44632226082956640174617588435500d-12/, &
       a(16) /-.27834973955429954872750658569980d-13/, &
       a(17) / .16340955723653371439330237807770d-14/, &
       a(18) /-.90528457869011239857100193879380d-16/, &
       a(19) / .47082745596897444393416714267310d-17/, &
       a(20) /-.21871593566850159497499482521600d-18/, &
       a(21) / .70434077120197016096355997013330d-20/
!
  data b(1)  / .61014308192320041792646581575600d+00/, &
       b(2)  /-.43484127271257747182818282088800d+00/, &
       b(3)  / .17635119364360550112584029812300d+00/, &
       b(4)  /-.60710795609249414860051215825000d-01/, &
       b(5)  / .17712068995694114486147141191000d-01/, &
       b(6)  /-.43211193855672938185998649680000d-02/, &
       b(7)  / .854216676887098678819832055000d-03/, &
       b(8)  /-.127155090609162742628893940000d-03/, &
       b(9)  / .112481672436711894688470720000d-04/, &
       b(10) / .313063885421820972630152000000d-06/
  data b(11) /-.270988068537762022009086000000d-06/, &
       b(12) / .307376227014076884409590000000d-07/, &
       b(13) / .251562038481762293731400000000d-08/, &
       b(14) /-.102892992132031912759000000000d-08/, &
       b(15) / .299440521199499393630000000000d-10/, &
       b(16) / .260517896872669362900000000000d-10/, &
       b(17) /-.263483992417196938600000000000d-11/, &
       b(18) /-.643404509890636443000000000000d-12/, &
       b(19) / .112457401801663447000000000000d-12/, &
       b(20) / .172815333899860980000000000000d-13/
  data b(21) /-.426410169494237500000000000000d-14/, &
       b(22) /-.545371977880191000000000000000d-15/, &
       b(23) / .158697607761671000000000000000d-15/, &
       b(24) / .208998378443340000000000000000d-16/, &
       b(25) /-.590052686940900000000000000000d-17/, &
       b(26) /-.941893387554000000000000000000d-18/, &
       b(27) / .214977356470000000000000000000d-18/, &
       b(28) / .466609850080000000000000000000d-19/, &
       b(29) /-.724301186200000000000000000000d-20/, &
       b(30) /-.238796682400000000000000000000d-20/
  data b(31) / .191177535000000000000000000000d-21/, &
       b(32) / .120482568000000000000000000000d-21/, &
       b(33) /-.672377000000000000000000000000d-24/, &
       b(34) /-.574799700000000000000000000000d-23/, &
       b(35) /-.428493000000000000000000000000d-24/, &
       b(36) / .244856000000000000000000000000d-24/, &
       b(37) / .437930000000000000000000000000d-25/, &
       b(38) /-.815100000000000000000000000000d-26/, &
       b(39) /-.308900000000000000000000000000d-26/, &
       b(40) / .930000000000000000000000000000d-28/
  data b(41) / .174000000000000000000000000000d-27/, &
       b(42) / .160000000000000000000000000000d-28/, &
       b(43) /-.800000000000000000000000000000d-29/, &
       b(44) /-.200000000000000000000000000000d-29/
!
  data e(1)  / .107797785207238315116833591035d+01/, &
       e(2)  /-.265598904091486733721465009040d-01/, &
       e(3)  /-.148707314669809950960504633300d-02/, &
       e(4)  /-.138040145414143859607708920000d-03/, &
       e(5)  /-.112803033322874914985073660000d-04/, &
       e(6)  /-.117286984274372522240537390000d-05/, &
       e(7)  /-.103476150393304615537382000000d-06/, &
       e(8)  /-.118991140858924382544470000000d-07/, &
       e(9)  /-.101622254498949864047600000000d-08/, &
       e(10) /-.137895716146965692169000000000d-09/
  data e(11) /-.936961303373730333500000000000d-11/, &
       e(12) /-.191880958395952534900000000000d-11/, &
       e(13) /-.375730172019937070000000000000d-13/, &
       e(14) /-.370537260269833570000000000000d-13/, &
       e(15) / .262756542349037100000000000000d-14/, &
       e(16) /-.112132287643793300000000000000d-14/, &
       e(17) / .184136028922538000000000000000d-15/, &
       e(18) /-.491302565748860000000000000000d-16/, &
       e(19) / .107044551673730000000000000000d-16/, &
       e(20) /-.267189366240500000000000000000d-17/
  data e(21) / .649326867976000000000000000000d-18/, &
       e(22) /-.165399353183000000000000000000d-18/, &
       e(23) / .426056266040000000000000000000d-19/, &
       e(24) /-.112558407650000000000000000000d-19/, &
       e(25) / .302561744800000000000000000000d-20/, &
       e(26) /-.829042146000000000000000000000d-21/, &
       e(27) / .231049558000000000000000000000d-21/, &
       e(28) /-.654695110000000000000000000000d-22/, &
       e(29) / .188423140000000000000000000000d-22/, &
       e(30) /-.550434100000000000000000000000d-23/
  data e(31) / .163095000000000000000000000000d-23/, &
       e(32) /-.489860000000000000000000000000d-24/, &
       e(33) / .149054000000000000000000000000d-24/, &
       e(34) /-.459220000000000000000000000000d-25/, &
       e(35) / .143180000000000000000000000000d-25/, &
       e(36) /-.451600000000000000000000000000d-26/, &
       e(37) / .144000000000000000000000000000d-26/, &
       e(38) /-.464000000000000000000000000000d-27/, &
       e(39) / .151000000000000000000000000000d-27/, &
       e(40) /-.500000000000000000000000000000d-28/
  data e(41) / .170000000000000000000000000000d-28/, &
       e(42) /-.600000000000000000000000000000d-29/, &
       e(43) / .200000000000000000000000000000d-29/, &
       e(44) /-.100000000000000000000000000000d-29/
!
  eps = epsilon ( eps )
!
!                     dabs(x) <= 1
!
  ax = dabs(x)
  if (ax > 1.d0) go to 20
  t = x*x
  w = a(21)
  do 10 i = 1,20
     k = 21 - i
     w = t*w + a(k)
   10 continue
  derfc = 0.5d0 + (0.5d0 - x*(1.d0 + w))
  return
!
!                 1 < dabs(x)  <  2
!
   20 if (ax >= 2.d0) go to 30
  n = 44
  if (eps >= 1.d-20) n = 30
  t = (ax - 3.75d0)/(ax + 3.75d0)
  derfc = dcsevl(t, b, n)
   21 derfc = dexp(-x*x) * derfc
  if (x < 0.d0) derfc = 2.d0 - derfc
  return
!
!                 2 < dabs(x)  <  12
!
   30 if (x < -9.d0) go to 60
  if (x >= 12.d0) go to 40
  n = 44
  if (eps >= 1.d-20) n = 25
  t = (1.d0/x)**2
  w = (10.5d0*t - 1.d0)/(2.5d0*t + 1.d0)
  derfc = dcsevl(w, e, n) / ax
  go to 21
!
!                      x >= 12
!
   40 if (x > 50.d0) go to 70
  t = (1.d0/x)**2
  an = -0.5d0
  c  =  0.5d0
  w  =  0.0d0
   50    c = c + 1.d0
     an = - c*an*t
     w = w + an
     if (dabs(an) > eps) go to 50
  w = (-0.5d0 + w)*t + 1.d0
  derfc = dexp(-x*x) * ((rpinv*w)/ax)
  return
!
!             limit value for large negative x
!
   60 derfc = 2.d0
  return
!
!             limit value for large positive x
!
   70 derfc = 0.d0
  return
end

function dcsevl (x, a, n)
!
!*****************************************************************************
!
!! DCSEVL: evaluate the n term chebyshev series a at x.
!          only half of the first coefficient is used.
!
  double precision dcsevl
  double precision a(n),x,x2,s0,s1,s2
!
  if (n > 1) go to 10
     dcsevl = 0.5d0 * a(1)
     return
!
   10 x2 = x + x
  s0 = a(n)
  s1 = 0.d0
  do 20 i = 2,n
     s2 = s1
     s1 = s0
     k = n - i + 1
     s0 = x2*s1 - s2 + a(k)
   20 continue
  dcsevl = 0.5d0 * (s0 - s2)
  return
end
```

A.5 Reciprocal Space Term

```
c ********************************************************************
c     recpTerm.F
c ********************************************************************
      subroutine CalcRecp
c ********************************************************************

      use global_parameters
      use scalars
      use serial_arrays
      use pmeVar

      implicit none

      integer, dimension(3) :: k
      integer i1, i2, i3, ksq
      real Qsq

      pi = two*asin(one)
      rtPi = sqrt(pi)
      pisq = pi*pi
      alphasq = alpha*alpha

      recpTerm = zero
      Call MeshInterp   !use atom(),qatom() to calculate realQmesh()
                        ![mesh=3d allocatable array]

      Call meshFFTforward   !transforms realQmesh() to recpQmesh(),
                            ! includes FFTW setup?
      do i1 = 1, meshSize
         do i2 = 1, meshSize
            do i3 = 1, meshSize
               k(1) = i1
               k(2) = i2
               k(3) = i3
               ksq = dot_product(k,k)
               Qsq = dot_product(Qmesh(k), Qmesh(k))
               recpTerm = recpTerm+exp(-pisq*ksq/alphasq)/(two*pi*ksq)*
     &          Qsq
            end do
         end do
      end do
      print *, "recp term = ",recpTerm

      return

      end
```

```
c *********************************************************************
      subroutine MeshInterp
c *********************************************************************

      use global_parameters
      use scalars
      use serial_arrays
      use pmeVar

      implicit none
      integer i,j,k,m,n,even, k1,k2,k3,nMP,MP
      real sum1, spline
      integer, dimension(:,:), allocatable :: nearestMesh
      complex test
      Qmesh = zero
      allocate(nearestMesh(p,3))

      pi = two*asin(one)
      rtPi = sqrt(pi)
      do i=1,numa(0)
         do j = 1, 3
            nMP = anint(atom(j,i)*meshSize)
            nearestMesh(1,j) = nMP
            if (nMP > meshSize) then
               print *, "error mesh point is ",nMP
               stop
            end if
            do k = 1, (p-1)/2
               if (nMP+k > meshSize) then
                  nearestMesh(1+k,j) = nMP+k - meshSize
                  nearestMesh(1+p/2+k,j) = nMP-k
               else
                  if (nMP-k < 0) then
                     nearestMesh(1+k,j) = nMP+k
                     nearestMesh(1+p/2+k,j) = nMP-k + meshSize
                  else
                     nearestMesh(1+k,j) = nMP+k
                     nearestMesh(1+p/2+k,j) = nMP-k
                  end if
               end if
            end do
            even = p/2 -(p-1)/2
            if (even .eq. 1) then
               if (nMP>meshSize) then
                  MP = nMP-meshSize
               else if (MP<0) then
```

```
                  MP = nMP+meshSize
               else
                  MP = nMP
               end if
               if (nMP>atom(j,i)) then
                  nearestMesh(p,j) = MP - k
                else
                  nearestMesh(p,j) = MP + k
               end if
            end if
         end do
         do n=1,p
            sum1 = zero
            do m=1,3
               sum1= sum1+qatom(i)*spline((atom(j,i)-
     .real(nearestMesh(n,m))/real(MeshSize))*meshSize,p)
            end do
            k1 = nearestMesh(n,1)
            k2 = nearestMesh(n,2)
            k3 = nearestMesh(n,3)
            Qmesh(k1,k2,k3) = Qmesh(k1,k2,k3)+cmplx(sum1,zero)
         end do
      end do

      return

      end


c ***********************************************************************
      subroutine meshFFTforward
c ***********************************************************************

      use global_parameters
      use scalars
      use pmeVar
      use serial_arrays

      implicit none
      include 'fftw_f77.i'

      scaleG = one/meshSize**3

      call fftwnd_f77_one(meshForward,Qmesh,0)
      Qmesh = Qmesh*scaleG

      return
```

```
end
```

A.5.1 Cardinal B-Spline

```
c ********************************************************************
c     spline.F
c ********************************************************************
      function spline(u,order)
c ********************************************************************
      implicit none
      real u, spline, M2,M3,M4,M5,M6
      integer order
      select case (order)
      case (6)
         spline = M6(u)
      case (5)
         spline = M5(u)
      case (4)
         spline = M4(u)
      case (3)
         spline = M3(u)
      case (2)
         spline = M2(u)
      case default
         print *, "spline not defined for order ",order
      end select

      return
      end

c ********************************************************************
      function M2(u)
c ********************************************************************
      implicit none
      integer, parameter :: one = 1.0d0
      real u, M2, M3
      M2 = 0
      if (u>0) then
         if (u<2) then
            M2 = one - abs(u-one)
         end if
      end if
      return
      end
c ********************************************************************
      function M3(u)
      implicit none
      integer, parameter :: one = 1.0d0
      real u, M2, M3
      M3 = u/2.0d0 * M2(u) + (3.0d0-u)/2.0d0 * M2(u-1.0d0)
      return
      end
c *********************************************************************
      function M4(u)
      implicit none
      integer, parameter :: one = 1.0d0
      real u, M3, M4

      M4 = u/3.0d0 * M3(u) + (4.0d0-u)/3.0d0 * M3(u-1.0d0)
      return
      end
c *********************************************************************
      function M5(u)
      implicit none
      integer, parameter :: one = 1.0d0
      real u, M4, M5

      M5 = u/4.0d0 * M4(u) + (5.0d0-u)/4.0d0 * M4(u-one)
      return
      end
c *********************************************************************
      function M6(u)
      implicit none
      integer, parameter :: one = 1.0d0
      real u, M5, M6

      M6 = u/5.0d0 * M5(u) + (6.0d0-u)/5.0d0 * M5(u-one)
      return
      end
```

B Appendix B: Input Lattice

```
latttice.dat, a file containing a particle geometry
fcc copper expressed as SC with 32-point basis
.10000000D-9
.25
5
0.343D0
5
    13.4808000000      .0000000000      .0000000000
      .0000000000    13.4808000000      .0000000000
      .0000000000      .0000000000    13.4808000000
      .4660840089      .0000000000      .0000000000
      .0000000000      .4660840089      .0000000000
      .0000000000      .0000000000      .4660840089
  32
   1    29      3.00      .0000000000      .0000000000      .0000000000
   1    29     -3.00     6.7404000000      .0000000000      .0000000000
   1    29      3.00      .0000000000     6.7404000000      .0000000000
   1    29     -3.00      .0000000000      .0000000000     6.7404000000
   1    29      3.00     3.3702000000     3.3702000000     6.7404000000
   1    29     -3.00    10.1106000000     3.3702000000     6.7404000000
   1    29      3.00     3.3702000000    10.1106000000     6.7404000000
   1    29     -3.00    10.1106000000    10.1106000000     6.7404000000
   1    29      3.00     6.7404000000     6.7404000000     6.7404000000
   1    29     -3.00     6.7404000000     3.3702000000     3.3702000000
   1    29      3.00     6.7404000000    10.1106000000     3.3702000000
   1    29     -3.00     3.3702000000     6.7404000000     3.3702000000
   1    29      3.00    10.1106000000     6.7404000000     3.3702000000
   1    29     -3.00     6.7404000000     3.3702000000    10.1106000000
   1    29      3.00     6.7404000000    10.1106000000    10.1106000000
   1    29     -3.00     3.3702000000     6.7404000000    10.1106000000
   1    29      3.00    10.1106000000     6.7404000000    10.1106000000
   1    29     -3.00     3.3702000000     3.3702000000      .0000000000
   1    29      3.00    10.1106000000     3.3702000000      .0000000000
   1    29     -3.00     3.3702000000    10.1106000000      .0000000000
   1    29      3.00    10.1106000000    10.1106000000      .0000000000
   1    29     -3.00     6.7404000000     6.7404000000      .0000000000
   1    29      3.00     3.3702000000      .0000000000     3.3702000000
   1    29     -3.00    10.1106000000      .0000000000     3.3702000000
   1    29      3.00     3.3702000000      .0000000000    10.1106000000
   1    29     -3.00    10.1106000000      .0000000000    10.1106000000
   1    29      3.00     6.7404000000      .0000000000     6.7404000000
   1    29     -3.00      .0000000000     3.3702000000     3.3702000000
   1    29      3.00      .0000000000    10.1106000000     3.3702000000
   1    29     -3.00      .0000000000     3.3702000000    10.1106000000
   1    29      3.00      .0000000000    10.1106000000    10.1106000000
   1    29     -3.00      .0000000000     6.7404000000     6.7404000000
```

C Appendix C: Program Outputs

Coulombic potential at the end of PME, and Ewald energy totii are the end values of the programs, and should be approximatly equal.

C.1 PME

```
 Box Length =    1.0000000000000000E-010
 cutoff radius =    0.2500000000000000
 mesh Exp =            5
 mesh size =            32
 alpha =    0.3430000000000000
 p            5
 number of atoms =            32

 Lattice input:  fcc copper expressed as SC with 32-point basis

   primitive lattice vectors:

     a1:    13.4808000000     0.0000000000     0.0000000000
     a2:     0.0000000000    13.4808000000     0.0000000000
     a3:     0.0000000000     0.0000000000    13.4808000000

     b1:     0.4660840089     0.0000000000     0.0000000000
     b2:     0.0000000000     0.4660840089     0.0000000000
     b3:     0.0000000000     0.0000000000     0.4660840089

   unit cell volume:  2449.8923228421

   atoms:
     number     atomic number                      position

        1            29          0.0000000000     0.0000000000     0.0000000000

        2            29          0.5000000000     0.0000000000     0.0000000000

        3            29          0.0000000000     0.5000000000     0.0000000000

        4            29          0.0000000000     0.0000000000     0.5000000000

        5            29          0.2500000000     0.2500000000     0.5000000000

        6            29          0.7500000000     0.2500000000     0.5000000000

        7            29          0.2500000000     0.7500000000     0.5000000000

        8            29          0.7500000000     0.7500000000     0.5000000000
```

```
         9              29          0.5000000000      0.5000000000      0.5000000000

        10              29          0.5000000000      0.2500000000      0.2500000000

        11              29          0.5000000000      0.7500000000      0.2500000000

        12              29          0.2500000000      0.5000000000      0.2500000000

        13              29          0.7500000000      0.5000000000      0.2500000000

        14              29          0.5000000000      0.2500000000      0.7500000000

        15              29          0.5000000000      0.7500000000      0.7500000000

        16              29          0.2500000000      0.5000000000      0.7500000000

        17              29          0.7500000000      0.5000000000      0.7500000000

        18              29          0.2500000000      0.2500000000      0.0000000000

        19              29          0.7500000000      0.2500000000      0.0000000000

        20              29          0.2500000000      0.7500000000      0.0000000000

        21              29          0.7500000000      0.7500000000      0.0000000000

        22              29          0.5000000000      0.5000000000      0.0000000000

        23              29          0.2500000000      0.0000000000      0.2500000000

        24              29          0.7500000000      0.0000000000      0.2500000000

        25              29          0.2500000000      0.0000000000      0.7500000000

        26              29          0.7500000000      0.0000000000      0.7500000000

        27              29          0.5000000000      0.0000000000      0.5000000000

        28              29          0.0000000000      0.2500000000      0.2500000000

        29              29          0.0000000000      0.7500000000      0.2500000000

        30              29          0.0000000000      0.2500000000      0.7500000000

        31              29          0.0000000000      0.7500000000      0.7500000000

        32              29          0.0000000000      0.5000000000      0.5000000000
 coulombic potential  =     -55.73290382118157
```

C.2 Conventional Ewald

```
 Lattice input:  fcc copper expressed as SC with 32-point basis

   primitive lattice vectors:

     a1:    13.4808000000     0.0000000000     0.0000000000
     a2:     0.0000000000    13.4808000000     0.0000000000
     a3:     0.0000000000     0.0000000000    13.4808000000

     b1:     0.4660840089     0.0000000000     0.0000000000
     b2:     0.0000000000     0.4660840089     0.0000000000
     b3:     0.0000000000     0.0000000000     0.4660840089

   unit cell volume:  2449.8923228421

   atoms:

     number     atomic number                      position

        1            29           0.0000000000     0.0000000000     0.0000000000

        2            29           6.7404000000     0.0000000000     0.0000000000

        3            29           0.0000000000     6.7404000000     0.0000000000

        4            29           0.0000000000     0.0000000000     6.7404000000

        5            29           3.3702000000     3.3702000000     6.7404000000

        6            29          10.1106000000     3.3702000000     6.7404000000

        7            29           3.3702000000    10.1106000000     6.7404000000

        8            29          10.1106000000    10.1106000000     6.7404000000

        9            29           6.7404000000     6.7404000000     6.7404000000

       10            29           6.7404000000     3.3702000000     3.3702000000

       11            29           6.7404000000    10.1106000000     3.3702000000

       12            29           3.3702000000     6.7404000000     3.3702000000

       13            29          10.1106000000     6.7404000000     3.3702000000

       14            29           6.7404000000     3.3702000000    10.1106000000
```

```
    15         29        6.7404000000    10.1106000000    10.1106000000

    16         29        3.3702000000     6.7404000000    10.1106000000

    17         29       10.1106000000     6.7404000000    10.1106000000

    18         29        3.3702000000     3.3702000000     0.0000000000

    19         29       10.1106000000     3.3702000000     0.0000000000

    20         29        3.3702000000    10.1106000000     0.0000000000

    21         29       10.1106000000    10.1106000000     0.0000000000

    22         29        6.7404000000     6.7404000000     0.0000000000

    23         29        3.3702000000     0.0000000000     3.3702000000

    24         29       10.1106000000     0.0000000000     3.3702000000

    25         29        3.3702000000     0.0000000000    10.1106000000

    26         29       10.1106000000     0.0000000000    10.1106000000

    27         29        6.7404000000     0.0000000000     6.7404000000

    28         29        0.0000000000     3.3702000000     3.3702000000

    29         29        0.0000000000    10.1106000000     3.3702000000

    30         29        0.0000000000     3.3702000000    10.1106000000

    31         29        0.0000000000    10.1106000000    10.1106000000

    32         29        0.0000000000     6.7404000000     6.7404000000
  nri1...3 =       3      3      3
  ngi1...3 =       3      3      3
total charge =     0.000000000000000

  nri1...3 =       3      3      3
  ngi1...3 =       3      3      3

  Ewald energy totii =  -55.71182724900044
```