Taylor Grubbs - PY525 Homework 3

1 Problem 1

1.1 Introduction

The goal of this assignment was to simulate a 2-dimensional system of particles whose inter-particle interaction is given by the Lennard-Jones potential:

$$V_{LJ} = 4\epsilon[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}] \tag{1}$$

where ϵ defines the depth of the potential well and σ roughly defines the range of the interaction. The motion of the particles is governed by the Energy-Verlet algorithm, given by:

$$\vec{r_i}(t_{k+1}) = 2\vec{r_i}(t_k) - \vec{r_i}(t_{k-1}) + \frac{d^2\vec{r_i}}{dt^2}\Delta t^2 \tag{2}$$

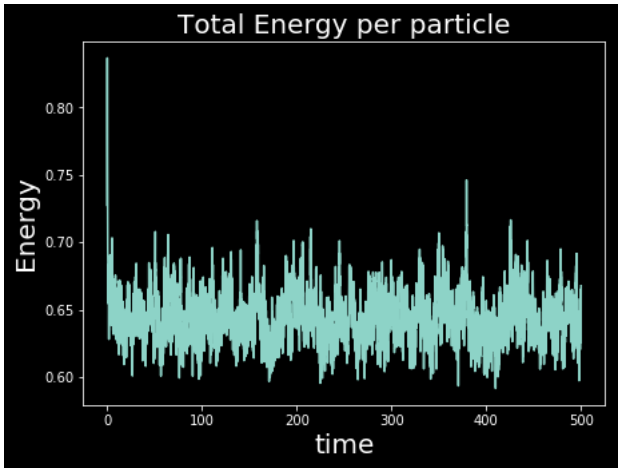
Here, $\vec{r_i}(t_k)$ denotes particle i's position at a discrete point in time denoted by k. And Δt is the time between consecutive steps of the algorithm. The time derivative term can be interpreted as the force on the particle divided by its mass. The force is easily calculated from the potential by the fact that $\vec{F} = -\nabla V$.

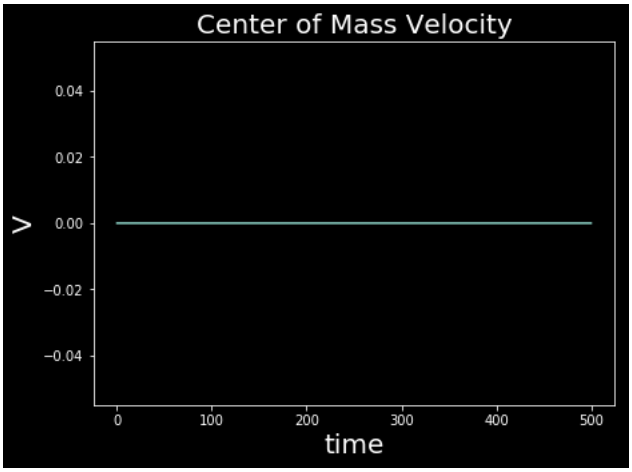
1.2 Methods and Results

C++ was used to program the main simulation while Python was utilized for analysis afterwards. Units were chosen as in the problem statement to make σ and ϵ equal to 1. Particles were initialized on a starting lattice within a 10×10 box to ensure that no particles started to close together. Starting velocities are also created from a uniform distribution such that $v_x, v_y \in (\frac{-3}{2}, \frac{3}{2})$. Periodic boundary conditions are also enforced by calculating particle interactions using the so-called minimum image distance. A cutoff for the potential is also used such that the energy between 2 particles, i and j is given by

$$V_{i,j} = \begin{cases} V_{LJ} & r \leq 3 \\ 0 & r > 3 \end{cases}$$

To test the validity of the simulation, I checked to make sure that the energy was constant and that the center-of-mass momentum was zero. Examples of these are shown below.



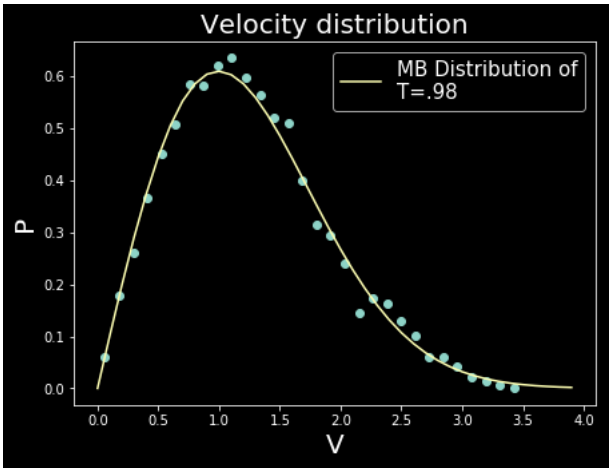


We see that, despite numerical fluctuations, the total energy of the system quickly settles down to an equilibrium value. Also the center of mass has 0 momentum throughout the entire simulation.

In 2 dimensions, the velocities of a group of particles are described by the modified Maxwell-Boltzmann distribution, which in our units is given by:

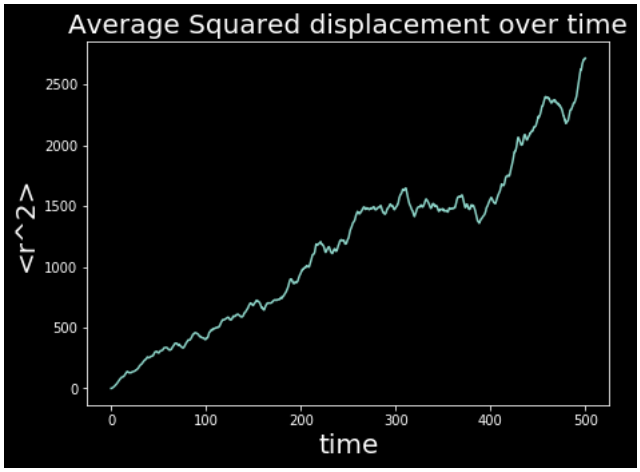
$$P = \frac{v}{T} e^{\frac{-v^2}{2T}} \tag{3}$$

By recording the speeds of each particle over multiple steps, we can create a histogram which describes the velocity distribution. This then can be fit to an appropriate Maxwell-Boltzmann curve. This is shown below.



The histogram was derived from velocities taken from between steps 20,000 and 30,000 so that the system had plenty of time to equilibrate. We see that the system seems to be fairly well described by the Maxwell-Boltzmann distribution at $T = 0.98$.

We can also get a value of the diffusion constant of the system by looking at the squared displacement over time.



From the relation $< \Delta r^2 >= 4Dt$. Fitting the above graph to this curve gives a diffusion constant of 1.16. We also see that this must be either a fluid or gas due to the increasing displacement. However, looking at the animation of the particles it seems more likely that this is a gas because there is no structure or order to the movement of the particles. In a fluid I would expect the particles to maintain some cohesion but should still be able to move around each other. Here the motion seems entirely random.

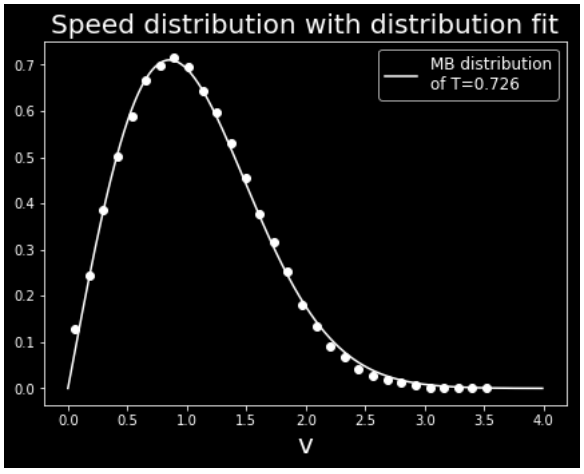
2 Problem 2

2.1 Introduction

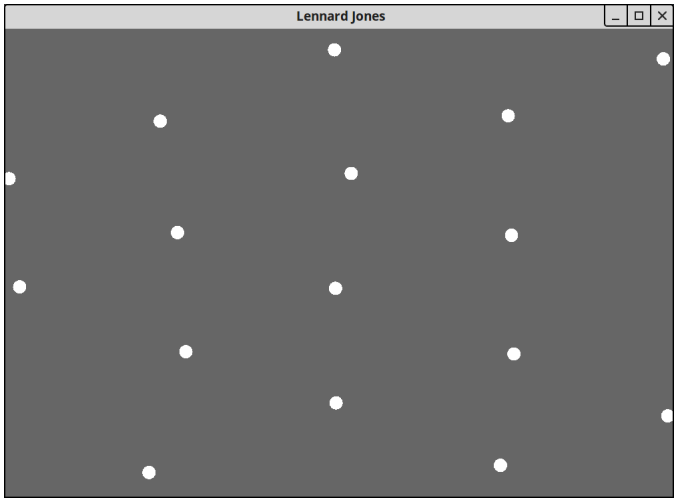
In this problem we are now looking at the same number of particles restricted to a smaller box and smaller starting velocities.

2.2 Methods and Results

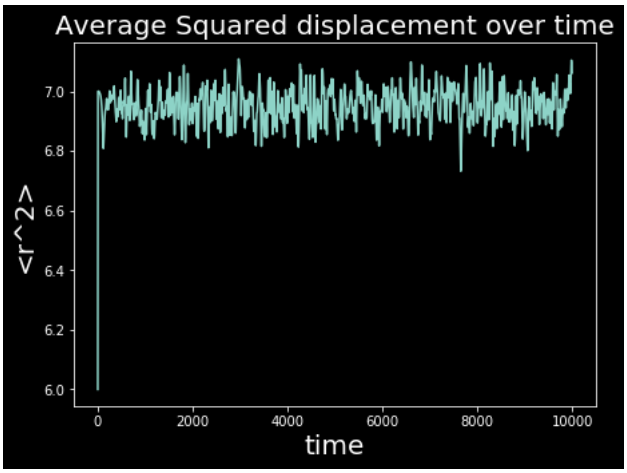
Using the same method of predicting temperature as in Problem 1, I obtain the below result.



In other words, the particles have a temperature of about 0.726. Looking at the animation of the particles, I see that they quickly form a hexagonal lattice.



Further proof that this is indeed a solid is given by the plot of $< \Delta r^2 >$ versus time.



We see that after quickly moving to an equilibrium position, the particles remain roughly the same distance from where they started- with some oscillation as expected from a real solid.

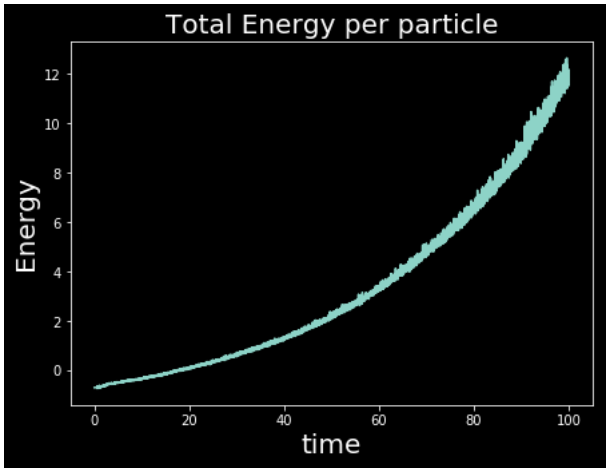
3 Problem 3

3.1 Introduction

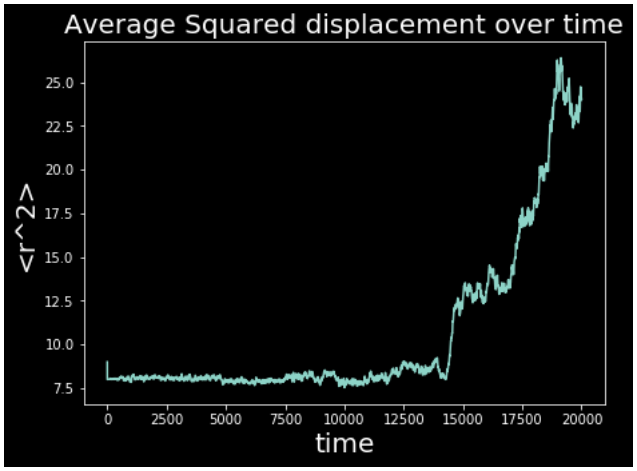
Here we are now gradually heating the system from Problem 2 using the method discussed in class. We are then asked to determine the approximate melting temperature of the system.

3.2 Methods and Results

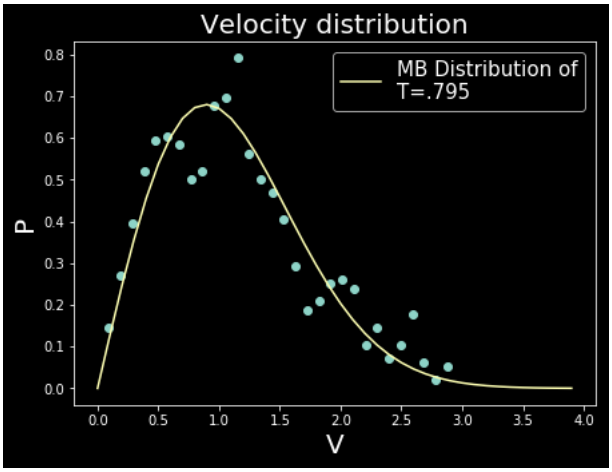
It is worth checking the energy over time to confirm that the system is indeed heating.



Now, looking at the graph of displacement versus time, we can identify approximately where the system shifts from a solid to a liquid.



We see that a significant transition occurs between steps 14000 and 15000. Looking at the distribution of speeds over this range we get an approximate temperature.



We see that our melting temperature is about 0.795.

4 Problem 4

For my final project I would like to create fluid dynamics simulation based on the Lattice-Boltzmann method: $https://en.wikipedia.org/wiki/Lattice_Boltzmann_methods$. I've wanted to write this on my own for a couple of years now. I also like its relation to the Boltzmann transport equation which is of importance in calculating macroscopic quantities in materials. I have found some papers which use the lattice Boltzmann method for studying phonon and electron transport[1,2]. To learn more about the method itself, I am referencing a book which is obtatined for free from NCSU's library[3].

1. A. Nabovati, D. P. Sellan, C. H. Amon, On the lattice Boltzmann method for phonon transport, Journal of Computational Physics 230 (15), 2011.

2. Coelho, Rodrigo Ilha, Anderson Doria, Mauro. (2016). A Lattice Boltzmann Method for Electrons in Metals. 10.20906/CPS/NSC2016-0039.

3. KrÃ$\frac{1}{4}$ger, T., Kusumaatmaja, H., Kuzmin, A.I., Shardt, O., Silva, G., Viggen, E.M. (2017). The Lattice Boltzmann Method.

5 Appendix: Code

```
// Programmer: Taylor Grubbs
// Start Date: Tuesday October 1, 2019
// HW 3
//

#include "GL/freeglut.h"
#include "GL/gl.h"
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <chrono>
#include <random>
#include <algorithm>
#include <omp.h>
// #include <iostream> //breaks when compiling in atom for some reason

using namespace std;

//global variables for all functions to easily access
//x and v represent the current values of the particle positions and
    velocities
//f stores the current value of force acting on all particles
//posList keeps tracks of all particle positions over time
double ** x;
double ** v;
double ** f;
double *** posList;
double ** realPosList;//stores actual particle positions
double currentPotential = 0.;

int numParticles = 16;

//timestep
double dt = .005;
double currtime = 0.;
```

```
35
36  //actually the minimum number of ms to wait between frames
37  double framerate = 1;
38
39  //used for dynamically changing timestep. Copied method from Ryan
        Wilmington
40  double sigma = .0034;
41  int step = 1;
42  int numSteps = 20000;
43
44  //half of Side length of "box". L0/2. Makes it easier for distributions
45  double boxSize = 2.0;
46
47  //Heating factor
48  double g = 1.01;
49
50  //files for recording stuff
51  FILE *energyFile;
52  FILE *tempFile;
53  FILE *pos2File; //stores r^2
54  FILE *timeFile; //records time. Important if using variable timestep
55  FILE *comFile;
56  FILE *speedDistFile;
57
58  bool runGraphics = false;
59
60  //graphics functions
61  void drawPoints();
62  void update(int);
63  void updateNoGraphics();
64
65  //Initialization functions
66  double ** create2dArray(int,int);
67  double *** create3dArray(int,int,int);
68  double ** createRandomPositions(int);
69  double ** createRandomVelocities(int);
70  double ** createRandomVFromDist(int);
71
72  //copies x to certain time in posList
73  void recordPositions(int);
74
75  //physics calculations
76  double findMinDistance(int,int);
77  inline double forceCalc(double);
78  void calculateAllForces();
79
80  int main(int argc, char **argv) {
81
82    auto begin = chrono::high_resolution_clock::now();
83
84    //initializing positions and velocities
85    energyFile = fopen("energyOverTime.csv", "w+");
86    tempFile = fopen("temp.csv", "w+");
87    pos2File = fopen("rSquared.csv", "w+");
88    timeFile = fopen("time.csv", "w+");
89    comFile = fopen("centerOfMassVelocity.csv", "w+");
90    speedDistFile = fopen("speeddistributionfile.csv", "w+");
91    x = createRandomPositions(numParticles);
92    v = createRandomVelocities(numParticles);
```

```
  f = create2dArray(numParticles, 2);
  posList = create3dArray(numSteps+1, numParticles, 2);
  realPosList = create2dArray(numParticles, 2);
  recordPositions(0);

  //using trick to get second position
  for(int i=0; i<numParticles; i++) {
    x[i][0] = x[i][0] + v[i][0]*dt;
    x[i][1] = x[i][1] + v[i][1]*dt;
    realPosList[i][0] = x[i][0];
    realPosList[i][1] = x[i][1];
  }
  recordPositions(1);

  //Graphics stuff
  if(runGraphics) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE);

    //initial window size and position
    glutInitWindowSize(1000, 700);
    glutInitWindowPosition(500, 100);

    //Window title and declaration of draw function
    glutCreateWindow("Lennard Jones");
    glutDisplayFunc(drawPoints);

    //calls update function every "framerate" milliseconds
    //still limited by the speed of the algorithm though
    glutTimerFunc(framerate, update, 0);

    //returns you back to main() after simlation is over
    glutSetOption(GLUT_ACTION_ON_WINDOW_CLOSE,
        GLUT_ACTION_GLUTMAINLOOP_RETURNS);
    glutMainLoop();
  }//else should run without graphics
  else for(int i=0; i<numSteps; i++) update(0);

  //closing things, freeing memory
  fclose(energyFile); fclose(tempFile); fclose(pos2File); fclose(
      timeFile); fclose(comFile); fclose(speedDistFile);
  free(x); free(v); free(posList); free(f);

  auto end = chrono::high_resolution_clock::now();
  auto duration = chrono::duration_cast<chrono::milliseconds>(end -
      begin);
  printf("Done in %lf seconds\n", duration.count()/1000.);
  // getchar();
  return 0;
}

double ** create2dArray(int xdim, int ydim) {
  double ** v;
  v = (double **) malloc(xdim * sizeof(double *));
  for(int i=0; i<xdim; i++) {
    v[i] = (double *) malloc(ydim * sizeof(double));
  }
  return v;
}
```

```
149
150 double *** create3dArray(int xdim, int ydim, int zdim) {
151   double *** v;
152   v = (double ***) malloc(xdim * sizeof(double **));
153   for(int i=0; i<xdim; i++) {
154     v[i] = (double **) malloc(ydim * sizeof(double *));
155     for(int j=0; j<ydim; j++) {
156       v[i][j] = (double*) malloc(zdim*sizeof(double));
157     }
158   }
159   return v;
160 }
161
162 double ** createRandomPositions(int numParticles) {
163   double ** xNew = create2dArray(numParticles, 2);
164
165   //creates evenly spaced grid of particles.
166   //bin divides separation
167   //xi and yi are starting points for lattice
168   double bin = 1.;
169   double xi = -2.;
170   double yi = 0.;
171   int j = 0;
172   int k = 0;
173   for(int i=0; i<numParticles; i++) {
174     xNew[i][0] = xi + k*bin;
175     xNew[i][1] = yi + j*bin;
176     k++;
177     if((xi+k*bin) >= 2.) {
178       k = 0;
179       j++;
180     }
181   }
182   return xNew;
183 }
184
185 double ** createRandomVelocities(int numParticles) {
186   std::random_device rd;
187   std::mt19937 gen(rd());
188   std::uniform_real_distribution<> dis(-.0001, .0001);
189   double ** vNew = create2dArray(numParticles, 2);
190   double vxTot = 0.;
191   double vyTot = 0.;
192   for(int i=0; i<numParticles-1; i++) {
193     vNew[i][0] = dis(gen);
194     vNew[i][1] = dis(gen);
195     vxTot += vNew[i][0];
196     vyTot += vNew[i][1];
197   }
198   //fixes center of mass velocity
199   vNew[numParticles-1][0] = -vxTot;
200   vNew[numParticles-1][1] = -vyTot;
201   return vNew;
202 }
203
204 //update function that runs for every frame
205 void update(int value) {
206   //calculating forces on all particles
207   currentPotential = 0.;
```

```
208   calculateAllForces();
209   // printf("%d\n", step);
210   for(int i=0; i<numParticles; i++) {
211
212     //allows for heating or cooling of system
213     if(step % 100 == 0) {
214       // printf("%d\n", step);
215       double rpxPrime = posList[step][i][0] - g*(posList[step][i][0] -
              posList[step-1][i][0]);
216       double rpyPrime = posList[step][i][1] - g*(posList[step][i][1] -
              posList[step-1][i][1]);
217       posList[step-1][i][0] = rpxPrime;
218       posList[step-1][i][1] = rpyPrime;
219     }
220     //updates particle positions
221     x[i][0] = 2.*posList[step][i][0] - posList[step-1][i][0] + f[i][0]*
          pow(dt,2);
222     x[i][1] = 2.*posList[step][i][1] - posList[step-1][i][1] + f[i][1]*
          pow(dt,2);
223
224     //if particle is outside of box, wrap it to other side
225     //also need to translate previous steps in order for velocity to be
          stable
226     if(x[i][0] > boxSize) {
227       x[i][0] = x[i][0] - 2.*boxSize;
228       posList[step][i][0] = posList[step][i][0] - 2.*boxSize;
229       posList[step-1][i][0] = posList[step-1][i][0] - 2.*boxSize;
230     } else if(x[i][0] < -boxSize) {
231       x[i][0] = x[i][0] + 2.*boxSize;
232       posList[step][i][0] = posList[step][i][0] + 2.*boxSize;
233       posList[step-1][i][0] = posList[step-1][i][0] + 2.*boxSize;
234     }
235     if(x[i][1] > boxSize) {
236       x[i][1] = x[i][1] - 2.*boxSize;
237       posList[step][i][1] = posList[step][i][1] - 2.*boxSize;
238       posList[step-1][i][1] = posList[step-1][i][1] - 2.*boxSize;
239     } else if(x[i][1] < -boxSize) {
240       x[i][1] = x[i][1] + 2.*boxSize;
241       posList[step][i][1] = posList[step][i][1] + 2.*boxSize;
242       posList[step-1][i][1] = posList[step-1][i][1] + 2.*boxSize;
243     }
244     //incrementing real position
245     realPosList[i][0] += x[i][0] - posList[step][i][0];
246     realPosList[i][1] += x[i][1] - posList[step][i][1];
247   }
248
249   // printf("%lf %lf\n", x[0][0], x[0][1]);
250   //lets me look at the initial positions
251   // if(step==2) getchar();
252   //records positions
253   //does other calculations
254   if(step < numSteps) {
255     step+=1;
256     recordPositions(step);
257     double totalE = 0.;
258     double totalVx = 0.;
259     double totalVy = 0.;
260     double totalr2 = 0.;
261     double totalSpeed = 0.;
```

```
262
263     for(int i=0; i<numParticles; i++) {
264       v[i][0] = (posList[step][i][0] - posList[step-2][i][0]) / (2.*dt);
265       v[i][1] = (posList[step][i][1] - posList[step-2][i][1]) / (2.*dt);
266       double E = (pow(v[i][0], 2.) + pow(v[i][1], 2.))*.5;
267
268       //technically starting timer at 2nd step
269       double rx = realPosList[i][0] - posList[1][i][0];
270       double ry = realPosList[i][1] - posList[1][i][1];
271       double r2 = pow(rx, 2.) + pow(ry, 2.);
272
273       totalVx += v[i][0];
274       totalVy += v[i][1];
275       totalSpeed += sqrt(pow(v[i][0], 2.) + pow(v[i][1], 2.));
276       fprintf(speedDistFile, "%lf\n", sqrt(pow(v[i][0], 2.) + pow(v[i
              ][1], 2.)));
277       totalE += E;
278       totalr2 += r2;
279     }
280     currtime+=dt;
281     // printf("%lf\n", currtime);
282     fprintf(energyFile, "%lf\n", (totalE+currentPotential)/numParticles)
          ;
283     fprintf(pos2File, "%lf\n", totalr2/numParticles);
284     fprintf(comFile, "%lf\n", sqrt(pow(totalVx/numParticles, 2.) + pow(
          totalVy/numParticles, 2.)));
285     fprintf(timeFile, "%lf\n", currtime);
286     fprintf(tempFile, "%lf\n", pow(totalSpeed/numParticles, 2.)/(2.));
287
288       // for(int i=0; i<numParticles; i++) {
289       //   double spd = sqrt(pow(v[i][0], 2.) + pow(v[i][1], 2.));
290       //   // fprintf(tempFile, "%lf\n", spd);
291       // }
292
293     if(runGraphics) {
294       glutPostRedisplay();
295       glutTimerFunc(framerate, update, 0);
296     }
297   }
298   else {
299     if(runGraphics) glutLeaveMainLoop();
300   }
301 }
302
303 void drawPoints() {
304   //drawing functions. Not sure what they all do
305   glClearColor(0.4, 0.4, 0.4, 0.4);
306   glClear(GL_COLOR_BUFFER_BIT);
307   glOrtho(-1.0, 1.0, -1.0, 1.0, -1.0, 1.0);
308
309   //creates circles for particles somehow
310   glEnable(GL_POINT_SMOOTH);
311   glHint(GL_POINT_SMOOTH_HINT, GL_NICEST);
312
313   //particle size
314   glPointSize(20);
315   glBegin(GL_POINTS);
316
317   // double com[2] = {0., 0.};
```

```
318
319   //draws particles
320   glColor3f(1., 1., 1.);
321   for(int i=0; i<numParticles; i++) {
322     //need to scale particle positions by boxsize since GL window is
          only 1x1
323     glVertex3f(x[i][0]/boxSize, x[i][1]/boxSize, 0);
324     // com[0] += x[i][0];
325     // com[1] += x[i][1];
326   }
327   glColor3f(1., 0., 0.);
328   // glVertex3f(com[0]/(boxSize*numParticles), com[1]/(boxSize*
        numParticles), 0);
329
330   glEnd();
331   glFlush();
332 }
333
334 inline double forceCalc(double r) {
335   return 24. * (2./pow(r, 13.) - 1./pow(r,7.));
336 }
337
338 inline double potentialCalc(double r) {
339   return 4. * (1./pow(r, 13.) - 1./pow(r,7.));
340 }
341
342 void calculateAllForces() {
343   //reset force matrix to zero
344   for(int i=0; i<numParticles; i++) {
345     f[i][0] = 0.;
346     f[i][1] = 0.;
347   }
348
349   //defines global minimum for r to control timestep
350   double globalMin = 10000.;
351
352   // #pragma omp parallel for
353   for(int i=0; i<numParticles; i++) {
354     for(int j=i+1; j<numParticles; j++) {
355
356       //use fancy trick here to calculate shortest image distance
357       //found at https://dasher.wustl.edu/ from Jay Ponder's lecture
            notes
358       double rx = x[i][0] - x[j][0];
359       double ry = x[i][1] - x[j][1];
360       rx = rx - 2.*boxSize*floor(rx/(2.*boxSize) + .5);
361       ry = ry - 2.*boxSize*floor(ry/(2.*boxSize) + .5);
362       double rMag = sqrt(pow(rx, 2.) + pow(ry, 2.));
363
364       if(rMag < globalMin) globalMin = rMag;
365
366       //don't count force if particle is too far away
367       if(rMag > 3.) continue;
368
369       //calculate force normally
370       double fmag = forceCalc(rMag);
371       currentPotential += potentialCalc(rMag);
372       double force[2];
373       force[0] = fmag * rx/rMag;
```

```
374       force[1] = fmag * ry/rMag;
375       f[i][0] += force[0];
376       f[i][1] += force[1];
377       f[j][0] += -1.*force[0];
378       f[j][1] += -1.*force[1];
379     }
380   }
381   // dt = log(1+globalMin*sigma);
382   // printf("%lf\n", dt);
383 }
384 
385 void recordPositions(int t) {
386   for(int i=0; i<numParticles; i++) {
387     posList[t][i][0] = x[i][0];
388     posList[t][i][1] = x[i][1];
389   }
390 }
```