Making animations with Python

The following script shows you how to animate a line with **`FuncAnimation`**. The line is a **`np.sin()`** function updated inside **`update_line()`** every time the function is called by **`FuncAnimation`**. The update consists in a horizontal shift set by variable **`num`**.

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

number_of_frames = 100
data = np.arange(0, 2*np.pi, 0.01)

fig = plt.figure()

def update_line(num):
    plt.cla()
    plt.plot(np.sin(data + num/10.0))

animation = animation.FuncAnimation(fig, update_line, num-
ber_of_frames,interval=1,repeat=False)
```

The problem with the approach above is that the function **`update_line()`** generates a brand new plot (axis, frame, lines, etc…) at every function call. This could result in slowing down your animation. One way around this is to update only the elements of a plot which need to be updated. The example below shows the same line animation where only the line us updated, while everything else stays the same. In order to do this we have to start making changes in Python *classes* for plotting operations, i.e. something we haven't discussed yet. But the example below should be clear enough: every Python object contains sub objects which can be altered individually. The example below shows how to alter the line in a plot using an instance of a plot:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

number_of_frames = 100
data = np.arange(0, 2*np.pi, 0.01)

fig, ax = plt.subplots()
line, = ax.plot(data, np.sin(data))

def update_line(num):
    line.set_ydata(np.sin(data + num/10.0))  # update the data
```

```
    return line,

animation = animation.FuncAnimation(fig, update_line, num-
ber_of_frames,interval=1,repeat=False)
```

It should be noted that the function **update_line()** now only returns the modified line, this is only for your own clarity because the whole script would work equally well without the **return** statement.

The following example shows how you can use **FuncAnimation** to generate animations which involve calculations in a separate function for each iteration. Similar to the line example above, we will use, and update an instance of **scatter()**, i.e. a scatter plot. The initial plot is a dot located at **xpos, ypos**, and its position is updated using the method **set_offsets** of the scatter plot instance **im**. The function **newypos()** updates the position of the dot, shifting vertically by an amount **dypos**:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

number_of_frames = 49
fig, ax = plt.subplots()

xpos,ypos=0.5,0.
dypos=1./50.

plt.xlim(0,1)
plt.ylim(0,1)
im = ax.scatter(xpos, ypos)

def newypos(i):
    global ypos
    ypos=ypos+dypos

def update_point(num):
    newypos(num)
    im.set_offsets((xpos,ypos))
    return im,

animation = animation.FuncAnimation(fig, update_point, num-
ber_of_frames,interval=1,repeat=False)
```

The following script shows how to animate a scatter plot. We have seen an example last week with a single point moving up across the plot window, but this time we have many

points. It uses some additional functionalities (numpy **vstack**) and methods of a scatter plot instance (e.g. **set_sizes()** and **set_facecolor()**).

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

npoint=500
nframe=100
fig, ax = plt.subplots()
plt.ylim(-0.2,1.2)
plt.xlim(-0.2,1.2)
x = np.random.random(npoint)
y = np.random.random(npoint)
s = 500 * np.random.random(npoint)
c = np.random.random(npoint * 3).reshape(npoint,3)

im = ax.scatter(x,y)
im.set_sizes(s)
im.set_facecolor(c)
dstep=400.

def update_point(num):
    newx=x+np.random.randn(npoint)/dstep
    newy=y+np.random.randn(npoint)/dstep
    data=np.stack((newx,newy),axis=-1)
    im.set_offsets(data)
    return im,

animation = animation.FuncAnimation(fig, update_point,
nframe,interval=1,repeat=False)
```

The itertool package

`itertool` is a powerful python package which can perform all sorts of iterations on arrays elements. The full description is given there:

https://docs.python.org/2/library/itertools.html

but here I will only discuss the method `combinations()` which you will need for project 4. For project 4, you will simulate the collision of a large number of particles in a box. In order to decide if two particles collide, you have to test how close they are, and

you have to do this for all particles in the gas. The method `combinations()` does this for you. Here is an example:

```
In [1]: import itertools

In [2]: import numpy as np

In [3]: x=np.array([1,2,3,4])

In [4]: itertools.combinations(x,2)
Out[4]: <itertools.combinations at 0x10942bc78>

In [5]: list(itertools.combinations(x,2))
Out[5]: [(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)]
```

The output is a list of all possible pairing coming of an input list. This is why `combinations()` is useful (see the program template at the end of project 4 instructions).

Project 4 (Due date Tuesday November 21st 12p.m.)

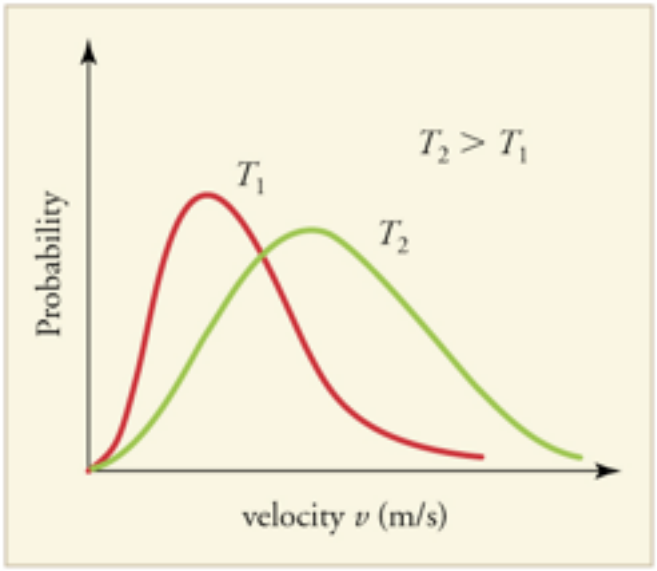
In introductory physics course for science students, kinetic theory of gases is an important subject area which provides the foundations of thermodynamics and statistical physics. A general description is given on the wikipedia page:

https://en.wikipedia.org/wiki/Kinetic_theory_of_gases

This theory explains the macroscopic properties of gases, such as pressure, temperature, entropy, etc… by considering their compositions and particles motion. Of primary importance is the Maxwell-Boltzmann (MB) distribution function which gives the probability distribution function of particle speeds in the gas:

https://en.wikipedia.org/wiki/Maxwell%E2%80%93Boltzmann_distribution

It is usually presented as an empirical formula with little to no discussion on its physical origin. In general students accept the fact that the distribution of speeds in a gas is given by the MB formula, but the whole concept remains an abstraction to many. The distribution of speed is related to the temperature of the gas T, as illustrated in the figure below:



In this project, you will simulate a 2-dimensional gas and recover the MB distribution of velocities. The goal is to develop some deep understanding as to where the MB distribution is really coming from microscopically. Your simulation will start with some arbitrary *very* non-MB velocity distribution and you will observe that, but no matter how the initial velocity conditions are set, the final velocity distribution will **always** be a MB distribution.

The repartition of energy in a gas is done through elastic collisions (collisions can be inelastic, but it is not relevant for the goal of this project). It means that every-time two particles in a gas collide, you will have to calculate the new velocities according to the conservation of momentum.

There is a simple formula which allows you to do this:

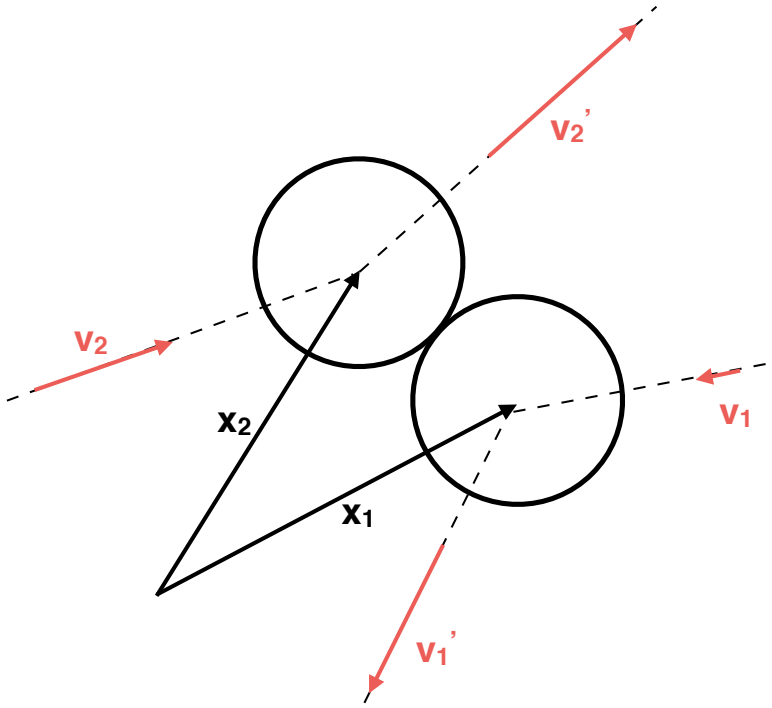
https://en.wikipedia.org/wiki/Elastic_collision#Two-dimensional

If $\mathbf{v_1}, \mathbf{v_2}, \mathbf{x_1}, \mathbf{x_2}$ are respectively the velocities of particle 1 and 2, and the positions of particle 1 and 2, then the new velocities after collision are going to be:

$$\mathbf{v_1'} = \mathbf{v_1} - \frac{\langle \mathbf{v_1} - \mathbf{v_2}, \mathbf{x_1} - \mathbf{x_2} \rangle}{\|\mathbf{x_1} - \mathbf{x_2}\|^2} (\mathbf{x_1} - \mathbf{x_2})$$

$$\mathbf{v_2'} = \mathbf{v_2} - \frac{\langle \mathbf{v_2} - \mathbf{v_1}, \mathbf{x_2} - \mathbf{x_1} \rangle}{\|\mathbf{x_2} - \mathbf{x_1}\|^2} (\mathbf{x_2} - \mathbf{x_1})$$

where the ⟨**A**,**B**⟩ refers to the dot product between **A** and **B**. We have assumed that the mass of the two particles are the same.The collision configuration is given by the following:



Objectives:

You will write a code **`collisions.py`** which shows the collisional motion of 400 particles in a box (see setup instructions below) for 1000 time steps. When the full time steps cycle has completed, your code will generate a plot showing the probability distribution of velocities `f(v)` of all particles and the probability distribution of kinetic energy `g(E)` of all particles (where `E=1/2 mv`^{2}). In two dimensions, the analytical expressions for `f(v)` (called the Maxwell-Boltzmann distribution of speed) and `g(E)` (called the Boltzmann distribution) are at the heart of classical statistical physics, they are given by:

$$f(\mathrm{v}) = \left(\frac{\mathrm{mv}}{k_B T}\right) \exp\left(-\frac{1}{2}\mathrm{mv}^2/k_B T\right)$$

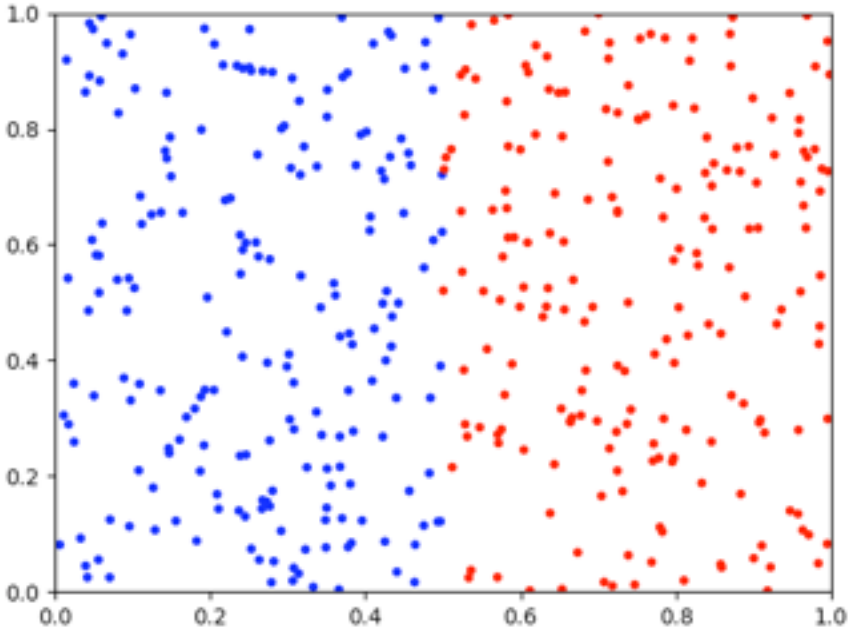
$$g(\mathrm{E}) = \frac{1}{k_B T} \exp\left(-\frac{\mathrm{E}}{k_B T}\right)$$

where k_B =1.38064852 × 10^{-23} m^2 kg s^{-2} K^{-1} is the Boltzmann constant and `T` is the temperature of the gas. Your code should also fit the analytical `f(v)` to the histogram generated by your simulation and it should return a temperature `T`.

Your code should generate the following files:a movie **`collisions.mp4`** showing the collisions over the entire time steps cycle. A plot called **`distributions.pdf`** which contains two panel, the top panel should be the MB distribution, the bottom panel should be the `g(E)` distribution. On the plots, you will also display the fitted `f(v)` and the analytical curve `g(E)` using the temperature you have measured from the MB distribution. In a file called **`collisions.txt`** your code should write the value of the temperature you found.
Note that all your histograms should be normalized to unity for the analytical formula to work.

Practical implementation:

1- You will prepare the gas particle distribution with following setup:



that is the left half of particles are blue, the right half are red. You will assign the initial velocity of the blue particles to `+500 m/s` and the red particles to `-500 m/s` along the x axis. You should start from the program template given at the end of these instructions. The chosen characteristics of the simulations are the following:

```
time step Dt = 0.00002 s
particle size radius = 0.00001 m
particle mass (this is the mass of an O2 molecule) = 2.672x10-26
kg
```

2- You will implement the particle-particle collision physics (velocity change) using the `itertool.combinaison()` method introduced above to identify the pairs of particle that undergo a collision (the instruction is already in the program template, you have to build from there).

Program template:

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from itertools import combinations

npoint=400
nframe=1000
xmin,xmax,ymin,ymax=0,1,0,1
fig, ax = plt.subplots()
plt.xlim(xmin,xmax)
plt.ylim(ymin,ymax)
Dt=0.00002

def update_point(num):
    global x,y,vx,vy

    print(num)
    indx=np.where((x < xmin) | (x > xmax))
    indy=np.where((y < ymin) | (y > ymax))
    vx[indx]=-vx[indx]
    vy[indy]=-vy[indy]
    xx=np.asarray(list(combinations(x,2)))
    yy=np.asarray(list(combinations(y,2)))
    dd=(xx[:,0]-xx[:,1])**2+(yy[:,0]-yy[:,1])**2

    dx=Dt*vx
    dy=Dt*vy
    x=x+dx
    y=y+dy
    data=np.stack((x,y),axis=-1)

    im.set_offsets(data)

x = np.random.random(npoint)
y = np.random.random(npoint)
vx=-500.*np.ones(npoint)
vy=np.zeros(npoint)
```

```
vx[np.where(x <= 0.5)]=-vx[np.where(x <= 0.5)]
s=np.array([10])
im = ax.scatter(x,y)
im.set_sizes(s)

animation = animation.FuncAnimation(fig, update_point,nframe,in-
terval=10,repeat=False)
```