2.37 Final Project: MD Simulation of Liquid Argon

Levi Lentz
Department of Mechanical Engineering
Massachusetts Institute of Technology
05/14/2013

Table of Contents

1.0 ABSTRACT **2**
2.0 INTRODUCTION **2**
3.0 CODE METHODOLOGY **2**
3.1 UNITS 3
3.2 CODE DESCRIPTION 3
3.2.1 INITIALIZATION 4
3.2.2 NEIGHBOR LIST 6
3.2.3 LENNARD-JONES FORCE AND POTENTIAL CALCULATION 7
3.2.4 VELOCITY VERLET INTEGRATION 9
3.2.5 EQUILIBRATION 10
3.2.6 PRODUCTION RUN 12
3.2.7 PROPERTY MEASUREMENT 13
3.2.8 ADDITIONAL CODE FEATURES 15
4.0 RESULTS **16**
4.1 INITIALIZATION 16
4.2 VELOCITY SCALING 17
4.3 ANDERSEN THERMOSTAT 20
4.4 OTHER TESTS 22
5.0 CONCLUSION **23**
APPENDIX: COMPLETE CODE **24**

1.0 Abstract

The code to be outlined in this report was programmed in C and models liquid-phase Argon. The potential was modeled as a classic Lennard-Jones potential with a cutoff radius, long-range correction, and Andersen Thermostat implemented. For a temperature of 140K and a density of $2.8\times10^{28}/\mathrm{m}^3$, the code found a pressure of 181-183MPa as well as a self-diffusion coefficient of $\sim 2-6\times10^{-9}\mathrm{m}^2/\mathrm{s}$. The internal energies were found to be within 5% of the Monte Carlo reference values. Additionally, the average fluctuations given by the created code were of the order 10^{-4}, showing a very strong convergence. The code also creates animations of the molecular movement process. This report, code, and animations can be found online at http://www.levilentz.com/2.37/

2.0 Introduction

For the final project of 2.37 at MIT, each student was tasked with creating a molecular dynamic simulation that would model liquid argon at 140K with a number density of $n = 2.8\times10^{28}/\mathrm{m}^3$. The project approximates the interactions between neighboring atoms with a classical Lennard-Jones potential. Molecular Dynamics applied in this fashion is a very powerful tool to predict properties of atomic systems by modeling the motion of each atom as classical Newtonian Particles. A complete understanding of fundamental physics as well as computational methods are required to accurately complete this project, reinforcing the methods taught throughout 2.37.

3.0 Code Methodology

3.1 Lennard-Jones Model

The Lennard-Jones model is a well-established model for calculating the potential energy and force between two atoms. A simple summation over all interacting atoms gives the total potential energy and force on an atom. The model can be described below:

$$U_i = 4\varepsilon \sum_{i=0}^{N-1} \sum_{j=i+1}^{N} \left[\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6} \right]$$

$$F_i = \frac{dU_i}{dr} = -24\epsilon \sum_{i=0}^{N-1} \sum_{j=i+1}^{N} \left[\left(2\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6} \right) \frac{1}{r_{ij}} \right]$$

The limits on the summation terms have been expressed in such a way to save computational time by limiting the number of atoms required to loop over. It is worth noting that the above is a vector equation, along the vector r_{ij}.

3.1 Units

All equations were non-dimensionalized into so called Lennard-Jones Units. This is used so as to check the output of the code at various stages: in non-dimensional terms, all expected values should be of order one. This also allows the code to be used on other atomic systems.

Below are the relative non-dimensional units for the Lennard-Jones model[1]:

$$L^* \equiv \frac{L}{\sigma}$$

$$\rho^* \equiv \frac{N\sigma^3}{V}$$

$$T^* \equiv T * \frac{k_b}{\varepsilon}$$

$$U^* \equiv \frac{U}{\varepsilon}$$

$$P^* \equiv P * \frac{\sigma^3}{\varepsilon}$$

$$t^* \equiv \frac{t}{\sigma}\sqrt{\frac{\varepsilon}{m}}$$

Because only one species is being modeled, a mass of 1.0 is assumed for all Newtonian equations. Additionally, the Equations of Motion (EOM) are integrated over a non-dimensional time-step corresponding to 1fs.

For the given values of T and n, this implies: $T^* = 1.171$ and $\rho^* = 0.904$. The code will also be checked against other known temperatures and densities to verify that it works over a wide range of values, and is not simply correctly predicting the properties for a single temperature and density.

3.2 Code Description

The code will follow the following flow-chart. Each main area has been functionalized so as to provide better readability to the end user. Each section will be explained in detail following subsections.

[1] http://www4.ncsu.edu/~franzen/public_html/CH795N/modules/ar_mod/comp_output.html

[2] Rapaport DC. (2004). *The Art of Molecular Dynamics Simulation.* 2nd ed. Cambridge University Press.

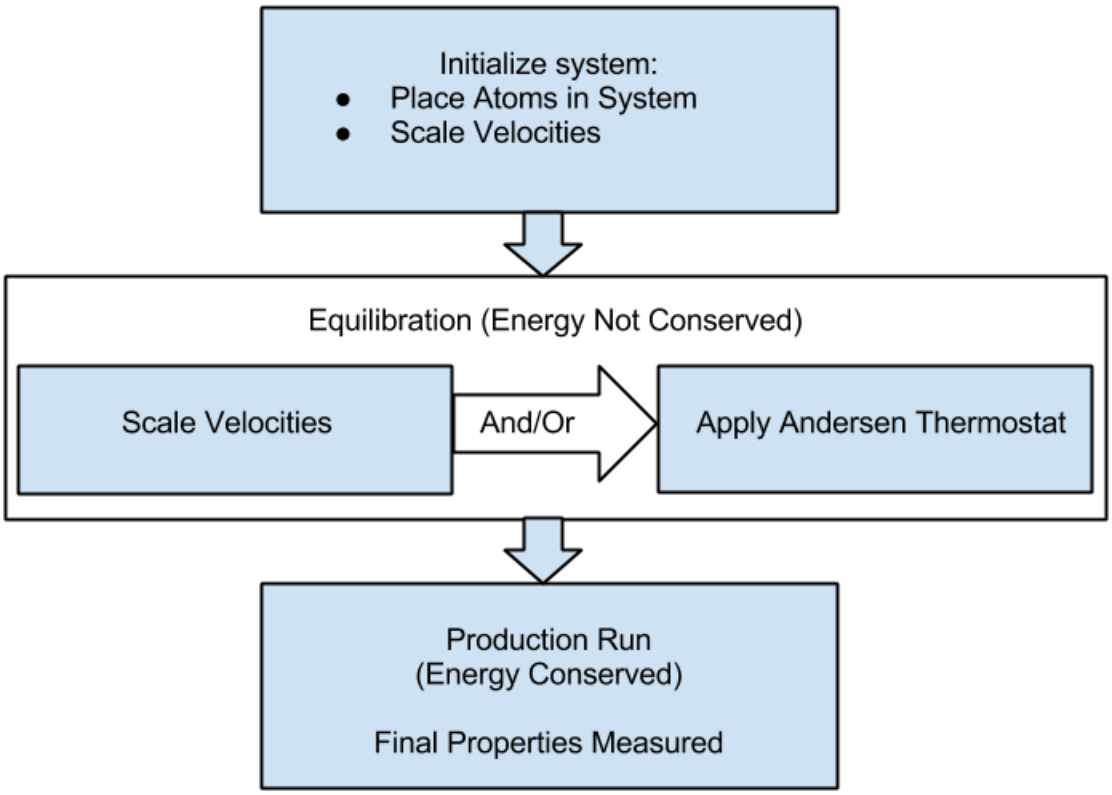

Figure 1. Flowchart of main components of the code.

3.2.1 Initialization

During initialization, the code places the N particles in a $n \times n \times n$ grid where $n = N^{1/3}$. The spacing between the atoms is governed by the given density of the system, where each side of the box is given by $L = \left(\frac{N}{\rho^*}\right)^{1/3}$. Each atom is then given a random velocity from the Maxwell distribution in the x,y, and z directions. Alternatively, the velocities can just as easily be chosen from a Gaussian distribution, as the final velocity will tend to the Maxwell-Boltzmann distribution. The velocities are then corrected for the center of mass drift and scaled to the proper temperature.

Below is the code for this step. The line numbers corresponds to the line numbers in the appendix.

```
242.    double random1() {
243.        return (double)random()/RAND_MAX;
244.    }
245.
246.    double maxwell() { //johnk's random number (from a maxwellian distribution) generator
247.    //referenced from: http://www.stat.rice.edu/~dobelman/textfiles/DistributionsHandbook.p
    df
248.        double x1, x2, w, y1, y2;
249.        do {
```

```
250.            x1 = 2.0 * random1() - 1.0;
251.            x2 = 2.0 * random1() - 1.0;
252.            w = x1 * x1 + x2 * x2;
253.        } while ( w >= 1.0 );
254.        w = sqrt( (-2.0 * log10( w ) ) / w );
255.        y1 = x1 * w;
256.        y2 = x2 * w;
257.        return y2;
258.
259.    }
260.
261.    double gauss(double std, double mean ) {
262.    //Used by Andersen (properly scaled velocities)
263.        double rand;
264.        rand = random1();
265.
266.        rand = mean + std*rand;
267.        return rand;
268.    }
269.
270.    void velscaler(double *velocity, double T, int n) {
271.    //Scales the velocity to proper temp, T.
272.        int i,j;
273.        double square_vel[nDim], velscale;
274.        for (i = 0; i<nDim; i++) {
275.            square_vel[i] = 0.0;
276.        }
277.
278.        for (i=0; i<n; i++) {
279.            for (j=0;j<nDim;j++) {
280.                square_vel[j] +=  *(velocity + j*n+ i) * *(velocity +j*n + i);
281.            }
282.        }
283.        velscale = sqrt(3*T/((square_vel[0]+square_vel[1]+square_vel[2])/n));
284.        for (i=0;i<n;i++) {
285.            for (j=0;j<nDim;j++) {
286.                *(velocity + j*n + i) *= velscale;
287.            }
288.        }
289.
290.    }
291.
292.    void initialization(double *positions, double *velocities, int n, double L, double T) {

293.        int N, i, j, k, flag;
294.        double del, ndouble, per_dimension, *sum_velocity, *squared_velocity, velscale, rcu
    toff;
295.        double r, tester;
296.        per_dimension = pow(n,1/3.0);
297.        rcutoff = 0.212; //Argon diameter divided by sigma
298.        del = L/per_dimension;
299.        //printf("%d %lf %lf\n", n, per_dimension, del);
300.        N = 0;
301.        flag = 0;
302.        sum_velocity = malloc(3*sizeof(double));
303.        squared_velocity = malloc(3*sizeof(double));
304.        for(i=0;i<nDim;i++) {
305.            *(sum_velocity + i) = 0.0;
306.            *(squared_velocity + i) = 0.0;
307.        }
308.
```

```
309.        for (i=0;i< per_dimension; i++) {
310.            for(j=0;j< per_dimension; j++) {
311.                for(k=0; k< per_dimension;k++) {
312.                    *(positions + 0*n + N) = i * del;
313.                    *(positions + 1*n + N) = j * del;
314.                    *(positions + 2*n + N) = k * del;
315.                    N++;
316.                }
317.            }
318.        }
319.
320.
321.        for(i=0; i<n; i++) {
322.            *(velocities + i) = maxwell();
323.            *(velocities + 1*n + i) = maxwell();
324.            *(velocities + 2*n + i) = maxwell();
325.            //below for calculating center of mass velocity
326.            *(sum_velocity) += *(velocities + i)/n;
327.            *(sum_velocity + 1) += *(velocities + 1*n + i)/n;
328.            *(sum_velocity + 2) += *(velocities + 2*n + i)/n;
329.        }
330.        //correcting for CM drift
331.        for (i=0;i<n;i++) {
332.            *(velocities + i) -= *(sum_velocity);
333.            *(velocities + 1*n + i) -= *(sum_velocity + 1);
334.            *(velocities + 2*n + i) -= *(sum_velocity + 2);
335.        }
336.        velscaler(velocities,T,n);
337.    }
```

3.2.2 Neighbor List

To speed up large calculations, a neighbor list has been implemented in the code. This is simply a $2xN_{MAX}$ matrix that tracks the nearest neighbors to every atom. Because this is not updated every time step, but rather ever ten time steps, it has a significant impact on the speed of the calculations.

The neighbor list is also responsible for ensuring that the periodic boundary condition is satisfied. The minimum image convention is used[2] where, if an atom is further than half the box length away, an image of the atom is moved by L to ensure the periodicity of the system. If the image or actual atom is in the cutoff radius, the pair is added to the neighbor list for force and potential calculation.

This is shown below:

```
338.
339.    int neighborhood(double *position, int *celllist, int n, double L, double rcut) {
340.    //Neighborhood list generation
341.        int i,j,k,xcell,ycell,zcell, Nmax;
342.        double x,y,z, halfL, distance[nDim], r;
343.        Nmax = 0;
344.        for (i = 0; i<n; i++) {
```

[2] Rapaport DC. (2004). *The Art of Molecular Dynamics Simulation.* 2nd ed. Cambridge University Press.

```
            for(j=i+1;j<n; j++) {
                r = 0.0;
                if(i == j) { break;}
                distance[0] = *(position + i) - *(position + j);
                distance[1] = *(position + n + i) - *(position + n + j);
                distance[2] = *(position + 2*n + i) - *(position + 2*n + j);
                //periodic boundary condition (minimum image convention):
                halfL = L/2.0;
                for (k=0;k<nDim;k++) {
                    if (distance[k] > halfL) {
                        distance[k] = distance[k] - L;
                    } else if( distance[k] < -halfL ) {
                        distance[k] = distance[k] + L;
                    }
                    r = r + distance[k]*distance[k];
                }
                r = sqrt(r);
                // generates a Nx2 matrix with the list of atoms
                //each row gives i,j corresponding to atoms near eachother
                if (r <= rcut) {
                    *(celllist + Nmax*2 + 1) = i;
                    *(celllist + Nmax*2 + 2) = j;
                    Nmax = Nmax + 1;
                }
            }
        }
        if (Nmax == 0 ) {
            printf("No near atoms\n");
            return 0;
        }
        return Nmax;
    }
```

3.2.3 Lennard-Jones Force and Potential Calculation

The Potential and Force are calculated in one function every time step. This function directly uses the neighbor list that was generated above.

Additionally, a switching function to truncate the potential and force on each atom. This ensures a smooth curve and continuous derivative at the cutoff, eliminating the need to add the cutoff potential to each atom. This function is shown below:

$$S(r)=\begin{cases} 1, & r < r_{max} \\ \dfrac{({r_{max}}^2 - r^2)^2({r_{max}}^2 + 2r^2 - 3r_{max}^2)}{(r_{max}^2 - r_{max}^2)^3}, & r_{cut} < r < r_{max} \\ 0, & r > r_{max} \end{cases}$$

This function simply times the normal Lennard-Jones potential. The force between the cutoff radius and the skin radius, or max radius, is then a longer differentiation because of the complexity of the switching function. The code for this calculation is shown below:

```

    void LJ_Forces( double *force, double *potential, double *position, int n, int Nmax, in
    t *celllist, double L, double rcut, double rmax, double *px){
```

```
379.        int i, j, k, x, y, z, num;
380.        double distance[nDim], halfL,ir6, U, F, r, r6, ir, r2, numerator, den, rmax2, rcut2
    , virial, tester;
381.        double ucut;
382.        ucut = 4*(1/(rcut*rcut*rcut*rcut*rcut*rcut))*((1/(rcut*rcut*rcut*rcut*rcut*rcut)) -
     1);
383.        virial = 0.0;
384.        memset(potential,0.0,n*sizeof(double));
385.        memset(force,0.0,nDim*n*sizeof(double));
386.        memset(px,0.0,nDim*sizeof(double));
387.
388.        virial = 0.0;
389.        halfL = L/2.0;
390.        for ( num = 0; num<Nmax; num++) {
391.                r = 0.0;
392.                i = *(celllist + num*2 + 1);
393.                j = *(celllist + num*2 + 2);
394.                if( i != j) {
395.                    distance[0] = *(position + i) - *(position + j);
396.                    distance[1] = *(position + n + i) - *(position + n + j);
397.                    distance[2] = *(position + 2*n + i) - *(position + 2*n + j);
398.                    for (k=0;k<nDim;k++) {
399.                        if (distance[k] > halfL) {
400.                            distance[k] = distance[k] - L;
401.                        }else if( distance[k] < -halfL ) {
402.                            distance[k] = distance[k] + L;
403.                        }
404.                    }
405.                    r2 = distance[0]*distance[0]+distance[1]*distance[1]+distance[2]*distan
    ce[2];
406.                    r = sqrt(r2);
407.                    //printf("%lf\n",r);
408.                    if (r <= rcut) {
409.                        ir = 1.0/r;
410.                        r6 = r2*r2*r2;
411.                        ir6 = 1.0/r6;
412.                        U = 4.0*ir6*(ir6-1.);
413.                        F = 48.0*(ir6*ir6-0.5*ir6);
414.                        //printf("%lf %lf %lf\n", r, U, F);
415.                        *(potential + i) += U;
416.                        for (k =0; k<nDim; k++) {
417.                            *(force + k*n + i) += F*distance[k]*ir*ir;
418.                            *(force + k*n + j) -
    = F*distance[k]*ir*ir; //equal and opposite forces
419.                        }
420.                        virial += F;
421.
422.                    }else if(r>rcut && r< rmax) {
423.                        r6 = r*r*r*r*r*r;
424.                        r2 = r*r;
425.                        rmax2 = rmax*rmax;
426.                        rcut2 = rcut*rcut;
427.                        numerator = 24.0*(rmax*rmax-r2)*(rmax2*rmax2*(r6-2.0)-rmax2*(r6-
    2.0)*(3.0*rcut2-r2)+rcut2*r2*(r6-4.0)+2.0*r2*r2);
428.                        den = r6*r6*r*(rmax2-rcut2)*(rmax2-rcut2)*(rmax2-rcut2);
429.                        ir6 = 1./r6;
430.                        U = 4.0*ir6*(ir6-1.0)*(rmax*rmax-r*r)*(rmax*rmax-
    r*r)*(rmax*rmax+2.0*r*r-3.0*rcut*rcut)/((rmax*rmax-rcut*rcut)*(rmax*rmax-
    rcut*rcut)*(rmax*rmax-rcut*rcut));
431.                        *(potential + i) += U;
432.                        F = -numerator/den;
```

```
                        for (k =0; k<nDim; k++) {
                            *(force + k*n + i) += F*distance[k]/r;
                            *(force + k*n + j) -
    = F*distance[k]/r; //equal and opposite forces
                        }
                    }

                }
        }
        *(px) = virial;
    }
```

3.2.4 Velocity Verlet Integration

This code utilizes a Velocity Verlet integration to integrate the equations of motion. This allows for the direct computation of the velocity and position of each particle at each time without having the need to track the particles at multiple time steps. This integration is shown below:

$$\boldsymbol{r}(t+\Delta t) = \boldsymbol{r}(t) + \boldsymbol{v}(t)\Delta t + \frac{\boldsymbol{F}(t)}{2m}\Delta t^2$$

$$\boldsymbol{v}(t+\Delta t) = \boldsymbol{v}(t) + \frac{\boldsymbol{F}(t) + \boldsymbol{F}(t+\Delta t)}{2m}\Delta t$$

To save on tracking the force at both t and $t + \Delta t$ this function is simply called twice each time step, once at to calculate the position at $t + \Delta t$, velocity at *t,* and then again to calculate the velocity at $t + \Delta t$. A force calculation is done between these steps to get the force at $t + \Delta t$. This function also tracks the particles as they move out of the box, correcting their new position to be back in our control volume. Additionally, this calculates the mean displacement in the array "tracker" that ignores movement across the periodic boundary. The tracker array will be used to calculate the self-diffusion coefficient to be described in following sections.

The function is called 'andersen' because a third switch is used to initiate the Andersen Thermostat, which will be covered in the next section.

```

    void andersen(int switch1, int n, double *force, double *velocity, double *positions,doubl
    e *tracer, double dt, double T, double L, int *blinker ) {
    //Andersen thermostat
        int i,j;
        double Temp, rand;
        double nu = 0.2; //collision with heat bath (needs to be a controllable variable)
        if (switch1 == 1) {
            for (i=0;i<n;i++) {
                for (j = 0; j<nDim; j++) {
                    *(positions + j*n + i ) += *(velocity +j*n + i)*dt + 0.5 * *(force +j*n
     + i)*dt*dt;
                    *(velocity + j*n + i ) += 0.5 * (*(force + j*n + i)) * dt;
                    *(tracer + j*n + i) += *(velocity +j*n + i)*dt + 0.5 * *(force +j*n + i
    )*dt*dt;
                    if( *(positions + j*n + i) < 0.0) {
                        *(positions + j*n + i) = *(positions + j*n + i) + L;
                        *(blinker + j*n + i) -= 1;
```

```
                    }
                    if( *(positions + j*n + i) > L ) {
                        *(positions + j*n + i) = *(positions + j*n + i) - L;
                        *(blinker + j*n + i) += 1;
                    }
                }
            }
        } else if(switch1 == 3 || switch1 == 2 ) {
            Temp = 0.0; //initialize temp
            for (i = 0;i<n;i++) {
                for (j=0;j<nDim;j++) {
                *(velocity + j*n + i ) = *(velocity + j*n + i ) + 0.5 * *(force + j*n + i)
    * dt;
                }
            }
            if(switch1 == 3) {
                for(i=0;i<n;i++) {
                    for(j=0;j<nDim;j++) {
                        rand = random1();
                        if(rand < nu*dt) {
                            //printf("%lf hit\n",rand);
                            *(velocity + j*n + i) = gauss(sqrt(T),0);
                        }
                    }
                }
            }
        }
    }
```

3.2.5 Equilibration

The initial configuration that the code places the atoms in is a FCC structure, therefore not a minimum energy configuration. Because of this, we have to let the system "melt" to a low energy configuration. During the equilibration process, only temperature is conserved, while energy is not conserved. This part of the code runs for a user-defined number of time steps. At each time step, the EOM are integrated using the Velocity Verlet integration, however because of the constant temperature nature, during these steps, the particles will not conform to Newtonian Physics as the velocity will be artificially changed. For testing purposes, the code implemented two equilibrium methods:

- Velocity Scaling
  - Every ten time-steps, the velocities are scaled to a velocity corresponding to the correct temperature. This method has the fastest convergence, but information is lost about how the velocity changes during the equilibrium time. This was shown above in the velscaler() function in section 3.2.1.
- Andersen Thermostat[3]
  - This is a stochastic process where the particles are randomly assigned a velocity from the Gaussian Distribution for temperature. Here we use a Gaussian Distribution because we do not scale the velocities at this step. This is as if the particle had a random collision with the heat bath.

[3] Hans C. Andersen "Molecular dynamics simulations at constant pressure and/or temperature", Journal of Chemical Physics 72, 2384-2393 (1980)

```
    initialization(positions,velocities,n,L,T);
    velocity_printer(velocityfp,velocities,n);
    position_printer(fp,positions,n,blinker,L,0);
    fclose(fp);
    fclose(velocityfp);
    melter = fopen("Melting.xyz","w");
    position_printer(melter,positions,n, blinker, L,1);
    nMax = neighborhood(positions,neighborlist,n,L,rmax);
    if (nMax == 0 ) {
        return 0;
    }
        LJ_Forces(force,potential,positions,n,nMax,neighborlist,L,rcut,rmax, pix);
        initend = clock();
        inittime = (double)(initend - initbegin) / CLOCKS_PER_SEC;
        timecounter = 0.0;
        neighborcounter = 0;
        if (strcmp(vyes,"Y") == 0) {
        scalebegin = clock();
        printf("\n\nMelting at constant Temp\n\n");
        fprintf(OUT,"\n\nMelting at constant Temp\n\n");
            for(i=0;i<cycles;i++) {
                neighborcounter += 1;
                andersen(1,n,force,velocities,positions,tracer, dt, T, L, blinker);
                LJ_Forces(force,potential,positions,n,nMax,neighborlist,L,rcut,rmax, pix);

                andersen(2,n,force,velocities,positions,tracer,dt, T, L, blinker);
                if (neighborcounter > 10) {
                    nMax = neighborhood(positions,neighborlist,n,L,rmax);
                    neighborcounter = 0;
                    velscaler(velocities,T,n);
                    position_printer(melter,positions,n, blinker, L,1);
                }
                timecounter += dt;
                properties(positions,tracer,velocities,force,potential,n,&U,&KE,&totale, av
    evel, &Temp,pix, &pressure, vol, rcut, Rho, L, &Ds, timecounter);
                printf("%lf %lf %lf %lf %lf %lf %lf\n", timecounter, KE, U, totale, pressur
    e, Ds, Temp);
                fprintf(OUT,"%lf %lf %lf %lf %lf %lf %lf\n", timecounter, KE, U, totale, pr
    essure, Ds, Temp);
            }
            velocityfp = fopen("Meltedvel","w");
            velocity_printer(velocityfp,velocities,n);
            fclose(velocityfp);
            sprintf(buffer,"%lf.xyz",dt*i);
            fp = fopen("melted.xyz","w");
            position_printer(fp,positions,n,blinker,L,0);
            fclose(fp);
        scaleend = clock();
        veltime = (double)(scaleend - scalebegin) / CLOCKS_PER_SEC;
        }
        if( strcmp(yes,"Y") == 0 ) {
        andersenbegin = clock();
            printf("\n\nAndersen Thermostat\n\n");
            fprintf(OUT,"\n\nAndersen Thermostat\n\n");
            velocityfp = fopen("Andersenvel","w");
            neighborcounter = 0;
            for(i=0;i<cycles;i++) {
                neighborcounter += 1;
                andersen(1,n,force,velocities,positions,tracer, dt, T, L, blinker);
                LJ_Forces(force,potential,positions,n,nMax,neighborlist,L,rcut,rmax, pix);
```

```
145.                andersen(3,n,force,velocities,positions,tracer,dt, T, L, blinker);
146.                if ((i % 10) == 0) {
147.                    nMax = neighborhood(positions,neighborlist,n,L,rmax);
148.                    neighborcounter = 0;
149.                    position_printer(melter,positions,n, blinker, L,1);
150.                }
151.                timecounter += dt;
152.                properties(positions,tracer,velocities,force,potential,n,&U,&KE,&totale, av
    evel, &Temp,pix, &pressure, vol, rcut, Rho, L, &Ds, timecounter);
153.                printf("%lf %lf %lf %lf %lf %lf %lf\n", timecounter, KE, U, totale, pressur
    e, Ds, Temp);
154.                fprintf(OUT,"%lf %lf %lf %lf %lf %lf %lf\n", timecounter, KE, U, totale, pr
    essure, Ds, Temp);
155.
156.            }
157.            velocity_printer(velocityfp,velocities,n);
158.            fclose(velocityfp);
159.        andersenend = clock();
160.        andersentime = (double)(andersenend - andersenbegin) / CLOCKS_PER_SEC;
161.        }
```

3.2.6 Production Run

The production run is where all the final properties will be calculated and measured. Time dependent properties, such as the self-diffusion coefficient, cannot be accurately calculated during the equilibrium phase because the EOM are not properly conserved. The production run operates under the same methodology as above by integrating the EOM, however the velocity corrections are not applied. This properly conserves energy and accurately models the behavior of argon atoms. The production run can be run as long as desired, as the system is stable during production; this code uses 10000 time steps in the production. This is shown below.

```
162.
163.        productionbegin = clock();
164.        memset(tracer,0.0,nDim*n*sizeof(double));
165.        cycles = 10000;
166.        tracker = fopen("tracking","w");
167.        fprintf(tracker,"0 0 0\n");
168.        neighborcounter = 0;
169.        aveke=avepe=avetotal=aveT=aveP=aveDs=0.0;
170.        printf("\n\nNVE Production\n\n");
171.        fprintf(OUT,"\n\nNVE Production\n\n");
172.        velocityfp = fopen("FinalVel","w");
173.        for(i=0;i<cycles;i++) {
174.            timecounter += dt;
175.            neighborcounter += 1;
176.            andersen(1,n,force,velocities,positions,tracer,dt, T, L, blinker);
177.            fprintf(tracker,"%lf %lf %lf\n", *(tracer + track), *(tracer + n + track), *(tr
    acer + 2*n + track));
178.            LJ_Forces(force,potential,positions,n,nMax,neighborlist,L,rcut,rmax,pix);
179.            andersen(2,n,force,velocities,positions,tracer,dt, T, L, blinker);
180.            if ((i % 10) == 0) {
181.                nMax = neighborhood(positions,neighborlist,n,L,rmax);
182.                neighborcounter = 0;
183.                position_printer(melter,positions,n, blinker, L,1);
184.            }
185.            timecounter += dt;
```

```
186.            properties(positions,tracer,velocities,force,potential,n,&U,&KE,&totale, avevel
    , &Temp,pix, &pressure, vol, rcut, Rho, L, &Ds, timecounter);
187.            printf("%lf %lf %lf %lf %lf %lf %lf\n", timecounter, KE, U, totale, pressure, D
    s, Temp);
188.            fprintf(OUT,"%lf %lf %lf %lf %lf %lf %lf\n", timecounter, KE, U, totale, pressu
    re, Ds, Temp);
189.            aveke += KE;
190.            avepe += U;
191.            avetotal += totale;
192.            aveP += pressure;
193.            aveT += Temp;
194.            aveDs += Ds;
195.        }
196.        productionend = clock();
197.        productiontime = (double)(productionend - productionbegin) / CLOCKS_PER_SEC;
198.        velocity_printer(velocityfp,velocities,n);
```

3.2.7 Property Measurement

At each time-step, the code measures the properties of the system to check for equilibrium and ensure convergence. It directly calculates the kinetic energy, potential energy, temperature, mean square displacement, and pressure.

During the property measurement routine, the long range corrections for the potential and pressure are computed and added to the final properties[4]:

$$u_{lr} = 2\pi \int_{r_c}^{\infty} r^2 u_{LJ}(r) dr = 8\pi \left(\frac{1}{9r_c^9} - \frac{1}{3r_c^3} \right)$$

$$P_{lr} = -\frac{2\pi}{3} \int_{r_c}^{\infty} r^3 \frac{du_{LJ}}{dr} dr = 16\pi \left(\frac{2}{9r_c^9} - \frac{1}{3r_c^3} \right)$$

All properties measured, where applicable, are in per element terms. To get the system properties, one needs only times by the number of elements that you have in your system.

The two non-obvious calculations that it performs are for the pressure and the self-diffusion coefficient.

The pressure is calculated using the virial portion of the pressure equation. The force is simply based on the Lennard-Jones model; hence the virial[5] can be written as:

$$Virial = \frac{1}{d} \left\langle \sum_{i}^{N-1} \sum_{j>i}^{N} r_{ij} \frac{dU}{dr}\Big|_{r_{ij}} \right\rangle$$

Where *d* is the dimensionality of the system (2 or 3). Because of this definition, the virial can readily take into account the periodicity of the system as well as any other cutoff methods used

[4] http://www.cstl.nist.gov/srs/LJ_PURE/
[5] http://www.fisica.uniud.it/~ercolessi/md/md/node37.html

in the Lennard-Jones model. Virial is calculated in the Force Calculation routine and passed to the properties where the pressure is measured as follows:

$$P^* = \frac{1}{V^*}(NT^* - Virial) + P^*_{lr}$$

Additionally, the code calculates the mean square displacement for the self-diffusion coefficient. The self-diffusion coefficient can be calculated by[6]:

$$D = \lim_{t\to\infty} \frac{1}{6t} \langle |\boldsymbol{r}(t) - \boldsymbol{r}(0)|^2 \rangle$$

Because of the vector nature of the radius, the delta-*r* is tracked at each time step, squared, and averaged. This also prevents errors in the measurement of the diffusion coefficient due to particles "jumping" across the periodic boundaries. At the end of the production run, the MSD is divided by the time to find the total diffusion coefficient.

```
485.
486.    void properties(double *position, double *tracer, double *velocities, double *forces, d
    ouble *potential, int n, double *U, double *KE, double *energy, double *avevel, double *Te
    mp, double *px, double *pressure, double v, double rcut, double rho, double L, double *Ds,
     double timecount) {
487.        int i,j,k;
488.        double fx, fy, fz, radius, radiust, radiusz, virial;
489.        double distance[3], F, radius6, radius12, ucor, pcor, rcut9, rcut3, halfL;
490.        rcut3 = rcut*rcut*rcut;
491.        rcut9 = rcut3*rcut3*rcut3;
492.        ucor = 8.0*3.1415926*rho*(1.0/(9.0*rcut9)-1.0/(3.0*rcut3));
493.        pcor = 16.0*3.1415926*rho*(2.0/(9.0*rcut9)-1.0/(3.0*rcut3));
494.        halfL = L/2.0;
495.        *U = 0.0;
496.        *KE = 0.0;
497.        *(avevel) = 0.0;
498.        *(avevel + 1 ) = 0.0;
499.        *(avevel + 2) = 0.0;
500.        radiust = 0.0;
501.        *Ds = 0.0;
502.        for (j=0;j<n;j++) {
503.            *(avevel) += *(velocities + j);
504.            *(avevel + 1 ) += *(velocities + n + j);
505.            *(avevel + 2) += *(velocities + 2*n + j);
506.            *U += *(potential + j);
507.            *Ds += *(tracer + j) * *(tracer + j) + *(tracer + n + j) * *(tracer + n +j) + *
    (tracer + 2*n + j) * *(tracer + 2*n +j);
508.
509.        }
510.        *Ds = *Ds / (2*nDim*n) ;
511.        // Below is just a sanity check to make sure that we have no "Flying Ice"
512.        *(avevel) /= n;
513.        *(avevel + 1 ) /= n;
514.        *(avevel + 2) /= n;
515.        //Above should all be identically 0.0, andersen thermostat may negate this
516.        for(j = 0; j<n; j++) {
```

[6] R. A. Fisher & R. O. Watts, (1972) Calculated Self Diffusion Coefficients for Liquid Argon. *Aust J Phys,***25**, 529-38

```
517.            *KE += (*(velocities + j) - *(avevel)) * (*(velocities + j) -
    *(avevel)) + (*(velocities + n + j)- *(avevel + 1)) * (*(velocities + n + j)-
    *(avevel + 1)) + (*(velocities + 2*n + j)-*(avevel + 2)) * (*(velocities + 2*n + j)-
   *(avevel + 2));
518.        }
519.        *U /= n;
520.        *U += ucor;
521.        *KE = *KE/(2.0 * n);
522.        *Temp = (2.0/3.0) * *KE ;
523.        *energy = *KE + *U;
524.        *pressure = 0.0;
525.        *pressure = *px / 3.0;
526.        *pressure += n * *Temp;
527.        *pressure /= v;
528.        *pressure += pcor;
529.    }
```

3.2.8 Additional Code Features

For visualization purposes, the code creates an outputted .xyz file for the positions of all the atoms at each major code point (initial positions, melted positions, fully relaxed structure). These files can be readily ready by Avogadro, Vesta, VMD and other freely available molecular visualization viewers.

Additionally, it creates an .xyz file formatted in such a way to visualized the real time movements of the atoms. Once imported into VMD, animations of the particle movement can be created. Because of the periodic boundary condition, an additionally tracking matrix had to be implemented to stop blinking in the animation. This effect can be seen on the website referenced in the abstract.

The last visualization tool that it can trace a single particle in three dimensions to make a three dimension scatter plot so as to visualize how one atom moves in three dimensions across the run of the simulation.

These features are shown below.

```
530.
531.    void position_printer(FILE * fp, double * positions, int n, int *blinker, double L, int
     flag) {
532.        int i;
533.        double length;
534.        length = sigma*10.0/(1.0e-9); //converting distance to angstrom (for animation)
535.        fprintf(fp,"%d\n\n",n);
536.        if (flag == 1) { //animation
537.            for (i=0;i<n;i++) {
538.                fprintf(fp,"Ar %lf %lf %lf\n", *(positions + i)*length + *(blinker + i)*L*l
    ength, *(positions + n + i)*length + *(blinker + n + i)*L*length, *(positions + 2*n + i)*l
    ength + *(blinker + 2*n + i)*L*length);
539.            }
540.        } else if(flag == 0) {
541.            for (i=0;i<n;i++) {
542.                fprintf(fp,"Ar %lf %lf %lf\n", *(positions + i)*length, *(positions + n + i
    )*length, *(positions + 2*n + i)*length);
543.            }
544.        }
```

```
545.    }
546.
547.    void velocity_printer(FILE *fp, double * velocity, int n ) {
548.        int i;
549.        double Vx, Vy, Vz;
550.        double V;
551.        for(i=0;i<n;i++) {
552.            Vx = *(velocity + i) * *(velocity + i);
553.            Vy = *(velocity + n + i) * *(velocity + n + i);
554.            Vz = *(velocity + 2*n + i) * *(velocity + 2*n + i);
555.            V = sqrt(Vx + Vy + Vz);
556.            fprintf(fp,"%lf\n",V);
557.        }
558.    }
```

4.0 Results

Each run was done with 3375 atoms (15 atoms per side of the cube) with 3000 steps to equilibrate for the Velocity Scaling and 5000 time steps to equilibrate for the Andersen Thermostat. Another 10000 time steps during the production run. For data clarity, the results are shown until t* = 5.0.

4.1 Initialization

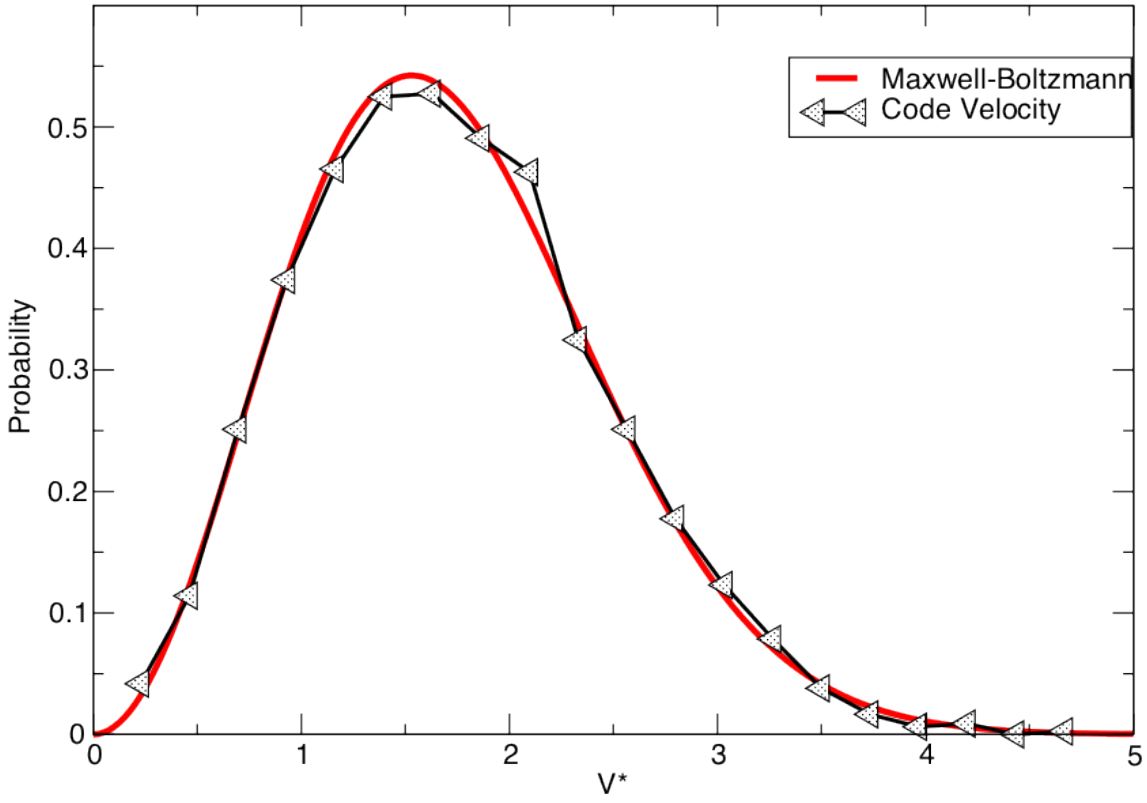

Figure 2. The initialization of the velocities matches the expected Maxwell-Boltzmann distribution.

Below are representations of the melting system. These are actual screen shots from the animation that the code produces. This will look similar for both equilibration processes.

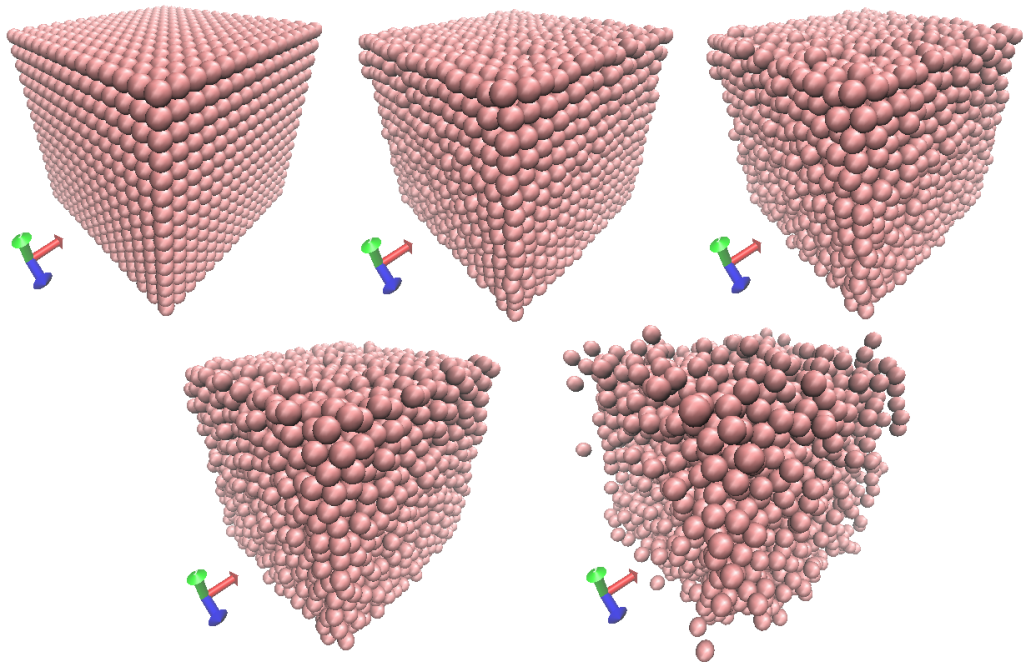

Figure 3. The above row shows the first 30 time steps, where the bottom shows the midway and final time step. Note, the animation allows the particles to leave the control volume, to prevent blinking of the atoms; the code properly corrects for the periodic boundary.

4.2 Velocity Scaling

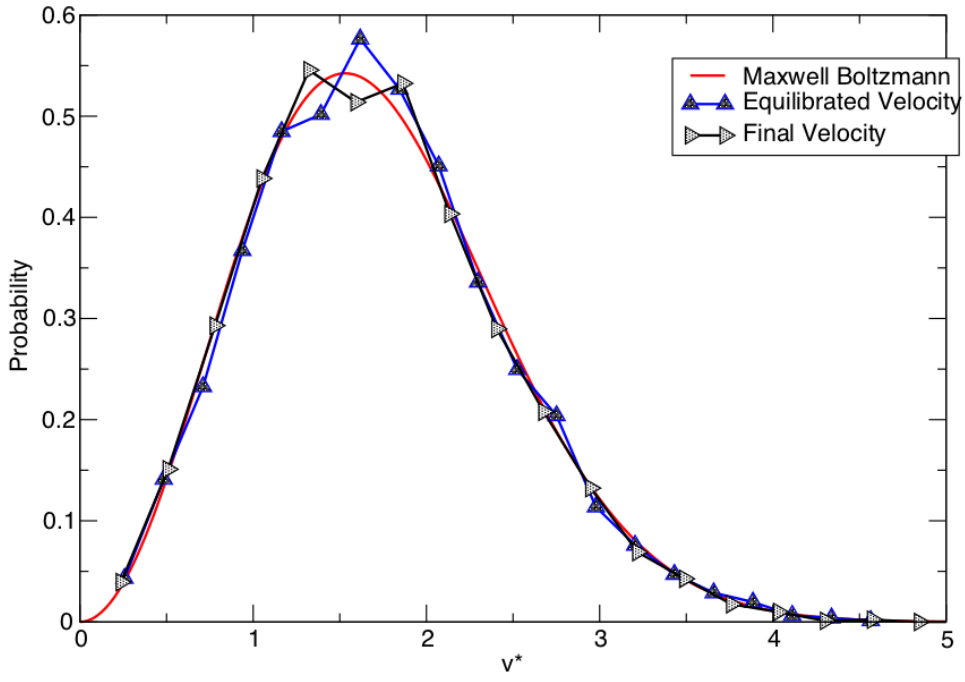

Figure 4, left. The velocity distribution matches the expected Maxwell-Boltzmann distribution at each stage. More particles would smooth out the curve

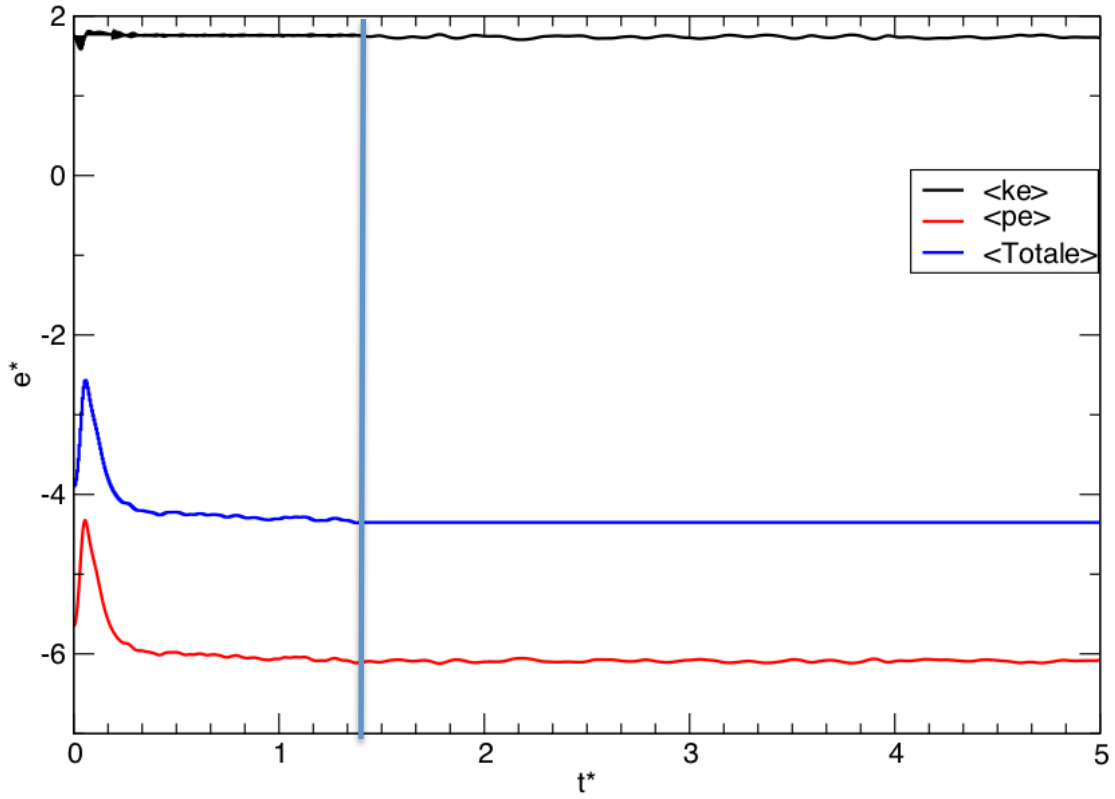

Figure 5. The energy converges very quickly in this method. The vertical line represents the end of equilibration.

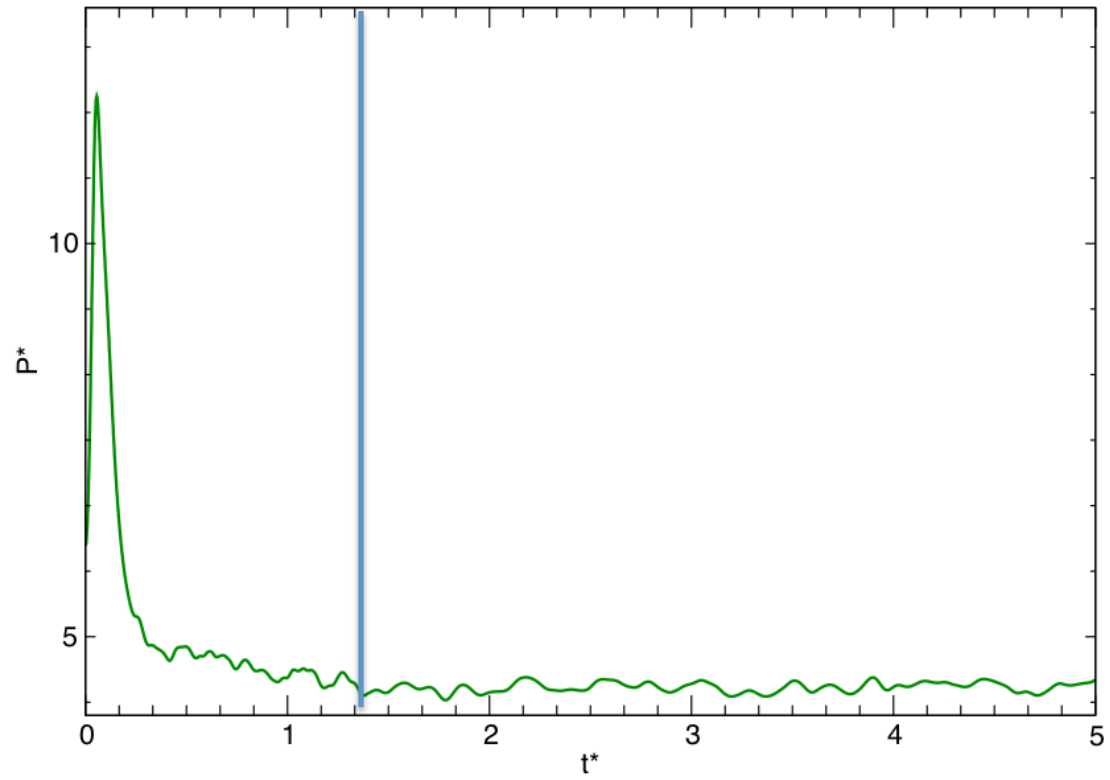

Figure 6. Pressure convergence for the Velocity Scaling method.

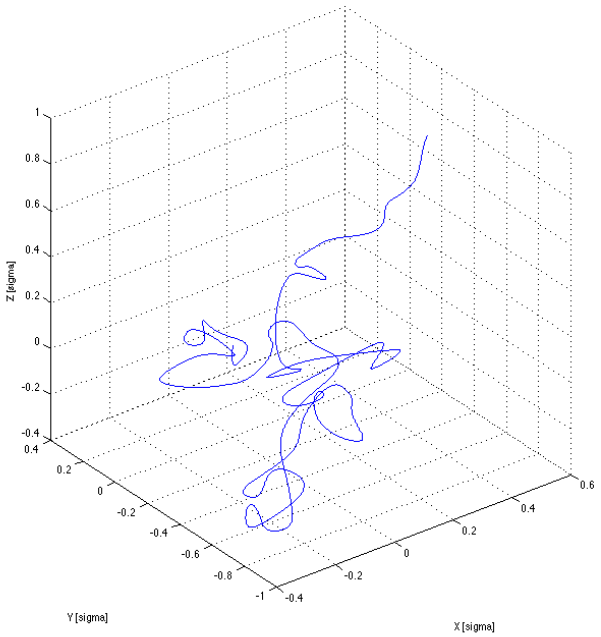

Figure 7. Trace of Particle 327. This is mainly just for visualization as each run; because of the stochastic nature of the process each run will result in a different trace.

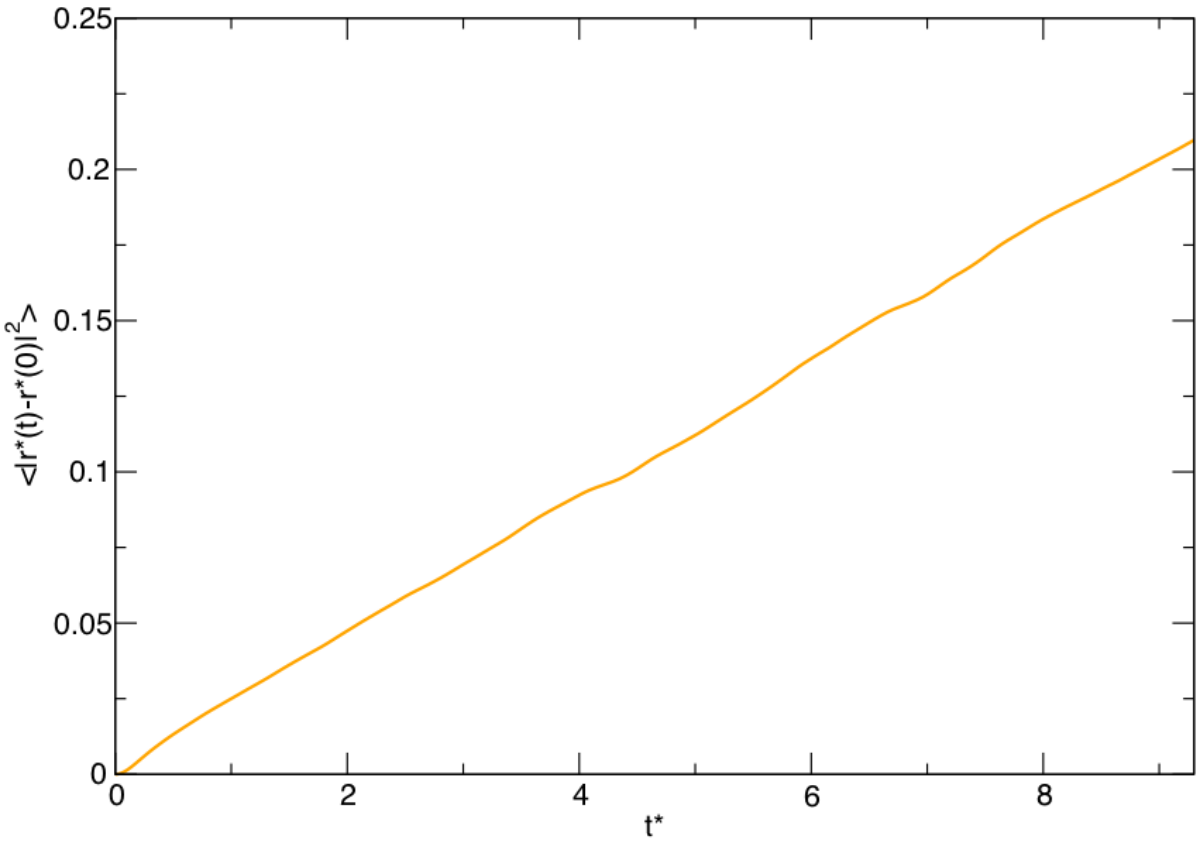

Figure 8. The MSD over the length of the production run. This takes into account the dimensionality of the system. Note the very smooth linearity of the system.

	Code	<Fluctuation>	Monte Carlo	Error
<ke> [zJ]	2.86	0.0003	-	-
<u> [zJ]	-10.04	0.00030316	-9.81	2.338%
<e> [zJ]	-7.18	4.88617E-10	-	-
<P> [MPa]	183.40	0.0068	174.03	5.380%
D $[m^2/s]$	2.42×10^{-9}	-	-	-

Table 1. Tabulated results of the code.

The above results show that the code converges quite well for the given values. Additionally, the MSD follows a very close linear profile, something to be expected using this law. Additionally, the calculated self-diffusion coefficient is of the same order of that offered in literature[7] ($\sim 2\times10^{-9} m^2/s$). However, it is worth noting that and exhaustive search could not find the self-diffusion coefficient for the pressure and temperature represented in this study. The found values ranged from $\sim 1.5 - 7\times10^{-9} m^2/s$.

4.3 Andersen Thermostat

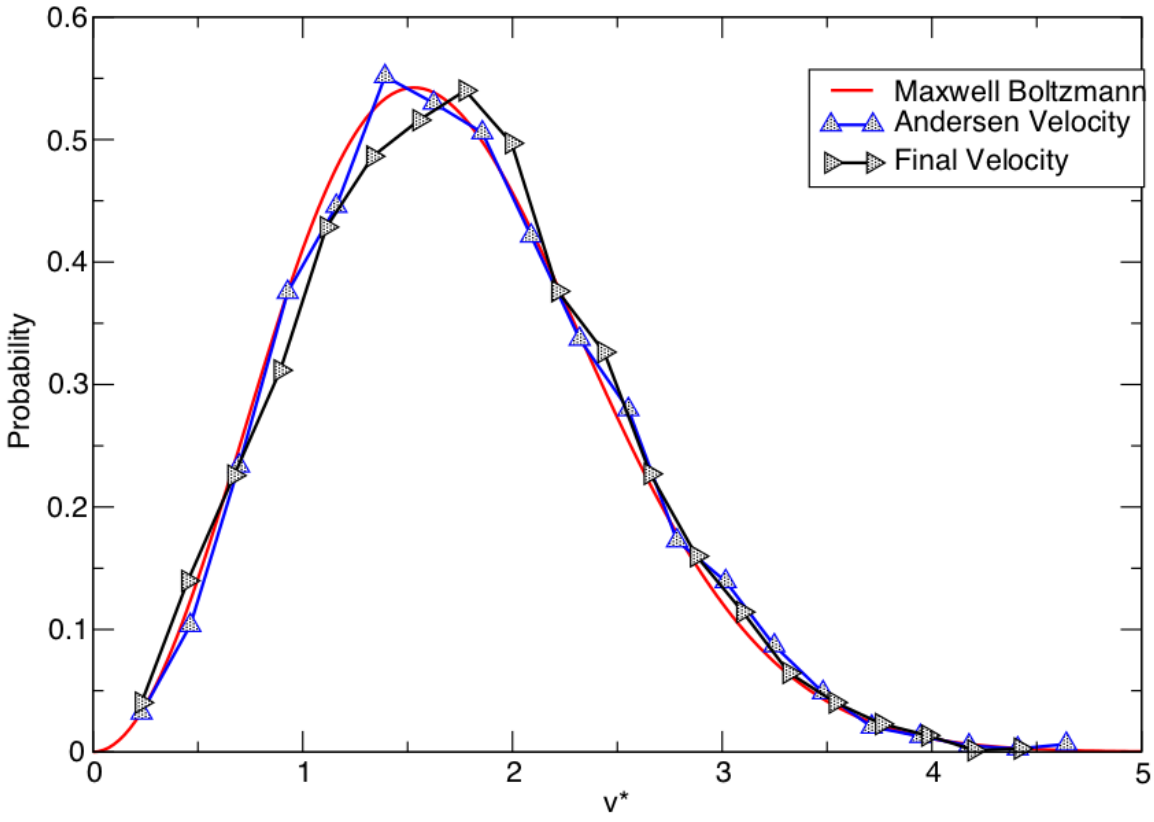

Figure 9. Velocity results of the Andersen Thermostat.

[7] R. A. Fisher & R. O. Watts, (1972) Calculated Self Diffusion Coefficients for Liquid Argon. *Aust J Phys,* **25**, 529-38

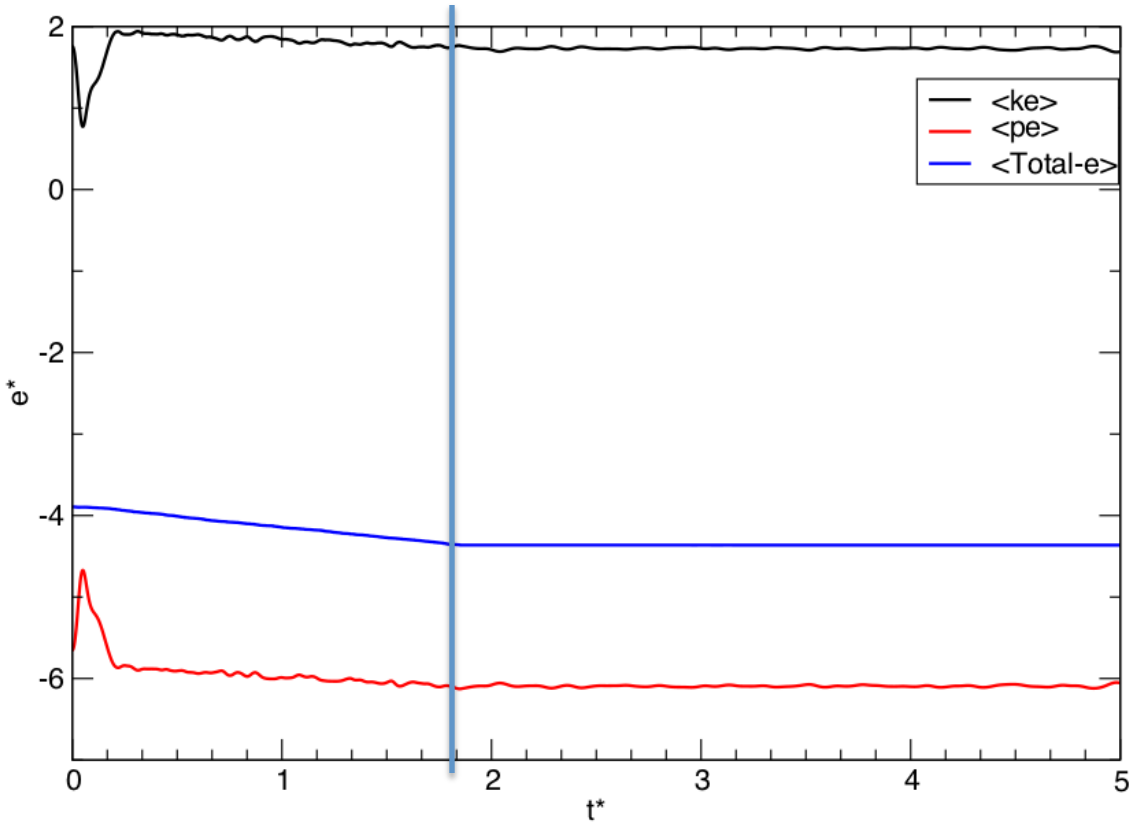

Figure 10. Energy convergence for Andersen Thermostat. Notice how we do not lose the information about how the velocity changes in the equilibration stage.

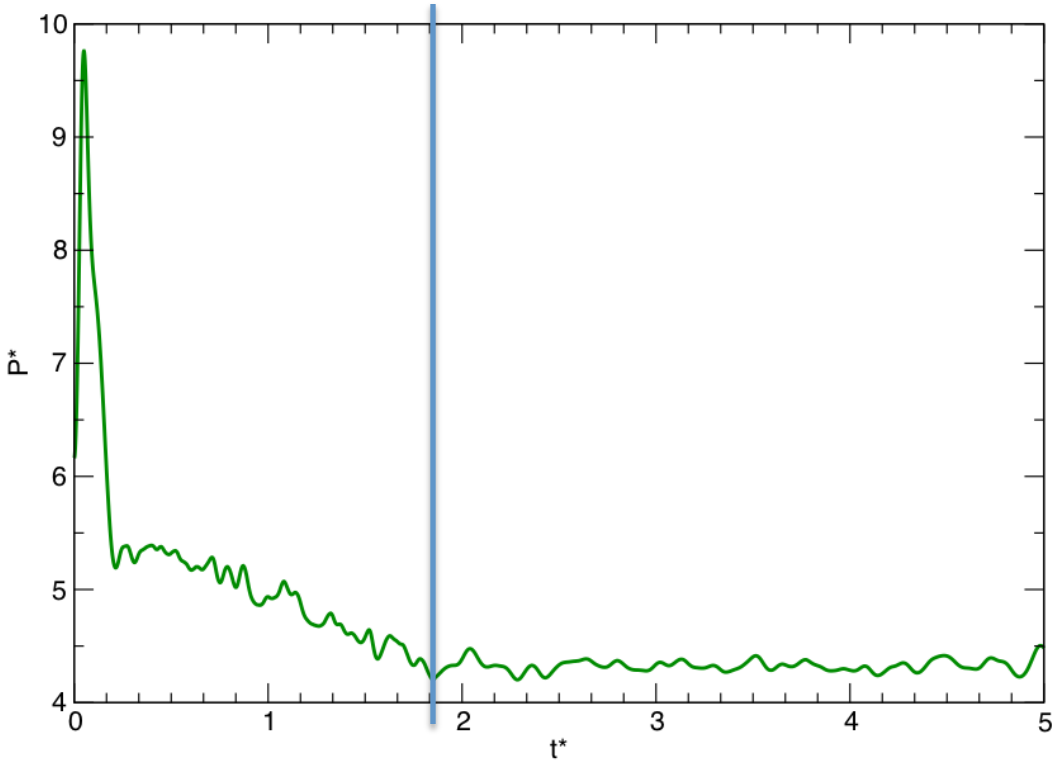

Figure 11. Andersen Thermostat Pressure convergence.

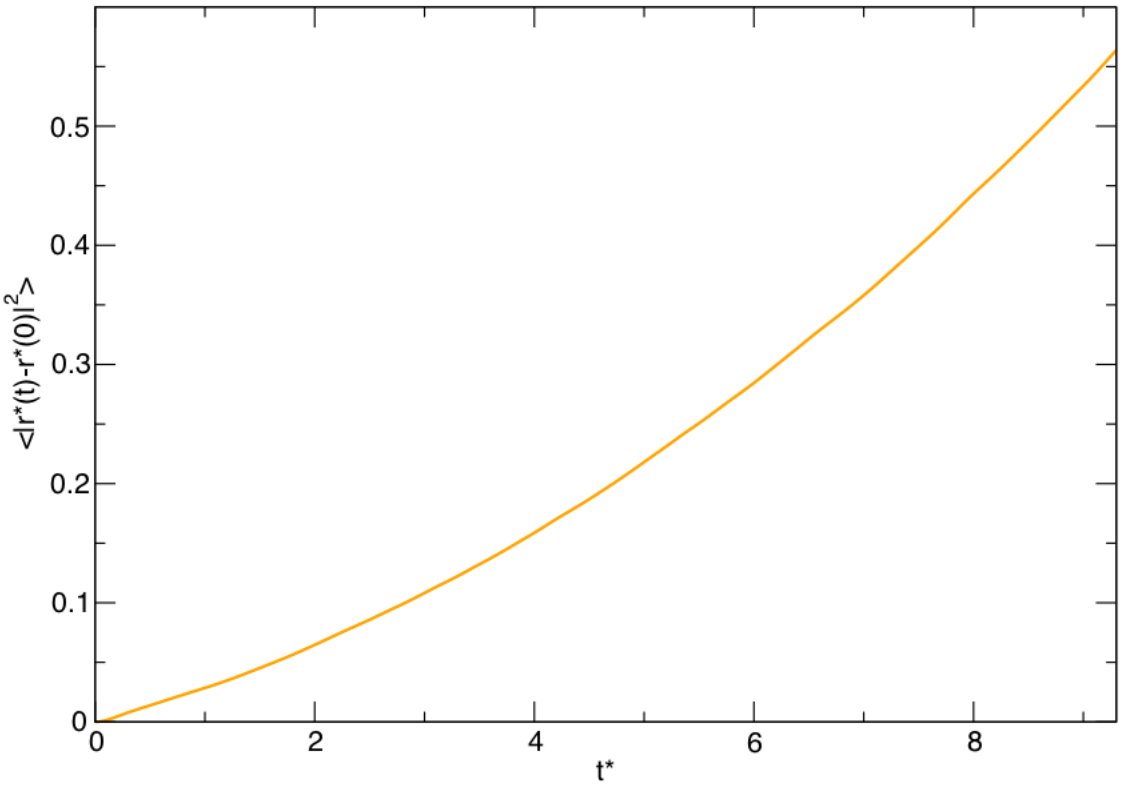

Figure 12. MSD for the Andersen Thermostat. Note that it does not follow a very close linear profile, something expected in MSD plots.

	Code	<Fluctuation>	Monte Carlo	Error
<ke> [zJ]	2.86	0.000224	-	-
<u> [zJ]	-10.06	0.00022	-9.81	2.487%
<e> [zJ]	-7.20	3.40524E-10	-	-
<P> [MPa]	181.39	0.003931552	174.03	4.229%
D $[m^2/s]$	6.52×10^{-9}	-	-	-

Table 2. Results of the application of the Andersen Thermostat.

Once again, the Andersen thermostat very closely matches the results of the velocity scaling method as well as the Monte Carlo results. The notable exception is the self-diffusion coefficient that is calculated at ~3x the coefficient found in the velocity scaling method. This may be because of the iterative nature of the thermostat. Further runs at different cycle counts should be tested for final values when using the Andersen Thermostat.

4.4 Other Tests

The code was additionally tested against known values of Argon given in homework 8 of 2.37. The results are shown on the next page.

Expected				Velocity Scale Calculation				
ρ^*	T*	P*	U*	P*	Error	U*	Error	D$[m^2/s]$
0.5	5	4.65	-2.37	4.40	5.48%	-2.47	4.38%	4.09E-08
0.9	2	9.1	-5	9.00	1.05%	-5.23	4.56%	5.18E-09
0.05	1	0.037	-0.478	-0.01	122.32%	-0.37	22.48%	7.74E-08

Table 3. Tabulated results for Velocity Scale thermostat at various densities, temperatures.

Expected				Andersen Calculation				
ρ^*	T*	P*	U*	P*	Error	U*	Error	D $[m^2/s]$
0.5	5	4.65	-2.37	3.00	35.44%	-2.77	16.78%	4.23E-08
0.9	2	9.1	-5	7.03	22.80%	-5.58	11.56%	8.04E-09
0.05	1	0.037	-0.478	-0.01	126.68%	-0.38	20.50%	7.62E-08

Table 4. Tabulated results for Andersen Thermostat at various densities, temperatures. Note, these tests were done with 3000 equilibrium steps.

From the above tables, it appears that the code has calculation issues in the low-density range. This is principally due to the cutoff radius needing to be larger at low pressures as the atoms become gaseous. To do these calculations, we would simply need to make the code iterate over the cutoff radius, or simply account for all atoms in the control volume. To test this theory, the code was rerun with a cutoff of 10, and a pressure of 0.048, or an error of 30%, was found. While this shows the code can handle gas-phase, it would need to be reevaluated to handle these densities.

Additionally, the code was tested for performance on a 2.7Ghz Intel i7 machine. The code was found to use ~44MB of memory each run while generating ~180MB of data for post processing. The timing of the main sub-processes are shown below:

	Velocity Scaling	**%**	**Andersen**	**%**
Initialization [s]	0.162246	0.035%	0.164914	0.034%
Equilibration [s]	97.711583	21.249%	110.753006	22.93%
Production [s]	361.941394	78.709%	372.020674	77.03%
Whole Calculation [s]	459.845685	100%	482.967538	100%

Table 5. All benchmarking was done with 3000 equilibrium steps along with 10000 production steps.

5.0 Conclusion

The above results indicate that the code has performed impeccably in the liquid densities that were requested of it. It runs extremely fast and is capable of being used on different atomic systems with minimum changes to the code. The code had errors in calculating gaseous densities, but increasing the cutoff radius helped with convergence in this range. Overall, the code performs as well as the Monte Carlo code at a fraction of the time required for calculation.

Appendix: Complete Code

This code can be found online at: http://www.levilentz.com/2.37/

```
1.  #include <stdio.h>
2.  #include <math.h>
3.  #include <stdlib.h>
4.  #include <time.h>
5.  #include <string.h>
6.  #define nDim 3
7.  #define sigma 3.4e-10
8.  #define epsilon 1.65e-21
9.  #define masskg 6.6e-26
10.
11. double random1();
12. double maxwell();
13. double gauss(double std, double mean );
14. void velscaler(double *velocity, double T, int n);
15. void initialization(double *positions, double *velocities, int n, double L, double T);
16. int neighborhood(double *position, int *celllist, int n, double L, double rcut);
17. void LJ_Forces( double *force, double *potential, double *position, int n, int Nmax, int *ce
    lllist, double L, double rcut, double rmax, double *px);
18. void andersen(int switch1, int n, double *force, double *velocity, double *positions,double
    *tracer, double dt, double T, double L, int *blinker );
19. void properties(double *position, double *tracer, double *velocities, double *forces, double
     *potential, int n, double *U, double *KE, double *energy, double *avevel, double *Temp, dou
    ble *px, double *pressure, double v, double rcut, double rho, double L, double *Ds, double t
    imecount);
20. void position_printer(FILE * fp, double * positions, int n, int *blinker, double L, int flag
    );
21. void velocity_printer(FILE *fp, double * velocity, int n );
22.
23. main() {
24.     int n, i, j, ncel, *link, *celllist, *neighborlist, nMax, neighborcounter, cycles, track
    , *blinker;
25.     double Rho, T, dt, dt_squared, L, vol, mass, ndouble, *positions, *velocities, rcut, *fo
    rce;
26.     double rcel, *potential, kb, kT, *acceleration, Lcell, rmax, timecounter, error, v, U, t
    otale;
27.     double *energies, KE, *avevel, Temp, *forcematrix, pressure, *pix, *tracer, Ds, tau, *fo
    rcestore;
28.     double squarevel[3], aveke, avepe, avetotal, aveT, aveP, aveDs;
29.     double andersentime,veltime, codetime,productiontime, inittime;
30.     char buffer[50], vyes[20];
31.     char yes[20];
32.     FILE * fp;
33.     FILE * tracker;
34.     FILE * melter;
35.     FILE * velocityfp;
36.     FILE * OUT;
37.     clock_t codebegin, codeend, initbegin, initend, scalebegin, scaleend, andersenbegin, and
    ersenend;
38.     clock_t productionbegin, productionend;
39.     codebegin = clock();
40.     srandom(time(NULL));
41.     printf("How many cycles?\n");
42.     scanf("%d",&cycles);
43.     //cycles = 3000;
44.     printf("How many atoms (perfect cube)?\n");
45.     scanf("%d",&n);
46.     //n = 3375;
```

```
    rcut = 2.5; //2.5sigma
    rmax = rcut + 1;
    printf("Rho (Lennard-Jones units)?\n");
    scanf("%lf",&Rho);
    //Rho = 0.90399;
    printf("Temp (Lennard-Jones units)?\n");
    scanf("%lf",&T);
    //T = 1.17145;
    mass = 1;
    tau = sigma*sqrt(masskg/epsilon);
    dt = 1e-15/tau;
    L = pow((double)n / Rho, 0.333333);
    vol = n/Rho;
    //printf("%lf %lf\n", L, vol);
    positions = (double*)malloc(nDim*n*sizeof(double));
    velocities = (double*)malloc(nDim*n*sizeof(double));
    force = (double*)malloc(nDim*n*sizeof(double));
    potential = (double*)malloc(n*sizeof(double));
    neighborlist = (int*)malloc(n*n*sizeof(int));
    avevel = (double*)malloc(nDim*sizeof(double));
    pix = (double*)malloc(nDim*sizeof(double));
    tracer = (double*)malloc(nDim*n*sizeof(double));
    blinker = (int*)malloc(nDim*n*sizeof(int));
    memset(positions,0.0,nDim*n*sizeof(double));
    memset(velocities,0.0,nDim*n*sizeof(double));
    memset(force,0.0,nDim*n*sizeof(double));
    memset(potential,0.0,n*sizeof(double));
    memset(neighborlist,0,n*n*sizeof(int));
    memset(avevel,0.0,3*sizeof(double));
    memset(tracer,0.0,nDim*n*sizeof(double));
    memset(tracer,0.0,nDim*n*sizeof(double));
    memset(blinker,0,nDim*n*sizeof(int));
    OUT = fopen("output","w");
    printf("Would you like to apply an Andersen Thermostat? [Y/N]\n");
    scanf("%s",yes);
    printf("Would you like to Velocity Scale? [Y/N]\n");
    scanf("%s",vyes);
    printf("What particle would you like to track?\n");
    scanf("%d",&track);
    initbegin = clock();
    fp = fopen("Initial.xyz","w");
    velocityfp = fopen("Initialvel","w");
    initialization(positions,velocities,n,L,T);
    velocity_printer(velocityfp,velocities,n);
    position_printer(fp,positions,n,blinker,L,0);
    fclose(fp);
    fclose(velocityfp);
    melter = fopen("Melting.xyz","w");
    position_printer(melter,positions,n, blinker, L,1);
    nMax = neighborhood(positions,neighborlist,n,L,rmax);
    if (nMax == 0 ) {
        return 0;
    }
        LJ_Forces(force,potential,positions,n,nMax,neighborlist,L,rcut,rmax, pix);
        initend = clock();
        inittime = (double)(initend - initbegin) / CLOCKS_PER_SEC;
        timecounter = 0.0;
        neighborcounter = 0;
        if (strcmp(vyes,"Y") == 0) {
        scalebegin = clock();
        printf("\n\nMelting at constant Temp\n\n");
```

```
        fprintf(OUT,"\n\nMelting at constant Temp\n\n");
            for(i=0;i<cycles;i++) {
                neighborcounter += 1;
                andersen(1,n,force,velocities,positions,tracer, dt, T, L, blinker);
          LJ_Forces(force,potential,positions,n,nMax,neighborlist,L,rcut,rmax, pix);
                andersen(2,n,force,velocities,positions,tracer,dt, T, L, blinker);
                if (neighborcounter > 10) {
                    nMax = neighborhood(positions,neighborlist,n,L,rmax);
                    neighborcounter = 0;
                    velscaler(velocities,T,n);
                    position_printer(melter,positions,n, blinker, L,1);
                }
                timecounter += dt;
                properties(positions,tracer,velocities,force,potential,n,&U,&KE,&totale, avev
    el, &Temp,pix, &pressure, vol, rcut, Rho, L, &Ds, timecounter);
                printf("%lf %lf %lf %lf %lf %lf %lf\n", timecounter, KE, U, totale, pressure,
     Ds, Temp);
                fprintf(OUT,"%lf %lf %lf %lf %lf %lf %lf\n", timecounter, KE, U, totale, pres
    sure, Ds, Temp);
            }
            velocityfp = fopen("Meltedvel","w");
            velocity_printer(velocityfp,velocities,n);
            fclose(velocityfp);
            sprintf(buffer,"%lf.xyz",dt*i);
            fp = fopen("melted.xyz","w");
            position_printer(fp,positions,n,blinker,L,0);
            fclose(fp);
        scaleend = clock();
        veltime = (double)(scaleend - scalebegin) / CLOCKS_PER_SEC;
        }
        if( strcmp(yes,"Y") == 0 ) {
        andersenbegin = clock();
            printf("\n\nAndersen Thermostat\n\n");
            fprintf(OUT,"\n\nAndersen Thermostat\n\n");
            velocityfp = fopen("Andersenvel","w");
            neighborcounter = 0;
            for(i=0;i<cycles;i++) {
                neighborcounter += 1;
                andersen(1,n,force,velocities,positions,tracer, dt, T, L, blinker);
                LJ_Forces(force,potential,positions,n,nMax,neighborlist,L,rcut,rmax, pix);
                andersen(3,n,force,velocities,positions,tracer,dt, T, L, blinker);
                if ((i % 10) == 0) {
                    nMax = neighborhood(positions,neighborlist,n,L,rmax);
                    neighborcounter = 0;
                    position_printer(melter,positions,n, blinker, L,1);
                }
                timecounter += dt;
                properties(positions,tracer,velocities,force,potential,n,&U,&KE,&totale, avev
    el, &Temp,pix, &pressure, vol, rcut, Rho, L, &Ds, timecounter);
                printf("%lf %lf %lf %lf %lf %lf %lf\n", timecounter, KE, U, totale, pressure,
     Ds, Temp);
                fprintf(OUT,"%lf %lf %lf %lf %lf %lf %lf\n", timecounter, KE, U, totale, pres
    sure, Ds, Temp);

            }
            velocity_printer(velocityfp,velocities,n);
            fclose(velocityfp);
        andersenend = clock();
        andersentime = (double)(andersenend - andersenbegin) / CLOCKS_PER_SEC;
        }

```

```
163.        productionbegin = clock();
164.        memset(tracer,0.0,nDim*n*sizeof(double));
165.        cycles = 10000;
166.        tracker = fopen("tracking","w");
167.        fprintf(tracker,"0 0 0\n");
168.        neighborcounter = 0;
169.        aveke=avepe=avetotal=aveT=aveP=aveDs=0.0;
170.        printf("\n\nNVE Production\n\n");
171.        fprintf(OUT,"\n\nNVE Production\n\n");
172.        velocityfp = fopen("FinalVel","w");
173.        for(i=0;i<cycles;i++) {
174.            timecounter += dt;
175.            neighborcounter += 1;
176.            andersen(1,n,force,velocities,positions,tracer,dt, T, L, blinker);
177.            fprintf(tracker,"%lf %lf %lf\n", *(tracer + track), *(tracer + n + track), *(trac
    er + 2*n + track));
178.            LJ_Forces(force,potential,positions,n,nMax,neighborlist,L,rcut,rmax,pix);
179.            andersen(2,n,force,velocities,positions,tracer,dt, T, L, blinker);
180.            if ((i % 10) == 0) {
181.                nMax = neighborhood(positions,neighborlist,n,L,rmax);
182.                neighborcounter = 0;
183.                position_printer(melter,positions,n, blinker, L,1);
184.            }
185.            timecounter += dt;
186.            properties(positions,tracer,velocities,force,potential,n,&U,&KE,&totale, avevel,
    &Temp,pix, &pressure, vol, rcut, Rho, L, &Ds, timecounter);
187.            printf("%lf %lf %lf %lf %lf %lf %lf\n", timecounter, KE, U, totale, pressure, Ds,
     Temp);
188.            fprintf(OUT,"%lf %lf %lf %lf %lf %lf %lf\n", timecounter, KE, U, totale, pressure
    , Ds, Temp);
189.            aveke += KE;
190.            avepe += U;
191.            avetotal += totale;
192.            aveP += pressure;
193.            aveT += Temp;
194.            aveDs += Ds;
195.        }
196.        productionend = clock();
197.        productiontime = (double)(productionend - productionbegin) / CLOCKS_PER_SEC;
198.        velocity_printer(velocityfp,velocities,n);
199.        fclose(velocityfp);
200.        fclose(tracker);
201.        fclose(melter);
202.        aveke /= cycles;
203.        avepe /= cycles;
204.        avetotal /=cycles;
205.        aveP /= cycles;
206.        aveT /= cycles;
207.        aveDs /= cycles;
208.
209.        Ds = Ds / (cycles*dt);
210.
211.        printf("<ke>\t<u>\t<e>\tD\t<P>\t<T>\t\n");
212.        printf("%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t\n",aveke,avepe,avetotal,Ds,aveP,aveT);
213.        fprintf(OUT,"<ke>\t<u>\t<e>\tD\t<P>\t<T>\t\n");
214.        fprintf(OUT,"%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t%lf\t\n",aveke,avepe,avetotal,Ds,aveP,aveT);

215.
216.        fclose(OUT);
217.
218.        free(positions);
```

```
219.        free(velocities);
220.        free(force);
221.        free(potential);
222.        free(neighborlist);
223.        free(avevel);
224.        free(pix);
225.        free(tracer);
226.        free(blinker);
227.        codeend = clock();
228.        codetime = (double)(codeend - codebegin) / CLOCKS_PER_SEC;
229.
230.        printf("Timing results\n");
231.        if (strcmp(vyes,"Y") == 0) {
232.            printf("Whole Code\tInitialization\tVelscale\tProduction\n");
233.            printf("%lf\t%lf\t%lf\t%lf\n",codetime,inittime,veltime,productiontime);
234.
235.        } else if( strcmp(yes,"Y") == 0 ) {
236.            printf("Whole Code\tInitialization\tAndersen\tProduction\n");
237.            printf("%lf\t%lf\t%lf\t%lf\n",codetime,inittime,andersentime,productiontime);
238.        }
239.
240.    }
241.
242.    double random1() {
243.        return (double)random()/RAND_MAX;
244.    }
245.
246.    double maxwell() { //johnk's random number (from a maxwellian distribution) generator
247.    //referenced from: http://www.stat.rice.edu/~dobelman/textfiles/DistributionsHandbook.pdf

248.        double x1, x2, w, y1, y2;
249.        do {
250.            x1 = 2.0 * random1() - 1.0;
251.            x2 = 2.0 * random1() - 1.0;
252.            w = x1 * x1 + x2 * x2;
253.        } while ( w >= 1.0 );
254.        w = sqrt( (-2.0 * log10( w ) ) / w );
255.        y1 = x1 * w;
256.        y2 = x2 * w;
257.        return y2;
258.
259.    }
260.
261.    double gauss(double std, double mean ) {
262.    //Used by Andersen (properly scaled velocities)
263.        double rand;
264.        rand = random1();
265.
266.        rand = mean + std*rand;
267.        return rand;
268.    }
269.
270.    void velscaler(double *velocity, double T, int n) {
271.    //Scales the velocity to proper temp, T.
272.        int i,j;
273.        double square_vel[nDim], velscale;
274.        for (i = 0; i<nDim; i++) {
275.            square_vel[i] = 0.0;
276.        }
277.
278.        for (i=0; i<n; i++) {
```

```
            for (j=0;j<nDim;j++) {
                square_vel[j] +=  *(velocity + j*n+ i) * *(velocity +j*n + i);
            }
        }
        velscale = sqrt(3*T/((square_vel[0]+square_vel[1]+square_vel[2])/n));
        for (i=0;i<n;i++) {
            for (j=0;j<nDim;j++) {
                *(velocity + j*n + i) *= velscale;
            }
        }

    }

    void initialization(double *positions, double *velocities, int n, double L, double T) {
        int N, i, j, k, flag;
        double del, ndouble, per_dimension, *sum_velocity, *squared_velocity, velscale, rcuto
    ff;
        double r, tester;
        per_dimension = pow(n,1/3.0);
        rcutoff = 0.212; //Argon diameter divided by sigma
        del = L/per_dimension;
        //printf("%d %lf %lf\n", n, per_dimension, del);
        N = 0;
        flag = 0;
        sum_velocity = malloc(3*sizeof(double));
        squared_velocity = malloc(3*sizeof(double));
        for(i=0;i<nDim;i++) {
            *(sum_velocity + i) = 0.0;
            *(squared_velocity + i) = 0.0;
        }

        for (i=0;i< per_dimension; i++) {
            for(j=0;j< per_dimension; j++) {
                for(k=0; k< per_dimension;k++) {
                    *(positions + 0*n + N) = i * del;
                    *(positions + 1*n + N) = j * del;
                    *(positions + 2*n + N) = k * del;
                    N++;
                }
            }
        }


        for(i=0; i<n; i++) {
            *(velocities + i) = maxwell();
            *(velocities + 1*n + i) = maxwell();
            *(velocities + 2*n + i) = maxwell();
            //below for calculating center of mass velocity
            *(sum_velocity) += *(velocities + i)/n;
            *(sum_velocity + 1) += *(velocities + 1*n + i)/n;
            *(sum_velocity + 2) += *(velocities + 2*n + i)/n;
        }
        //correcting for CM drift
        for (i=0;i<n;i++) {
            *(velocities + i) -= *(sum_velocity);
            *(velocities + 1*n + i) -= *(sum_velocity + 1);
            *(velocities + 2*n + i) -= *(sum_velocity + 2);
        }
        velscaler(velocities,T,n);
    }

```

```
339.    int neighborhood(double *position, int *celllist, int n, double L, double rcut) {
340.    //Neighborhood list generation
341.        int i,j,k,xcell,ycell,zcell, Nmax;
342.        double x,y,z, halfL, distance[nDim], r;
343.        Nmax = 0;
344.        for (i = 0; i<n; i++) {
345.            for(j=i+1;j<n; j++) {
346.                r = 0.0;
347.                if(i == j) { break;}
348.                distance[0] = *(position + i) - *(position + j);
349.                distance[1] = *(position + n + i) - *(position + n + j);
350.                distance[2] = *(position + 2*n + i) - *(position + 2*n + j);
351.                //periodic boundary condition:
352.                halfL = L/2.0;
353.                for (k=0;k<nDim;k++) {
354.                    if (distance[k] > halfL) {
355.                        distance[k] = distance[k] - L;
356.                    } else if( distance[k] < -halfL ) {
357.                        distance[k] = distance[k] + L;
358.                    }
359.                    r = r + distance[k]*distance[k];
360.                }
361.                r = sqrt(r);
362.                // generates a Nx2 matrix with the list of atoms
363.                //each row gives i,j corresponding to atoms near eachother
364.                if (r <= rcut) {
365.                    *(celllist + Nmax*2 + 1) = i;
366.                    *(celllist + Nmax*2 + 2) = j;
367.                    Nmax = Nmax + 1;
368.                }
369.            }
370.        }
371.        if (Nmax == 0 ) {
372.            printf("No near atoms\n");
373.            return 0;
374.        }
375.        return Nmax;
376.    }
377.
378.    void LJ_Forces( double *force, double *potential, double *position, int n, int Nmax, int
    *celllist, double L, double rcut, double rmax, double *px){
379.        int i, j, k, x, y, z, num;
380.        double distance[nDim], halfL,ir6, U, F, r, r6, ir, r2, numerator, den, rmax2, rcut2,
    virial, tester;
381.        double ucut;
382.        ucut = 4*(1/(rcut*rcut*rcut*rcut*rcut*rcut))*((1/(rcut*rcut*rcut*rcut*rcut*rcut)) -
     1);
383.        virial = 0.0;
384.        memset(potential,0.0,n*sizeof(double));
385.        memset(force,0.0,nDim*n*sizeof(double));
386.        memset(px,0.0,nDim*sizeof(double));
387.
388.        virial = 0.0;
389.        halfL = L/2.0;
390.        for ( num = 0; num<Nmax; num++) {
391.                r = 0.0;
392.                i = *(celllist + num*2 + 1);
393.                j = *(celllist + num*2 + 2);
394.                if( i != j) {
395.                    distance[0] = *(position + i) - *(position + j);
396.                    distance[1] = *(position + n + i) - *(position + n + j);
```

```
397.                     distance[2] = *(position + 2*n + i) - *(position + 2*n + j);
398.                     for (k=0;k<nDim;k++) {
399.                         if (distance[k] > halfL) {
400.                             distance[k] = distance[k] - L;
401.                         }else if( distance[k] < -halfL ) {
402.                             distance[k] = distance[k] + L;
403.                         }
404.                     }
405.                     r2 = distance[0]*distance[0]+distance[1]*distance[1]+distance[2]*distance
    [2];
406.                     r = sqrt(r2);
407.                     //printf("%lf\n",r);
408.                     if (r <= rcut) {
409.                         ir = 1.0/r;
410.                         r6 = r2*r2*r2;
411.                         ir6 = 1.0/r6;
412.                         U = 4.0*ir6*(ir6-1.);
413.                         F = 48.0*(ir6*ir6-0.5*ir6);
414.                         //printf("%lf %lf %lf\n", r, U, F);
415.                         *(potential + i) += U;
416.                         for (k =0; k<nDim; k++) {
417.                             *(force + k*n + i) += F*distance[k]*ir*ir;
418.                             *(force + k*n + j) -
    = F*distance[k]*ir*ir; //equal and opposite forces
419.                         }
420.                         virial += F;
421.
422.                     }else if(r>rcut && r< rmax) {
423.                         r6 = r*r*r*r*r*r;
424.                         r2 = r*r;
425.                         rmax2 = rmax*rmax;
426.                         rcut2 = rcut*rcut;
427.                         numerator = 24.0*(rmax*rmax-r2)*(rmax2*rmax2*(r6-2.0)-rmax2*(r6-
    2.0)*(3.0*rcut2-r2)+rcut2*r2*(r6-4.0)+2.0*r2*r2);
428.                         den = r6*r6*r*(rmax2-rcut2)*(rmax2-rcut2)*(rmax2-rcut2);
429.                         ir6 = 1./r6;
430.                         U = 4.0*ir6*(ir6-1.0)*(rmax*rmax-r*r)*(rmax*rmax-
    r*r)*(rmax*rmax+2.0*r*r-3.0*rcut*rcut)/((rmax*rmax-rcut*rcut)*(rmax*rmax-
    rcut*rcut)*(rmax*rmax-rcut*rcut));
431.                         *(potential + i) += U;
432.                         F = -numerator/den;
433.                         for (k =0; k<nDim; k++) {
434.                             *(force + k*n + i) += F*distance[k]/r;
435.                             *(force + k*n + j) -
    = F*distance[k]/r; //equal and opposite forces
436.                         }
437.                     }
438.
439.                 }
440.         }
441.         *(px) = virial;
442.     }
443.
444.     void andersen(int switch1, int n, double *force, double *velocity, double *positions,doub
    le *tracer, double dt, double T, double L, int *blinker ) {
445.     //Andersen thermostat
446.         int i,j;
447.         double Temp, rand;
448.         double nu = 0.2; //collision with heat bath (needs to be a controllable variable)
449.         if (switch1 == 1) {
450.             for (i=0;i<n;i++) {
```

```
451.                 for (j = 0; j<nDim; j++) {
452.                     *(positions + j*n + i ) += *(velocity +j*n + i)*dt + 0.5 * *(force +j*n +
    i)*dt*dt;
453.                     *(velocity + j*n + i ) += 0.5 * (*(force + j*n + i)) * dt;
454.                     *(tracer + j*n + i) += *(velocity +j*n + i)*dt + 0.5 * *(force +j*n + i)*
    dt*dt;
455.                     if( *(positions + j*n + i) < 0.0) {
456.                         *(positions + j*n + i) = *(positions + j*n + i) + L;
457.                         *(blinker + j*n + i) -= 1;
458.                     }
459.                     if( *(positions + j*n + i) > L ) {
460.                         *(positions + j*n + i) = *(positions + j*n + i) - L;
461.                         *(blinker + j*n + i) += 1;
462.                     }
463.                 }
464.             }
465.         } else if(switch1 == 3 || switch1 == 2 ) {
466.             Temp = 0.0; //initialize temp
467.             for (i = 0;i<n;i++) {
468.                 for (j=0;j<nDim;j++) {
469.                 *(velocity + j*n + i ) = *(velocity + j*n + i ) + 0.5 * *(force + j*n + i) *
    dt;
470.                 }
471.             }
472.             if(switch1 == 3) {
473.                 for(i=0;i<n;i++) {
474.                     for(j=0;j<nDim;j++) {
475.                         rand = random1();
476.                         if(rand < nu*dt) {
477.                             //printf("%lf hit\n",rand);
478.                             *(velocity + j*n + i) = gauss(sqrt(T),0);
479.                         }
480.                     }
481.                 }
482.             }
483.         }
484.    }
485.
486.    void properties(double *position, double *tracer, double *velocities, double *forces, dou
    ble *potential, int n, double *U, double *KE, double *energy, double *avevel, double *Temp,
    double *px, double *pressure, double v, double rcut, double rho, double L, double *Ds, doubl
    e timecount) {
487.         int i,j,k;
488.         double fx, fy, fz, radius, radiust, radiusz, virial;
489.         double distance[3], F, radius6, radius12, ucor, pcor, rcut9, rcut3, halfL;
490.         rcut3 = rcut*rcut*rcut;
491.         rcut9 = rcut3*rcut3*rcut3;
492.         ucor = 8.0*3.1415926*rho*(1.0/(9.0*rcut9)-1.0/(3.0*rcut3));
493.         pcor = 16.0*3.1415926*rho*(2.0/(9.0*rcut9)-1.0/(3.0*rcut3));
494.         halfL = L/2.0;
495.         *U = 0.0;
496.         *KE = 0.0;
497.         *(avevel) = 0.0;
498.         *(avevel + 1 ) = 0.0;
499.         *(avevel + 2) = 0.0;
500.         radiust = 0.0;
501.         *Ds = 0.0;
502.         for (j=0;j<n;j++) {
503.             *(avevel) += *(velocities + j);
504.             *(avevel + 1 ) += *(velocities + n + j);
505.             *(avevel + 2) += *(velocities + 2*n + j);
```

```
506.            *U += *(potential + j);
507.            *Ds += *(tracer + j) * *(tracer + j) + *(tracer + n + j) * *(tracer + n +j) + *(t
    racer + 2*n + j) * *(tracer + 2*n +j);
508.
509.        }
510.        *Ds = *Ds / (2*nDim*n) ;
511.        // Below is just a sanity check to make sure that we have no "Flying Ice"
512.        *(avevel) /= n;
513.        *(avevel + 1 ) /= n;
514.        *(avevel + 2) /= n;
515.        //Above should all be identically 0.0, andersen thermostat may negate this
516.        for(j = 0; j<n; j++) {
517.            *KE += (*(velocities + j) - *(avevel)) * (*(velocities + j) -
     *(avevel)) + (*(velocities + n + j)- *(avevel + 1)) * (*(velocities + n + j)-
     *(avevel + 1)) + (*(velocities + 2*n + j)-*(avevel + 2)) * (*(velocities + 2*n + j)-
    *(avevel + 2));
518.        }
519.        *U /= n;
520.        *U += ucor;
521.        *KE = *KE/(2.0 * n);
522.        *Temp = (2.0/3.0) * *KE ;
523.        *energy = *KE + *U;
524.        *pressure = 0.0;
525.        *pressure = *px / 3.0;
526.        *pressure += n * *Temp;
527.        *pressure /= v;
528.        *pressure += pcor;
529.    }
530.
531.    void position_printer(FILE * fp, double * positions, int n, int *blinker, double L, int f
    lag) {
532.        int i;
533.        double length;
534.        length = sigma*10.0/(1.0e-9); //converting distance to angstrom (for animation)
535.        fprintf(fp,"%d\n\n",n);
536.        if (flag == 1) { //animation
537.            for (i=0;i<n;i++) {
538.                fprintf(fp,"Ar %lf %lf %lf\n", *(positions + i)*length + *(blinker + i)*L*len
    gth, *(positions + n + i)*length + *(blinker + n + i)*L*length, *(positions + 2*n + i)*lengt
    h + *(blinker + 2*n + i)*L*length);
539.            }
540.        } else if(flag == 0) {
541.            for (i=0;i<n;i++) {
542.                fprintf(fp,"Ar %lf %lf %lf\n", *(positions + i)*length, *(positions + n + i)*
    length, *(positions + 2*n + i)*length);
543.            }
544.        }
545.    }
546.
547.    void velocity_printer(FILE *fp, double * velocity, int n ) {
548.        int i;
549.        double Vx, Vy, Vz;
550.        double V;
551.        for(i=0;i<n;i++) {
552.            Vx = *(velocity + i) * *(velocity + i);
553.            Vy = *(velocity + n + i) * *(velocity + n + i);
554.            Vz = *(velocity + 2*n + i) * *(velocity + 2*n + i);
555.            V = sqrt(Vx + Vy + Vz);
556.            fprintf(fp,"%lf\n",V);
557.        }
558.    }
```