Astro 410 Extra-Credit HomeWork 4

Swapnil Dubey

List of included files:

- hw4-dubey.ipynb
- hw4-dubey.py (Python code not as a Jupyter Notebook)
- hw4-dubey.pdf (contains printed pdf of html file)
- hw4-dubey.html (contains html version of Jupyter Notebook for better readability)

(a lot of the inspiration for the code has been taken from the leap-frog.c file on canvas and various internet portals, although I have done my best to mold it to suit the assignment's needs by disecting it the best I can and annotating for easy understanding)

The following report attempts to approximate the behaviours of N-bodies in orbit around each other.

We begin by calculating acceleration.

In [1]:

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

def getAcc( pos, mass, G, softening ):
    # positions r = [x,y,z] for all particles
        x = pos[:,0:1]
        y = pos[:,1:2]
        z = pos[:,2:3]

        # matrix that stores all pairwise particle separations: r_j - r_i
        dx = x.T - x
        dy = y.T - y
        dz = z.T - z

        # matrix that stores 1/r^3 for all particle pairwise particle separati
ons
        inv_r3 = (dx**2 + dy**2 + dz**2 + softening**2)
        inv_r3[inv_r3>0] = inv_r3[inv_r3>0]**(-1.5)

        ax = G * (dx * inv_r3) @ mass
        ay = G * (dy * inv_r3) @ mass
        az = G * (dz * inv_r3) @ mass

        # pack together the acceleration components
        a = np.hstack((ax,ay,az))

        return a
```

Building a function to calculate KE and PE seperately for each body. Where formula for KE is:

$$\frac{1}{2}\sum_i mv^2$$

And potential energy (PE) is calculated as below:

$$\frac{1}{2}\sum_{1<i<j<N}\frac{Gm_im_j}{|r_j - r_i|}$$

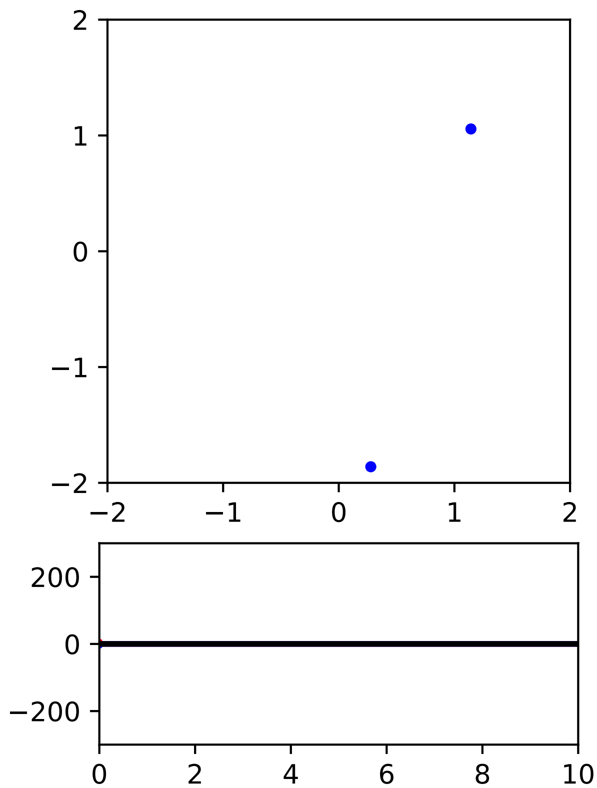
In [3]:
```
def getEnergy( pos, vel, mass, G ):

    # Kinetic Energy:
    # kinetic energy of the system
    KE = 0.5 * np.sum(np.sum( mass * vel**2 ))


    # Potential Energy:

    # positions r = [x,y,z] for all particles
    x = pos[:,0:1]  # Selecting Column 1 from pos matrix
    y = pos[:,1:2]  # Selecting Column 2 from pos matrix
    z = pos[:,2:3]  # Selecting Column 3 from pos matrix

    # matrix that stores all pairwise particle separations: r_j - r_i
    dx = x.T - x    # Transpose of array X - array X
    #################### where each position X is subtracted from every other
    dz = z.T - z    # X in the matrix
    dy = y.T - y    # Same with Y and Z to make a 2D matrix

    # matrix that stores 1/r for all particle pairwise particle separations
    inv_r = np.sqrt(dx**2 + dy**2 + dz**2)
    inv_r[inv_r > 0] = 1.0 / inv_r[inv_r > 0]

    # sum over upper triangle, to count each interaction only once
    # potential energy of the system
    PE = G * np.sum(np.sum(np.triu(-(mass*mass.T)*inv_r,1)))
    # np.triu function: Return a copy of an array with the elements below the
k-th diagonal zeroed

    return KE, PE
```

To run the simulation, we need to first input initial parameters, for example, gravitaitonal constant G, the number of bodies N and so on as listed below. Tweaking these values allow us to view a wide variety of variations by changing one or a combination of parameters at a time. Currently, the code does not run a simulation for N bodies, but instead for 2 particles mimicking the Earth-Sun orbit. Uncommenting the lines above the specified mass will let us add more bodies in the simulation.

In [19]:
```
# Simulation parameters
N         = 2    # Number of particles
t         = 2      # current time of the simulation
tEnd      = 10.0   # time at which simulation ends
dt        = 0.01   # timestep
softening = 0.1    # softening length
G         = 1.0    # Newton's Gravitational Constant
plotRealTime = True # switch on for plotting as the simulation goes along

# Generate Initial Conditions
np.random.seed(17)            # set the random number generator seed

# mass = 20.0*np.ones((N,1))/N  # total mass of particles is 20
mass = np.array(([19.999_999_9], [0.000_000_1]))/N # 20.0*np.ones((N,1))/N  #
 total mass of particles is 20
pos  = np.random.randn(N,3)   # randomly selected positions and velocities
pos[0][0]  = 0   # This is so that the first particle has x = 0
pos[0][1]  = 0   # y = 0
pos[0][2]  = 0   # and z = 0
pos[1][0]  = 1   # This is so that the first particle has x = 0
pos[1][1]  = 0   # y = 0
pos[1][2]  = 0   # and z = 0
vel  = np.random.randn(N,3)

# Convert to Center-of-Mass frame
vel -= np.mean(mass * vel,0) / np.mean(mass)

# calculate initial gravitational accelerations
acc = getAcc( pos, mass, G, softening )

# calculate initial energy of system
KE, PE  = getEnergy( pos, vel, mass, G )

# number of timesteps
Nt = int(np.ceil(tEnd/dt))

# save energies, particle orbits for plotting trails
pos_save = np.zeros((N,3,Nt+1))
pos_save[:,:,0] = pos
KE_save = np.zeros(Nt+1)
KE_save[0] = KE
PE_save = np.zeros(Nt+1)
PE_save[0] = PE
t_all = np.arange(Nt+1)*dt

# prep figure
fig = plt.figure(figsize=(4,5), dpi=80)
grid = plt.GridSpec(3, 1, wspace=0.0, hspace=0.3)
ax1 = plt.subplot(grid[0:2,0])
ax2 = plt.subplot(grid[2,0])

# Simulation Main Loop
for i in range(Nt):
    # (1/2) kick
    vel += acc * dt/2.0
```

```
    # drift
    pos += vel * dt

    # update accelerations
    acc = getAcc( pos, mass, G, softening )

    # (1/2) kick
    vel += acc * dt/2.0

    # update time
    t += dt

    # get energy of system
    KE, PE  = getEnergy( pos, vel, mass, G )

    # save energies, positions for plotting trail
    pos_save[:,:,i+1] = pos
    KE_save[i+1] = KE
    PE_save[i+1] = PE

    # plot in real time
    if plotRealTime or (i == Nt-1):

        # Scatter plot following N bodies
        plt.sca(ax1)
        plt.cla()
        xx = pos_save[:,0,max(i-50,0):i+1]
        yy = pos_save[:,1,max(i-50,0):i+1]
        plt.scatter(xx,yy,s=1,color=[.7,.7,1])
        plt.scatter(pos[:,0],pos[:,1],s=10,color='blue')
        ax1.set(xlim=(-2, 2), ylim=(-2, 2))
        ax1.set_aspect('equal', 'box')
        ax1.set_xticks([-2,-1,0,1,2])
        ax1.set_yticks([-2,-1,0,1,2])

        # Scatter plot following KE, PE and Totale Energy
        plt.sca(ax2)
        plt.cla()
        plt.scatter(t_all,KE_save,color='red',s=1,label='KE' if i == Nt-1 else "")
        plt.scatter(t_all,PE_save,color='blue',s=1,label='PE' if i == Nt-1 else "")
        plt.scatter(t_all,KE_save+PE_save,color='black',s=1,label='Etot' if i == Nt-1 else "")
        ax2.set(xlim=(0, tEnd), ylim=(-300, 300))
        ax2.set_aspect(0.007)

        plt.pause(0.001)



# add labels/legend
plt.sca(ax2)
plt.xlabel('time')
plt.ylabel('energy')
ax2.legend(loc='upper right')
```

```
# Save figure
plt.savefig('nbody.png',dpi=240)
plt.show()
```




```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-19-1145540d4350> in <module>
     69     # plot in real time
     70     if plotRealTime or (i == Nt-1):
---> 71         plt.sca(ax1)
     72         plt.cla()
     73         xx = pos_save[:,0,max(i-50,0):i+1]

~\AppData\Local\Programs\Python\Python37\lib\site-packages\matplotlib\pyplot.
py in sca(ax)
    856             m.canvas.figure.sca(ax)
    857             return
--> 858     raise ValueError("Axes instance argument was not found in a figur
e")
    859 
    860 

ValueError: Axes instance argument was not found in a figure
```