N-Body Simulation of Ideal and Non-Ideal Gas

Shivam Garg[1]

[1]*Indian Institute of Science Education and Research Kolkata*

We simulate a 2-D gas using the basic assumptions of the kinetic theory of gases. The Boyle's law is proven and the Maxwell-Boltzmann curve is also calculated. We also use the Lennard-Jones potential and simulate N particles interacting via the potential.

INTRODUCTION

We perform an N body simulation of an ideal gas. All the basic assumptions and the techniques used (and not used) are described in the article. This project was done under the supervision of Dr. Ananda Dasgupta for the PH4201 Advanced Experimental Physics course. The program was written in Python and the plotting was done in Gnuplot.

DISTRIBUTION OF RANDOM VECTOR

Our first task was to calculate the distribution of random points in a 2-D box. It was expected that in a rectangular 2-D box, we would find more number of particles along the diagonal rather than the coordinate axes. We were expecting a distribution of the kind

$$f(\theta) = \begin{cases} \frac{1}{\sin\theta} & 0 < \theta < \frac{\pi}{4} \\ \frac{1}{\cos\theta} & \frac{\pi}{4} < \theta < \frac{\pi}{2} \end{cases}$$

where θ is the angle (positive in the counterclockwise direction).We expect from geometry of the system that there will $\sqrt{2}$ times number of particles more in the diagonal direction as compared to the coordinate axes

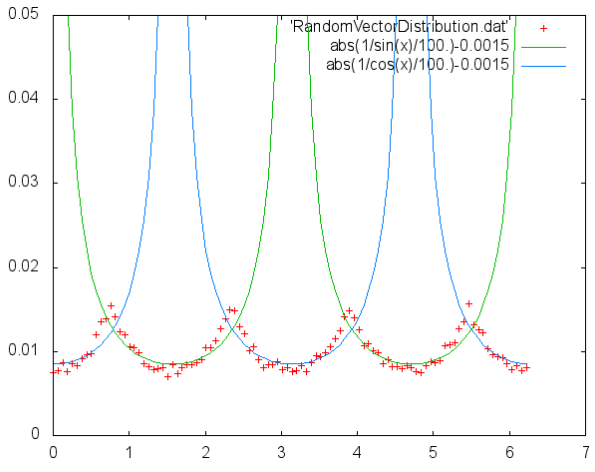

FIG. 1: Distribution of random points along the unit circle (x axis - θ (in radians) y axis - normalised vector count)

direction.

The reason we wanted to calculate the distribution was that we planned to take velocity vector for each particle along the direction of its position vector. If there was indeed a bias in the distribution we intended to remove it.

The method was as follows. We distributed N number of particles (where N was of the order of 10000) in a box by taking random x and y position components between [-1,1]. Then we projected all the position vectors onto its corresponding place on the unit circle (keeping the direction fixed, but changing the magnitude to unit magnitude). Henceforth, for each small interval dθ we calculated the number of points lying in that interval and plotted it. We fitted the resulting data with the expected function and we find that there is certainly a distribution as we expected. Figure 1 shows the data and the fit.

COLLISION OF PARTICLES WITH THE WALLS

Since the previous approach did not work out we proceeded to take random velocity magnitude and random direction to produce a random velocity vector for each particle. We were successful but at the cost of increased computation. We had random velocity magnitude and directions, but to make our job simpler and for us to comprehend the problem more easily, we split the velocity into it's x and y components taking the cosine and sin respectively. This increases the amount of computation significantly. But as stated, to make things more comprehensible we went ahead with the component approach. At each time step the x and y position component were updated according to the corresponding velocity component.

We first tackled the problem of particles colliding with the walls. The basic idea was simple. If a particle collides with the wall , i.e., crosses the boundary of the box, invert the velocity component corresponding to the wall it hit. Suppose the particle hits the wall along the x-direction. Then we switch the sign of the x component of the velocity. Similarly for the walls along the y direction.

This approach was ideal except for the fact that

our program did not have a continuous time parameter. Each iteration was done for a certain time step dt. We ran into collision detection problems. It can be explained as follows. Suppose we have a particle that moves a very high velocity. If the time step is not small enough, then between the first and the next interval the particle may cross the wall partially or completely without getting reflected. To counter this, we could have taken a very small dt. But unfortunately, that would take up a lot of processor speed. So we could not help the flaw with our approach. We relied on the fact that when we actually do simulations we take large enough number of particles so that these effects cancel out on an average. The collision detection problem reoccurs in a bigger way during collision between two particles.

COLLISION OF TWO PARTICLES

Next, we tried to incorporate the innocent looking but quite complex phenomenon of collision between two particles into our program. Higher order collisions (3 particles or more) were ignored for the sake of simplicity.

Collision was done as follows. Each particle was tested against every other particle for collision. This was an aspect where we could clearly improve. We could test each particle against the particles only in its vicinity and not every particle howsoever far away. This would greatly reduce the computational complexity. There are algorithms such as the Barnes-Hut algorithms which look into these issues. That is a future direction we would want to look into.

Two particles collided if the distance between their centers was less than twice the radius (the radius of each particle is considered the same in our analysis). If the particles fulfilled the above condition, i.e., they were undergoing collision, the new particle velocities were found as follows. We know from classical mechanics that if two particles undergo a collision, the force that acts on both the particles is along the direction connecting the two centers. Moreover, if the masses of both the particles are same, then basically the velocity components exchange along the line joining the centers of the particles. We used this fact to calculate the final velocity components.

We calculated the angle between the line joining the centers of the particles and the x axis. We transformed the velocity components in the original frame to the rotated frame by multiplying with the rotation matrix. We exchanged the velocity component along the x direction (of the rotated frame) and then transformed the components in the rotated frame to the original frame using the inverse transformation. We then advance both the particles by one or two time steps so that they don't appear to collide in the next iteration.

As mentioned in the previous section, here also we run into problems involving collision detection due to having a finite time step. The major problem here is that if the time step is not sufficiently small enough, the particles may actually go across each others boundaries without colliding. Even if the program detects in the next iteration that they have collided (or maybe gone through) and runs the collision module, they may not be able to go far enough so that they are not colliding any more. If the velocity is not high enough, then the particles may not be able to go far enough and in the next iteration also they will be detected as collided. Due to this, particles sometimes clump together. But again, we expect that if we take a large number of particles, these irregularities will be averaged out.

We could try other techniques for solving the problem of collision detection as well. I again propose some suggestions for pursuing in the future. We could try to use an adaptive time step sort of technique. The adaptive time step technique is the technique where we keep changing our time step according to collisions. If the distance between any two particles is less than a certain value (say 4 times the radius), i.e., the particles might collide, we reduce the time step. We actually tried it but we had nominal success. This is because any two particles might be in the vicinity quite often which would reduce the time step for every particle. That would mean that effectively the time step would be reduced time step for most of the cases. That increased computational complexity frequently.

Another technique we could try is calculating when exactly the particles would collide and then calculate the velocities after collision. We did not try this method but this could have been a very effective technique for detecting collision. Another approach could be taking "softening parameters", basically certain parameters which help early collision detection. This approach also had nominal success.

MAXWELL BOLTZMANN CURVE

With the above code written, we had the framework laid for an ideal gas. Now we set to obtain the Maxwell-Boltmann distribution using the ideal gas simulation. The Maxwell-Boltmann distribution for a 2-D system is

$$f(v) = 2\pi \frac{m}{2\pi kT} v e^{-\frac{mv^2}{2kT}} \tag{1}$$

We needed to produce the speed distribution of the

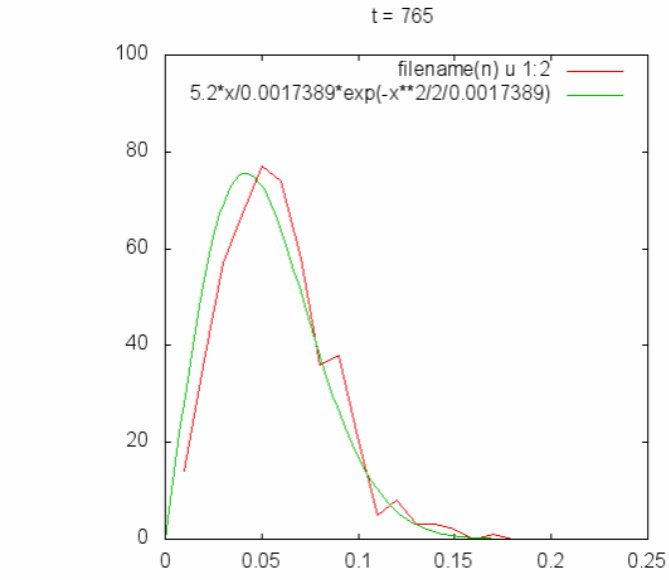

FIG. 2: 2-D Maxwell Boltzmann distribution with fit
(x axis - speed v, y axis - Number of particles with speed v)

particles. For each time step iteration, we counted the number of particles within a certain velocity range and plotted it. The graph shown is for time t=765. The initial plots were discarded because the system needs some time to reach equilibrium. We see that the obtained curve (in red) matches well with the theoretical prediction (in green). (The theoretical distribution needs to be multiplied by the total number of particles, since the graph we obtained is not normalised.)

Important Note : To calculate the theoretical distribution, we need to calculate the temperature first. We know from kinetic theory of gases that the average kinetic energy per particle is equal to $k_B T$ in 2 dimensions. So, we calculate the total kinetic energy at the start of the program. Then divide that by the number of particles taken N and $k_(B)$ to get the temperature T. The major problem here is that during calculation of kinetic energy we set mass to 1. Due to that reason the temperature turns out to be of the order 10^{20} which is ridiculous in normal units but since we set our mass to be 1 (which is ridiculous in itself, a small particle having a mass of 1 kg) we do not have any inconsistency in our result. It's just that we failed to properly define our units and kept doing so as per our convenience. In the whole program, the mass is set to unity.

BOYLE'S LAW

Next, we turn our attention to calculating the pressure on the box walls due to the particles and hence verifying Boyle's law by finding out the P vs. V curve. We should note that since this is a 2-D calculation, volume here is replaced by area and pressure as force per unit area is replaced by force per unit length. This, as we'll show, will not affect our calculations or the result.

Until now, we had kept the box length fixed. But to calculate P vs. V, we need to vary the box length as well. If we square the box length, we get the area of the box. For calculating the pressure, we need to keep track of the particles hitting the wall. We apply basic concepts of kinetic theory of gases. Whenever a particle hits the wall, it experiences a momentum change of $2mv_x$ or $2mv_y$ depending on which wall it hit. We keep track of all such momentum changes, both in the x and y direction (ideally we should have equal contributions to pressure from both walls, since the initial velocity vector of any particle is random). Then, we need to calculate the pressure using this momentum change. In principal, what we should do is divide the momentum change by dt and then divide by the box length to obtain the pressure.

Again, since we do not have continuous variables for momentum change (we have δp not dp) we need to average it over a few time steps. What we do is, we keep averaging after every 10 time steps but over the whole momentum change (including those in the previous time steps). Suppose our time step is 0.1. So we first calculate pressure at 1.0. Next time, we calculate pressure at 2.0. But the pressure at 2.0 includes the full contribution(from 0.0 to 2.0). We do get stabilization of the pressure after a certain time interval.

As we see in the figure below (Figure 3), the averaged pressure stabilizes after a long enough period of time. Note that this does not happen for box length = 1. This is due to the fact that because of such a small box length, we lose too many particles to the outside. Even if we correct for that, by putting the particle randomly inside the box again, we still get a lot many particles which stay outside (this is because of time not being continuous in our program, as explained earlier) which contribute to the diverging pressure. This phenomenon is relatively insignificant in bigger box lengths, hence the converging pressure.

Finally, we plot the pressure obtained versus area and get the following plot (Figure 4).

INCLUDING LENNARD-JONES POTENTIAL

Till this stage, we have not included any interaction whatsoever between the molecules (except the collision). In the next stage, we evolve the particles under a Lennard-Jones potential and we hope to see a phase transition.

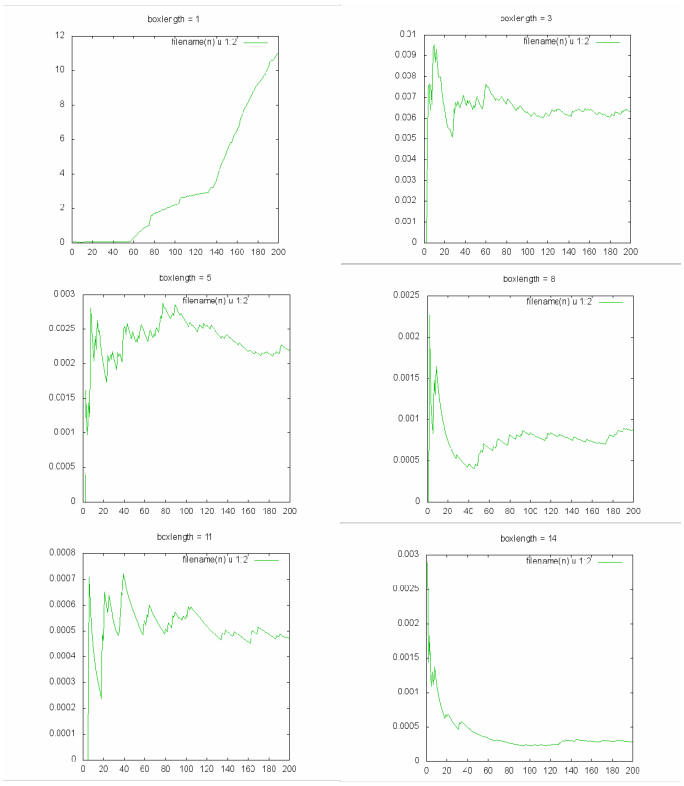

FIG. 3: Pressure vs. Time for various boxlengths

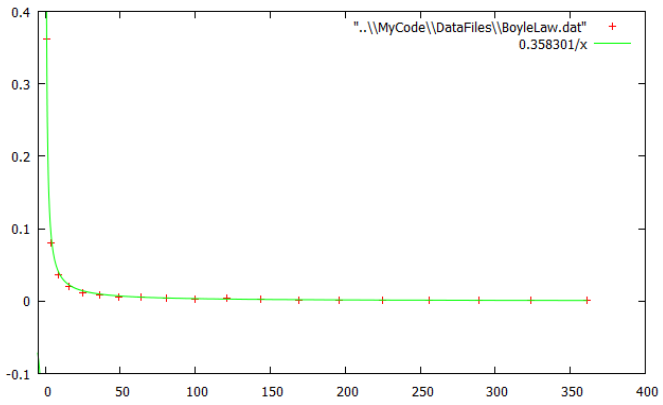

FIG. 4: P vs. V graph. Verified Boyle's Law

The Lennard-Jones potential is given by

$$V(r) = \epsilon[(\frac{r_m}{r})^{12} - 2(\frac{r_m}{r})^6] \tag{2}$$

where ϵ is the depth of the potential well and r_m is the equilibrium distance between two particles. The LJ potential is a mathematically simple model that can quite accurately represent the interaction between a pair of neutral atoms or molecules. The r^{-12} term is the repulsive term which describes Pauli repulsion at short ranges. The r^{-6} term is the attractive long-range term

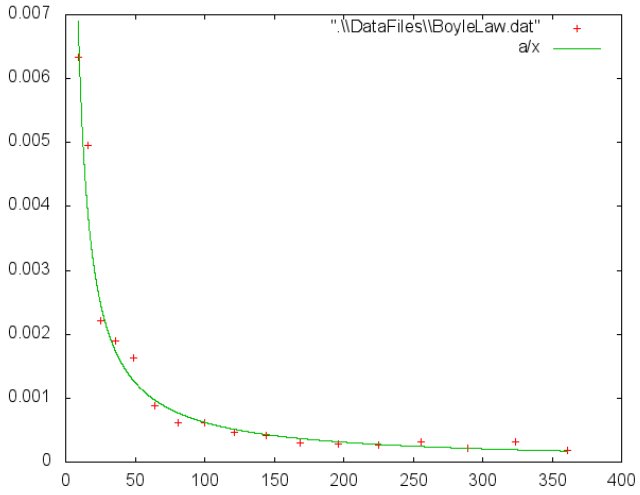

FIG. 5: P vs. V graph with Lennard-Jones Potential

which describes attraction at long ranges.

The force due to LJ potential is given by,

$$F(r) = 12\epsilon[(\frac{r_m^6}{r^7}) - (\frac{r_m^{12}}{r^{13}})] \tag{3}$$

This force is incorporated into the program. We set the parameters of the potential as $\epsilon = 0.0003$ and $r_m = 4radius$. We are actually not using the exact LJ potential, since beyond 2*radius, there is collision. So basically, for r¡2*radius there is the hard wall potential and beyond that it's the LJ potential. We calculate (at each time step) the force components on each particle due to every other particle and update the velocity components accordingly.

We expect to see a phase transition. For that, the temperature of the gas and the energy scale of the potential need to be comparable. If the temperature is too high, we will again get Boyle's law back. The plot obtained was as follows. Even for this case, pressure and area were calculated the same way as before.

Notice that we did get some points off the Boyle's law curve. But we did not see the phase transition we were expecting to see. That may be because the temperature of our gas is so high that the energy scale of the potential well is negligible as compared to it, which again emphasizes the fact that we need to scale our calculations properly.

CONCLUDING REMARKS

I would like to make a few concluding remarks about the project. We can explore several areas for improvement. Some of them are listed below.

- We did not ensure during the random particle creation that two particles should not overlap while being created. A simple code could be written to ensure that does not happen.
- We can very certainly make our code much faster and efficient. For calculating the LJ force we could consider the force due to particles only within a certain range of the particle itself. That would greatly reduce the computational complexity.
- We were working in a 2-D space. We did start some work on 3-D Ideal gas but we could not extend it. We would like to extend all these results to their 3-D counterparts.
- As we noticed, we got into a lot of trouble because we didn't define our units well. We did not run into any inconsistency because we kept our units constant over the whole program but our lack of defining units cost us sometimes. We need to properly maintain all the units of every variable in the program.

We were successful in predicting the Maxwell-Boltzmann distribution and the Boyle's Law from our simulations. We were close to seeing the phase transition in Lennard-Jones potential but ultimately could not see it.
The final code has been attached.

ACKNOWLEDGEMENTS

I would like to thank Dr. Ananda Dasgupta for his brilliant guidance and insight throughout the project duration. I would also thank my colleague, Sharon Felix, for collaborating with me on this project.

```
from math import *
import random
def arctan(m):
	res=atan(m)
	if(res<0):
		return pi+res
	return res

N=300
boxlength=1.0
while(boxlength<20.):
		radius=0.0001
		x=[random.uniform(-boxlength+radius,boxlength-radius) for i in range(N)]
		y=[random.uniform(-boxlength+radius,boxlength-radius) for i in range(N)]
		v=[random.uniform(0,1)*0.003 for i in range(N)]
		theta=[random.uniform(0,1)*2.0*pi for i in range(N)]

		#For calculating Energy
		sum=0.0
		for i in range(N):
			sum=sum+v[i]**2./2.
		print(sum)
		#End calculating Energy

		vx=[0 for i in range(N)]
		vy=[0 for i in range(N)]
		#outfile=open("..\\MyCode\\DataFiles\\RandomParticleDistribution.datDistribution.dat",'w')
		#outfile=open("./DataFiles/RandomParticleDistribution.dat",'w')
		for i in range(N):
			vx[i]=v[i]*cos(theta[i])
			vy[i]=v[i]*sin(theta[i])
		    #print >>outfile,x[i],y[i]

		t=0.0
		tf=200.0
		dt=0.1

		steps=int(round(tf/dt,0))

		epsbox=boxlength/30.0#softening parameter
		epsrad=radius/10.0

		Fxs=0.0
		Fys=0.0
		kp=1
		outfile1=open("./DataFiles/Force"+str(round(boxlength,0))+".dat",'w')
		#outfile1=open(".\\DataFiles\\Force"+str(round(boxlength,0))+".dat",'w')
		epsilon=0.0003
		rm=4.*radius


		while(t<tf):
			s="./DataFiles/T"+str(round(t,0))+".dat"
```

```
        #s=".\\DataFiles\\T"+str(round(t,0))+".dat"
        f=open(s,'w')

        #----For calculating Maxwell Boltmann distribution
        sp=".\\DataFiles\\tp"+str(round(t,0))+".dat"
f=open(s,'w')
    fp=open(sp,'w')
k=0
    dk=0.01
while(k<2.0*max(v)):
        counter=0
        for kp in range(N):
        if(k<hypot(vx[kp],vy[kp])<k+dk):
                counter=counter+1
                        fp.write(str(k+dk)+"\t"+str(counter)+"\n")
        k=k+dk
fp.close()
        #----End calculating Maxwell Boltzmann distribution


        for i in range(N):
                if (abs(y[i])>1.3*boxlength or abs(x[i])>1.3*boxlength):
                        x[i]=random.uniform(-boxlength+radius,boxlength-radius)
                        y[i]=random.uniform(-boxlength+radius,boxlength-radius)
                f.write(str(x[i])+"\t"+str(y[i])+"\t"+str(radius)+"\n")


                if (y[i]>=(boxlength-radius) or y[i]<=-boxlength+radius):
                        #print(abs(2.*vy[i]))
                        #print(Fys)
                        Fys=Fys+abs(2.*vy[i])
                        vy[i]=-vy[i]
                elif(x[i]>=(boxlength-radius) or x[i]<=-boxlength+radius):
                        Fxs=Fxs+abs(2.*vx[i])
                        vx[i]=-vx[i]
                else:
                        Fxtemp=0.0
                        Fytemp=0.0
                        for j in range(N):
                                if(j!=i):
                                        if(hypot(x[i]-x[j],y[i]-y[j])<=2.0*radius):
                                                phi=arctan((y[i]-y[j])/(x[i]-x[j]))
                                                vxp1=cos(phi)*vx[i]-sin(phi)*vy[i]
                                                vyp1=sin(phi)*vx[i]+cos(phi)*vy[i]
                                                vxp2=cos(phi)*vx[j]-sin(phi)*vy[j]
                                                vyp2=sin(phi)*vx[j]+cos(phi)*vy[j]
                                                vxp1,vxp2=vxp2,vxp1
                                                vx[i]=cos(phi)*vxp1+sin(phi)*vyp1
                                                vy[i]=-sin(phi)*vxp1+cos(phi)*vyp1
                                                vx[j]=cos(phi)*vxp2+sin(phi)*vyp2
                                                vy[j]=-sin(phi)*vxp2+cos(phi)*vyp2
                                                x[i]=x[i]+vx[i]*dt#x=x+vt
                                                y[i]=y[i]+vy[i]*dt
```

```
                                        x[j]=x[j]+vx[j]*dt#x=x+vt
                                        y[j]=y[j]+vy[j]*dt
                                        x[i]=x[i]+vx[i]*dt#x=x+vt
                                        y[i]=y[i]+vy[i]*dt
                                        x[j]=x[j]+vx[j]*dt#x=x+vt
                                        y[j]=y[j]+vy[j]*dt
                                        break
                                else:
                                        r=hypot(x[i]-x[j],y[i]-y[j])
                                        phi=arctan((y[i]-y[j])/(x[i]-x[j]))#*180./pi
                                        if (x[i]>x[j] and phi<=pi/2.):
                                                phi=phi+pi
                                        elif (x[i]<x[j] and phi>pi/2.):
                                                phi=phi+pi

                                        Fxtemp=Fxtemp-(-12.*epsilon*(rm**6./r**7.-rm**12./r**13.))*cos(phi)
                                        Fytemp=Fytemp-(-12.*epsilon*(rm**6./r**7.-rm**12./r**13.))*sin(phi)

                        vx[i]=vx[i]+Fxtemp*dt
                        vy[i]=vy[i]+Fytemp*dt
                x[i]=x[i]+vx[i]*dt#x=x+vt
                y[i]=y[i]+vy[i]*dt
        if(kp>1 and kp%10==0):
                outfile1.write(str(t)+"\t"+str((Fxs+Fys)/boxlength/kp/dt)+"\n")
                print(str(kp)+"\t"+str(boxlength))
        f.close()
        kp=kp+1
        t=t+dt

outfile1.close()
boxlength=boxlength+1.
```