Solar System Dynamics

Krittika Summer Projects 4.0

Suryansh Srijan

Solar System Dynamics

Krittika Summer Projects 4.0

Summer 2023

Project Student:

Suryansh Srijan

Computer Science and Engineering

Indian Institute of Technology Bombay

Mentors:

Adarsh Reddy Madur

Dhananjay Raman

Cover image: Edge of Night by Paul Wilson

Template style: Thesis style by Richelle F. van Capelleveen

Abstract

This project (as its name suggests) delves into solar system dynamics, focusing on simulating n-body motion and interesting phenomena such as analemma and Kirkwood gaps. Through computational simulations and analytical methods, the project explores celestial bodies' gravitational interactions, revealing intricate orbits and stability. It investigates the Earth's axial tilt and elliptical orbit's influence on the analemma, uncovering the underlying causes of its shape. The project also take a detour into general relativity, and shows how different it is from Newtonian gravity. Additionally, the project examines the resonance effects shaping Kirkwood gaps in the asteroid belt. By comprehensively studying these phenomena, the report advances our understanding of solar system mechanics and their significant role in shaping celestial bodies' motions and distributions.

Contents

1. Theory

1.1 Introduction

The goal of this project is to simulate interesting phenomena in the Solar System due the motion of its individual components. For this, we need to understand the concepts of orbital motion and different ways of numerical integration which will allow us to efficiently reach our goal. The various phenomena that we will simulate in this project are:

1. N-Body Motion
2. Circular Restricted Three-Body Motion (A Special Case) and Lagrange points
3. Analemma
4. Gravity due to large dense masses such as Black Holes (A deviation from Solar System but an interesting one)
5. Kirkwood Gaps

1.2 N-Body Motion

Considering a case of N bodies, for any two bodies i and j, according to Newton's Law of universal gravitation:

$$\vec{F}_{ij} = -\frac{Gm_i m_j(\vec{r}_i - \vec{r}_j)}{|\vec{r}_i - \vec{r}_j|^3}, \quad \vec{F}_{ji} = -\frac{Gm_i m_j(\vec{r}_j - \vec{r}_i)}{|\vec{r}_i - \vec{r}_j|^3}$$

where,

G = Universal Gravitaional constant,
m_i, m_j = masses of the two bodies,
$\vec{F}_{ij}, \vec{F}_{ji}$ = Force exerted on i by j and vice verca,
$\vec{r}_i, \vec{r}_j$ = Position vectors of i and j.

More parameters related to motion are:

- Kinetic Energy: $KE = \sum_{i=1}^{N} \frac{1}{2} m_i |\frac{d\vec{r}_i}{dt}|^2$
- Potential Energy: $PE = \sum_{t=1}^{N} \sum_{j>i}^{N} -\frac{Gm_i m_j}{|\vec{r}_i - \vec{r}_j|}$
- Energy: $E = KE + PE$
- Linear Momentum: $\vec{P} = \sum_{i=1}^{N} m_i \frac{d\vec{r}_i}{dt}$
- Angular Momentum: $\vec{L} = \sum_{i=1}^{N} m_i (\vec{r}_i \times \frac{d\vec{r}_i}{dt})$

Under absence of external force and torque the energy, linear momentum and angular momentum remain conserved. To simulate N-body motion and other attributes related to it, we apply the above formulae.

1.3 Circular Restricted three-body problem

In the case of two bodies, we can precisely find their equations of motion given the initial conditions. But in case of three or more bodies, the situation becomes chaotic.

The circular restricted three-body problem has two large masses in a circular orbit at their common center of mass. A third relatively smaller body is then introduced.

We will look at a case where all the bodies lie on the X-Y plane. There are two large bodies with mass m_1 and m_2 with their center of mass at the origin, which are moving around it in a circle, and a third small body with a mass m_3 being p_1 and p_2 distant from the two bodies respectively.

We will shift to a rotaional frame of reference, centered at the origin rotating counter-clockwise with angular velocity $\omega = \sqrt{\frac{G(m_1+m_2)}{(r_1+r_2)^3}}$, such that the positions of the larger bodies remain fixed. We will now study the motion of the third smaller body. Its acceleration in this frame is given by:

$$\frac{d^2x}{dt^2} = 2\omega\frac{dy}{dt} + \omega^2 x - \frac{Gm_1(x-x_1)}{p_1^3} - \frac{Gm_2(x-x_2)}{p_2^3}$$

$$\frac{d^2y}{dt^2} = -2\omega\frac{dx}{dt} + \omega^2 y - \frac{Gm_1(y-y_1)}{p_1^3} - \frac{Gm_2(y-y_2)}{p_2^3}$$

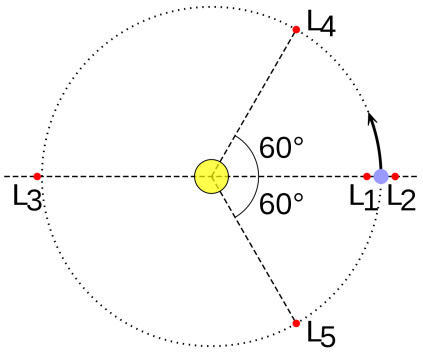
The kinetic and potential energy of this body is given by:

$$KE = \frac{1}{2} m_3 (\dot{x}^2 + \dot{y}^2 + 2x\omega\dot{y} - 2y\omega\dot{x} + \omega^2(x^2+y^2))$$

$$PE = -\frac{Gm_1m_3}{p_1} - \frac{Gm_2m_3}{p_2}$$

These equations will prove handy to do faster calculations, in case of simulating asteroids for kirkwood gaps.

There are somethings called Lagrange points. In the rotational frame of the two bodies, these points remain stationary at all times. There are 5 lagrange points as shown in the figure. Out of these L1, L2 and L2 are points of unstable equilibrium.



1.4 Analemma

An analemma is a diagram showing the position of the Sun in the sky as seen from a fixed location on Earth at the same mean solar time, as that position varies over the course of a year. Analemma looks like a lopsided "8". The reason being that it is the combined effect of the tilt of the Earth's axis and elliptical orbit of the Earth.

We will try to find analemma observed on Earth's north pole (just to simplify conversion between equatorial and horizontal co-ordinates). To do this, we calculate the α (Right ascension) and δ (Declination) of Sun on various days and then find the corresponding altitude and azimuth.

First we find the ecliptic longitude of Sun (λ). Since, we have to find it on different dates, we need to solve the following differential equation:

$$\dot{\theta} = \frac{2\pi(1 + e\cos\theta)^2}{T(1 - e^2)^{3/2}}$$

This equation is obtained by using the equation of ellipse and the expression of areal velocity in two body elliptical orbits. e is the eccentricity of Earth's orbit equal to 0.0167. The ecliptic longitude $\lambda = \theta - \theta_0$, where θ_0 is the angular distance of the vernal equinox and Earth's perihelion with respect to the Sun and $\theta_0 \approx 77°$.

Next, $\delta = \sin^{-1}(\sin\lambda \sin\epsilon)$ and $\alpha = \tan^{-1}(\tan\lambda \cos\epsilon)$. And Altitude($a$) = δ and Azimuth(A) = $\alpha - \omega t$, where $\omega = 2\pi/T$, t is the time of observation and ϵ is the inclination of the Earth's axis with respect to the vertical. We then plot A vs a to obtain analemma.

1.5 Gravity near massive bodies

We know that Newtonian gravity falls short in explaining multiple phenomena such as deviation of light due to massive celestial objects, and the precision of Mercury's orbit around the Sun. These things are explained by general relativity.

In general relativity, the metric tensor (denoted by $g_{\mu\nu}$) is the fundamental object of study. The metric captures everything about spacetime. It is used to define notions such as time, distance, volume, curvature, angle, and separation of the future and the past.

In general relativity, we deal with 4d space i.e., we consider time as a co-ordinate and the spacetime interval between two points (analogous to "distance" in 3d space) is given by[1]

$$ds^2 = g_{\mu\nu}dx^\mu dx^\nu = -c^2dt^2 + dx^2 + dy^2 + dz^2, \qquad x^\mu \in (t, x, y, z)$$

in the case of flat spacetime. This can be converted to polar co-ordinates as:

$$ds^2 = g_{\mu\nu}dx^\mu dx^\nu = -c^2dt^2 + dr^2 + r^2d\theta^2 + r^2\sin^2\theta\, d\phi^2, \qquad x^\mu \in (t, r, \theta, \phi)$$

The metric $g_{\mu\nu}$ in this case is given by $\begin{pmatrix} -c^2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & r^2 & 0 \\ 0 & 0 & 0 & r^2\sin^2\theta \end{pmatrix}$.

The metric tensor in the case of uncharged, non-rotating body, centered at the origin is given by the Schwarzschild metric:

$$g_{\mu\nu} = \begin{pmatrix} -(1 - \frac{2GM}{rc^2}) & 0 & 0 & 0 \\ 0 & (1 - \frac{2GM}{rc^2})^{-1} & 0 & 0 \\ 0 & 0 & r^2 & 0 \\ 0 & 0 & 0 & r^2\sin^2\theta \end{pmatrix} \text{ for co-ordinates } (ct, r, \theta, \phi)$$

Here, the space is said to be curved due the mass of the body M.

The motion of bodies in a metric is given by a geodesic (a path that a particle which is not accelerating would follow for example, a straight line in flat space, a great circle on a sphere). The general equation of a geodesic is[2]:

$$\frac{d^2x^\mu}{d\lambda^2} = -\Gamma^\mu_{\alpha\beta}\frac{dx^\alpha}{d\lambda}\frac{dx^\beta}{d\lambda}$$

where λ is a path parameter that increases monotonically along the particle's path and $\Gamma^\mu_{\alpha\beta} = \frac{g^{\mu\nu}}{2}\left[\frac{\partial g_{\alpha\nu}}{\partial x^\beta} + \frac{\partial g_{\beta\nu}}{\partial x^\alpha} - \frac{\partial g_{\alpha\beta}}{\partial x^\nu}\right]$, $g^{\mu\nu}$ is the inverse of $g_{\mu\nu}$

For simplicity, we will consider a photon in X-Y plane ($\theta = \pi/2$). The geodesic equations in this case of a Schwarzschild metric after simlification are:

$$E = \left(1 - \frac{r_s}{r}\right)\frac{dt}{d\lambda}, \quad r_s = \frac{2GM}{c^2} \quad \text{Energy conservation}$$

[1] whenever greek letters are introduced, it implicitly implies summation over all co-ordinates.
[2] Here summation is implied only on α and β

$$L = r^2 \frac{d\phi}{d\lambda} \quad \text{Angular Momentum conservation}$$

$$\frac{1}{2}\left(\frac{dr}{d\lambda}\right)^2 + V_{eff} = \frac{c^2 E^2}{2}, \quad V_{eff} = \frac{c^2}{2}\left(1 - \frac{r_s}{r}\right) + \frac{L^2}{2r^2} - \frac{L^2 r_s}{2r^3}$$

The $\frac{L^2 r_s}{2r^3}$ is different from newtonian gravity. This V_{eff} leads us to write the acceleration of a photon as:

$$\vec{a} = -\frac{3L^2 r_s \vec{r}}{2r^5}$$

This formula will be used later to simulate orbits of photons around black holes.

1.6 Kirkwood Gaps

A Kirkwood gap is a gap or dip in the distribution of the semi-major axes (or equivalently of the orbital periods) of the orbits of main-belt asteroids. They correspond to the locations of orbital resonances with Jupiter. Two bodies are

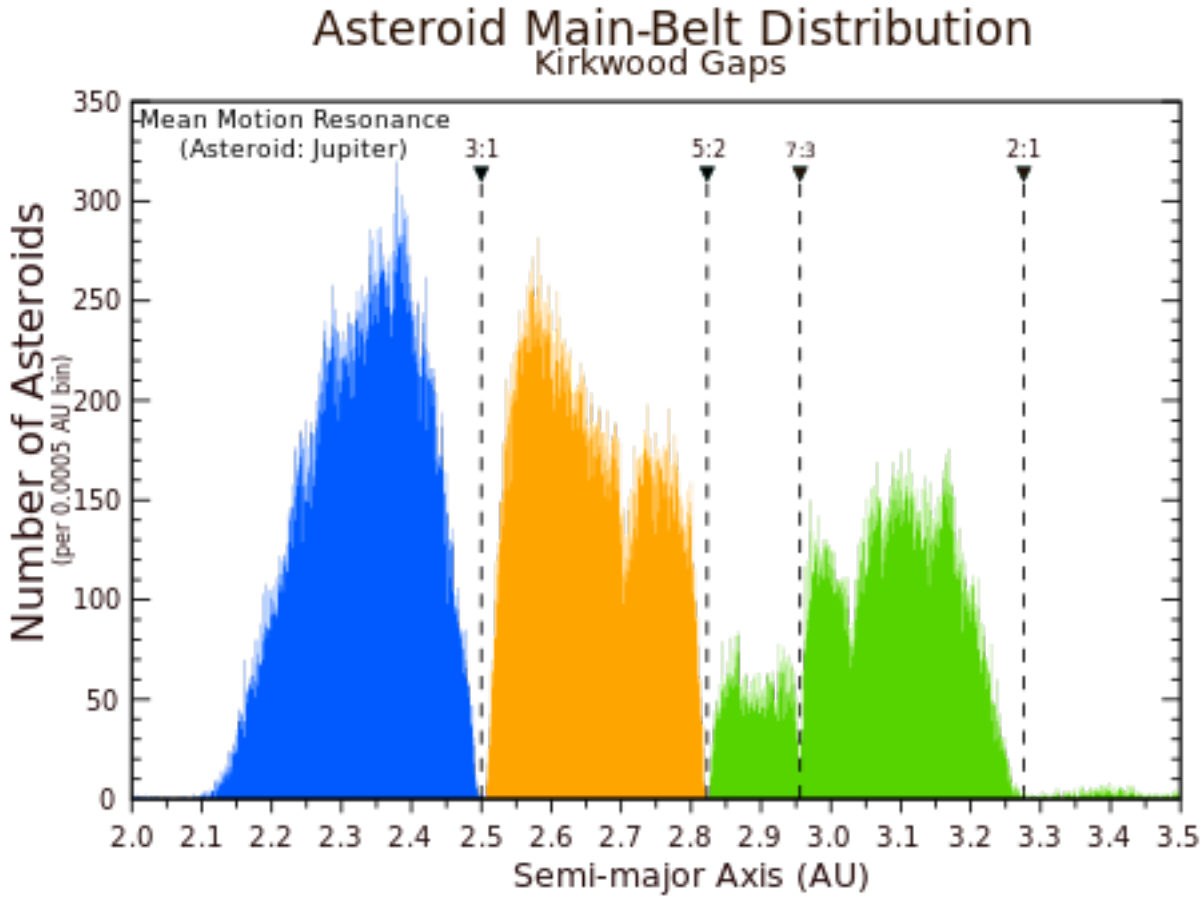


Figure 1.1: *Histogram showing the four most prominent Kirkwood gaps and a possible division into inner, middle and outer main-belt asteroids.*

said to be in orbital resonance if their orbital periods can be expressed as a ratio of two simple integers. The main gaps observed in the asteroid belt are at:

- 3.279 AU (2:1 resonance)
- 2.502 AU (3:1 resonance)
- 2.825 AU (5:2 resonance)
- 2.958 AU (7:3 resonance)

1.7 Numerical integration methods

In this project we will be using numerical methods to solve inital value problems. Most of the time we will dealing with 2nd orger differential equations of the form:

$$\frac{d^2\vec{r}(t)}{dt^2} = f(\vec{r}, t), \quad \vec{r}(0) = \vec{r}_0, \quad \frac{d\vec{r}(0)}{dt} = \vec{v}_0$$

To solve these, we will use the following three algorithms:

1. **Euler**: The following algorithm is used to solve for $\vec{r}$ at times with a time step Δt:
$$v(t + \Delta t) = v(t) + a(t) \cdot \Delta t$$
$$r(t + \Delta t) = r(t) + v(t) \cdot \Delta t$$
2. **Euler-Chromer**: This is a slight modification of Euler method and is also called the backward-Euler method:
$$v(t + \Delta t) = v(t) + a(t) \cdot \Delta t$$
$$r(t + \Delta t) = r(t) + v(t + \Delta t) \cdot \Delta t$$
3. **Euler-Richardson**: In this, instead of using $v(t)$ or $v(t + \Delta t)$ to calculare r, we use the v at the mid of the time step:
$$a(t) = f(r(t), v(t))$$
$$v_{mid} = v(t) + a(t) \cdot \Delta t/2$$
$$r_{mid} = r(t) + v(t) \cdot \Delta t/2$$
$$a_{mid} = f(r_{mid}, v_{mid})$$
$$v(t + \Delta t = v_{mid} + a_{mid} \cdot \Delta t/2)$$
$$r(t + \Delta t) = r(t) + v_{mid} \cdot \Delta t$$

I have used Euler-Chromer algorithm for most of the simulations. But I have used Euler-Richardson for simulating asteroids to obtain a greater accuracy.

2. Results and Code

Most of the coding is done in python, due to ease of plotting and libraries like numpy. But we shift to C with CUDA to use parallel processing for asteroid simulation.

2.1 Two body simulations

Simulating two-body motion is relatively simple. First we define a particle class:

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

class Particle:
    def __init__(self, mass,r,v): # Constructor
        self.mass = mass
        self.pos = r
        self.vel = v

    def speed(self): # Returns speed of the particle
        return math.sqrt(self.vel[0]**2+self.vel[1]**2)

    def update(self, step, f):
        # Updates the particle using Euler-Chromer algorithm
        self.vel = self.vel + f*step/self.mass
        self.pos = self.pos + step*self.vel
```

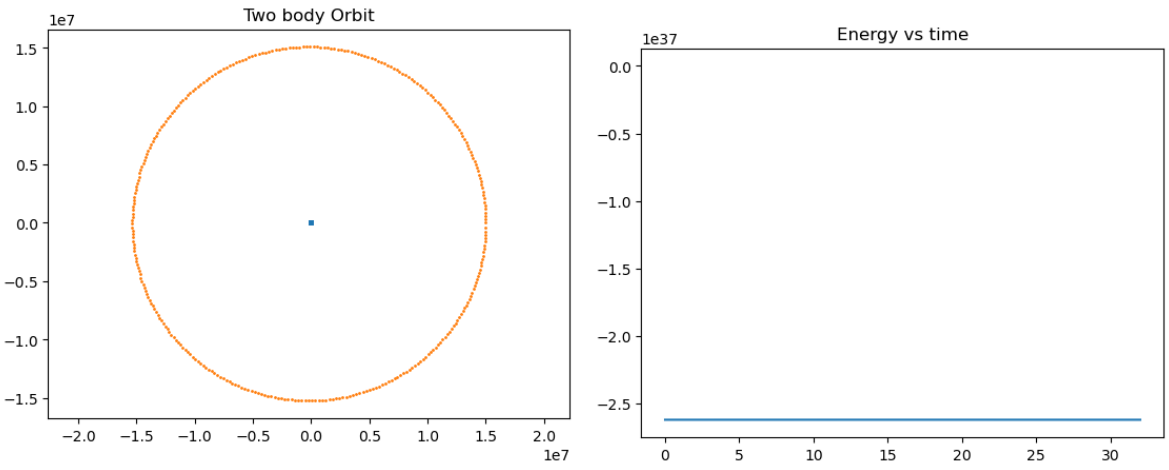
I make another class for two-body simulation and plotting the gravitational potential due to the two bodies

```
class GravSimul_twobody:
    global G
    G=6.674e-11

    def __init__(self, pp1, pp2): # Constructor
        self.p1 = pp1
```

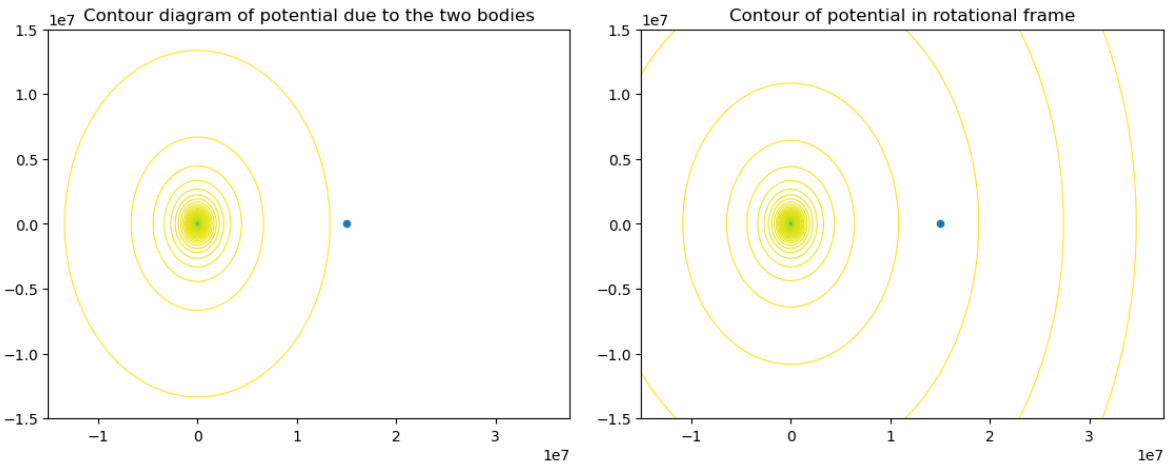
```
        self.p2 = pp2

    def dist(self):
    # Returns distance between the two bodies

    def calcF(self):
    # Calculates force for 1 body which can also be used for 2nd body

    def Update(self,step):
    # Updates the two bodies with time step

    def energy(self):
    # Returns total energy

    def simulate(self, del_t, num):
    # Simulate motion and also plot the position of the two bodies
    # along with total energy vs time plot

    def contourplot(self) :
    # Create contour plot of the graviational potential in
    # inertial frame and also in rotational frame
    # for reduced circular 3-body problem
```

I obtained the following orbits for the following initial conditions.

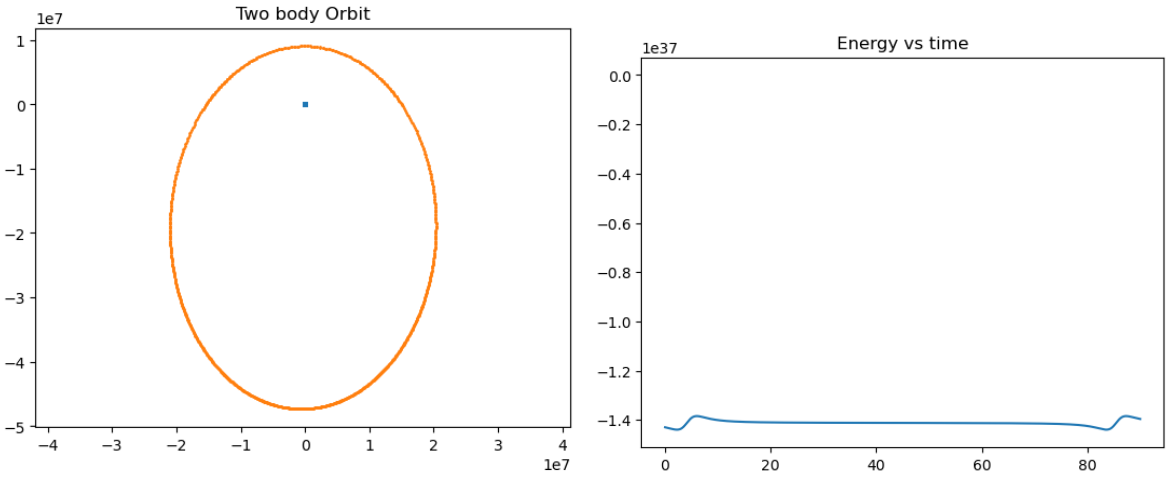
```
p1 = Particle(2e30,np.array([0,0]),np.array([0,0]))
p2 = Particle(5.96e24,np.array([1.5e7,0]),np.array([0,3e6]))
```



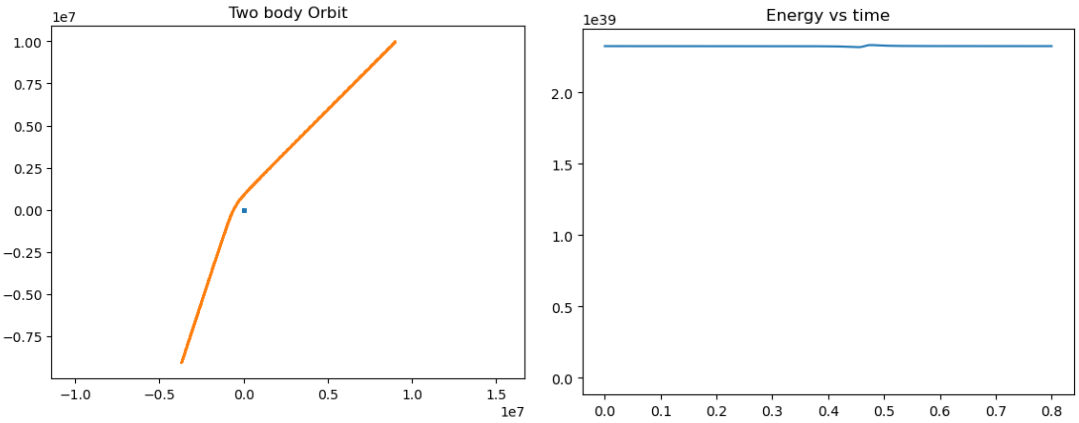
And the potential due two these particles:



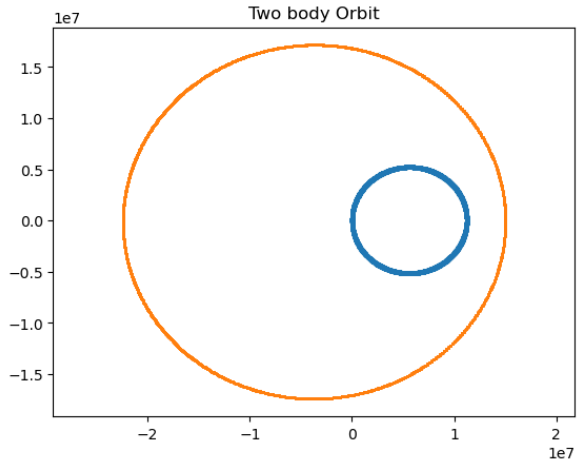

```
p1 = Particle(2e30,np.array([0,0]),np.array([0,0]))
p2 = Particle(5.96e24,np.array([1.5e7,0]),np.array([-2e6,3e6]))
```

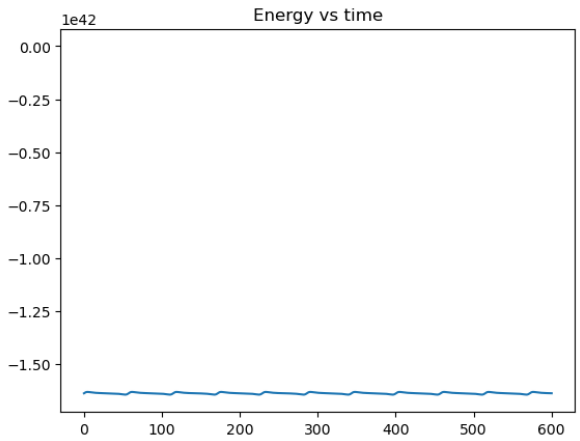



```
p1 = Particle(2e30,np.array([0,0]),np.array([0,0]))
p2 = Particle(5.96e24,np.array([0.9e7,1e7]),np.array([-2e7,-2e7]))
```

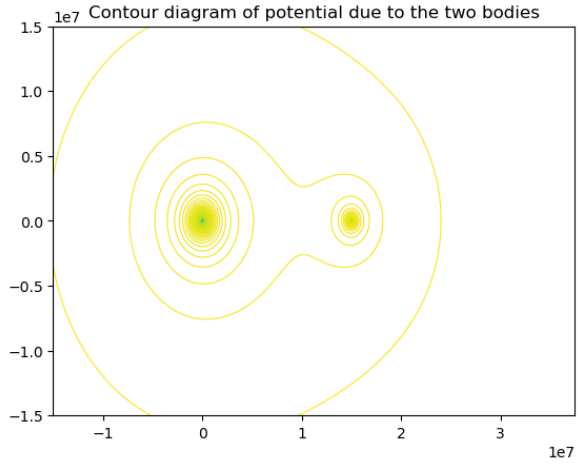


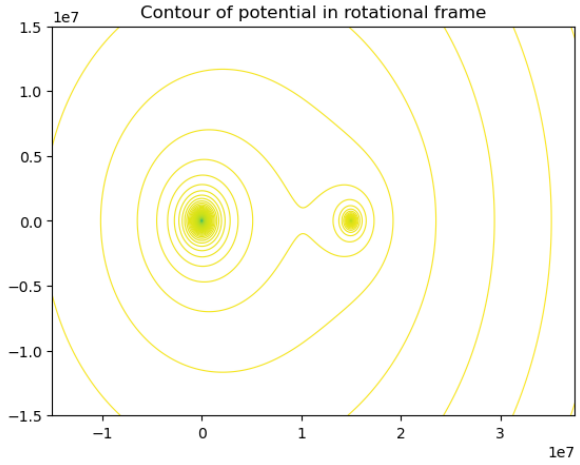
```
p1 = Particle(2e30,np.array([0,0]),np.array([0,-1e6]))
p2 = Particle(6e29,np.array([1.5e7,0]),np.array([0,3e6]))
```





And the potential due two these particles:





2.2 N-body simulations

I used the same particle class but a different n-body simulation class.

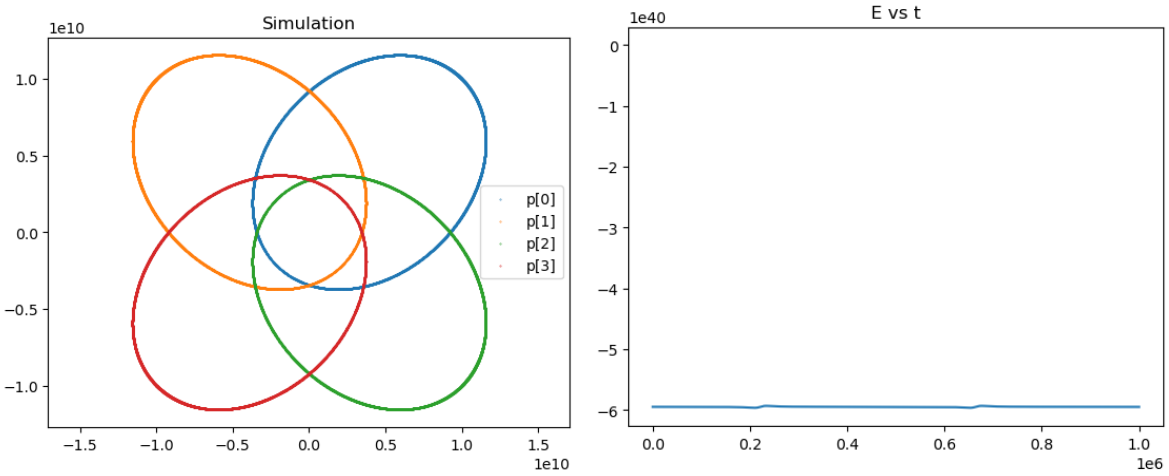
```
class GravSimul_nbody:
    global G
    G = 6.674e-11
    num = 0
    p = []
    def __init__(self,n:int,m,r,v): # Constructor
        self.num = n
        self.p = []
        for i in range(n):
            self.p.append(Particle(m[i],r[i],v[i]))
```

```
    @staticmethod
    def dist(a, b):
    # Returns distance between two particles a and b

    @staticmethod
    def angle(a, b):
    # Returns the angle the displacement vector
    # of a and b makes with the x axis

    def calcF(self,i:int):
    # Calculates the force experienced by the
    # ith particle due to all other particles

    def energy(self):
    # Returns the total energy of the system

    def Update(self,del_t):
    # Updates the parametters of all the parameters after given time step

    def simulate(self,del_t,steps):
    # Simulates the motion of all the particles
    # and also plot their positions with time

    def simulate_restricted(self, del_t, steps):
    # In this function,  we do the same simulation as before
    # the first two particles are the heavy particles of the system
    # the rest are lighter
    # the parameters of the heavy bodies should be of a circular orbit
    # based on that we plot the positions of
    # all the small bodies in the rotaional frame
    # by rotating their positions about the centre of mass
    # by the angle theta which is angular_velocity*time_elapsed
```

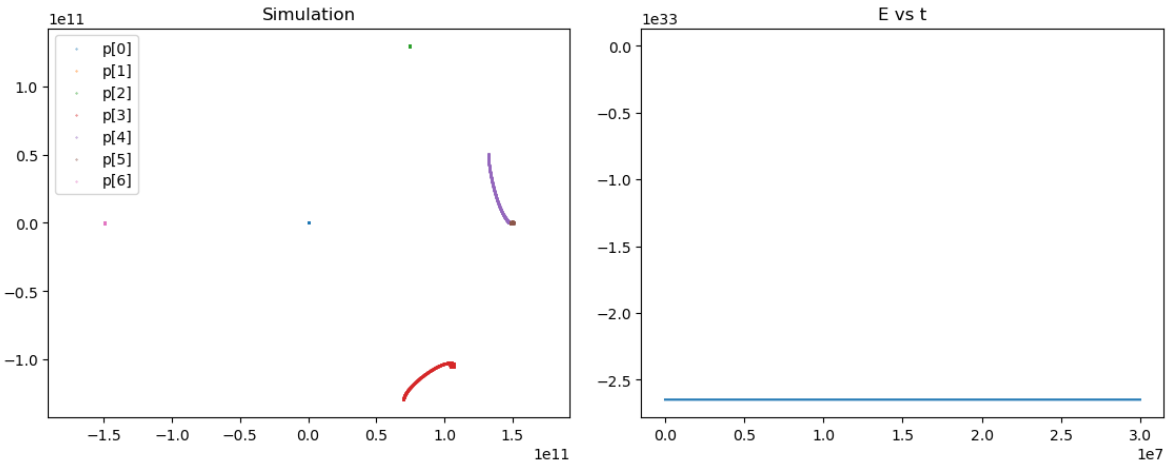
A beautiful example of a 4-body system:

```
m = np.array([2e30,2e30,2e30,2e30])
r = np.array([np.array([1.0e10,1.0e10]),
              np.array([-1.0e10,1.0e10]),
              np.array([1.0e10,-1.0e10]),
              np.array([-1.0e10,-1.0e10])])
v = np.array([np.array([-4.0e4,4.0e4]),
              np.array([-4.0e4,-4.0e4]),
              np.array([4.0e4,4.0e4]),
              np.array([4.0e4,-4.0e4])])
```

An example of Lagrange points and how they remain stationary in rotational frame and are not stationary even with small deviation:

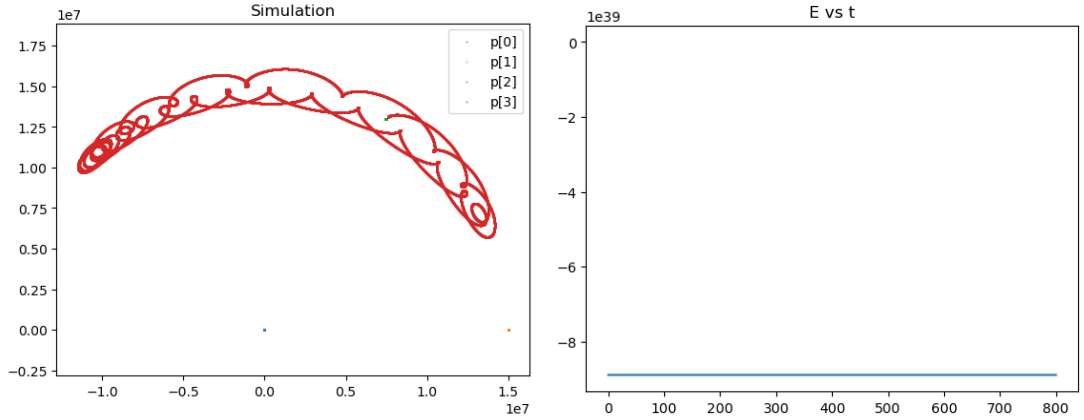



```
m = np.array([1.98847e30,5.9722e24,6e10,4e6,2e6,2e6,2e6])
r = np.array([np.array([0.0,0.0]),
              np.array([1.49598e11,0.0]),
              np.array([7.48e10,1.2956e11]),
              np.array([7e10,-1.2956e11]),
              np.array([1.481e11,0.0]),
              np.array([1.511e11,0.0]),
              np.array([-1.496e11,0.0])])
v = np.array([np.array([0.0,0.0]),
              np.array([0.0,2.978446e4]),
              np.array([-2.579e4,1.489e4]),
              np.array([2.579e4,1.489e4]),
              np.array([0.0,2.948e4]),
              np.array([0.0,3.008e4]),
              np.array([0.0,-2.978e4])])
ob = GravSimul_nbody(7,m,r,v) ob.simulate_restricted(10000,3000)
```



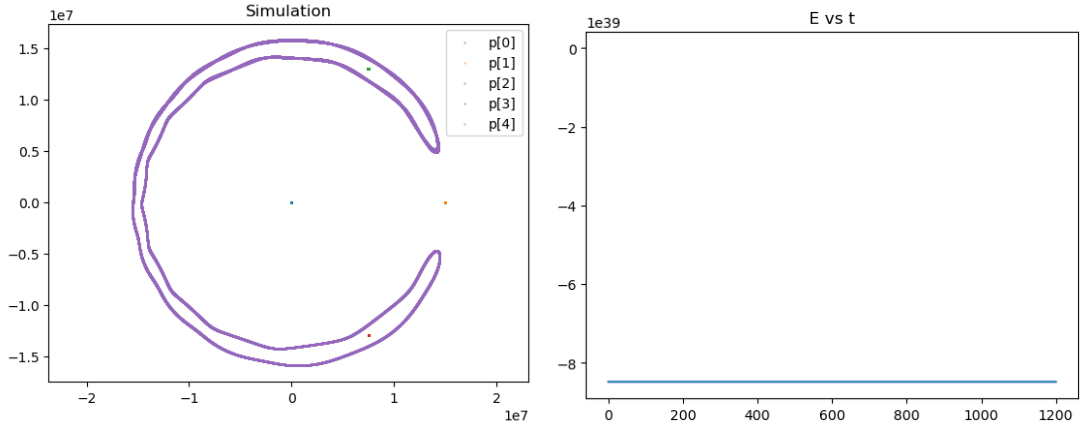
Next are some special orbits in restricted three body problem. Tadpole orbit:

```
m = np.array([2e30,2e27,6e7,6e7])
r = np.array([np.array([0.0,0.0]),
              np.array([1.5e7,0.0]),
              np.array([7.5e6,12990381.056766579701456]),
              np.array([7.62e6,13110381.056766579701456])])
v = np.array([np.array([0.0,0.0]),
              np.array([0.0,2984554.461445348350461]),
              np.array([-2584699.982589855630612,1492277.230722674175231]),
              np.array([-2608576.418281418417416,1516153.666414236962034])])
```



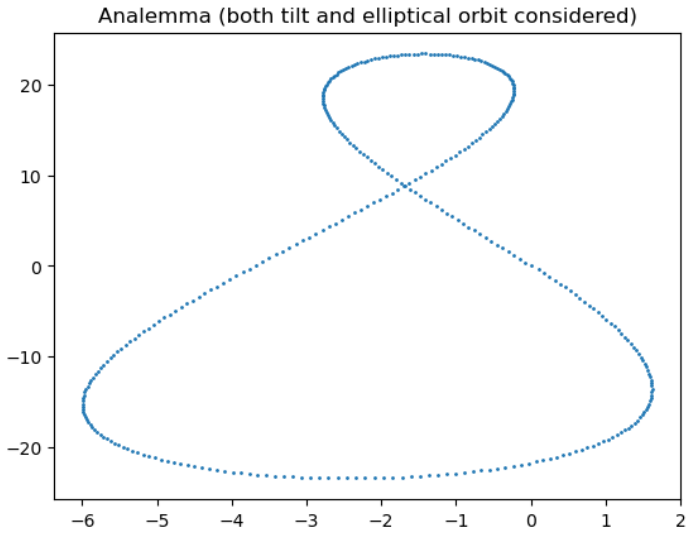
Horseshoe orbit:

```
m = np.array([2e30,1.90775e27,6e7,6e7,6e7])
r = np.array([np.array([0.0,0.0]),
              np.array([1.5e7,0.0]),
              np.array([7.5e6,12990381.056766579701456]),
              np.array([7.5e6,-12990381.056766579701456]),
              np.array([-1.541175e7,0.0])])
v = np.array([np.array([0.0,0.0]),
              np.array([0.0,2984485.698128462021083]),
              np.array([-2584640.431810583589904,1492242.849064231010542]),
              np.array([2584640.431810583589904,1492242.849064231010542]),
              np.array([0.0,-2946075.367193548714872])])
```



2.3 Analemma

I simply used the formulas used in the theory to simulate analemma.




```
from scipy.integrate import solve_ivp

num = 365
eps = 23.44*math.pi/180
w = 2*math.pi/num
t = np.arange(num)
lam = w*t
dec = np.zeros(num,dtype="float64")
alpha = np.zeros(num,dtype="float64")
alt = np.zeros(num,dtype="float64")
azi = np.zeros(num,dtype="float64")

# Using scipy to solve differential eqn for theta
theta0 = 77*math.pi/180
def dth_dt(t, y):
    e = 0.0167
    return (2*math.pi*((1+e*math.cos(y+theta0))**2))/(num*((1-e**2)**(1.5)))

sol = solve_ivp(dth_dt,[0,num],[0],t_eval=t)

lam = sol.y.flatten()
for i in range(num):
    x = 0
    if lam[i] > 3*math.pi/2: x = 2
    elif lam[i] > math.pi/2: x = 1
    dec[i] = math.asin(math.sin(eps)*math.sin(lam[i]))
```

```
    alpha[i] = math.atan(math.tan(lam[i])*math.cos(eps)) + math.pi*x
    alt[i],azi[i] = dec[i],alpha[i]-w*i

plt.scatter(np.rad2deg(azi),np.rad2deg(alt),s = 1)
plt.title("Analemma (both tilt and elliptical orbit considered)")
plt.show()
```

2.4 Photons near black-holes

We setup the initial condition of the photon to be moving towards the black hole from the x-direction at some impact parameter $p \cdot r_{bh}$. To simulate this I create a class Photon which accepts an impact parameter and sets an initial position at $(200r_{bh}, pr_{bh})$ (200 is assumed to be far enough).

The radius of shadow of a black hole is found to be $\frac{3\sqrt{3}}{2} r_{bh} \approx 2.598 r_{bh}$. Thus, for $p < 2.598$ the photon should go inside the black hole.

```
c = 3e8
class Photon:
    def __init__(self, r, p): # Constructor
        self.pos = np.array([200*r,p*r])
        self.dir = np.array([-1,0],dtype="float64")
        self.rbh = r
        self.l = c*p*r

    def r(self): # Returns polar co-ordinate r
        return math.sqrt(self.pos[0]**2+self.pos[1]**2)

    def theta(self): # Returns polar co-ordinate theta
        return math.atan2(self.pos[1],self.pos[0])

    def speed(self):
    # Returns speed of photon due to
    # the Schwarzchild metric in units of c
        th = self.theta()
        vr = self.dir[0]*math.cos(th) + self.dir[1]*math.sin(th)
        vth = self.dir[1]*math.cos(th) - self.dir[0]*math.sin(th)
        x = 1-self.rbh/self.r()
        return math.sqrt(vr**2+vth**2*x)

    def update(self,del_t): # Updates photon parameter after time step
        a = np.zeros(2,dtype="float64")
        a = -1.5*(self.l**2)*self.rbh*self.pos/(self.r()**5)
        # Used the acceleration obtained in the theory section
        self.dir += a*del_t/c
```

```
        self.pos += self.dir*c*del_t

Mbh = 1e38
Rbh = 6.674e-11*Mbh/(c**2)
n = 9
steps = 100000
del_t = 1
para = np.array([4.5,4,3.5,3.0,2.8,2.675,2.6,2.59,2])
data = np.zeros(((2*n),steps+1),dtype="float64")
speed = np.zeros((n,steps+1),dtype="float64")
time = del_t*np.arange(steps+1)
fig, ax = plt.subplots()
ax.add_patch(plt.Circle((0,0),Rbh,color="black"))
ax.add_patch(plt.Circle((0,0),1.5*Rbh,color="black",fill=False))
for i in range(n):
    ph = Photon(Rbh,para[i])
    data[2*i][0] = ph.pos[0]
    data[2*i+1][0] = ph.pos[1]
    speed[i][0] = ph.speed()
    for j in range(steps):
        if ph.r()>Rbh:
            ph.update(del_t)
            speed[i][j+1] = ph.speed()
        data[2*i][j+1] = ph.pos[0]
        data[2*i+1][j+1] = ph.pos[1]
max = np.max(para) + 1
ax.set_xlim([-1.5*max*Rbh, 1.5*max*Rbh])
ax.set_ylim([-1.5*max*Rbh, 1.5*max*Rbh])
for i in range(n):
    ax.plot(data[2*i],data[2*i+1],label=para[i])
ax.set_aspect("equal")
plt.legend()
plt.show()
for i in range(n):
    plt.plot(time,speed[i],label=para[i])
    plt.xlim(47000,53000)
    plt.ylim(0.997,1.003)
plt.title("Speed of photons according to Schwarschild Metric")
plt.show()
```

There is variation in speed only when the photens are near the blackhole. After a photon goes inside it, its speed is set to 0.

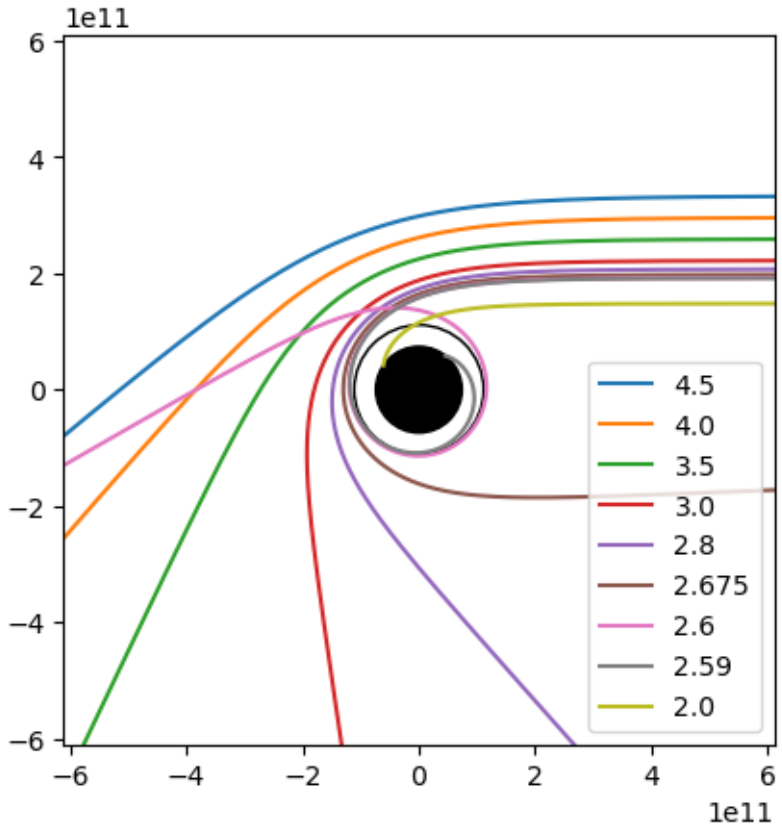
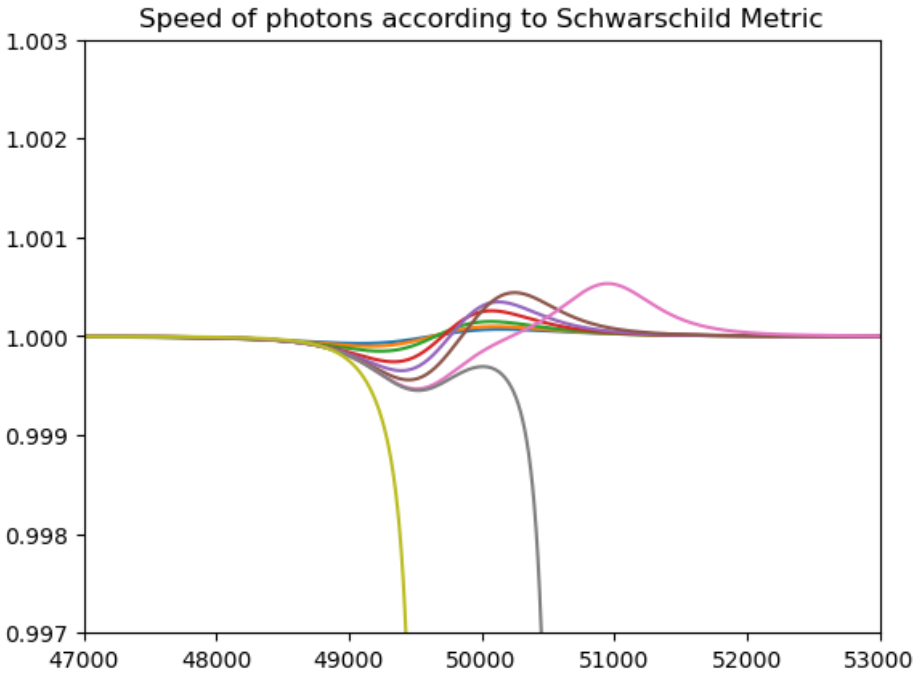


Figure 2.11: Trajectories of photons with various impact parameters under the influence of a black hole



2.5 Kirkwood Gaps

Simulating Kirkwood gaps is quite a heavy task as it involves finding parameters for >1000 asteroids and for >1000 years of time with a relatively small time step ≈ days. So, I shifted to CUDA for parallel processing of asteroids and ignored mass of asteroids (as asteroid-asteroid iinteraction is negligible). It is also assumed that Jupiter is in a circular orbit, so we can fix its position. The CUDA code for the following:

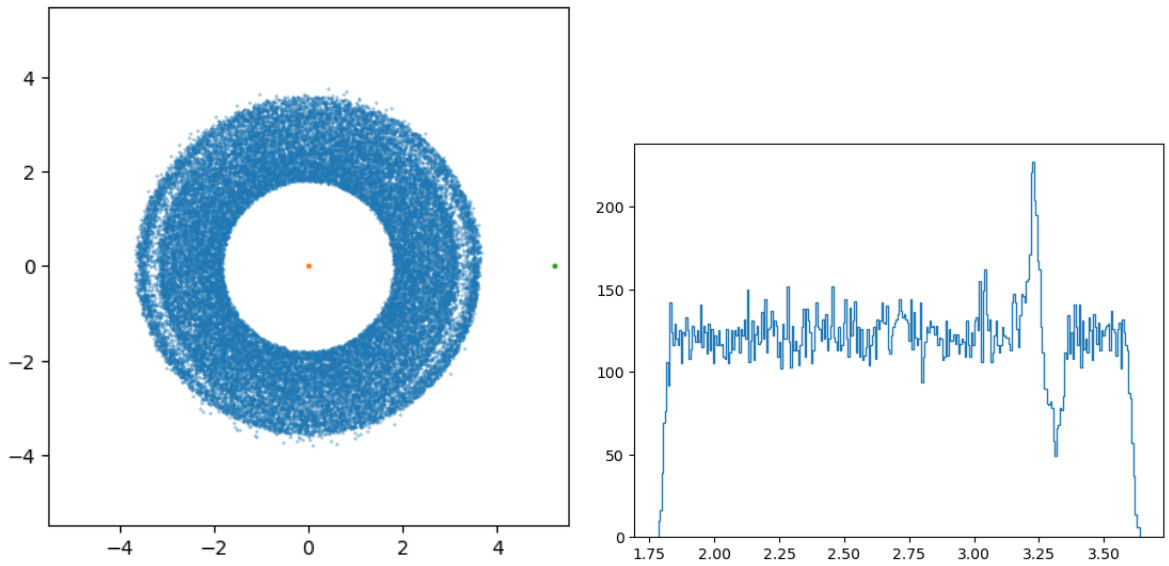
```
#include "cuda_runtime.h"
#include "device_launch_parameters.h"

#include <stdio.h>
#include <cstdlib>
#include <math.h>
#include <time.h>

double drand() {
    return (double)rand()/(double)RAND_MAX;
}

// function runs for every epoch on GPU
__global__ void calcKernel(double *x, double *y, double *vx,
            double *vy, int n_ast, long long n_step,
            double del_t, double r1, double r2, double w,
            double w2, double c1, double c2) {

    int i = blockIdx.x*blockDim.x+threadIdx.x;
    if(i<n_ast) {
        for(int j = 0;j<n_step/10;++j) {
            // Implements Euler-Richardson technique
            double p1 = (x[i]-r1)*(x[i]-r1)+y[i]*y[i],
            p2 = (x[i]-r2)*(x[i]-r2)+y[i]*y[i],
            ax = 2*w*vy[i] + w2*x[i] - c1*(x[i]-r1)*rsqrt(p1)/p1
                - c2*(x[i]-r2)*rsqrt(p2)/p2,
            ay = -2*w*vx[i] + w2*y[i] - c1*y[i]*rsqrt(p1)/p1
                - c2*y[i]*rsqrt(p2)/p2;
            double vxmid = vx[i] + ax*del_t/2;
            double vymid = vy[i] + ay*del_t/2;
            double xmid = x[i] + vxmid*del_t/2;
            double ymid = y[i] + vymid*del_t/2;
            p1 = (xmid-r1)*(xmid-r1)+ymid*ymid;
            p2 = (xmid-r2)*(xmid-r2)+ymid*ymid;
            ax = 2*w*vymid + w2*xmid - c1*(xmid-r1)*rsqrt(p1)/p1
                - c2*(xmid-r2)*rsqrt(p2)/p2;
            ay = -2*w*vxmid + w2*ymid - c1*ymid*rsqrt(p1)/p1
```

```
                - c2*ymid*rsqrt(p2)/p2;
            x[i] += vxmid*del_t;
            y[i] += vymid*del_t;
            vx[i] += ax*del_t;
            vy[i] += ay*del_t;
        }
    }
}

int main() {
    clock_t start,end;
    double cpu_time_used;
    srand(0);

    double pi = 3.14159265358979323851;
    const int n_ast = 16384;
    double m1 = 1.989e30, m2 = 1.898e27, rj = 7.78479e11,
           r1 = -m2*rj/(m1+m2), r2 = m1*rj/(m1+m2),
           G = 6.674e-11, w2 = G*(m1+m2)/pow(rj,3),
           w = sqrt(w2), c1 = G*m1, c2 = G*m2;
    // constants to make calculations faster
    long long n_step = 262800000, del_t = 14400;
    // A total time of approx 1,20,000 years

    // Allocate memory for parameters of asteroids
    double *x, *y, *vx, *vy;
    cudaMallocManaged(&x, n_ast * sizeof(double));
    cudaMallocManaged(&y, n_ast * sizeof(double));
    cudaMallocManaged(&vx, n_ast * sizeof(double));
    cudaMallocManaged(&vy, n_ast * sizeof(double));

    for(int i=0;i<n_ast;++i) {
        // Initialize the position and velocities
        // of the asteroids
    }

    // file pointer for printing in csv file
    FILE *fptr;
    if ((fptr = fopen("outcuda1.csv","w")) == NULL){
       printf("Error! opening file");
       exit(1);
   }

    start = clock();
    for(int j=0;j<n_ast;++j) {
```

```
        // Semi-major axis
        double a = 1/(2*rsqrt(x[j]*x[j]+y[j]*y[j])
                -((vx[j]-w*y[j])*(vx[j]-w*y[j])
                +(vy[j]+w*x[j])*(vy[j]+w*x[j]))/(c1));
        fprintf(fptr, "%6.9e, %6.9e, %6.9e \n",x[j],y[j],a);
    }

    // Calculates and saves the position of asteroids
    // after every epoch
    for(int i=0;i<10;++i) {
        // Runs the code on GPU
        calcKernel <<<64,256>>> (x,y,vx,vy,n_ast,n_step,
                                 del_t,r1,r2,w,w2,c1,c2);
        cudaDeviceSynchronize();

        printf("%d ",i);
        for(int j=0;j<n_ast;++j) {
            double a = 1/(2*rsqrt(x[j]*x[j]+y[j]*y[j])
                    -((vx[j]-w*y[j])*(vx[j]-w*y[j])
                    +(vy[j]+w*x[j])*(vy[j]+w*x[j]))/(c1));
            fprintf(fptr, "%6.9e, %6.9e, %6.9e \n",x[j],y[j],a);
        }

    }
    end = clock();

    cudaFree(x);
    cudaFree(y);
    cudaFree(vx);
    cudaFree(vy);

    cpu_time_used = ((double) (end - start)) / CLOCKS_PER_SEC;
    printf("%4.3f sec",cpu_time_used);
    return 0;
}
```

I used two methods of initializing the positions and velocities. First method: assumed all the asteroids are in circular orbit with the Sun with radius between 1.8 AU and 3.6 AU. This gave me the following result for n_ast = 35840, del_t = 53200 and n_steps = 43800000 which is approxiamately for 60000 yrs. We can clearly see the KirkWood gap at 2:1 orbital resonance. However nothing is observed at other orbital resonances.

The second method was to make a more realistic case, with all the asteroids having semi-major axes between 1.8 AU and 3.6 AU, and eccentricity less than 0.35. For n_ast = 16384, del_t = 14400 and n_steps = 262800000 which is ap-



(a) *Position of asteroids, the Sun and Jupiter* **(b)** *Histogram of semi-major axes of asteroids*

Figure 2.12: *Asteroids after 60,000 years of simulation using the first initialization method*

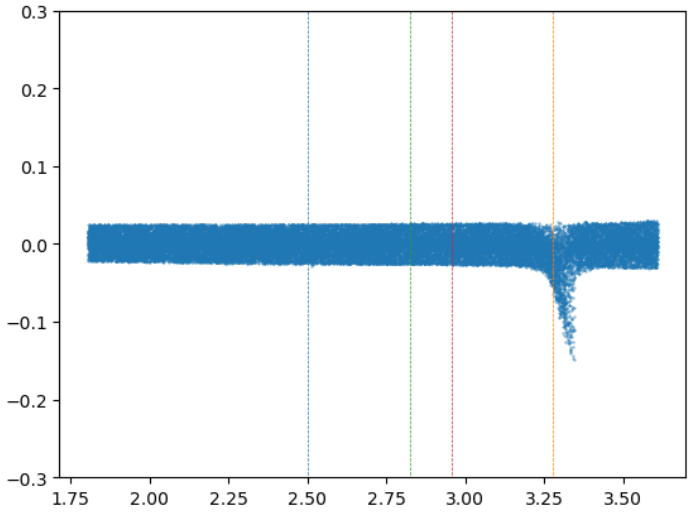
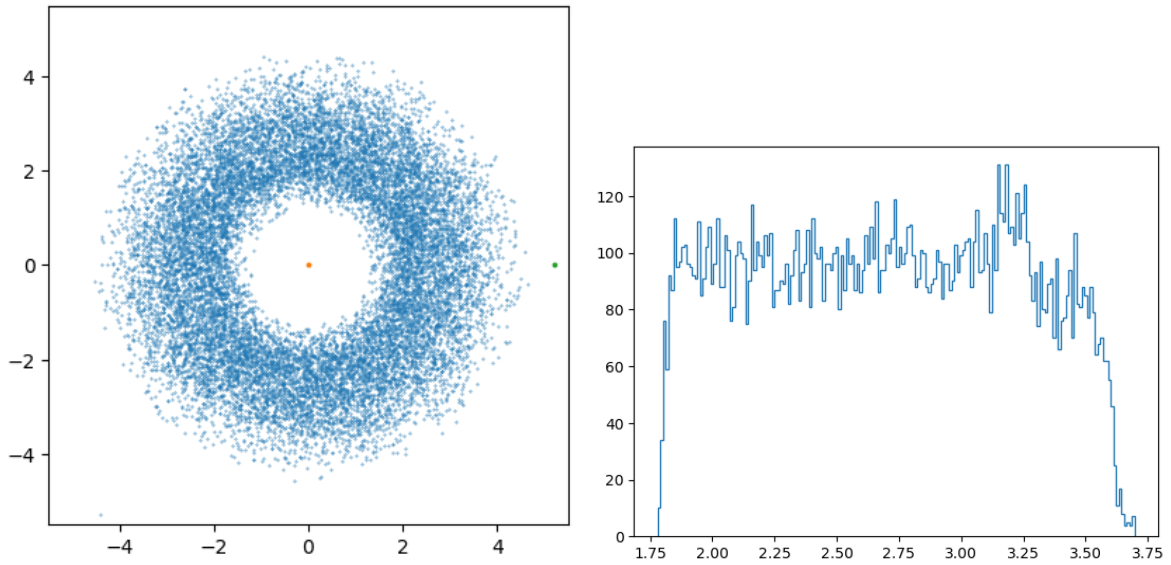


Figure 2.13: *This plot is the differences in semi-major axes of all the asteroid from initial to final states along with lines marking the prominent Kirkwood gaps in the Solar System*

proxiamately for 1,20,000 yrs, the following result is obtained: Here, we can see some asteroids that have deviated from the positions of other Kirkwood gaps.

Limitations:
(1) We are assuming Jupiter to be in a circular orbit around the Sun.
(2) We are neglecting the forces due to other planets.
(3) A better numerical integration technique can be used.
(4) A better initial set of asteroids can be prepared to give a better final result.



(a) *Position of asteroids, the Sun and Jupiter* **(b)** *Histogram of semi-major axes of asteroids*

Figure 2.14: *Asteroids after 1,20,000 years of simulation using the 2nd initialization method*

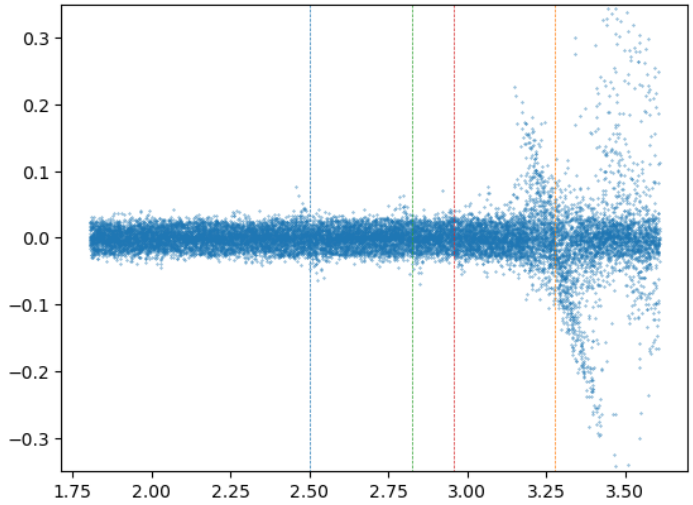


Figure 2.15: *Differences between semi-major axes of all the asteroid from initial to final states along with lines marking the prominent Kirkwood gaps in the Solar System*

3. References

1. https://en.wikipedia.org/wiki/Newton%27s_law_of_universal_gravitation
2. https://en.wikipedia.org/wiki/Celestial_mechanics
3. https://jan.ucc.nau.edu/~ns46/student/2010/Frnka_2010.pdf
4. https://en.wikipedia.org/wiki/Analemma
5. https://en.wikipedia.org/wiki/Euler_method
6. https://www.physics.udel.edu/~bnikolic/teaching/phys660/numerical_ode/node4.html
7. https://www.diva-portal.org/smash/get/diva2:566736/FULLTEXT01.pdf
8. https://en.wikipedia.org/wiki/Kirkwood_gap