Assignment 5

Tyler Shendruk

April 4, 2010

1 Tyler Problem 1

1.1 Part a) Entropy

N hard disks in a box. Each disk excludes an area ω. There is hard-core repulsion between disks. So the number of states is

$$\Omega \propto \int_{\mathcal{H}=E} d^2q_1 d^2q_2 \dots d^2q_N d^2v_1 d^2v_2 \dots d^2v_N$$

As with an ideal gas, the momenta must lie on a hyper-sphere of radius

$$\sum_i |p_i| = \sqrt{2mE}. \tag{1}$$

This is analogous to an ideal gas. The surface area of a d-dimensional sphere is

$$\begin{aligned} A_d &= S_d R^{d-1} \\ &= \frac{2\pi^{d/2}}{(d/2-1)!} R^{d-1}. \end{aligned} \tag{2}$$

So the allowed momentum states must combine to fall on that sphere when $d = 2N$ and the radius is $R = \sqrt{2mE}$. When you do this also remember to just add by hand the over counting term $1/N!$.

$$\begin{aligned} \Omega &= \frac{1}{N!} \int_{\mathcal{H}=E} d^2q_1 d^2q_2 \dots d^2q_N d^2v_1 d^2v_2 \dots d^2v_N \\ &= \frac{1}{N!} \frac{2\pi^{2N/2}}{(2N/2-1)!} (2mE)^{(2N-1)/2} \int d^2q_1 d^2q_2 \dots d^2q_N. \end{aligned} \tag{3}$$

Of course, for an ideal gas $\int d^2q_1 d^2q_2 \dots d^2q_N = A^N$ but now the position is limited due to the presence of the other disks (A is area). We **approximate** this by placing them one after another:

- The first particle has area A availiable
- The second particle has $A - \omega$
- The third particle has $A - 2\omega$
- *etc...*

- The last particle has $A-(N-1)\omega$.

So then

$$\int d^2q_1 d^2q_2 \ldots d^2q_N \approx \prod_{i=1}^{N} \left[A-(i-1)\omega\right].$$

We use the approximation (not a very great one) that

$$(A-a\omega)\left[A-(N-a)\omega\right] \approx \left(A-\frac{N\omega}{2}\right)^2.$$

So then the number of states availiable to a hard-core disk is

$$\Omega = \frac{1}{N!}\frac{2\pi^N}{(N-1)!}(2mE)^{(2N-1)/2}\left[A-\frac{N\omega}{2}\right]^N.$$

We take the limit where $N \gg 1$ then

$$\Omega = \frac{2}{(N!)^2}(2\pi mE)^N\left[A-\frac{N\omega}{2}\right]^N \tag{4}$$

and therefore the entropy is

$$\begin{aligned}
S &= k_{\mathrm{B}}\ln\Omega \\
&= k_{\mathrm{B}}\ln\left[\frac{2}{(N!)^2}(2\pi mE)^N\left(A-\frac{N\omega}{2}\right)^N\right] \\
&= k_{\mathrm{B}}\left\{N\ln\left[2\pi mE\left(A-\frac{N\omega}{2}\right)\right]+\ln\left[\frac{2}{(N!)^2}\right]\right\} \\
&= k_{\mathrm{B}}\left\{N\ln\left[2\pi mE\left(A-\frac{N\omega}{2}\right)\right]+\ln 2-2\ln N!\right\} \\
&= k_{\mathrm{B}}\left\{N\ln\left[2\pi mE\left(A-\frac{N\omega}{2}\right)\right]+\ln 2-2N\ln N+2N\right\} \\
&= k_{\mathrm{B}}\left\{N\ln\left[2\pi mE\left(A-\frac{N\omega}{2}\right)\right]-N\ln N^2+N\ln e^2+\ln 2\right\} \\
&= Nk_{\mathrm{B}}\ln\left[\frac{2\pi e^2 mE}{N^2}\left(A-\frac{N\omega}{2}\right)\right]+k_{\mathrm{B}}\ln 2
\end{aligned} \tag{5}$$

and when we assume $N \gg 1$ then the first term is significantly larger than the constant $k_{\mathrm{B}}\ln 2$ and the entropy is

$$\boxed{S = Nk_{\mathrm{B}}\ln\left[\frac{2\pi e^2 mE}{N^2}\left(A-\frac{N\omega}{2}\right)\right]} \tag{6}$$

1.2 Part b) Equation of State

From

$$\begin{aligned}
dE &= đQ + dW \\
&= TdS + \sum Jdx \\
&= TdS - \Pi dA
\end{aligned}$$

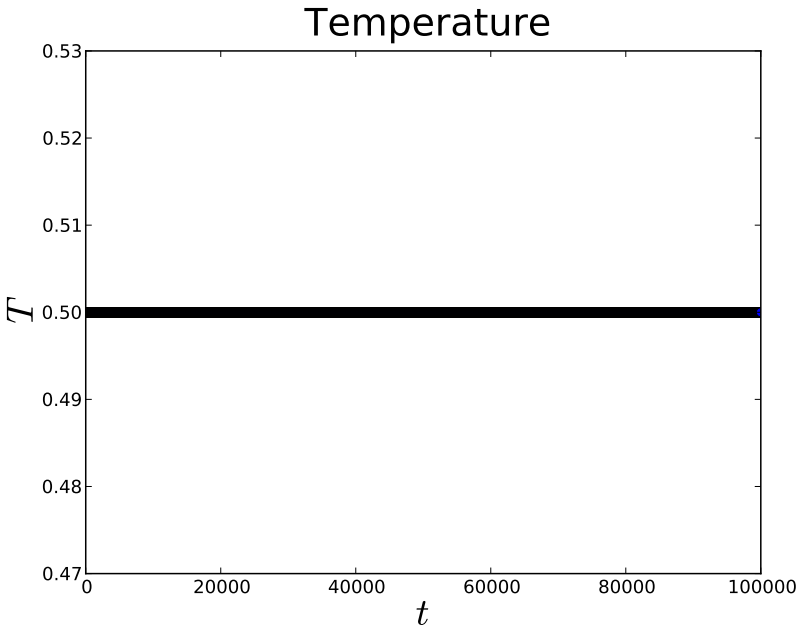


Figure 1: The temperature has not fluctuations.

where Π is the surface pressure (*i.e.* force per unit length) we must have

$$\frac{\Pi}{T} = \left.\frac{\partial S}{\partial A}\right|_{E,N} \tag{7}$$

This gives us an equation of state

$$\boxed{\Pi\left(A - \frac{Nw}{2}\right) = Nk_{\mathrm{B}}T} \tag{8}$$

1.3 Part c) Modify Code

It is nice if you define a constant parameter maybe called `DIM` and altered the program to allow for any dimension but it is perfectly acceptable if you altered the code to be just for a 2D gas of disks. The appendix to this assigment shows what I did.

The other way to do this is to **only** alter the position and velocity initialization routines to explicitly set the $\hat{z}$-components to zero. The code can then run in 3D all it wants but the spheres stay in a plane and act like disks.

1.4 Part d) Temperature Fluctuations

There are no temperature fluctuations. The closed system is initiated with some energy E_0 and each collision conserves energy so there is no source or sink of energy. Using the values in Part 1.5, I get Fig. 1.

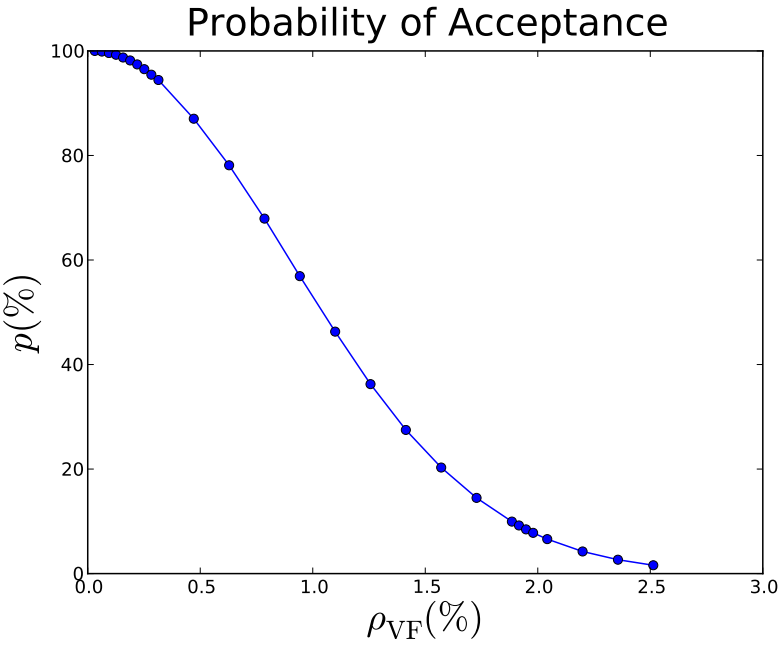


Figure 2: The probability for acceptance in a 100×100 box decays rapidly with the area occupied by disks.

1.5 Part e) System Set-up

How many disks should we be able to put into the box? Since they are disks, they close-pack at high enough density. So if you consider them as tight together as possible and draw a unit cell around one of the disks, then the unit-cell forms a hexagon. A hexagon is made of 6 equilateral triangles each of area $A_T = \sqrt{3}r^2$. Each triangle has 3 sixths (or one half) circle. Draw it. It's easy to see drawn. So the close-packed volume fraction is

$$V_0 = \frac{A_C/2}{A_T} = \frac{\pi r^2}{2\sqrt{3}r^2} = \frac{\pi}{2\sqrt{3}} \approx 0.91 \tag{9}$$

Of course, the walls will have an edge effect on that number but the spirit is that we should be able to fit lots more disks than you might have been able to. The limiting factor isn't the simulation but the initialization. Therefore, the big worry isn't the population but the volume fraction of disks. In order to demonstrate this to you, I've made a little loop:

```
tries = 0;
for(i=0;i<ACCEPT;i++) {
  tries = tries + setPos(pos);
}
fprintf(fileAcc,"%e %e\n",density(),ACCEPT/tries);
```

Fig. 2 shows they way that this develops. The probability of getting a successful initialization becomes smaller and smaller quite rapidly until it takes you a large amount of time to initialize.

The point of all this is to say that you should use a large volume/area and a large number of disks for this project. A volume fraction of only 0.025 (80

dimension	population	length	radius	mass	energy	simulation time	time step
2	125	100.	1.	1.	2.5	10000	1.

Table 1: Values inputted for simulation unless otherwise stated.

disks) has only 1.5% probability of accpetance. I used the values in Table 1

1.6 Part f) Physical Units

When dealing with computers, I think it is best to think of the variables as having units that you define *i.e.* you might have chosen

$$\begin{aligned} m &= 1 \\ r &= 1 \\ k_{\mathrm{B}}T &= 1 \end{aligned}$$

which define your units as

$$\begin{aligned} m &= 1[m] \\ r &= 1[\ell] \\ k_{\mathrm{B}}T &= 1[\epsilon]. \end{aligned}$$

So if the atomic weight is $40g/\mathrm{mol}$, we know 1[m]

$$\begin{aligned} 1[m] &= \frac{40g}{\mathrm{mol}} \times \frac{\mathrm{mol}}{6 \times 10^{23}} \times \frac{1kg}{1000g} \\ &= 6.6 \times 10^{-26} kg. \end{aligned}$$

The length is

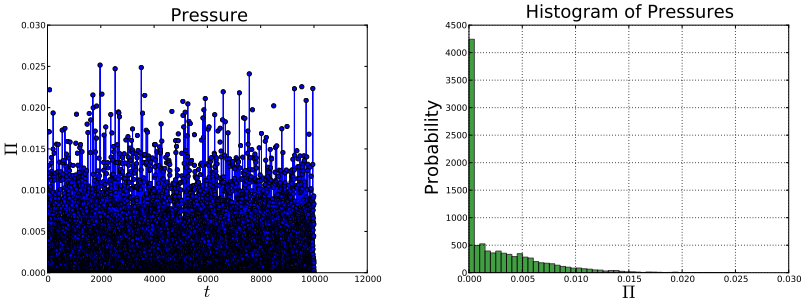
$$\begin{aligned} 1[\ell] &= 4\text{Å} \\ &= 4 \times 10^{-10} m \end{aligned}$$

and the energy scale is

$$\begin{aligned} 1[\epsilon] &= k_{\mathrm{B}}T \\ &= 1.38 \times 10^{-23} \frac{J}{K} \times 303K \\ &= 4.2 \times 10^{-21} J \end{aligned}$$

Now we have more clearly all the information to find $t = 1[t]$ in seconds. The link is energy

$$\begin{aligned} [\epsilon] &= \frac{[m][\ell]^2}{[t]^2} \\ [t] &= \sqrt{\frac{[m][\ell]^2}{[\epsilon]}} \\ [t] &= \sqrt{\frac{6.7 \times 10^{-26} kg \times (4 \times 10^{-10} m)^2}{4.2 \times 10^{-21} J}} \end{aligned}$$



(a) Variations in surface pressure over time. (b) Distribution of surface pressure.

Figure 3: Surface pressure.

which means

$$\boxed{[t] = 1.6 \times 10^{-12} s}. \tag{10}$$

Only 1.6 picoseconds.

1.7 Part g) Pressure

The surface pressure is expected to be close to Eq. 8 but it has that terrible approximation in it. On the other hand, the pressure you measure from the program should be the actual pressure of your system but uses a very finite number of disks.

To measure the pressure in your system, you must add a running sum of the impulse on the walls to `pwCollision`. This should resemble

```
impulse[w] += 2.*MASS*dot(norm,vel1)/2.;
```

if you keep track of the impulse on each wall or just

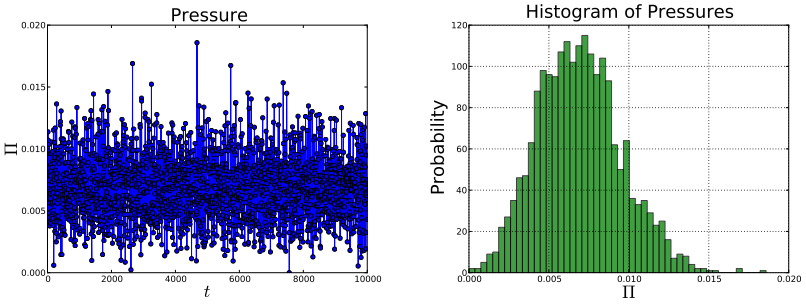
```
impulse += \fabs(2.*MASS*dot(norm,vel1)/2.);
```

if you only record the total magnitude. Within the routine `snapShot` the impulse should then be transformed into pressure

```
for(j=0;j<2*DIM;j++) pressure += fabs(impulse[j]/(STEP*area));
pressure /= (double)(2*DIM);
fprintf( fp,"%e %e\n",stepTime,pressure );
for(j=0;j<2*DIM;j++) impulse[j] = 0.;
```

The pressure is recorded and then the running sum of the impulse is zeroed for the next time. For the input values in Table 1, the output should resemble Fig. 3a. Unlike the temperature, the pressure should fluctuate but notice if we do some quick statistics on the distribution of surface pressure with time the fluctuations aren't symmetric as seen in Fig. 3b and even worse most of the time the pressure is zero. That's because between most snap shots no particle has hit the wall. The solution to this is to make the variable `Step` larger, say alter it from $\Delta t = 1$ to $\Delta t = 5$ then we get much better results as shown in Fig. 4. There may still be a bit of skewness but it is much better than in Fig. 3. The pressure is determined to be

$$\Pi = 0.0069 \pm 0.0026 \tag{11}$$



(a) Variations in surface pressure over time. (b) Distribution of surface pressure.

Figure 4: Surface pressure.

which is pretty close to what it was with $\Delta t = 1$ too.

How does Eq. 11 compare to Eq. 8? By substituting in the values in Table 1 we get

$$\begin{aligned}\Pi &= \frac{Nk_{\mathrm{B}}T}{\left(A - \frac{Nw}{2}\right)} \\ &= \frac{Nk_{\mathrm{B}}T}{\left(A - \frac{N(\pi(2r)^2)}{2}\right)} \\ &= \frac{125 \times 0.5}{(100^2 - 125 \times 2 \times \pi)} \\ &= 0.00678\end{aligned}$$

which is easily within the uncertainty of the simulation. In fact, they're under 2% different.

A input.h

```
#ifndef INPUT_H
#define INPUT_H
/* Input parameters - they are the only all capitalized variables */
# define DIM 2
# define POP 125
# define LENGTH 100.
# define RADIUS 1.
# define MASS 1.
# define ENERGY 0.5*POP*MASS * 1.
# define TIME 10000
# define STEP 1.
# define BIG_NUM 1E250
# define ACCEPT 100000
/* Numerical constants */
# define pi acos(-1.)
# define e 2.7182818284590
#endif
```

B experiments.h

```
#include "input.h"
#ifndef EXPERIMENTS_H
#define EXPERIMENTS_H
double temperature(double vel[][DIM]);
double energy(double vel[][DIM]);
void volume();
double density();
#endif
```

C experiments.c

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

/* My headers */
#include "experiments.h"

double temperature(double vel[][DIM]) {
  double kbt,v2;
  int i,j;

  kbt = 0.;
  for(i=0;i<POP;i++) {
    v2 = 0.;
    for(j=0;j<DIM;j++) v2 += vel[i][j]*vel[i][j];
    kbt = kbt + MASS*v2;
  }
  kbt = kbt/((double)(DIM*POP));
  return kbt;
}
double energy(double vel[][DIM]) {
  double energy,v2;
  int i,j;

  energy = 0.;
  for(i=0;i<POP;i++) {
    v2 = 0.;
    for(j=0;j<DIM;j++) v2 += vel[i][j]*vel[i][j];
    energy = energy + MASS*v2/2.;
  }
  return energy;
}
void volume() {
  double vol, excluded;
  if(DIM==3) {
    vol = 4.*pi*RADIUS*RADIUS*RADIUS/3.;
```

```
    excluded = 8.*4.*pi*RADIUS*RADIUS*RADIUS/3.;
  }
  else if (DIM==2) {
    vol = pi*RADIUS*RADIUS;
    excluded = pi*4.*RADIUS*RADIUS;
  }
  else if (DIM==1) {
    vol = RADIUS;
    excluded = 2.*RADIUS;
  }
  else printf("Hyper-spheres not allowed\n");

  printf("The volume of each hard sphere is: %lf\n",vol);
  printf("The excluded volume of each hard sphere is: %lf\n",excluded);
}
double density() {
  double dens;
  if(DIM==3) {
    dens = 4.*pi*RADIUS*RADIUS*RADIUS/3.;
    dens = dens*(double)POP;
    dens = dens / (LENGTH*LENGTH*LENGTH);
  }
  else if(DIM==2) {
    dens = pi*RADIUS*RADIUS;
    dens = dens*(double)POP;
    dens = dens / (LENGTH*LENGTH);
  }
  else if(DIM==1) {
    dens = RADIUS*(double)POP/LENGTH;
  }
  else {
    printf("Hyper-spheres not allowed\n");
    dens = 0.;
  }
  return dens;
}
```

D hard.c

```
/*************************************************************************
*  hard.c: Simulation of hard sphere gas                                 *
*          Includes: Temperature measurements                            *
*                    Pressure measurements                               *
*                    Density and acceptance measurements                 *
*                    Collision rate                                      *
*                    Steroscopic output                                  *
*                    Mean square displacement                            *
*                    Variable dimension                                  *
*                    Cleaned up to show class                            *
*                                                                        *
*  written by: Tyler Shendruk                                            *
*              at University of Ottawa                                   *
*  writen for: PHY5330 Statistical Mechanics                             *
*  writen on:  Jan/Feb 2010                                              *
*                                                                        *
*  Compile by: gcc hard.c experiments.c -o hard.out -lm                  *
*************************************************************************/

/* C-headers */
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <math.h>
#include <time.h>

/* My headers */
#include "experiments.h"
#include "input.h"

/* Initialize subroutines */
void stream(double streamTime,double pos[][DIM],double vel[][DIM]);
```

```
double ppTime(double pos1[DIM],double vel1[DIM],double pos2[DIM],double vel2[DIM]);
double pwTime(double pos[DIM],double vel[DIM]);
void ppCollision(double pos1[DIM],double vel1[DIM],double pos2[DIM],double vel2[DIM]);
void pwCollision(double pos1[DIM],double vel1[DIM],double impulse[2*DIM]);
double dRand(void);
float gRand(void);
int setPos(double pos[][DIM]);
void setVel(double vel[][DIM]);
void snapShot(double runTime,double collisionTime,double pos[][DIM],double vel[][DIM],double impuls
double dot(double x[DIM],double y[DIM]);
void print2Terminal(double pos[][DIM],double vel[][DIM]);

/*****************************************************************
*  The() main routine:                                           *
*    1. Places the particles                                     *
*    2. Sets velocities                                          *
*    3. Determines the time to the next pair collision           *
*    4. Determines the time to the next wall collision           *
*    5. Streams the particles till the next collision            *
*    6. Solves the implact problem                               *
*    7. Returns to Step 3                                        *
*****************************************************************/
int main(int argc, char* argv[])
{
  double tempTime=0.,pairTime=0., wallTime=0.,collisionTime=0., runTime=0.;
  double pos[POP][DIM],vel[POP][DIM];
  double impulse[2*DIM] = {0.};
  int i,j,p1,p2,w1;
  int tries=0,numCollisions=0;
  char ft[] = "temperature.dat", fp[] = "pressure.dat", fa[] = "prob.dat";;
  FILE *fileTemp, *filePres, *fileAcc;

  fileTemp=fopen(ft, "w+");
  filePres=fopen(fp, "w+");
  fileAcc=fopen(fa, "a+");

  printf("Initialize system ...\n");
  srand((unsigned)time(NULL));                                  /* Initialize random number generat
//    for(i=0;i<ACCEPT;i++) {
//      tries = tries + setPos(pos);                             /* Initiialize positions */
//    }
//    fprintf(fileAcc,"%e %e\n",density(),((double)ACCEPT/(double)tries));
  setPos(pos);                                                  /* Initiialize positions */
  setVel(vel);                                                  /* Initialize velocities */
  printf("kbt=%lf\n",temperature(vel));

  printf("Simulation ..\n");
  while(runTime < TIME) {                                       /* The main temporal loop */

    pairTime = BIG_NUM;                                         /* Initialize the time to "infiity"
    for(i=0;i<POP;i++) for(j=i;j<POP;j++) {                     /* Determine the time to the next p
        tempTime = ppTime(pos[i],vel[i],pos[j],vel[j]);         /* Calculate the collision time for
        if(tempTime < pairTime && tempTime > 0.) {              /* See if it is the current minimum
          pairTime = tempTime;
          p1=i;
          p2=j;
        }
    }

    wallTime = BIG_NUM;                                         /* Initialize the time to "infiity"
    for(i=0;i<POP;i++) {                                        /* Determine the time of the next w
      tempTime = pwTime(pos[i],vel[i]);                         /* Calculate the collision time */
      if(tempTime < wallTime && tempTime > 0.) {                /* See if it is the current minimum
        wallTime = tempTime;
        w1=i;
      }
    }

    if (pairTime < wallTime) collisionTime = pairTime;          /* The next collision is min(pairTi
    else collisionTime = wallTime;

    snapShot(runTime,collisionTime,pos,vel,impulse,fileTemp,filePres);

    stream(collisionTime,pos, vel);                             /* Stream till then */
```

```
    runTime = runTime + collisionTime;

    if (pairTime < wallTime) {                              /* Apply the collision */
      ppCollision(pos[p1],vel[p1],pos[p2],vel[p2]);
      numCollisions = numCollisions + 1;
    }
    else {
      pwCollision(pos[w1],vel[w1],impulse);
    }

    for(i=0;i<POP;i++) for(j=0;j<DIM;j++) {                 /* Check for leak */
      if(pos[i][j]<0. || pos[i][j]>LENGTH) {
        printf("Particle escaped box\n");
        return 0;
      }
    }
  }

  fclose(fileTemp);
  fclose(filePres);
  return 0;
}


/*************************************************************************
*  stream() subroutine moves the particles forward in time               *
*************************************************************************/
void stream(double streamTime,double pos[][DIM],double vel[][DIM]) {
  int i,j;
  for(i=0;i<POP;i++) for(j=0;j<DIM;j++) pos[i][j] = pos[i][j] + streamTime*vel[i][j];

}

/*************************************************************************
*  ppTime() returns the time of collision between particle 1 and 2      *
*************************************************************************/
double ppTime(double pos1[DIM],double vel1[DIM],double pos2[DIM],double vel2[DIM])
{
  double collisionTime = BIG_NUM;
  double deltaX[DIM], deltaV[DIM], dotXX, dotVV, dotXV, Y;
  int i;

  for(i=0;i<DIM;i++) {
    deltaX[i] = pos2[i]-pos1[i];
    deltaV[i] = vel2[i]-vel1[i];
  }
  dotXX = dot(deltaX,deltaX);
  dotVV = dot(deltaV,deltaV);
  dotXV = dot(deltaX,deltaV);                               /* The particles must be approachin
  Y = dotXV*dotXV - fabs(dotVV)*(fabs(dotXX)-4.*RADIUS*RADIUS); /* The square root must be real */

  if(Y > 0. && dotXV < 0.) {
    collisionTime = -(dotXV+sqrt(Y))/dotVV;

    if( collisionTime < 0. ) {                              /* Reject negative times */
      collisionTime = -(dotXV-sqrt(Y))/dotVV;
    }
  }
  else collisionTime = BIG_NUM;

  return collisionTime;
}

/*************************************************************************
*  pwTime() returns time of collision between a particle and a wall     *
*************************************************************************/
double pwTime(double pos[DIM],double vel[DIM]) {
  double tempTime,collisionTime;
  int i;

  collisionTime = BIG_NUM;

  for(i=0;i<DIM;i++) {                                      /* The three walls that are in the
    tempTime = (0.+RADIUS-pos[i])/vel[i];
```

```
    if(tempTime > 0. && tempTime < collisionTime && vel[i] < 0.) {
      collisionTime = tempTime;
    }
  }

  for(i=0;i<DIM;i++) {                                          /* The three other walls */
    tempTime = (LENGTH-RADIUS-pos[i])/vel[i];
    if(tempTime > 0. && tempTime < collisionTime && vel[i] > 0.) {
      collisionTime = tempTime;
    }
  }

  return collisionTime;
}

/*****************************************************************
*  ppCollision() computes velocities after a pair collision      *
*****************************************************************/
void ppCollision(double pos1[DIM],double vel1[DIM],double pos2[DIM],double vel2[DIM])
{
  double norm[DIM]={0.}, deltaX[DIM]={0.}, deltaV[DIM]={0.}, changeV[DIM]={0.};
  double dotXX;
  int i;

  for(i=0;i<DIM;i++) deltaX[i] = pos2[i]-pos1[i];
  dotXX = dot(deltaX,deltaX);
  for(i=0;i<DIM;i++) norm[i] = deltaX[i]/sqrt(dotXX);           /* Find normal vector */

  for(i=0;i<DIM;i++) deltaV[i] = vel2[i]-vel1[i];

  for(i=0;i<DIM;i++) changeV[i] = norm[i]*dot(deltaV,norm);     /* New velocities */

  for(i=0;i<DIM;i++) {
    vel1[i] = vel1[i]+changeV[i];
    vel2[i] = vel2[i]-changeV[i];
  }
}

/*****************************************************************
*  pwCollision() computes velocity after collision with a wall   *
*                and adds the impulse to the running total       *
*****************************************************************/
void pwCollision(double pos1[DIM],double vel1[DIM],double impulse[2*DIM])
{
  int i,j,w;
  double test, norm[DIM]={0.};
  double dist=LENGTH;

  for(i=0;i<DIM;i++) {                                          /* The three walls that are in the
    test = pos1[i];
    if(test < dist ) {
      for(j=0;j<DIM;j++) norm[j] = 0.;
      norm[i] = -2.;
      dist = test;
      w = i;
    }
  }

  for(i=0;i<DIM;i++) {                                          /* The three other walls */
    test = LENGTH-pos1[i];
    if(test < dist ) {
      for(j=0;j<DIM;j++) norm[j] = 0.;
      norm[i] = -2.;
      dist = test;
      w = i+DIM;
    }
  }

  impulse[w] += 2.*MASS*dot(norm,vel1)/2.;                      /* Record the impulse */

  for(i=0;i<DIM;i++) {                                          /* Alter the velocity */
    vel1[i] = vel1[i] + norm[i]*vel1[i];
  }
}
```

```
/*************************************************************************
*  dRand() returns a type double random number                           *
*************************************************************************/
double dRand(void) {
    return rand()/((double)(RAND_MAX));
}

/*************************************************************************
   gRand() is a Box-Muller transformation to                             *
   turn a uniform random number between 0-1 into                         *
   a gaussian distribution of mean 0 and StDev of 1.                     *
*************************************************************************/
float gRand(void){
        float x1,x2,w,y1,y2;
        do {
                x1=2.0*dRand()-1.0;
                x2=2.0*dRand()-1.0;
                w=x1*x1+x2*x2;
        }while (w>=1.0);
        w=sqrt((-2.0*log(w))/w);
        y1=x1*w;
        y2=x2*w;
        return y1;
}

/*************************************************************************
*  setPos() randomly initializes the particle positions                  *
*************************************************************************/
int setPos(double pos[][DIM]) {
  int i,j,k;
  int check=0,tries=0;
  double dist;

  do {
    check = 0;
    tries = tries + 1;
    for(i=0;i<POP;i++) {
      for(j=0;j<DIM;j++) pos[i][j] = dRand() * (LENGTH-2.*RADIUS)+RADIUS;

      for(j=0;j<i;j++) {
        dist = 0.;
        for(k=0;k<DIM;k++) dist += (pos[i][k]-pos[j][k])*(pos[i][k]-pos[j][k]);
        dist = sqrt(dist);
        if(dist < 2.*RADIUS) check = 1;
      }
    }
  } while(check);
  return tries;
}

/*************************************************************************
*  setVel() randomly initializes velocities on hypersphere               *
*************************************************************************/
void setVel(double vel[][DIM]) {
  double stdev,sqrtDim,sum;
  int i,j;

  stdev = ENERGY;
  stdev = sqrt(2.*stdev/MASS);

  sum = 0.;
  sqrtDim = 1./sqrt( (double)(DIM*POP) );
  for(i=0;i<POP;i++) {
    for(j=0;j<DIM;j++) vel[i][j] = gRand() * sqrtDim;
    for(j=0;j<DIM;j++) sum += (vel[i][j]*vel[i][j]);
  }
  sum = sqrt(sum);
  for(i=0;i<POP;i++) for(j=0;j<DIM;j++) vel[i][j] = vel[i][j]*stdev / sum;
}

/*************************************************************************
*  snapShot() writes values to files at regular intervals                *
*************************************************************************/
```

```
void snapShot(double runTime,double collisionTime,double pos[][DIM],double vel[][DIM],double impuls
  int imin,imax,i,j;
  double stepTime;
  double pressure = 0.;
  double area;

  imin = (int) (runTime/STEP) + 1;
  imax = (int) ((runTime+collisionTime)/STEP);
  if(DIM == 3) area = (double)(LENGTH*LENGTH);
  else if(DIM == 2) area = (double)(LENGTH);
  else if(DIM == 1) area = 1.;

  for(i=imin;i<=imax;i++) {
    stepTime = STEP*(double)i;
    fprintf( ft,"%e %e\n",stepTime,temperature(vel) );

    for(j=0;j<2*DIM;j++) pressure += fabs(impulse[j]/(STEP*area));
    pressure /= (double)(2*DIM);
    fprintf( fp,"%e %e\n",stepTime,pressure );
    for(j=0;j<2*DIM;j++) impulse[j] = 0.;

  }
}

/*************************************************************************
*  dot() returns the dot product between two vectors                     *
*************************************************************************/
double dot(double x[DIM],double y[DIM]) {
  int i;
  double d = 0.;
  for(i=0;i<DIM;i++) d += x[i]*y[i];
  return d;
}

/*************************************************************************
*  print2Terminal() prints coordinates and velocity to the terminal     *
*************************************************************************/
void print2Terminal(double pos[][DIM],double vel[][DIM]) {
  int i,j;
  for(i=0;i<POP;i++) {
    printf("%d\t[%lf",i,pos[i][0]);
    for(j=1;j<DIM;j++) printf(",%lf",pos[i][j]);
    printf("]\n");
    printf("%d\t[%lf",i,vel[i][0]);
    for(j=1;j<DIM;j++) printf(",%lf",vel[i][j]);
    printf("]\n");
  }
}
```