IDEAL GAS LAW IN PYTHON

February 26, 2018

Joey Latta, Joe Gibson
Python-Based Physics

Introduction

The goal of this project is to recreate the physical derivation of the Ideal gas law $PV = nrT$. To do this we have to simulate a closed environment that has a number of particles bouncing off the walls and each other. As an ideal gas, it can be assumed that there are no intermolecular attractions between the molecules and therefor the potential energy is zero. Therefore the system can be modeled by simply using the kinetic energy of each particle and simulate multibody interactions as inelastic collisions. We can then calculate the force from the balls on the wall in order to find the pressure. The volume is set from the dimensions of the volume we simulate. n and r are constants in this case. Temperature can be calculated by looking at the average kinetic energy of the balls. So all we need to do is show $P = \frac{neT}{V}$ over some period of time.

Methods

In order to code this simulation we have to create moving balls and an environment. This can be done by creating a box object, or in python a class. This box is an empty volume with the properties size, state of balls and some additional constants that will aid in the calculation of the ideal gas law. The balls are a property of the box represented by a matrix. This matrix contains information about the balls position velocity and mass.

Now we have the state of the balls encoded we need a function to move forward one time step. This function is made inside the box object so when box.timestep() is called it alters the state of the ball matrix. To move forward we need to increment the position of the balls by $v * dt$, we can also change the velocity of the balls but since we are in a closed system no energy is being added to the system so there can't be any acceleration except for collisions. Collisions with the wall must also be accounted for. We set the balls to all have a set radius so we can check if the position of the ball plus the radius is past the wall then give the ball an inelastic collision with the wall, at this time we calculate a force from the ball and add it to the pressure calculation for this step. Next we have to calculate collisions between balls. This is done in a similar manner as we have the positions of all the balls and can use a set of linear equation functions to get the distance in-between all the balls which gives us a list of distances. Then, we check if any of these distances are less than $2 * (radius)$ which would be a collision. Then we calculate an inelastic collision between the respective balls and change their velocities accordingly. These processes are calculated for each step in time.

We also need to calculate the temperature of the assembly. To do this we use $\frac{1}{2} * m * v^2$

to get the kinetic energy of the each ball, take the average of this and multiply this by a constant of $\frac{2}{3}$ because we have rotationally symmetric particles. To obtain a calculated pressure, we take this temperature value and multiply it by the number of particles we have, the Boltzman gas constant then divide by the volume of our system. To prove the Ideal Gas Law within Python, we have to do is compare the calculated value to the value we get by hitting ($animating$) balls against a wall.

Results

The simple system of one ball in a box works but doesn't give us an equivalence of the ideal gas law. This provides the ability to see the ball bouncing off the walls and if we give that ball a friend they can knock each other around as we would expect.

Now that we've shown our simulation works with small numbers of balls we can scale up to n balls for simulation which unfortunately makes our computers cry for mercy. Using 1000 balls our simulation get fairly poor results as "hitting the wall" pressure seems to oscillate around the "calculated" pressure but with some significant deviations. Since 1000 $<<<<<<<<$ mol we didn't expect the simulation to be precise or accurate, but it was decently close a majority of times.

Figure 1 shows the calculated pressure ($orange$) versus the measured pressure ($blue$). It can be seen that while some measured pressure data is consistent with the value calculated by the Ideal Gas Law, there is significant error in the measured data. Unfortunately we have been unable to optimize this code to be able to run effectively on our laptop computers so the simulation stops running effectively at about 5000 particles. It is expected that the deviations from calculated pressure would decrease as we simulate larger numbers of particles.

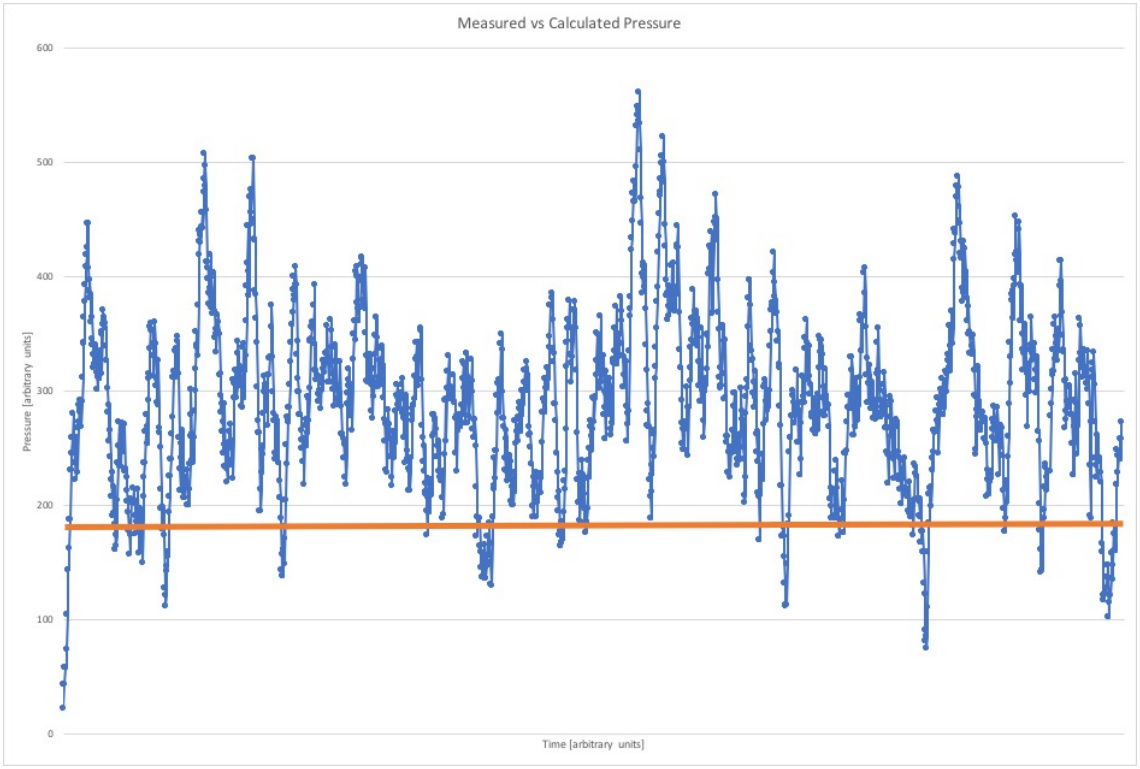

Figure 1

Listing 1: Python code

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import pdist, squareform
import scipy.integrate as integrate
import matplotlib.animation as animation

#constants
radiusH=0.1#120*10e-12
massH=1#1.6737*10e-27
box_length=100*radiusH
gasconstant=8.3144590
BoltzmanConst=1.38065e-23
#this one is actually important, how many particles we put in the box
numparticles=500
##############################################################################
```

```
# Pv=nrT v=box_length**3  [Volume] n =number of particles [mol] \
# r=8.3144598[J/(K mol)]
# T=2/3*E/Kb  E= mean kinetic energy[J] kb =1.38065*10e-23 [J/K]
#p=1/3*N/V*mv**2
################################################################################
#### area and physics ###
#### x,y,z,vx,vy,vz,m ###
class physicsvolume:
    def __init__(self,
                 cornors=[-box_length/2,box_length/2,-box_length/2,box_length/2,\
                 -box_length/2,box_length/2],
                 init_state=[[0,0,0,0,0,0,1],[0,0,0,0,0,0,0,1]],
                 radius=radiusH):

        #things internal to the box
        self.init_state=np.asarray(init_state,dtype=float)
        self.state=self.init_state.copy()
        self.radius=radiusH
        self.time_elapsed=0
        self.cornors=cornors
        self.ForceAtTime=0
        self.kineticEnergy=0
        self.Temperature=0
        self.Volume=box_length**3
        self.Area=(box_length**2)*6
        self.measuredpressure=0
        self.pressure=0
        #self.magVelocityinit
        #self.magVelocityfinal

    def step(self, dt):
        #move time
        self.ForceAtTime=0
        self.time_elapsed +=dt
```

```
        #move particles
    self.state[:,:3]+=dt*self.state[:,3:6]

    #calculates the total kineticEnergy of the system
    self.kineticEnergy=np.sum(0.5*self.state[:,6]*(self.state[:,3]**2+\
    self.state[:,4]**2+self.state[:,5]**2))
    #calculates the temperature of the particles
    self.Temperature=self.kineticEnergy*2.0/3.0
    #calculates the theoretical pressure of the system
    self.pressure=self.Temperature*gasconstant*numparticles/self.Volume
    #self.pressure=2/3*numparticles/self.Volume*self.kineticEnergy


#Simple Collision Detection thing
        #finds distance between particles
            #produces an NxN array of distances between particles
    D=squareform(pdist(self.state[:,:3]))
        #checks if distance is less than the radius
    #for i in range(50):
    #    for j in range(50):
    #        if D[i,j]<2*self.radius+5 and i!=j:
    #            print("hey",str(i),str(j),str(D[i,j]))
    #            print(2*self.radius)

    #this creates an array of bool values that just say if this particle is
    # hitting another particle or not
    ind1,ind2 = np.where(D<2*self.radius)
    unique = (ind1<ind2)
    ind1 = ind1[unique]
    ind2 = ind2[unique]

        #calculate collisions
    for i1,i2 in zip(ind1,ind2):
        m1=massH
        m2=massH
```

```
        # positions
        r1=self.state[i1,:3]
        r2=self.state[i2,:3]
        # velocities
        v1=self.state[i1,3:6]
        v2=self.state[i2,3:6]
        #relative location and velocities
        r_rel=r1-r2
        v_rel=v1-v2
        #momentum of com
        v_cm=(m1*v1+m2*v2)/(m1+m2)
        #collisions of spheres
        rr_rel = np.dot(r_rel,r_rel)
        vr_rel = np.dot(v_rel,r_rel)
        v_rel=2*r_rel*vr_rel / rr_rel - v_rel

        #new velocities
        self.state[i1,3:6]=(v_cm+v_rel*m2/(m1+m2))
        self.state[i2,3:6]=(v_cm-v_rel*m1/(m1+m2))
        #print((v_cm+v_rel*m2/(m1+m2)))

    #this makes sure the balls don't clip through the wall
    crossed_x1 = (self.state[:,0] < self.cornors[0] + self.radius)
    crossed_x2 = (self.state[:,0] > self.cornors[1] - self.radius)
    crossed_y1 = (self.state[:,1] < self.cornors[2] + self.radius)
    crossed_y2 = (self.state[:,1] > self.cornors[3] - self.radius)
    crossed_z1 = (self.state[:,2] < self.cornors[4] + self.radius)
    crossed_z2 = (self.state[:,2] > self.cornors[5] - self.radius)

    #print(crossed_x1)
    #### THIS ADDS A FORCE ON THE WALL ####
    # force from a collition F=m*dv/dt          dv=2*v dt=step length
    if len(self.state[crossed_x1 | crossed_x2])==1:
        #print(abs(self.state[crossed_x1|crossed_x2,3]))
        self.ForceAtTime=2*float(abs(self.state[crossed_x1|crossed_x2,3]))*\
```

```
            abs(self.state[crossed_x1|crossed_x2,6])/dt+self.ForceAtTime

        if len(self.state[crossed_y1 | crossed_y2])==1:
            #print(abs(self.state[crossed_x1|crossed_x2,3]))
            self.ForceAtTime=2*float(abs(self.state[crossed_y1|crossed_y2,4]))*\
            abs(self.state[crossed_y1|crossed_y2,6])/dt+self.ForceAtTime
        if len(self.state[crossed_z1 | crossed_z2])==1:
            #print(abs(self.state[crossed_x1|crossed_x2,3]))
            self.ForceAtTime=2*float(abs(self.state[crossed_z1|crossed_z2,5]))*\
            abs(self.state[crossed_z1|crossed_z2,6])/dt+self.ForceAtTime

        #this means something is really broken
        if self.ForceAtTime<0:
            print("ahh")
        #calculates pressure from the FORCE
        self.measuredpressure=self.ForceAtTime*numparticles/self.Area

        self.state[crossed_x1,0]=self.cornors[0] + self.radius
        self.state[crossed_x2,0]=self.cornors[1] - self.radius

        self.state[crossed_y1,1]=self.cornors[2]+self.radius
        self.state[crossed_y2,1]=self.cornors[3]-self.radius

        self.state[crossed_z1,2]=self.cornors[4]+self.radius
        self.state[crossed_z2,2]=self.cornors[5]-self.radius

        #makes the balls "bounce" back the way they came from
        self.state[crossed_x1 | crossed_x2,3]*=-1
        self.state[crossed_y1 | crossed_y2,4]*=-1
        self.state[crossed_z1 | crossed_z2,5]*=-1

################################################################################

#this creates an initial state of numparticle number of balls with an initial
#position velocity and mass
```

```
np.random.seed(0)
red=-0.5+np.random.random((numparticles,7))
#makes sure they are in the box
red[:,:3]*=box_length-0.1
#sets all the masses equal
red[:,6]=massH
#print("hey")
#print(0.5*red[:,6]*(red[:,3]**2+red[:,4]**2+red[:,5]**2))
#red=[[0,0,0,0,0,0,0],[10,10,0,-1,-1,0,0]]
#says let there be a box
box = physicsvolume(init_state=red,radius=1)
dt=1./30


################################################################################
#plots for animation
fig = plt.figure()
fig.subplots_adjust(left=0,right=1,bottom=0,top=1)
ax = fig.add_subplot(111,aspect='equal',autoscale_on=False,
                     xlim=(-box_length/2-5,box_length/2+5),
                     ylim=(-box_length/2-5,box_length/2+5))

#plots for the particles
particles, = ax.plot([],[],'bo',ms=6)

#text displayed showing the calculated and measured pressure
pressure_template = 'pressure␣measured␣=␣%.00001f'
law_template = 'pressure␣␣calculated␣=␣%.00001f'
pressure_text = ax.text(0.05, 0.95, '', transform=ax.transAxes,size=28)
law_text = ax.text(0.05, 0.05, '', transform=ax.transAxes,size=28)

#this line ill explain later
pressurearr=np.zeros(20)

rect = plt.Rectangle(box.cornors[::2],box.cornors[1] - box.cornors[0],
```

```
                                        box.cornors[3] - box.cornors[2], \
                                        ec='none',lw=2,fc='none')
ax.add_patch(rect)


########################## animation stuff

def init():
    global box, rect
    particles.set_data([],[])
    rect.set_edgecolor('none')

    pressure_text.set_text('')
    law_text.set_text('')
    return particles, rect, pressure_text, law_text

def animate(i):
    global box, rect, dt, ax, fig, avgpress
    box.step(dt)
    ms = int(fig.dpi*box.radius*fig.get_figwidth()/np.diff(ax.get_xbound())[0])

    rect.set_edgecolor('k')
    particles.set_data(box.state[:,0],box.state[:,1])
    particles.set_markersize(ms)

    #takes ten frames of hits and calculates a pressure from that
    #if i%10==0:
    pressurearr[i%20]=box.measuredpressure
    avgpress=np.sum(pressurearr)
    #avgpress=box.measuredpressure

    calcpress=box.pressure

    pressure_text.set_text(pressure_template % (avgpress))
    law_text.set_text(law_template % (calcpress))
```

```
    return particles, rect, pressure_text, law_text

ani=animation.FuncAnimation(fig,animate,frames=600,interval=1,blit=True,\
init_func=init)
ani.save('particle_box.mp4', fps=30, extra_args=['-vcodec', 'libx264'])
plt.show()
```

References

Kinder, Jesse M., and Philip Charles Nelson. A Student's Guide to Python for Physical Modeling. Princeton University Press, 2018.

Vanderplas, Jake. "Pythonic Perambulations". http://jakevdp.github.com.