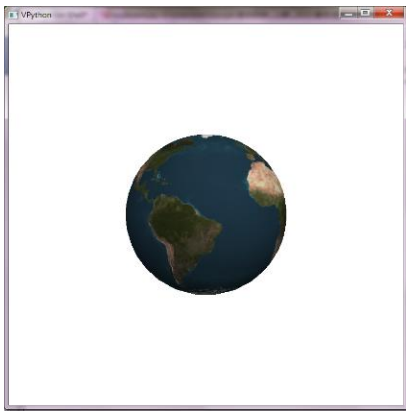
VP9 Adiabatic Compression and Free Expansion [Vpython: Interrupt Callback]

I. Callback and keyboard control:

In some programs, e.g. games, you use mice or keyboards to control things but at the same time, the programs continue running. An interrupt callback function can help you do this kind of work. The following example shows how to use arrow keys and letter keys ('i', 'o', 'r', 'c') to control the earth's position and rotation axis while the earth continues spinning. However, in **Jupyter** this will **not work** since Jupyter console has a higher priority accessing to the keyboard than your vpython program. As a result, your program will not know if a keyboard is pressed or not. You can use the method marked as ##(for JUPYTER also for VIDLE).

##(NOT FOR JUPYTER)



```
from vpython import *
pos, angle = vector(0, 0, 0), 0

def keyinput(evt):                  #keyboard callback function
    global pos, angle
    move = {'left': vector(-0.1, 0, 0), 'right': vector(0.1, 0, 0),
            'up': vector(0, 0.1, 0),
          'down': vector(0, -0.1, 0), 'i' : vector(0, 0, -0.1),
            'o': vector(0, 0, 0.1)}
    roa = {'c' : pi / 90.0 , 'r': - pi / 90.0}
    s = evt.key
    if s in move : pos = pos + move[s]
    if s in roa:
        ball.rotate(angle = roa[s], axis = vector(0, 0, 1), origin= ball.pos)
        angle = angle - roa[s]

scene = canvas(width=800, height=800, range = 5, background=color.white)
ball = sphere(radius = 2.0, texture=textures.earth )
scene.bind('keydown', keyinput)                 # setting for the binding function

while True:
    rate(1000)
    ball.rotate(angle=pi/600, axis= vector(sin(angle),cos(angle),0),
                origin=pos)
    ball.pos = pos
```

In def keyinput(evt), the interrupt callback function, 'global' declares the variables (pos, angle) to be global variables, meaning their values will be known to the entire program (exceeding the scope of this function). s=evt.key gets the key that is pressed-down. Then you decide what to do with s. ball.rotate(angle = roa[s], axis = vector(0, 0, 1), origin= ball.pos) will rotate ball with an angle, around axis that passes through origin. Any vpython drawn object, such as sphere or box, can be rotated by this 'rotate' method.

In the main program, command scene.bind('keydown', keyinput) binds keyinput with scene. When your active window is scene and the event 'keydown' (here, a key of the keyboard is pressed) happens, no matter at which part of the program it is now being executed, the bound function, keyinput(), will interrupt and be executed right away. After the bound function has finished, the program will go back to the program where it has left just before the interrupt event has happened. More information about keyboard operation can be found at http://www.glowscript.org/docs/VPythonDocs/keyboard.html. Similar working principle applies to mouse. You can also find mouse operation at http://www.glowscript.org/docs/VPythonDocs/mouse.html.

##(for Jupyter also for VIDLE)

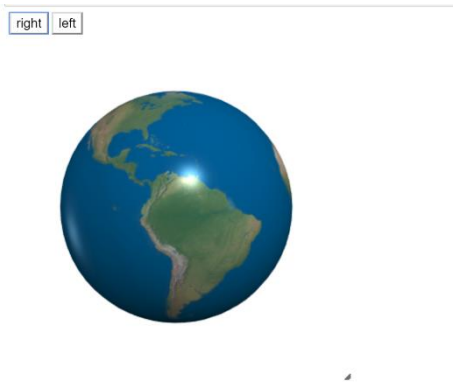
```
from vpython import *
pos, angle = vector(0, 0, 0), 0

def right(b):                   #callback function
    global pos, angle
    pos = pos + vector(0.1, 0, 0)
```

```
def left(b):
    global pos, angle
    pos = pos + vector(-0.1, 0, 0)

scene = canvas(width=400, height=400, range = 5, background=color.white)
ball = sphere(radius = 2.0, texture=textures.earth )
button(text='right', pos=scene.title_anchor, bind = right)
button(text='left', pos=scene.title_anchor, bind = left)

while True:
    rate(1000)
    ball.rotate(angle=pi/600, axis= vector(sin(angle),cos(angle),0), origin=pos)
    ball.pos = pos
```



In def right(b), the interrupt callback function, 'global' declares the variables pos and angle to be global variables, meaning their values will be known to the entire program (exceeding the scope of this function). When this function is executed, pos is added by vector(0.1, 0, 0). The same is applied similarly to def left(b). You can write similar functions to control the pos or the angle.

In the main program, command button(text='right', pos=scene.title_anchor, bind = right) creates a button with text='right' with the button's position set by pos=scene.title_anchor. The most importantly, this button is bound to function right by bind = right, meaning that if this button is clicked, the bound function will be executed no matter at which part of the program it is now being executed, the bound function, right(), will interrupt and be executed right away. After the bound function has finished, the program will go back to the program where it has left just before the interrupt event has happened. More information about buttons and other similar features can be found http://www.glowscript.org/docs/VPythonDocs/controls.html. In the main loop ball.rotate(angle=pi/600, axis= vector(sin(angle),cos(angle),0), origin=pos) will rotate ball with an angle, around axis that passes through origin. Any vpython drawn object, such as sphere or box, can be rotated by this 'rotate' method.

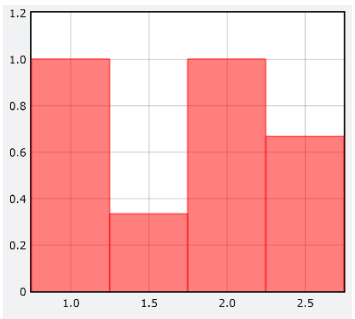
II. Histogram module (Do save this as 'histogram.py' in the same folder of your main program):

```
import numpy as np
import vpython as vp
class ghistogram:
    def __init__(self, graph, bins, color = vp.color.red):
        self.bins = bins
        self.slotnumber = len(bins)
        self.slotwidth = bins[1]-bins[0]
        self.n = 0
        self.slots = np.zeros(len(bins))
        self.bars = vp.gvbars(graph = graph, delta=self.slotwidth, color=color)

    def plot(self, data):
        currentslots = np.zeros(self.slotnumber)
        for value in data:
            currentslots[min(max(int((value - self.bins[0])/self.slotwidth), 0), self.slotnumber-1)] += 1

        self.slots = (self.slots * self.n + currentslots)/(self.n + 1)
        self.n += 1
        if self.n == 1:
            for (currentbin, barlength) in zip(self.bins, self.slots):
                self.bars.plot( pos = (currentbin, barlength))
        else:
            self.bars.data = list(zip(self.bins, self.slots))

if __name__ == '__main__':
    vdist = vp.graph(width = 450)
    observation = ghistogram(graph = vdist, bins = np.arange(1, 3, 0.5))
    observation.plot(data=[1.2, 2.3, 4])
```



```
    observation.plot(data=[1, 1.7, 2.6])
    observation.plot (data=[-0.5, 2, 2.3])
```

This module provides you the ghistogram class to plot the histogram for a series of data. Remember to put this file 'histogram.py' in the same folder of your main program. If you are interested in how this class is written, you can look at the details of the program, otherwise, you just use this module to finish your homework. Unter if __name__ == '__main__': shows how to use it. You first import vpython and create a graph (here it is vdist). Then you can create an ghistogram object (here it is observation), in which it must specified in which graph it should display (graph = vdist), the bins (here it is 1-1.5, 1.5-2, 2-2.5, 2.5-3), and the color of the histogram. If you do not specify what color the histogram should be, it defaults to red. Then in the method observation.plot, you give the data (can be list or array), it then counts the number of occurrence of the values in data, averages the number of occurrence over times this method has been called and plots the latest results. If a value in data is smaller than the lower bound, it is counted as in the lowest bin, similarly, if a value is larger than the upper bound, it is counted as in the highest bin. For example, the 3 rows of data will yield the histogram in the figure.

III. Template file for gas in a container (VP9.py):

```
from vpython import *
import numpy as np
from histogram import *

N = 200
m, size = 4E-3/6E23, 31E-12*10                  # He atom are 10 times bigger for easiear collision but not too big for accuracy
L = ((24.4E-3/(6E23))*N)**(1/3.0)/2 + size      # 2L is the cubic container's original length, width, and height
k, T = 1.38E-23, 298.0                          # Boltzmann Constant and initial temperature
t, dt = 0, 3E-13
vrms = (2*k*1.5*T/m)**0.5                       # the initial root mean square velocity
stage = 0                                       # stage number
atoms = []                                      # list to store atoms

# histogram setting
deltav = 50.                                    # slotwidth for v histogram
vdist = graph(x=800, y=0, ymax = N*deltav/1000.,width=500, height=300, xtitle='v', ytitle='dN', align = 'left')
theory_low_T = gcurve(color=color.cyan)         # for plot of the curve for the atom speed distribution
dv = 10.
for v in arange(0.,4201.+dv,dv):                # theoretical speed distribution
    theory_low_T.plot(pos=(v,(deltav/dv)*N*4.*pi*((m/(2.*pi*k*T))**1.5)*exp((-0.5*m*v**2)/(k*T))*(v**2)*dv))
observation = ghistogram(graph = vdist, bins=arange(0.,4200.,deltav), color=color.red)    # for the simulation speed distribution

#initialization
scene = canvas(width=500, height=500, background=vector(0.2,0.2,0), align = 'left')
container = box(length = 2*L, height = 2*L, width = 2*L,    opacity=0.2, color = color.yellow )
p_a, v_a = np.zeros((N,3)), np.zeros((N,3))     # particle position array and particle velocity array, N particles and 3 for x, y, z
for i in range(N):
    p_a[i] = [2 * L*random() - L, 2 * L*random() - L, 2 * L*random() - L]        # particle is initially random positioned in container
    if i== N-1:                                 # the last atom is with yellow color and leaves a trail
        atom = sphere(pos=vector(p_a[i, 0], p_a[i, 1], p_a[i, 2]), radius = size, color=color.yellow, make_trail = True, retain = 50)
    else:                                       # other atoms are with random color and leaves no trail
        atom = sphere(pos=vector(p_a[i, 0], p_a[i, 1], p_a[i, 2]), radius = size, color=vector(random(), random(), random()))
    ra = pi*random()
    rb = 2*pi*random()
    v_a[i] = [vrms*sin(ra)*cos(rb), vrms*sin(ra)*sin(rb), vrms*cos(ra)]         # particle initially same speed but random direction
    atoms.append(atom)

def vcollision(a1p, a2p, a1v,a2v):              # the function for handling velocity after collisions between two atoms
    v1prime = a1v - (a1p - a2p) * sum((a1v-a2v)*(a1p-a2p)) / sum((a1p-a2p)**2)
    v2prime = a2v - (a2p - a1p) * sum((a2v-a1v)*(a2p-a1p)) / sum((a2p-a1p)**2)
    return v1prime, v2prime

while True:
```

```
    t += dt
    rate(10000)

    p_a += v_a*dt                                   # calculate new positions for all atoms
    for i in range(N): atoms[i].pos = vector(p_a[i, 0], p_a[i, 1], p_a[i, 2])       # to display atoms at new positions
    if stage != 1 : observation.plot(data = np.sqrt(np.sum(np.square(v_a),-1)))       ## freeze histogram for stage != 1

### find collisions between pairs of atoms, and handle their collisions
    r_array = p_a-p_a[:,np.newaxis]                 # array for vector from one atom to another atom for all pairs of atoms
    rmag = np.sqrt(np.sum(np.square(r_array),-1))   # distance array between atoms for all pairs of atoms
    hit = np.less_equal(rmag,2*size)-np.identity(N) # if smaller than 2*size meaning these two atoms might hit each other
    hitlist = np.sort(np.nonzero(hit.flat)[0]).tolist()  # change hit to a list
    for ij in hitlist:                              # i,j encoded as i*Natoms+j
        i, j = divmod(ij,N)                         # atom pair, i-th and j-th atoms, hit each other
        hitlist.remove(j*N+i)                       # remove j,i pair from list to avoid handling the collision twice
        if sum((p_a[i]-p_a[j])*(v_a[i]-v_a[j])) < 0 :   # only handling collision if two atoms are approaching each other
            v_a[i], v_a[j] = vcollision(p_a[i], p_a[j], v_a[i], v_a[j])      # handle collision

#find collisions between the atoms and the walls, and handle their elastic collisions
    for i in range(N):
        if abs(p_a[i][0]) >= L - size and p_a[i][0]*v_a[i][0] > 0 :
            v_a[i][0] = - v_a[i][0]
        if abs(p_a[i][1]) >= L - size and p_a[i][1]*v_a[i][1] > 0 :
            v_a[i][1] = - v_a[i][1]
        if abs(p_a[i][2]) >= L - size and p_a[i][2]*v_a[i][2] > 0 :
            v_a[i][2] = - v_a[i][2]
```

Read the template program for simulation of a gas of 200 He atoms in a container. The most difficult part is the section commented by ### find collisions between pairs of atoms, and handle their collisions. If you do not understand this part, it is ok to skip it. The entire program is written very straight forward: setting the suitable parameters, preparing for the plotting for histogram, initialization of the atoms, and the main program that includes the movements of the atoms, handling the atom-atom collisions, and the handling the atom-wall collisions. In the program, we use numpy's array to expedite the execution. When you run this program, you will see a container and within it 200 He atoms running and colliding, with one of them traveling with a trail. In the histogram, you will see the theoretical speed distribution curve (i.e. Maxwell-Boltzmann distribution) whose equation is copied from the textbook. After running the program for some time, you will see that the speed distribution histogram looks just like the theoretical curve. Remember that, the program starts with all the atoms to have the same initial speed. However, only after certain time span and certain number of collisions to allow the atoms to exchange energy, the speed distribution becomes the Maxwell-Boltzmann distribution, without any special treatment.

IV. Homework must:

Now you need to complete the simulation for the adiabatic compression and free expansion with the above template file. Read carefully the template program (without ### part is ok). You need to understand the logic behind the program in order to complete this simulation. In this program, you also need to implement a section of code for an interrupt callback function with 'n' key to advance in stages if you are using keyboard control or with a button 'n' to advance in stages if you are using button control.

In each stage, do the following:

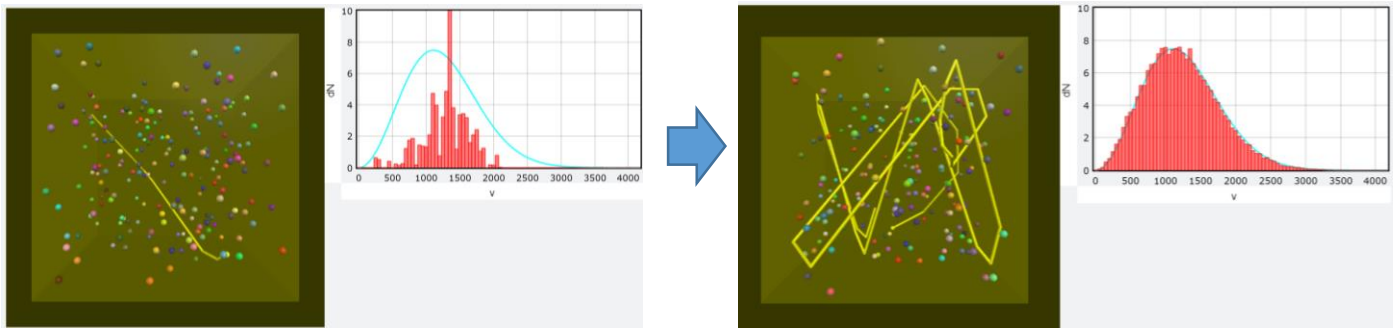
For every 1000*dt, in addition to the main animation that shows the gas of atoms in the container and the graph that shows the histogram of the speed distribution, print the following

(a) T (temperature), calculated by T = Total Translational Kinetic Energy / (3*N*k/2), N is the number of molecules and k the Boltzmann constant;

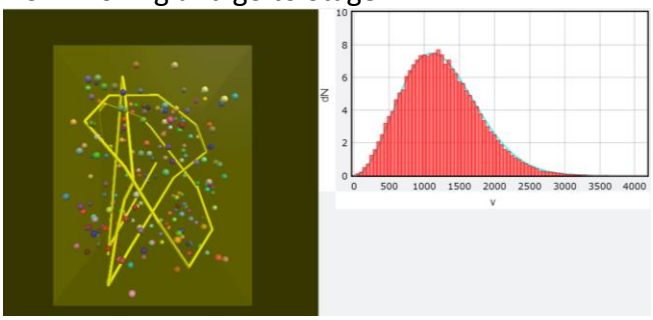
(b) p (pressure), which is calculated by the total momentum impacted on the 6 walls within the 1000*dt time

span, divided by 1000*dt, divided by total wall area. Note: $p = \frac{Force}{area} = (\Delta P/\Delta t)/area$;
(c) V (volume), which is equal to length * height * width of the container at any moment;
(d) p*V;
(e) N*k*T; You will see at any moment p*V is very close to N*k*T;
(f) p*(V**gamma), the product of pressure and the volume to the power of gamma. Since the motion is three-dimensional, therefore the degree of freedom is 3 (without rotation and vibration for monatomic molecule) the ratio of the constant-pressure heat capacity to the constant-volume heat capacity is **gamma = (1+3/2) / (3/2) = 5/3**.

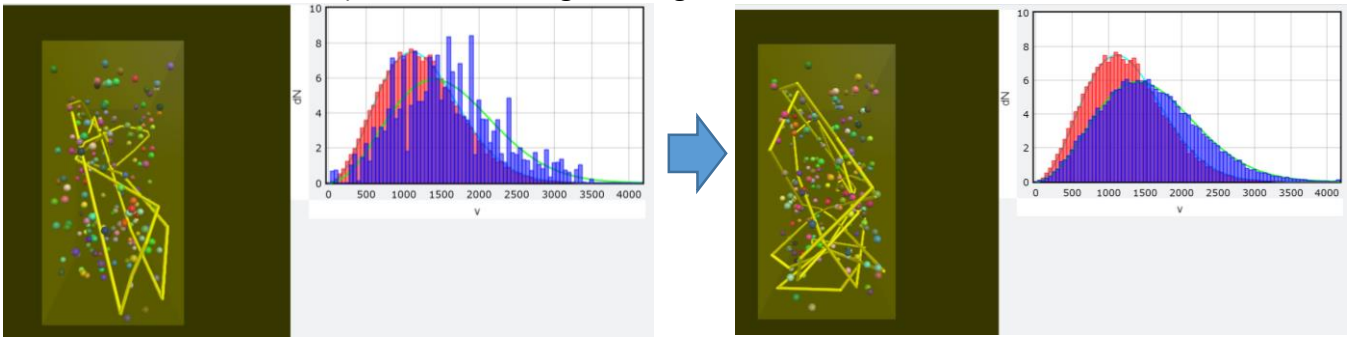
In Stage 0 (initial stage): You wait the system to reach the equilibrium, which is indicated by the speed distribution histogram very close to the theoretical curve. Then you hit the 'n', freeze the current histogram plot (this is already done by ##) and go to Stage 1.



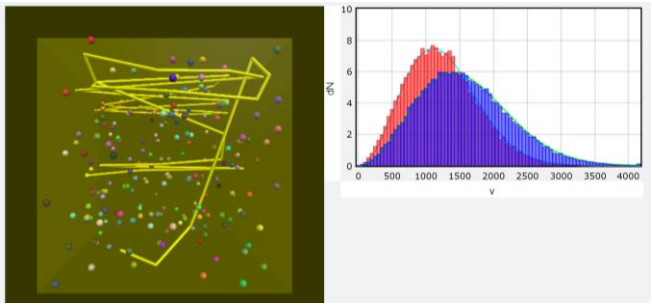
In Stage 1 (adiabatic compression): Make the left and right walls (in x-axis) move towards the center with a speed v_W = L / (20000.0*dt). This has two effects: First, the available space for the atoms to move is smaller. Second, since the walls have mass much larger than that of the atoms, therefore when the atoms hit **elastically** on the moving walls, they are bounced back with a velocity that the walls' velocity needs to be considered (you need to implement this in the program). Until the total length of the container becomes 1/2 of that of the original container or until you hit 'n' again (depending on which comes first), stop the walls from moving and go to Stage 2.



In Stage 2: Start a new histogram for the speed of the atoms and add a new theoretical curve according to the current temperature. Wait until the histogram reaches the new equilibrium (i.e. the histogram is very close the theoretical value), hit the 'n' and go to Stage 3.



In Stage 3 (free expansion): The walls suddenly go to their original positions as in Stage 0, mimicking a free expansion, in which the gas expands in an infinitesimal time without any resistance.



Run your program, observe carefully about the physics and find what happens to T, p, V, p*V, N*k*T and p*(V**gamma). Are these value matching to the theoretically prediction by your text book? Are the speed distribution matching the Maxwell-Boltzmann Distribution for the same temperature? Think about the physics why you get these results. Do Notice that, in this simulation, nothing but exchange of energy of elastic collision causes these results.