Simulating Matter with Molecular Dynamics

Mitchell Miller

Physics 240

This assignment explores the applications and methods molecular dynamics simulations. Molecular dynamics allows one to simulate the behavior of atoms and molecules under different conditions. These simulations are used in many different fields, including physics, chemistry, and biology. By simulating the behavior of matter, one can predict the outcomes of all sorts of applications, whether it be thermodynamic equilibrium or drug effects. In this assignment, the Verlet Algorithm is used to model the speeds and forces that argon atoms exert on one another. In the course of this assignment, both one dimensional and two dimensional simulations were explored. Using these simulations, the usefulness of molecular dynamics is clear.

1. Introduction

Computer simulations are a vital to solving most complex problems in physics. The ability to effectively apply them is a vital tool for developing physicists. Using simulations, one can solve all sorts of problems; however, designing code for complicated simulations can be extremely difficult. A simulation of a realistic number of atoms in a simple box requires hundreds of calculations for just one time step. To conduct a simulation of hundreds of thousands of atoms over a long period of time would require incredible computing power and an enormous amount of time. Fortunately, there are a few simple tricks to significantly reduce the amount of calculation and time required to perform these simulations. Using methods like periodic boundary conditions and the Lennard – Jones potential, one can substantially simplify the work required.

The focus of this assignment is a two – dimensional simulation of the behavior of argon atoms. In this simulation, the atoms are modeled as hard spheres that obey the Lennard – Jones potential.

2. Theory

Simulating atoms is a relatively simple process if one uses a few easy tricks. The first and possibly most important is the use of periodic boundary conditions. Since computer memory is inherently finite, one cannot simulate an infinite number of atoms that would be necessary to get a true picture of their behavior. Instead, one simulates a box of atoms that is duplicated in all directions surrounding the area of interest[1]. This can be imagined that if an atom exits the left side of the 'box', then an atom with same speed and height enters from the right. Using this method, one can simulate an essentially infinite number of particles with significantly less computational power. The second part of boundary conditions is the minimum image condition. For this, one checks to make sure that the distance between any two atoms is less than half the length of the simulation region. If the length is greater, one creates a virtual particle outside the box on the opposite side of the particle in question.

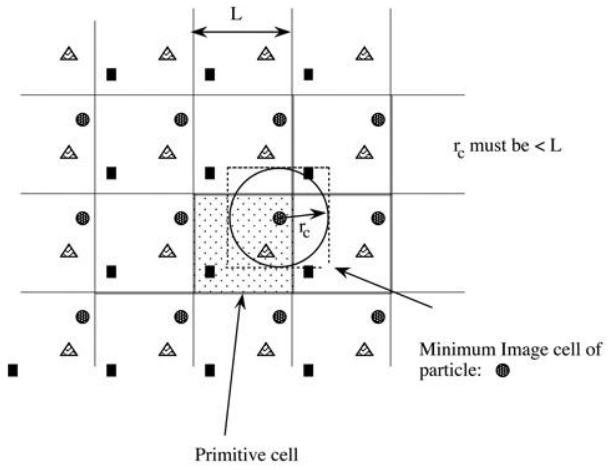

Figure A –This figure displays the periodic boundary conditions used in molecular dynamics simulations. The shaded box represents the simulation region. The dotted box represents the minimum image region, showing which particles actually have an effect[2].

Figure A is a simple visualization of these boundary conditions.

The next trick used to reduce computation needs of these simulations is the Lennard – Jones Potential and cut – off potential. To conduct a true simulation, one must consider the interaction of every electron of every argon atoms and their effects of every other electron in the system. For just two argon atoms, one would need to consider over 1000 interactions between electrons and the nuclei. It is obviously impossible to compute all these interactions for a statistically relevant sample size. Instead, one must use the Lennard – Jones potential formula[3]:

$$u(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] \qquad (1)$$

$$f(r) = -\frac{du}{dr} = \frac{48\epsilon}{r}\left[\left(\frac{\sigma}{r}\right)^{12} - \frac{1}{2}\left(\frac{\sigma}{r}\right)^{6}\right] \qquad (2)$$

This formula allows one to calculate the effective potential between two atoms without needing to consider all the sub – atomic interactions. One must only specify the parameters σ and ϵ, which determine the length scale and strength of the interaction, respectively. With user controlled parameters, this form for potential is incredibly versatile. This form also successfully balances the attractive and repulsive components of the interactions, generating a very accurate potential simulation.

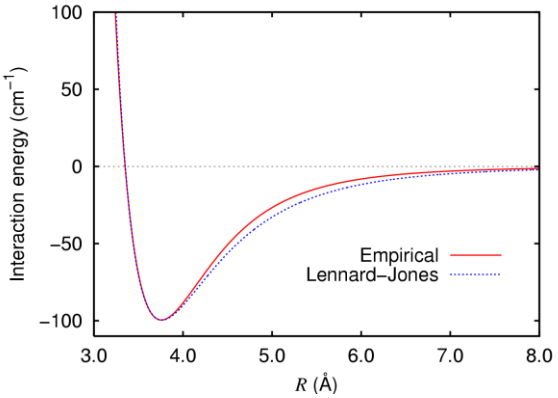

Figure B – This figure displays the Lennard – Jones Potential as a function of position in blue and empirically collected data in red. It is clear that the Lennard – Jones is a very accurate method for potential calculations[4].

The second part to the simplification of potential calculations is the implementation of a cutoff radius. Since the force between atoms becomes minimal at relatively small distances, a simulation can consider the potential to be zero beyond a certain distance specified by the user. Since this causes the derivative of the potential to be infinite, the force calculation at this point would be incorrect and thus the energy of the system would not be conserved. Since the force is so small, however, the amount of energy gained or lost should be negligible.

The final simplification used in this simulation is the Verlet Velocity Algorthim. This is based on the Verlet Algorithm, which is a simplified third order Taylor expansion[5]. This algorithm allows one to quickly calculate the position and velocity of a particle as it is affected by forces[6].

$$r(t+h) \cong r(t) + h * v(t) + \frac{h^2}{2} F(t) \quad (3)$$

$$v(t+h) \cong v(t) + h * \left[\frac{F(t+h)+F(t)}{2}\right] \quad (4)$$

Both the position and the velocity of the particle are dependent on the velocity and force of the particle in the previous time step. Using this algorithm and the previously mentioned methods, one can develop a relatively simple simulation for the behavior and interactions of a group of nearly infinite atoms.

3. Experimental Method

The first part of this assignment involved the simulation of a one – dimensional box of argon atoms. This simulation was conducted using the methods described above and successfully demonstrated this highly theoretical behavior.

Next, the significantly more difficult task of a two – dimensional system was attempted. Using the Verlet Velocity Algorthim, periodic boundary conditions, and the Lennard – Jones Potential, a system was successfully created and interactions simulated. The expected equilibrium state was not successfully reached however.

4. Data and Analysis

The first section of this lab explored a one – dimensional simulation of argon atoms. This simulation successfully demonstrated the behavior of particles in a one – dimensional box, however this situation is highly theoretical and not very useful. The starting and equilibrium positions of an eight atom simulation are shown in Table 1. The atoms shift a substantial amount but remained approximately one unit apart. This is the maximum distance they can all be from one another since the 'box' they are in is 8 units long.

Next, a two – dimensional simulation was conducted. This proved to be substantially more difficult than the one – dimensional version. There were many unforeseen pitfalls and problems. Eventually, a partially working simulation was developed. This system reached a sort of equilibrium, but not the one predicted. Ideally, all the atoms in a two – dimensional simulation would reach the points of a face – centered – cubic lattice. The actual results of this simulation are shown in Figure 2. The atoms appear to cluster into groups, indicating that the repulsive force was accounted for too much over the repulsive force of the atoms.

In addition to the final position not being correct, there was a problem with the energy calculation. The total energy, which is the sum of the kinetic and potential energy, should remain constant throughout the simulation.

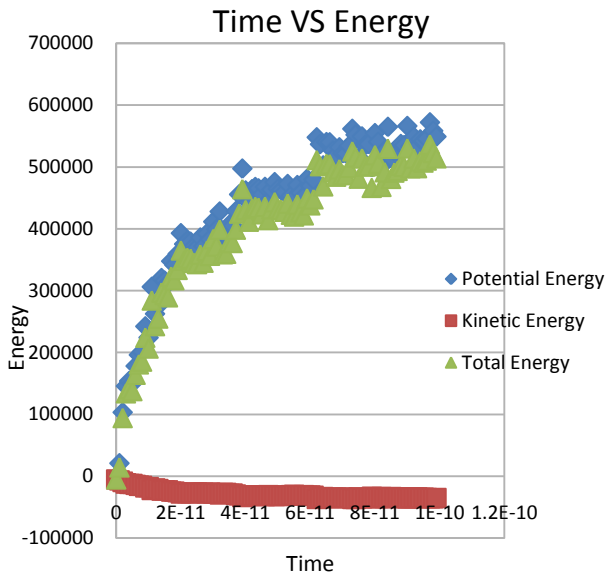

Figure B – This figure displays the change is potential, kinetic, and total energy versus time of the simulation described in Figure 1 and 2. The total energy of this system should be a constant, horizontal line that is the sum of the potential and kinetic energy. The kinetic energy changes significantly more than the potential.

Figure B shows that this is not correct. As the kinetic energy increases, the potential energy does not match its change. The temperature does follow the kinetic energy as was expected.

$$T = \frac{2(KE)}{3kN} \qquad (5)$$

The energy does appear to reach an equilibrium state, indicating that the error lies in the initial calculation and not the simulation.

Finally, the trajectory of a single atom is displayed in Figure 4. The atom that is being tracked begins in the center of the system and slowly works its way to the outer edge. From there it demonstrates the periodic boundary condition imposed with the vertical and horizontal lines across the plot where it exited one side and was entered back on the other.

5. Conclusion

This lab successfully demonstrated the applications of molecular dynamics simulations. First, a simple one dimensional simulation was conducted. Next, a two dimensional system was simulated unsuccessfully. Although this simulation was unsuccessful, the concepts were properly demonstrated.

6. Bibliography

[1]"Periodic Boundary Conditions." *Wikipedia, the Free Encyclopedia*. Web. 09 May 2010. <http://en.wikipedia.org/wiki/Periodic_boundary_conditions>.

[2]"Chapter 6. Computer Simulation Algorithms." *ANU E Press*. Web. 09 May 2010. <http://epress.anu.edu.au/sm/html/ch06.html>.

[3]"Lennard-Jones Potential." *Wikipedia, the Free Encyclopedia*. Web. 09 May 2010. <http://en.wikipedia.org/wiki/Lennard-Jones_potential>.

[4]"File:Argon Dimer Potential and Lennard-Jones.png." *Wikipedia, the Free Encyclopedia*. Web. 09 May 2010. <http://en.wikipedia.org/wiki/File:Argon_dimer_potential_and_Lennard-Jones.png>.

[5]"The Verlet Algorithm." *Department of Physics, University of Udine, Italy*. Web. 09 May 2010. <http://www.fisica.uniud.it/~ercolessi/md/md/node21.html>.

[6]"Velocity Verlet Algorithm." *Directory of Web Sites on Xbeams.chem.yale.edu*. Web. 09 May 2010. <http://xbeams.chem.yale.edu/~batista/vaa/node60.htm

APPENDIX A: TABLES AND FIGURES

Atom	Initial Position	Final Position
1	0	4.396752
2	1	5.396751
3	2	6.396775
4	3	7.396810
5	4	0.396822
6	5	1.396823
7	6	2.396799
8	7	1.396596

Table 1 – This table shows the initial and final positions of a one – dimensional simulation of eight atoms. The atoms have all shifted substantially from their starting positions and are approximately the same distance apart as they started.

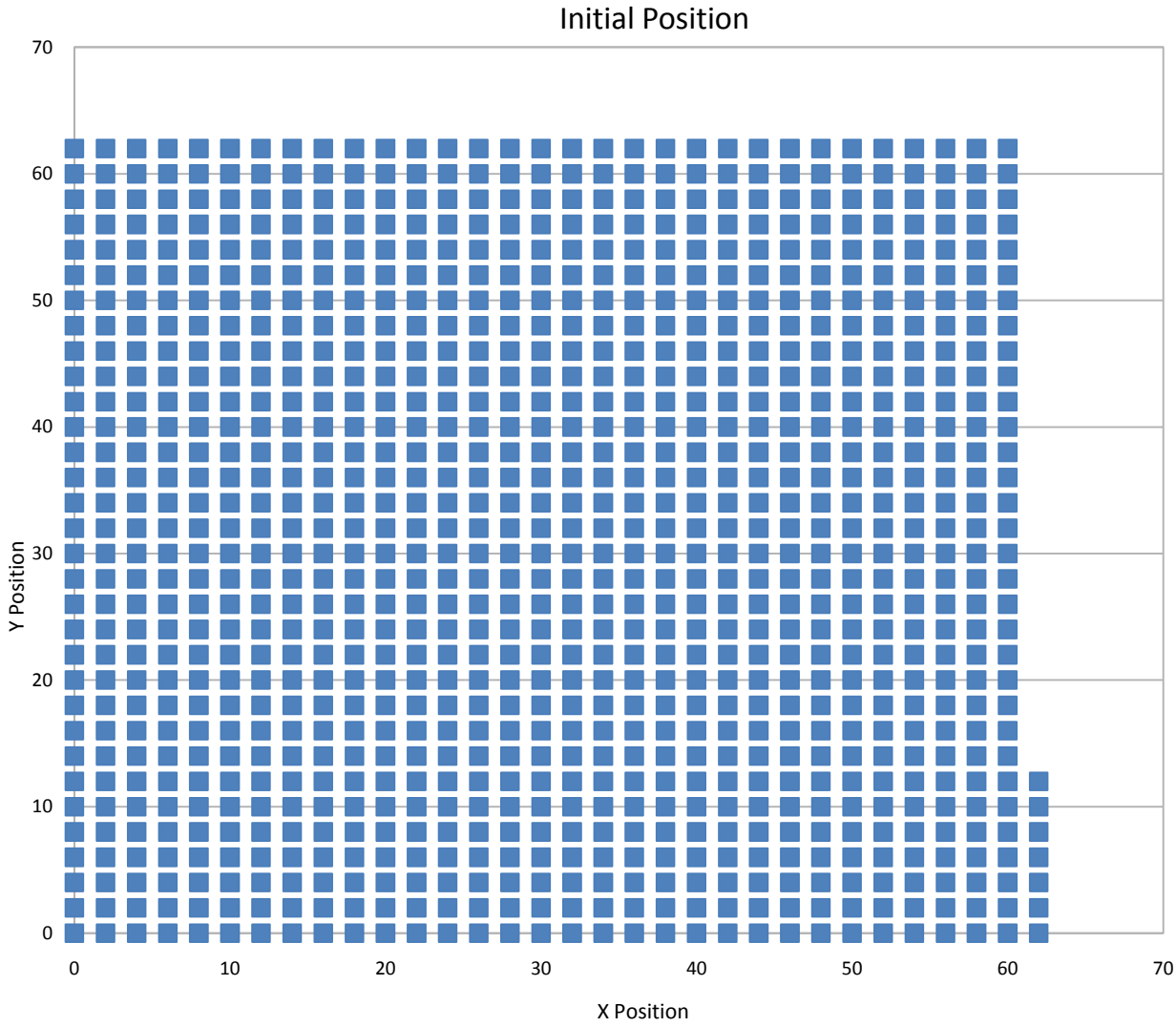

Figure 1 – This figure displays the initial positions of the argon atoms in a two – dimensional simulation of 1000 atoms. The atoms are initially spaced in a grid with two units between each atom.

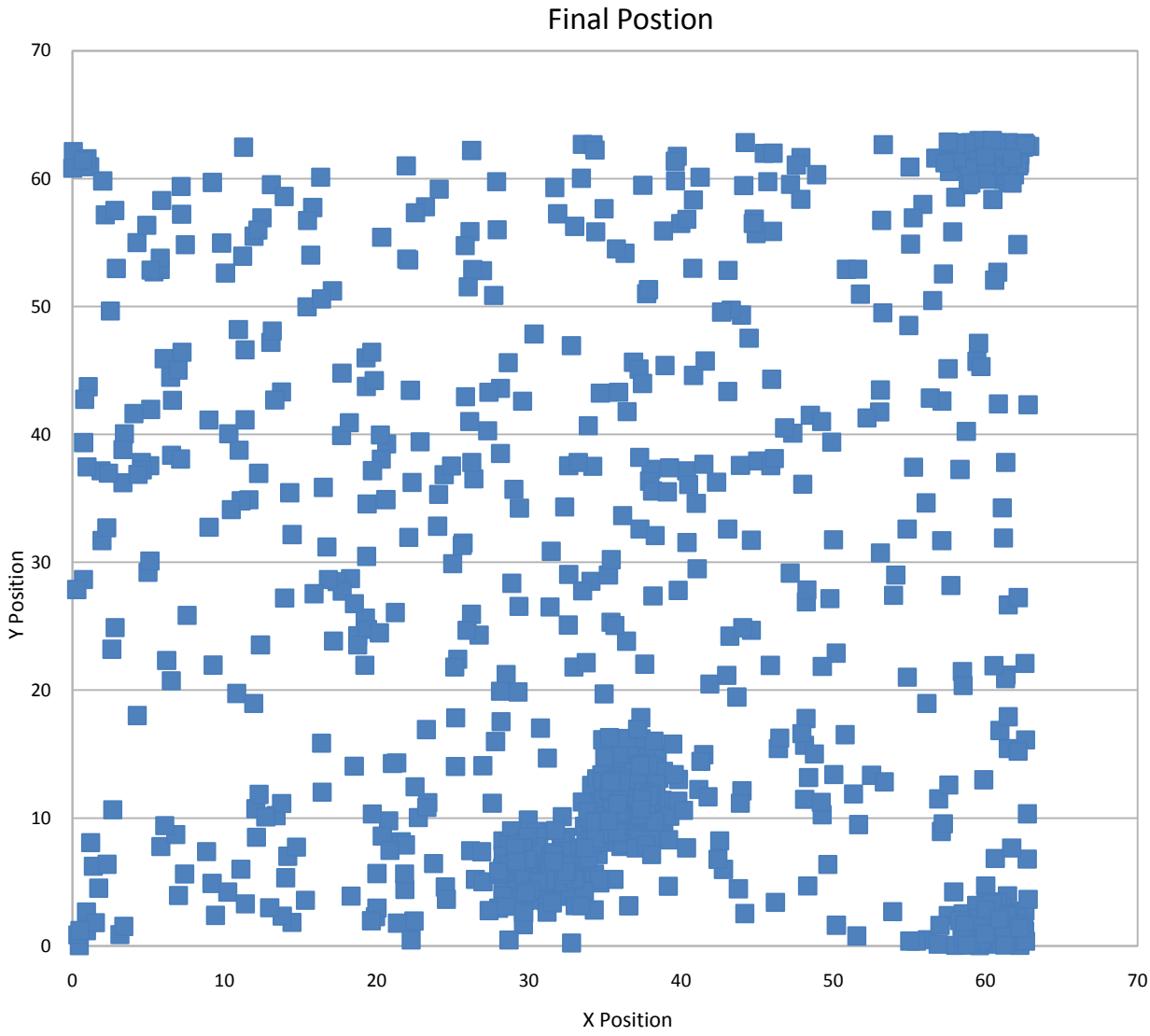

Figure 2 – This figure displays the final position of a simulation of 1000 atoms after 10^{-10} seconds. The atoms appear to cluster in an incorrect manner. The proper equilibrium position for this simulation is all atoms sitting in a face – centered – cubic lattice.

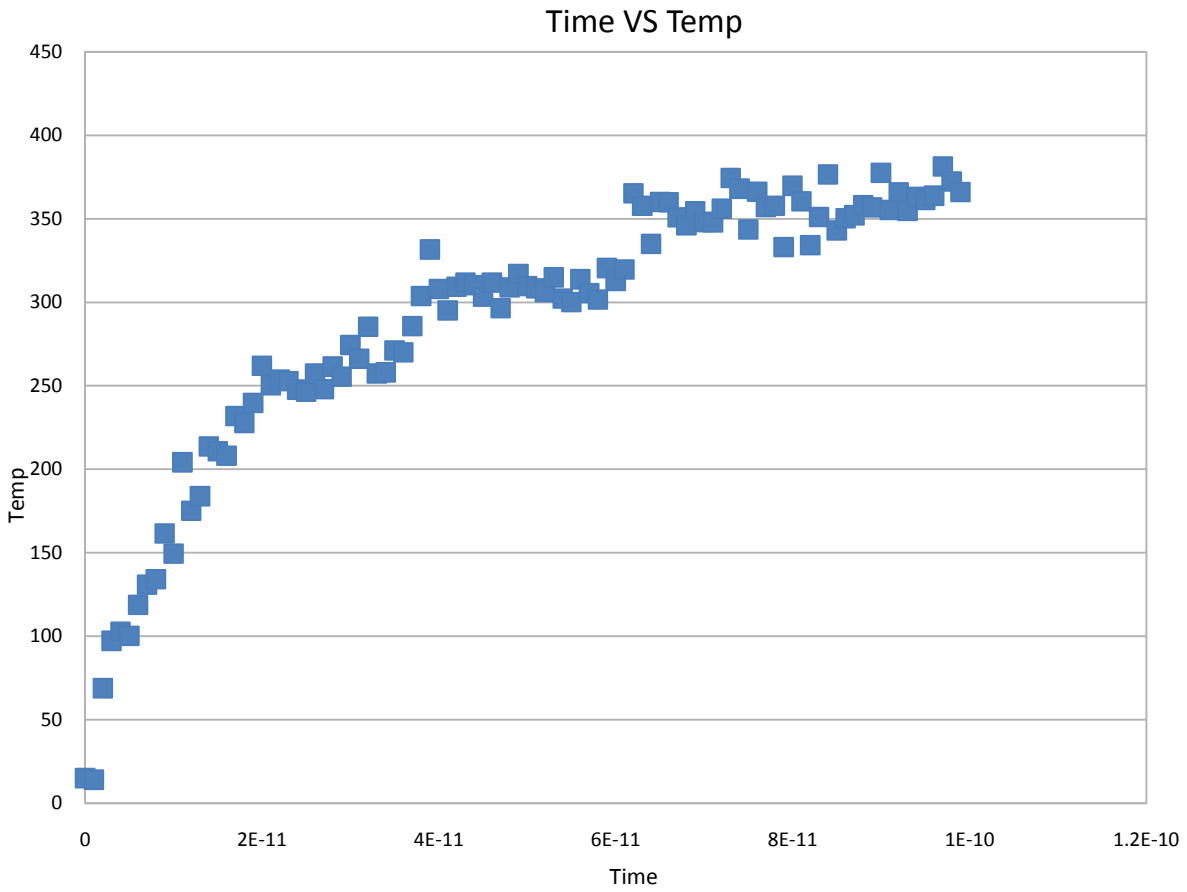

Figure 3 – This figure displays the plot of temperature versus time of the simulation described in Figure 1 and 2. As the velocity of the particles increases, the temperature does too, until leveling off between 350 and 400.

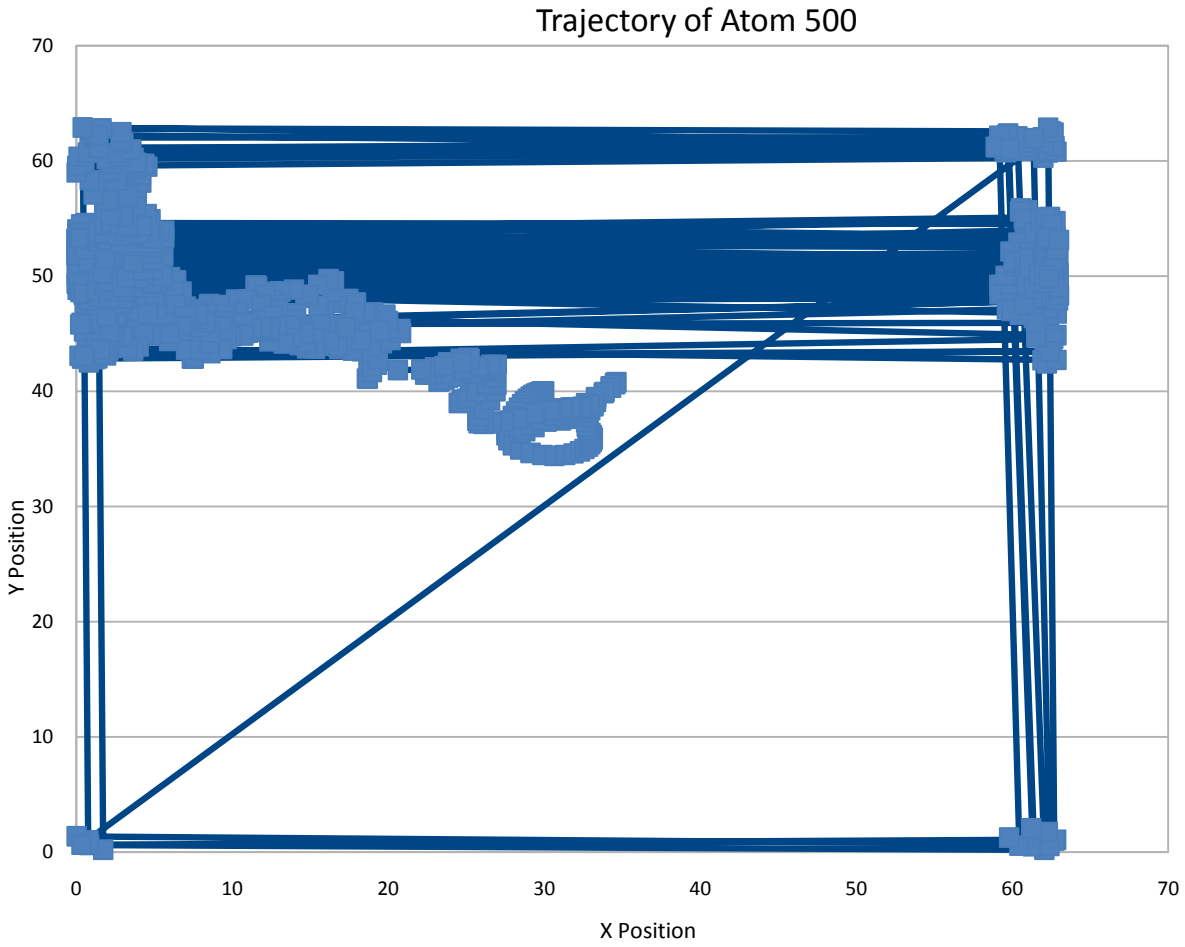

Figure 4 – This figure displays the trajectory of atom 500 in the simulation described in Figure 1 and 2. The atom starts in the center and works its way to the outer edge of the simulation region. The long lines crossing the plot indicate when the particle exited the region and entered on the opposite side. Note that, at one point, the particle exited in both the X and Y directions, and therefore crossed diagonally.

APPENDIX B: SOURCE CODE

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>

//*********************************\\
//Mitchell Miller                  \\
//PHYS 240                         \\
//Final Project 3/30/10              \\
//2 Dimensional Molecular Dynamics \\      **Currently 1 Dimensional Version as of 4/25/10**
//*********************************\\

//Declare global constants
#define nAtoms 8
#define nMax 513
#define drand48 1.0/RAND_MAX*rand
#define srand48 srand

double forces(int time, double potentialEnergy,double *x, double *forceX){

        //Intialize Variables
        int counter1,counter2,length,nDimensions=1;
        double deltaForce, radiusSquared, inverseRadiusSquared=0, radiusSquaredCutoff=9, dx;
        length = floor(pow(1.0*nAtoms, 1.0/nDimensions));
        double sign(double a, double b);

        //Intialize Forces
        potentialEnergy = 0.0;
        for(counter1=0;counter1<nAtoms;counter1++){

                *(forceX+counter1+nAtoms*time)=0;

        }

        //Calculate Forces
        for(counter1=0;counter1<nAtoms;counter1++){

                for(counter2=counter1+1; counter2<nAtoms; counter2++){

                        dx = *(x+counter1) - *(x+counter2);
```

```
                        if(fabs(dx) > 0.5*length){

                                dx = dx - sign(length,dx);

                        }

                        radiusSquared = pow(dx,2);

                        //Check for cutoff distance
                        if(radiusSquared < radiusSquaredCutoff){

                                if(radiusSquared == 0.0)        radiusSquared = 0.0001;
                                inverseRadiusSquared = 1.0 / radiusSquared;
                                deltaForce = (48.0 * (pow(inverseRadiusSquared,3) - 0.5) *
pow(inverseRadiusSquared,3)) * inverseRadiusSquared * dx;
                                *(forceX+counter1+time*nAtoms) =
*(forceX+counter1+time*nAtoms) + deltaForce;
                                *(forceX+counter2+time*nAtoms) =
*(forceX+counter2+time*nAtoms) - deltaForce;
                                potentialEnergy += 4 * pow(inverseRadiusSquared,3) *
(pow(inverseRadiusSquared,3) - 1);

                        }

                }

        }

        return potentialEnergy;

}

double sign(double a, double b){

        if(b >= 0.0)      return fabs(a);
        else      return -fabs(a);

}

int main(){

        //Intialize variables
```

```
        double *positionX, *forceXMain, velocityX[nAtoms];
        double timeStep=0.004, stepOver2=(timeStep/2.0), potentialEnergy=0, kineticEnergy=0,
temp, initialTemp=10;
        int counter=0, xCounter, time1=0, time2=1, dummyTime, timeMain=0,
numberOfSteps=10000, stepsBetweenPrint=100, nDimensions=1, length;
        length = floor(pow(1.0*nAtoms, 1.0/nDimensions));
        srand48(123456);
        FILE *outfile;
        outfile = fopen("final.1.1.txt","w");

        positionX = calloc(nAtoms,sizeof(double));
        forceXMain = calloc(nAtoms*2,sizeof(double));

        printf("nAtoms = %d\tLength = %d\n", nAtoms, length);


        for(xCounter=0; xCounter<length; xCounter++){

                //Set initial positions
                *(positionX+counter) = xCounter;
                printf("Intial X_%d = %f\n", counter, *(positionX+counter));

                //Set initial velocities with temperature scaling
                velocityX[counter] =((drand48()+drand48()+drand48()+
            drand48()+drand48()+drand48()+
            drand48()+drand48()+drand48()+
            drand48()+drand48()+drand48())/(12.0))-0.5;
                velocityX[counter] = velocityX[counter]*sqrt(initialTemp);
                //velocityX[counter] =
(((rand()+rand()+rand()+rand()+rand()+rand()+rand()+rand()+rand()+rand()+rand()+rand()) / (12
* RAND_MAX)) - 0.5) * sqrt(initialTemp);
                printf("Initial X Velocity %d = %f\n", counter, velocityX[counter]);

                counter++;

        }

        //Calculate inital Potential Energy
        potentialEnergy = forces(time1, potentialEnergy, positionX, forceXMain);
        for(counter=0; counter<nAtoms; counter++){

                kineticEnergy += (velocityX[counter]*velocityX[counter]) * 0.5;

        }
```

```
	printf("Time: %d\tPE = %f\tKE = %f\tTE = %f\n", timeMain,
potentialEnergy,kineticEnergy, potentialEnergy+kineticEnergy);

	//Main Loop
	for(timeMain=1;timeMain<numberOfSteps;timeMain++){

		for(counter=0;counter<nAtoms;counter++){

			potentialEnergy = forces(time1, potentialEnergy, positionX, forceXMain);
			//printf("PE = %lf\n",potentialEnergy);
			//printf("X = %lf\n",*(positionX+counter));
			*(positionX+counter) = *(positionX+counter) + timeStep *
(velocityX[counter] + stepOver2*(*(forceXMain+counter+time1*nAtoms)));
			//printf("Next X = %lf\n",*(positionX+counter));
			if (*(positionX+counter) <= 0.)		*(positionX+counter) =
*(positionX+counter) + length;
			if (*(positionX+counter) >= length)	*(positionX+counter) =
*(positionX+counter) - length;

		}

		potentialEnergy = forces(time2, potentialEnergy, positionX, forceXMain);
		kineticEnergy = 0;

		for(counter=0;counter<nAtoms;counter++){

			velocityX[counter] = velocityX[counter] + stepOver2 *
((*(forceXMain+counter+time1*nAtoms))+(*(forceXMain+counter+time2*nAtoms)));
			//printf("Velocity_%d = %lf\n",counter,velocityX[counter]);
			kineticEnergy = kineticEnergy +
(velocityX[counter]*velocityX[counter]/2);

		}

		temp = 2*kineticEnergy / (3*nAtoms);

		if(timeMain<100)	printf("Time: %d\tPE = %lf\tKE = %lf\tPE+KE =
%lf\n",timeMain,potentialEnergy,kineticEnergy,potentialEnergy+kineticEnergy);

		if(timeMain%stepsBetweenPrint == 0)		printf("Time: %d\tPE = %lf\tKE =
%lf\tPE+KE = %lf\n",timeMain,potentialEnergy,kineticEnergy,potentialEnergy+kineticEnergy);

		dummyTime = time1;
```

```
                time1 = time2;
                time2 = dummyTime;


        }
        for(counter=0;counter<nAtoms;counter++) printf("Final X_%d =
%lf\n",counter,*(positionX+counter));

        free(positionX);
        free(forceXMain);
        fclose(outfile);
   getchar();

}


#include<stdio.h>
#include<math.h>
#include<stdlib.h>

//*********************************\\
//Mitchell Miller                  \\
//PHYS 240                         \\
//Final Project 3/30/10              \\
//2 Dimensional Molecular Dynamics \\
//*********************************\\

//Declare global constants
#define nAtoms 1000
#define nMax 513
#define Pi 3.14159265358979323846264338327950288419716939937510582097 49
#define argonRadius 1.1225
#define nDimensions 2
#define drand48 1.0/RAND_MAX*rand
#define srand48 srand

//Forces function for calculating forces and
//potential energy with Lennard-Jones Potential

//
//            [  1      1  ]
//  U(r) = 4 * | ---  -  --- |
//            [ r^12    r^6 ]
//
```

```
double forces(int time, double length, double potentialEnergy, double x[nAtoms], double
forceX[nAtoms][2], double y[nAtoms], double forceY[nAtoms][2]){

	//Intialize Variables
	int counter1,counter2;
	double deltaForceX, deltaForceY, radiusSquared, inverseRadiusSquared=0,
radiusSquaredCutoff=20, radius, dx, dy;

	double sign(double a, double b);

	//Intialize Potential Energy and Forces
	potentialEnergy = 0.0;
	for(counter1=0;counter1<nAtoms;counter1++){

		forceX[counter1][time]=0;
		forceY[counter1][time]=0;

	}

	//Calculate Forces
	for(counter1=0;counter1<nAtoms;counter1++){

		for(counter2=counter1+1; counter2<nAtoms; counter2++){

			dx = x[counter2] - x[counter1];
			dy = y[counter2] - y[counter1];

			//Minimum image condition
	if(fabs(dx) > 0.5*length){

				dx = dx - sign(length,dx);

			}

			if(fabs(dy) > 0.5*length){

				dy = dy - sign(length,dy);

			}


			radiusSquared = dx*dx + dy*dy;
			radius = sqrt(radiusSquared);
```

```
                        //Check for cutoff distance
                        if(radiusSquared < radiusSquaredCutoff){

                                //Check for overlapping atoms
        if(radiusSquared == 0.0){

                                        dx = dx * argonRadius * cos(drand48());
                                        dy = dy * argonRadius * cos(drand48());
                                        radiusSquared = argonRadius;

                                }
                                else{

                                        dx = dx * sqrt(argonRadius/radiusSquared);
                                        dy = dy * sqrt(argonRadius/radiusSquared);
                                        radiusSquared = argonRadius;

                                }

                                inverseRadiusSquared = 1.0 / radiusSquared;

                                //Calculate change in force based on Lennard - Jones Potential
derivative
        deltaForceX = 48.0 * (pow(inverseRadiusSquared,3) - 0.5) *
pow(inverseRadiusSquared,3) * dx;
                                deltaForceY = 48.0 * (pow(inverseRadiusSquared,3) - 0.5) *
pow(inverseRadiusSquared,3) * dy;

                                forceX[counter1][time] = forceX[counter1][time] + deltaForceX;
                                forceX[counter2][time] = forceX[counter2][time] - deltaForceX;
                                forceY[counter1][time] = forceY[counter1][time] + deltaForceY;
                                forceY[counter2][time] = forceY[counter2][time] - deltaForceY;

                                //Lennard - Jones Potential
        potentialEnergy += 4 * pow(inverseRadiusSquared,3) * (pow(inverseRadiusSquared,3)
- 1);

                        }

                }

        }
```

```
	return potentialEnergy;

}

double sign(double a, double b){

	if(b >= 0.0)	return fabs(a);
	else	return -fabs(a);

}

int main(){

	//Intialize variables
	double positionX[nAtoms], forceXMain[nAtoms][2], velocityX[nAtoms];
	double positionY[nAtoms], forceYMain[nAtoms][2], velocityY[nAtoms];
	double velocityTotal[nAtoms];
	double timeStep=0.004, stepOver2=(timeStep/2.0), potentialEnergy=0, kineticEnergy=0,
temp, initialTemp=15, length;
	int counter=0, xCounter, yCounter, time1=0, time2=1, dummy, timeMain=0,
numberOfSteps=10000, stepsBetweenPrint=100, stop=0;

	if(floor(pow(1.0*nAtoms, 1.0/nDimensions)) == pow(1.0*nAtoms, 1.0/nDimensions))
	length = 2*pow(1.0*nAtoms, 1.0/nDimensions);
	else	length = 2*floor(pow(1.0*nAtoms, 1.0/nDimensions))+1;

	srand48(123456);

	FILE *outfile1;
	FILE *outfile2;
	outfile1 = fopen("final.2.2.txt","w");
	outfile2 = fopen("final.2.2.path.txt","w");


	printf("nAtoms = %d\tLength = %lf\n", nAtoms, length);


	for(xCounter=0; xCounter<length; xCounter+=2){

		for(yCounter=0; yCounter<length; yCounter+=2){

			//Set initial positions
			positionX[counter] = xCounter;
			positionY[counter] = yCounter;
```

```
                        printf("Intial X_%d = %f, Inital Y_%d = %f\n", counter, positionX[counter],
counter, positionY[counter]);

                        //Set initial velocities with temperature scaling
                        velocityX[counter]
=((drand48()+drand48()+drand48()+drand48()+drand48()+drand48()+drand48()+drand48()+dra
nd48()+drand48()+drand48()+drand48())/(12.0))-0.5;
                        velocityX[counter] = velocityX[counter]*sqrt(initialTemp);
                        printf("Initial X Velocity %d = %f\n", counter, velocityX[counter]);

                        velocityY[counter]
=((drand48()+drand48()+drand48()+drand48()+drand48()+drand48()+drand48()+drand48()+dra
nd48()+drand48()+drand48()+drand48())/(12.0))-0.5;
                        velocityY[counter] = velocityY[counter]*sqrt(initialTemp);
                        printf("Initial Y Velocity %d = %f\n", counter, velocityY[counter]);

                        velocityTotal[counter] =
sqrt(velocityX[counter]*velocityX[counter]+velocityY[counter]*velocityY[counter]);

                        fprintf(outfile1, "%lf\t%lf\n", positionX[counter],positionY[counter]);

                        counter++;
                        if(counter>=nAtoms-1){
                                stop = 1;
                                break;
                        }

                }

                if(stop==1)     break;
        }

        fprintf(outfile1,"\n\n\n");

        //Calculate inital Potential and Kinetic Energy
        potentialEnergy = forces(time1, length, potentialEnergy, positionX, forceXMain,
positionY, forceYMain);

        for(counter=0; counter<nAtoms; counter++){

                velocityTotal[counter] =
sqrt(velocityX[counter]*velocityX[counter]+velocityY[counter]*velocityY[counter]);
                kineticEnergy += (velocityTotal[counter]*velocityTotal[counter]) * 0.5;
```

```
	}

	printf("Time:   0\tPE = %f\tKE = %f\tTE = %f\n", potentialEnergy, kineticEnergy,
potentialEnergy+kineticEnergy);
	fprintf(outfile1, "%d\t%lf\t%lf\t%lf\t%lf\n", timeMain, initialTemp, kineticEnergy,
potentialEnergy, kineticEnergy+potentialEnergy);
	fprintf(outfile2, "%lf\t%lf\n", positionX[500], positionY[500]);

	//Main Loop
	for(timeMain=1;timeMain<numberOfSteps;timeMain++){

		//Check for collisions
		/*for(xCounter=0;xCounter<nAtoms;xCounter++){

			for(yCounter=xCounter+1;yCounter<nAtoms;yCounter++){

				if(*(positionX+xCounter)==*(positionX+yCounter) &&
*(positionY+xCounter)==*(positionY+yCounter)){

					dummy = velocityX[xCounter];
					velocityX[xCounter] = velocityX[yCounter];
					velocityX[yCounter] = dummy;

					dummy = velocityY[xCounter];
					velocityY[xCounter] = velocityY[yCounter];
					velocityY[yCounter] = dummy;

				}

			}

		}*/

		//Change position
		for(counter=0;counter<nAtoms;counter++){

			positionX[counter] = positionX[counter] + timeStep * (velocityX[counter]
+ stepOver2*(forceXMain[counter][time1]));
			positionY[counter] = positionY[counter] + timeStep * (velocityY[counter]
+ stepOver2*(forceYMain[counter][time1]));

			if (positionX[counter] <= 0.)		positionX[counter] =
positionX[counter] + length;
```

```
                        if (positionX[counter] >= length)       positionX[counter] =
positionX[counter] - length;
                        if (positionY[counter] <= 0.)           positionY[counter] =
positionY[counter] + length;
                        if (positionY[counter] >= length)       positionY[counter] =
positionY[counter] - length;

                }


                potentialEnergy = forces(time2, length, potentialEnergy, positionX, forceXMain,
positionY, forceYMain);
                kineticEnergy = 0;

                //Change velocity
                for(counter=0;counter<nAtoms;counter++){

                        velocityX[counter] = velocityX[counter] + stepOver2 *
(forceXMain[counter][time1]+forceXMain[counter][time2]);
                        velocityY[counter] = velocityY[counter] + stepOver2 *
(forceYMain[counter][time1]+forceYMain[counter][time2]);

                        velocityTotal[counter] =
sqrt(velocityX[counter]*velocityX[counter]+velocityY[counter]*velocityY[counter]);

                        kineticEnergy +=
(velocityX[counter]*velocityX[counter]+velocityY[counter]*velocityY[counter]) * 0.5;

                }

                temp = 2*kineticEnergy / (3*nAtoms);

                //Troubleshoot print loop
                //if(timeMain<100)     printf("Time: %3.d\tPE = %lf\tKE = %lf\tPE+KE =
%lf\n",timeMain,potentialEnergy,kineticEnergy,potentialEnergy+kineticEnergy);

                //Main print loop
                if(timeMain%stepsBetweenPrint == 0){

                        printf("Time: %d\tPE = %lf\tKE = %lf\tPE+KE =
%lf\n",timeMain,potentialEnergy,kineticEnergy,potentialEnergy+kineticEnergy);
                        fprintf(outfile1, "%d\t%lf\t%lf\t%lf\t%lf\n",timeMain, temp,
kineticEnergy, potentialEnergy, kineticEnergy+potentialEnergy);
```

```
                }

                //Trajectory print loop
                if(timeMain%10 == 0) fprintf(outfile2, "%lf\t%lf\n", positionX[500],
positionY[500]);

                dummy = time1;
                time1 = time2;
                time2 = dummy;


        }

        fprintf(outfile1, "\n\n\n");

        for(counter=0;counter<nAtoms;counter++)  fprintf(outfile1,
"%lf\t%lf\n",positionX[counter],positionY[counter]);

        printf("Final Temp = %lf\n",temp);

        fclose(outfile1);

}
```