A Computational Astronomy Project on

Modelling Gravitational Systems

Submitted by: Pratham Srivastava

B. Tech, Electrical Engineering

IIT Bombay

Dated: 12-01-2025

Abstract:

Have you ever wondered how planets orbit stars, galaxies take shape, or why some celestial systems are orderly while others descend into chaos? This project focuses on computationally modelling gravitational systems using Python, PyTorch and Matplotlib. It includes the simulation of 2-body, 3-body and in general, n-body systems to study their dynamics. The project also provides the functionality to generate contour plots of gravitational potential, offering insights into the spatial distribution of gravitational potential. These simulations leverage numerical methods for solving differential equations, providing an effective way to explore celestial mechanics and gravitational interactions.

1. Introduction

Gravitational systems are fundamental in understanding the dynamics of celestial bodies. From planetary orbits to galaxy formation, the study of such systems has been pivotal in astrophysics. Simulating these systems computationally allows us to predict behaviours that are otherwise difficult to observe. The objective of this project is to create an efficient and versatile tool for simulating 2-body, 3-body, and n-body systems, along with visualizing the gravitational potential distribution through contour plots.

---

2. Theoretical Background

Newton's Laws of Gravitation

Newton's law of gravitation states that every point mass attracts every other point mass with a force directly proportional to the product of their masses and inversely proportional to the square of the distance between them. The force is given by:

$$\mathbf{F}_{12} = -G\frac{m_1 m_2}{r^2}\hat{\mathbf{r}}_{12}$$

Where G is the universal gravitational constant, and m_1 and m_2 the masses, and r is the distance between them.

Kepler's Laws

Kepler's First Law A planet orbits the Sun in an ellipse, with the Sun at one focus of the ellipse.

Kepler's Second Law A line connecting a planet to the Sun sweeps out equal areas in equal time intervals.
i.e.

$$\frac{dA}{dt} = \frac{l}{2\mu} \tag{2.1}$$

where l is the angular momentum and μ is the reduced mass
We will get back to this later when we will see Two Body Problem
Kepler's Third Law The Harmonic Law

$$P^2 \propto a^3 \tag{2.2}$$

where P is the time period and a is the semi-major axis of the ellipse

Gravitational Potential and Potential Energy

In classical mechanics, two or more masses always have a gravitational potential. Conservation of energy requires that this gravitational field energy is always negative, so that it is zero when the objects are infinitely far apart. The gravitational potential energy is the potential energy an object has because it is within a gravitational field.

$$U = -\frac{GMm}{R}$$

The total potential energy for an n-body system is the sum of pairwise interactions.

Centripetal Force

A **centripetal force** is a force that makes a body follow a curved path. The direction of the centripetal force is always orthogonal to the motion of the body and towards the fixed point of the instantaneous centre of curvature of the path. Isaac Newton described it as "a force by which bodies are drawn or impelled, or in any way tend, towards a point as to a centre". In Newtonian mechanics, gravity provides the centripetal force causing astronomical orbits. It has a magnitude:

$$F_c = ma_c = m\frac{v^2}{r}$$

---

3. Implementation

Programming Language and Libraries

The project is implemented in Python, utilizing the following libraries:

- **NumPy**: For efficient numerical computations.
- **Matplotlib**: For visualizing trajectories and contour plots.
- **Math**: For providing access to the mathematical functions.

Code Structure

1. **2-Body Simulation**: Models the interaction between two masses under gravitational force.
2. **3-Body Simulation**: Extends the logic to three interacting bodies, demonstrating chaotic dynamics.
3. **n-Body Simulation**: Generalizes the equations for any number of interacting bodies.
4. **Contour Plot**: Computes and visualizes the gravitational potential field in a specified region.

Algorithm

1. **Initialization**: The masses, positions and velocities of the bodies are initialised.
2. **Numerical Integration**: The equations of motion are solved using the Euler method as follows:

   Consider a differential equation of the form

   $$\ddot{x} = f(x, \dot{x}, t)$$

   Here we denote $a = \ddot{x}$ and $v = \dot{x}$. Suppose that the initial conditions are given. That is, (x_0, v_0, t_0). The task is to find $x(t)$ for $t_0 \leq t \leq T$.

   Initialize $x \leftarrow x_0$, $v \leftarrow v_0$ and $t \leftarrow t_0$. Choose a small dt appropriately (depending on the accuracy that you desire). While $t \leq T$,
   - $a \leftarrow f(x, v, t)$
   - $v \leftarrow v + a\,dt$
   - $x \leftarrow x + v\,dt$
   - $t \leftarrow t + dt$
3. **Visualization**: Animation is created (frame by frame) to understand the different trajectories of the bodies under different conditions. Also, the gravitational potential contours for a particular state of the given system can be plotted as well.

A Note on the Computational Libraries Used

In this project, I used **NumPy** and **PyTorch** to handle the calculations and make the simulations efficient and scalable.

NumPy

NumPy was my go-to library for most of the initial computations. It helped me:

1. Perform calculations on multiple bodies at once using **vectorized operations**, which saved a lot of time compared to writing loops.
2. Use **broadcasting** to easily handle arrays of different shapes, especially when calculating distances and forces between bodies.
3. High-performance arrays were critical for handling large-scale computations in n-body simulations.

PyTorch

For the more advanced parts of the project, I switched to PyTorch. It allowed me to:

1. Use **tensors**, which are like arrays but work even faster, especially with a GPU (graphics card).
2. Scale up the simulations to handle more bodies without slowing down too much.

NumPy and **PyTorch** were pivotal in implementing the computational aspects of gravitational system simulations. Combining these two libraries made it easier to handle complex calculations efficiently while keeping the code clean and flexible for future extensions.

2-Body Simulation

1. **Constants and Initialization**:
   - The universal gravitational constant (G) and time step (dt) are used.
   - Two celestial bodies are defined with specific masses, positions, and velocities.

```
N=2
G=1
dt=0.1
```

```
body1 = Body(1,-7.5,0,0,+math.sqrt(4)/15) #define body 1 with attributes mass and velocity
body2 = Body(1,+7.5,0,0,-math.sqrt(4)/15) #define body 2 with attributes mass and velocity
red_mass = reducedMass(body1.mass, body2.mass)
COM = getCOM()
```

2. **Class Definitions**:
   - A Body class represents a celestial body interacting with other bodies in space.
   - It has attributes such as mass, position (initialised as coordinates- x0 and y0), and velocity (initialised as velocity components- vx0 and vy0).
   - There are several object methods defined as well:
     1. updateAcc(): Updates the acceleration of the body using the force acting on it.
     2. updateVel(): Updates the velocity of the body using its acceleration.
     3. updatePos(): Updates the position of the body using its velocity.
     4. getKE(): Returns the kinetic energy of the body.

```
class Body:
  def __init__(self, m, x0, y0, vx0, vy0):
    self.mass = m
    self.pos = np.array([x0,y0])
    self.vel = np.array([vx0,vy0])
    self.acc = np.array([0,0])

  def updateAcc(self, force):
    self.acc = force / self.mass

  def updateVel(self):
    self.vel = self.vel + self.acc * dt

  def updatePos(self):
    self.pos = self.pos + self.vel * dt

  def getKE(self):
    return 0.5 * self.mass * (magnitude(self.vel))**2
```

   - Functions are also defined for some other general purposes such as calculating the gravitational forces between the bodies, calculating their reduced mass, calculating the magnitude of a vector (eg: speed can be determined from the velocity vector), positioning the centre of mass of the bodies and the potential energy of the system.

```
def calcForce(dist_vec):
  return -(G*body1.mass*body2.mass/(magnitude(dist_vec)**3))*dist_vec

def reducedMass(m1, m2):
  return m1*m2/(m1+m2)

def magnitude(vector):
  return math.sqrt((vector[0])**2 + (vector[1])**2)

def getCOM():
  return (body1.mass*body1.pos + body2.mass*body2.pos) / (body1.mass+body2.mass)

def getPE(r):
  return -G * body1.mass * body2.mass / r
```

3. **Numerical Integration**:

- The equations of motion are solved using a simple integration method: Euler method, given as:

Initialize $x \leftarrow x_0$, $v \leftarrow v_0$ and $t \leftarrow t_0$. Choose a small dt appropriately (depending on the accuracy that you desire). While $t \leq T$,

- $a \leftarrow f(x, v, t)$
- $v \leftarrow v + a\,dt$
- $x \leftarrow x + v\,dt$
- $t \leftarrow t + dt$

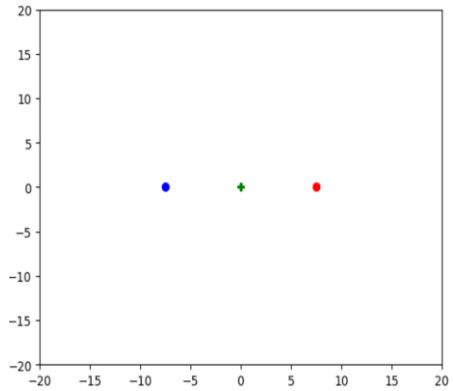
```
def updateAcc(self, force):
  self.acc = force / self.mass

def updateVel(self):
  self.vel = self.vel + self.acc * dt

def updatePos(self):
  self.pos = self.pos + self.vel * dt
```

4. **Visualization**:

- Trajectories of the two bodies are animated using matplotlib.animation. Firstly, a plot figure is created and the bodies are initially plotted for the initial animation frame.

```
# create a figure, axis and plot element
fig = plt.figure()
axis = plt.axes(xlim =(-20, 20), ylim =(-20, 20))
mass1, = axis.plot(body1.pos[0], body1.pos[1], 'bo')
mass2, = axis.plot(body2.pos[0], body2.pos[1], 'ro')
com, = axis.plot(COM[0], COM[1], 'gP')
# initialization function
def init():
  # creating an empty plot/frame
  mass1.set_data([body1.pos[0]], [body1.pos[1]])
  mass2.set_data([body2.pos[0]], [body2.pos[1]])
  return mass1, mass2
```

The initial frame hence looks as follows:



The animate() function is then defined to plot the graph frame by frame which then be compiled to make an animation video. Hence, the forces acting, acceleration, velocities and positions of the bodies are updated each frame.

```
#animate function
def animate(i):
    # t is a parameter
    t = i*dt

    dist_vec = body1.pos - body2.pos
    r = magnitude(dist_vec)
    f12 = calcForce(dist_vec)
    f21 = -f12

    body1.updateAcc(f12)
    body2.updateAcc(f21)

    body1.updateVel()
    body2.updateVel()

    body1.updatePos()
    body2.updatePos()
```

In addition to that, the potential energy, kinetic energies and the total energy are also calculated each frame and stored in a list. This would be used to plot an energy vs time plot later on.

```
    # set positions of the two masses
    mass1.set_data(body1.pos[0], body1.pos[1])
    mass2.set_data(body2.pos[0], body2.pos[1])
    COM = getCOM()
    com.set_data(COM[0], COM[1])

    #energy
    PE.append(getPE(r))
    KE1 = body1.getKE()
    KE2 = body2.getKE()
    totalEnergy=getPE(r)+KE1+KE2
    TE.append(totalEnergy)

    # return mass object
    return mass1, mass2
```

The animation is then saved as a ‘mp4’ file, with the name given by user. The length of the video varies depending on the values of frames and fps.

```
# setting a title for the plot
plt.title('2-Body Simulation')
# hiding the axis details
plt.axis('off')

# call the animator
anim = animation.FuncAnimation(fig, animate, init_func=init,
                               frames=2000, interval=10, blit=False)

# save the animation as mp4 video file
anim.save('twobodysim.mp4', writer = 'ffmpeg', fps = 100)

energyPlot()
plt.show()
```

The energyPlot() defined at the end is used to plot an energy vs time curve of the system of the bodies as move along other trajectories.

```
#energy plotter
def energyPlot():
  time = np.linspace(0, 20, len(PE))
  axis_energy = plt.axes()
  axis_energy.plot(time, PE, 'g-', label='Potential Energy')  # PE line
  axis_energy.plot(time, TE, 'r-', label='Total Energy')      # TE line
  axis_energy.set_xlim(0, 20)
  axis_energy.set_ylim(np.min(PE)-0.2, np.max(TE)+0.2)
  axis_energy.set_title("Energy vs Time")
  axis_energy.set_xlabel("Time (s)")
  axis_energy.set_ylabel("Energy (J)")
  axis_energy.legend()
  axis_energy.grid(True)
```

3-Body Simulation

1. **Constants and Initialization**:
   - Similar setup to the 2-body system, extended to include three interacting bodies.
   - Additionally, a list of bodies has been created to help in dealing with each body individually.

```
bodies = [body1, body2, body3]
```

2. **Class Definitions**:
   - The Body class and related functions are adapted for three-body interactions.
   - Pairwise distances and forces are calculated dynamically for all three bodies. This is done with the help of the following functions:

```
def findDistVector(body1, body2):
  return body1.pos - body2.pos

def calcForce(dist_vec):
  return -(G*body1.mass*body2.mass/(magnitude(dist_vec)**3))*dist_vec
```

```
def findForceOn(body):
  force = np.array([0,0])
  for b in bodies:
    if b != body:
      dist_vec = findDistVector(body, b)
      force = force + calcForce(dist_vec)
  return force
```

```
def getCOM(bodies):
  num, den = np.array([0,0]), 0
  for body in bodies:
    num = num + body.mass*body.pos
    den = den + body.mass
  return num/den
```

3. **Simulation Setup**:

- Initial conditions are chosen to illustrate motion with three bodies interacting gravitationally (generally chaotic).
- Unlike the two-body problem, the three-body problem has **no general closed-form solution**, meaning there is no equation that always solves it. Still, there are some stable systems possible. A famous example is the figure-8 orbit (attached in the Appendix).

4. **Visualization**:

- Animations are created for the trajectories of the three bodies, showcasing their complex and often unpredictable motion.
- In addition to this, provision has been made to plot the contour of the gravitation potential field (U). It helps in visualizing the distribution of the field over 2D space due the given system of bodies.

```
def plotContour():
  x_points = np.linspace(-2, 2, 50)
  y_points = np.linspace(-2, 2, 50)
  X, Y = np.meshgrid(x_points, y_points)

  U = -(G*body1.mass/np.sqrt((X-body1.pos[0])**2 + (Y-body1.pos[1])**2) + G*body2.mass/np.sqrt((X-body2.pos[0])**2 + (Y-body2.pos[1])**2) + G*body3

  contour = axis.contour(X, Y, U, levels=500)

  plt.colorbar(contour)
  plt.show()
```

- For a grid of points, the potential energy due to all masses is computed. A contour plot is generated using matplotlib, highlighting potential wells and stable/unstable regions.

N-Body Simulation

A similar setup is created for simulation gravitational system of N bodies- each now stored in an array of objects of class Body. Certain modifications were made in the code for calculation of force acting on a body and the total potential energy due to the system enabling us to take account of the effect of the N bodies, with N being a variable.

These bodies could either be initialised by the user or could be randomly plotted using scatter plot, and the motion thereafter could be observed.

---

4. Results

Simulations

1. **2-Body System**:
    - Analysis of several 2-body systems was done (see Appendix).
    - Additionally, the energy vs time plots were also studied for the same and valuable inferences were gained.
2. **3-Body System**:
    - Demonstrated the general chaotic behaviour of the 3-body system depending on initial conditions.
    - Some stable solutions to the 3-body problem were analysed through animations.
    - The Lagrange points were observed in special configuration of the reduced three body problem.
    - Potential contour plots made for the system shows potential wells corresponding to mass locations and helps in understanding system stability.
3. **N-Body System**:
    - Provides usage in simulating dynamic systems such as star clusters.
    - Helps in visualizing the formation of stable and unstable regions.

---

5. Challenges and Solutions

1. **Generalising the code**:
    - Problem: While it was easier to manage the code for a system of two bodies, there was eventually a need to generalise the code in order to simulate N-bodies simultaneously.
    - Solution: Implemented arrays and used them to store information for each body.
2. **Computational Limitations**:
    - Problem: There was a high computational cost for N-body simulations (where N>3).
    - Solution: Optimized calculations using vectorization in NumPy, and subsequently modifying the code to use the PyTorch library.

6. Applications and Future Work

Applications

1. Studying planetary motion and orbital dynamics.
2. Simulating the behaviour of star clusters and galaxies.
3. Could be used as a teaching tool for astrophysics and celestial mechanics.

Future Work

1. Simulating satellite motion, as well as lunar and inter-planetary transfer orbits.
2. Incorporating dynamic potential field and contour plotting in simulations.
3. Creating simulations in 3D space.

---

7. Learnings from this Project

This project deepened my understanding of gravitational dynamics and celestial interactions, such as the behaviour of multi-body systems. I gained hands-on experience with Python libraries like NumPy and PyTorch, using techniques like vectorization, broadcasting, and tensor operations to optimize simulations. I also enhanced my knowledge of numerical methods for solving differential equations and visualized data effectively through dynamic animations and contour plots.

Through this experience, I improved my problem-solving skills, tackling challenges like computational efficiency. Managing the project from implementation to documentation strengthened my ability to balance theoretical concepts with practical coding.

---

8. Conclusion

This project successfully models gravitational systems of varying complexities, providing valuable insights into their dynamics and potential energy distributions. The simulations and visualizations developed are robust tools for exploring celestial mechanics and pave the way for more advanced studies.

---

9. References

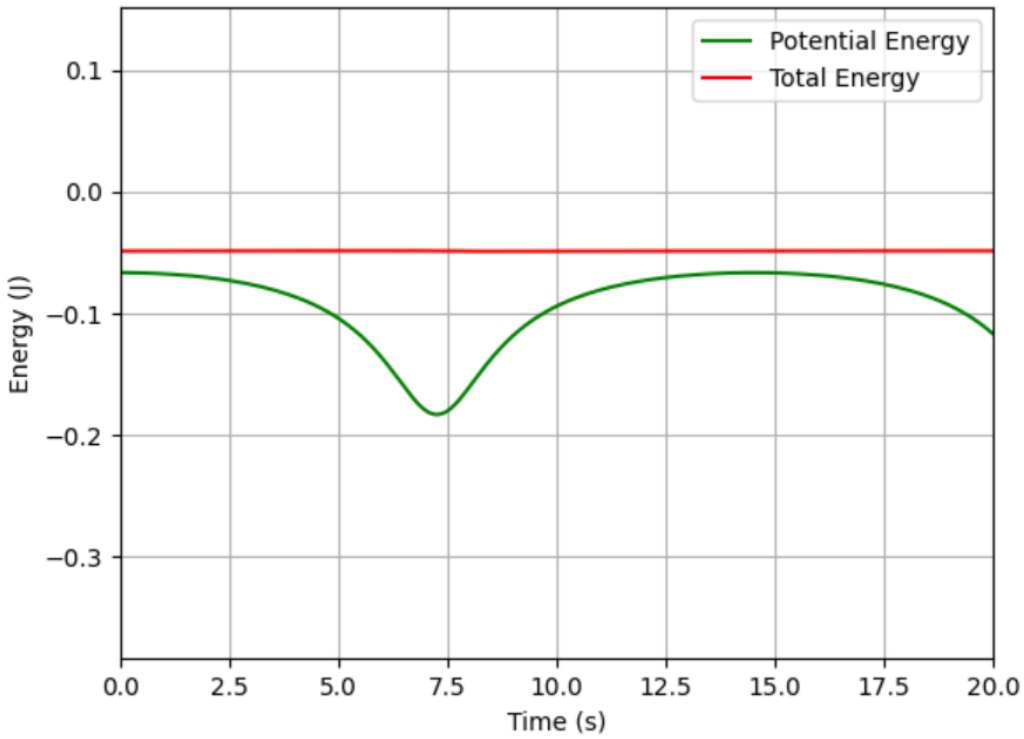
1. Krittika Tutorials on Theoretical Astronomy
2. Reference: https://docs.astropy.org/en/stable/timeseries/lombscargle.html
3. Reference: https://web.iucaa.in/~dipankar/CMA2020/MF_files/VanderPlas_1703.09824.pdf
4. GeeksForGeeks
5. Online Documentation: NumPy, Matplotlib, and PyTorch.

Appendix

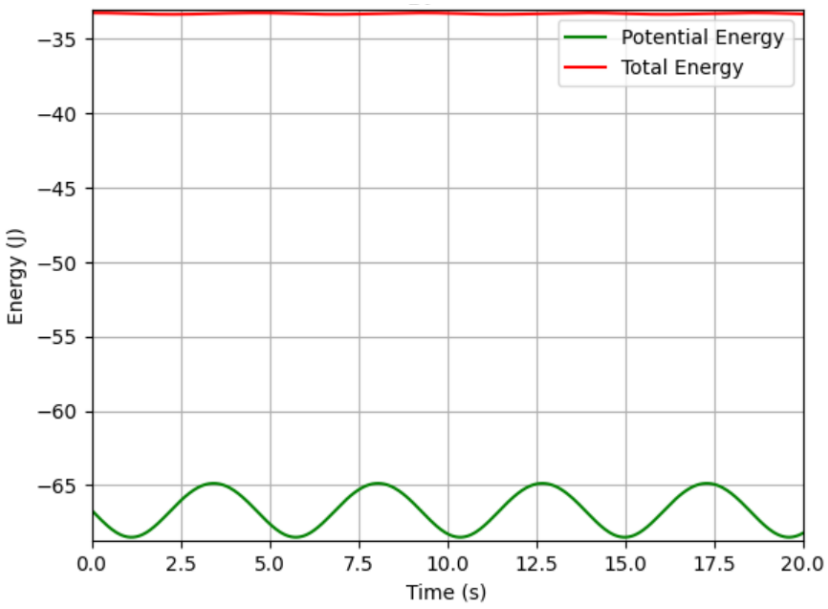
This section includes some of the interesting 2-body and 3-body models simulated by certain initialisation for some calculated orbits and motions. Alongside, there is also data regarding the energy vs time plots or contour plots that I have included as well. Enjoy having a look at them! The animations could be found here.

(Google Drive link: https://drive.google.com/drive/folders/1RlVID6Vcu1PTDSorRS4XeKkMf6hUbmqt?usp=sharing)

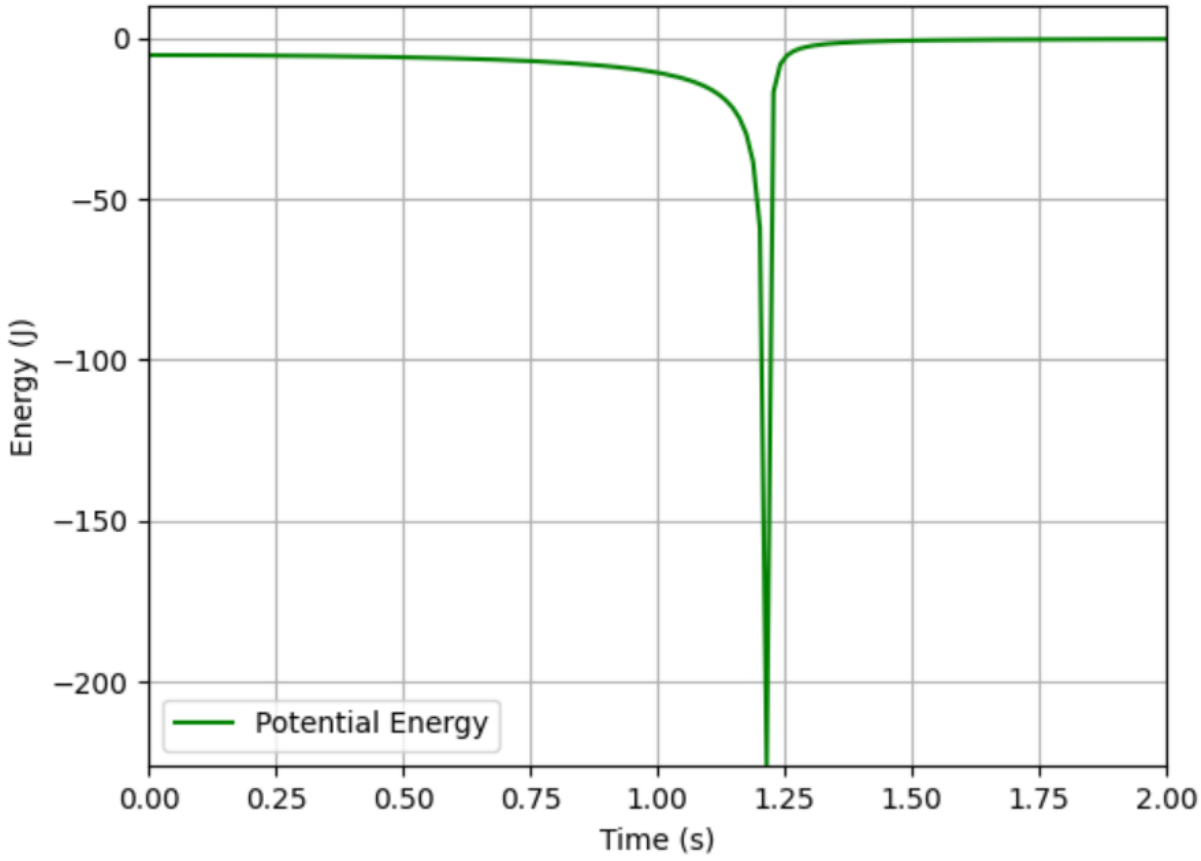
Following are some of the energy vs time plots and the contour plots that I came across:



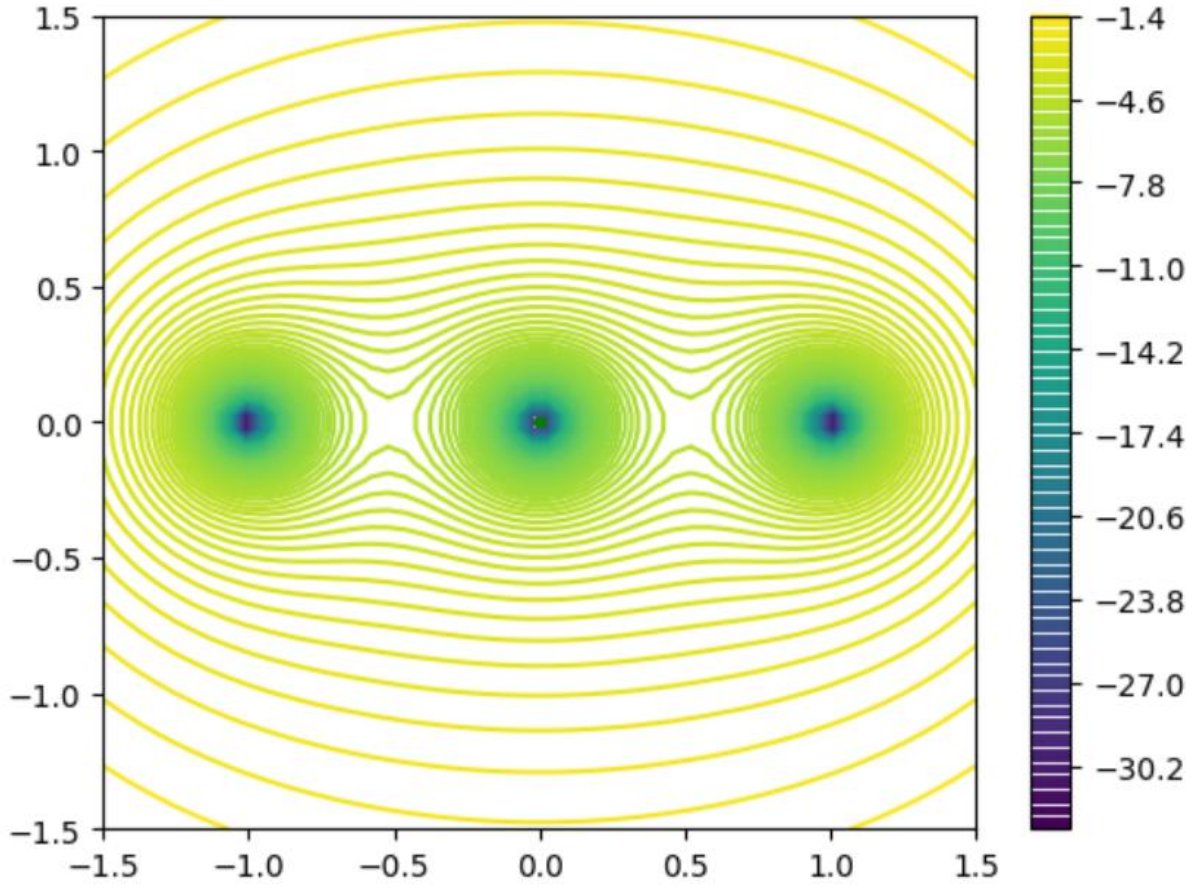
Periodic motion of a planet around Sun. The curve shows sharper dips in potential energy, suggesting moments when the bodies come closer together, i.e., **periapsis (closest approach)**, and then move farther apart, i.e., **apoapsis (farthest separation)**



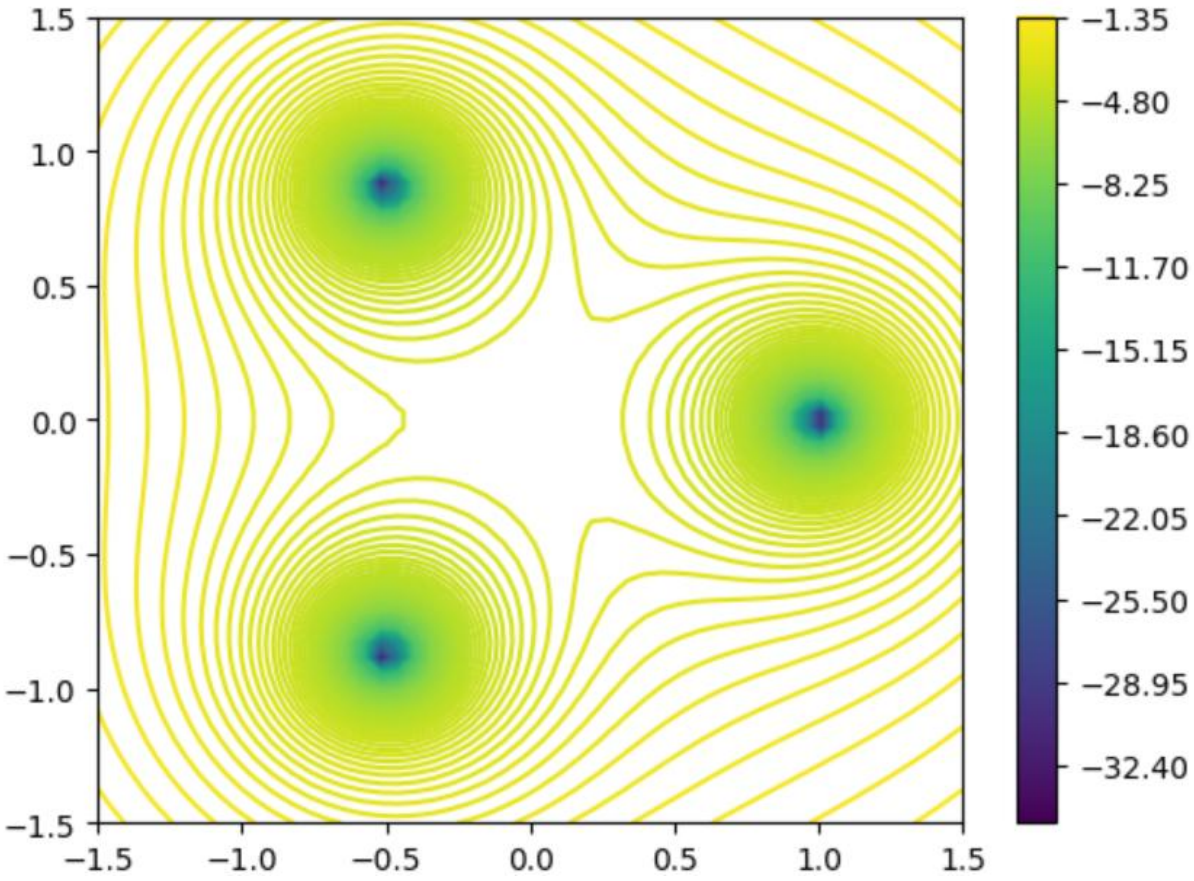
A planet revolving in an elliptic orbit around Sun



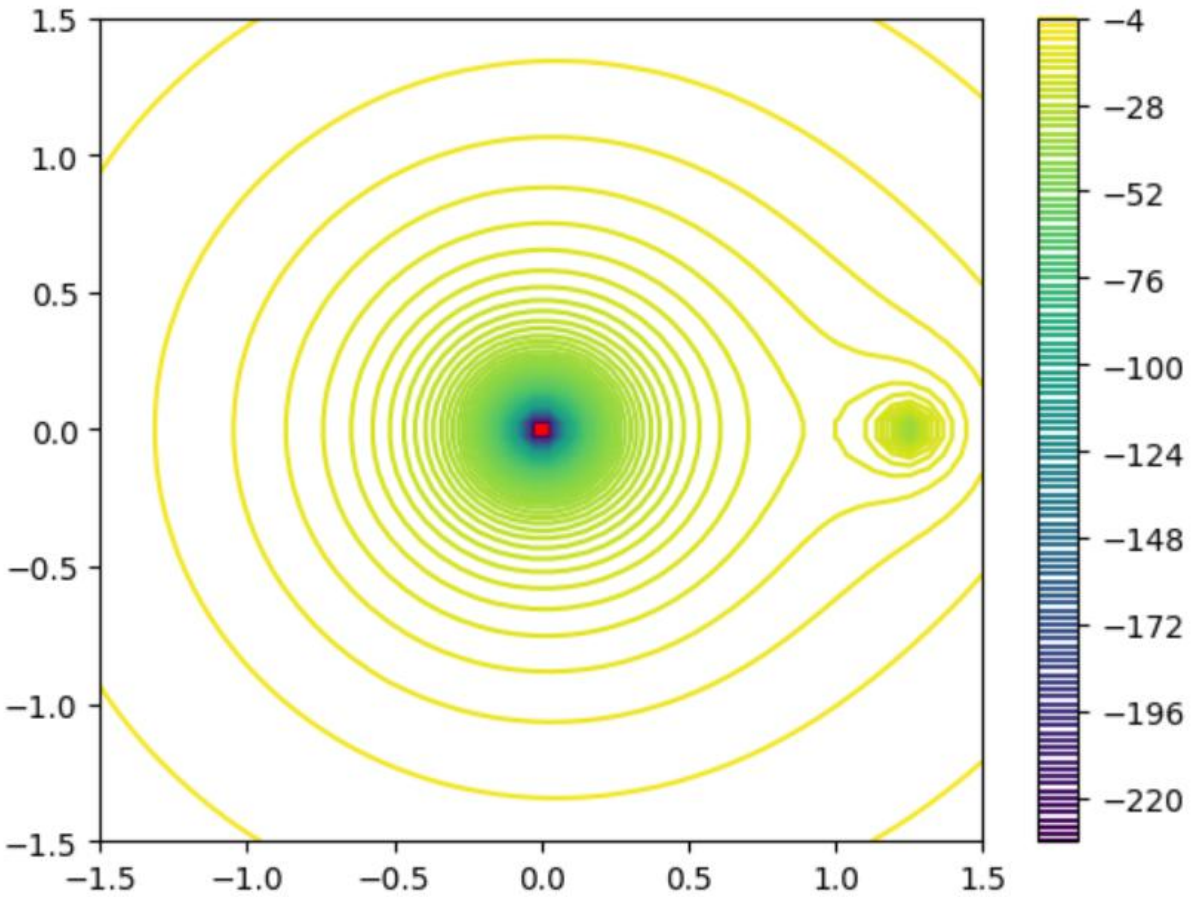
Two bodies moving freely under gravity (Note that sharp dip in potential energy when the bodies come in close proximity)



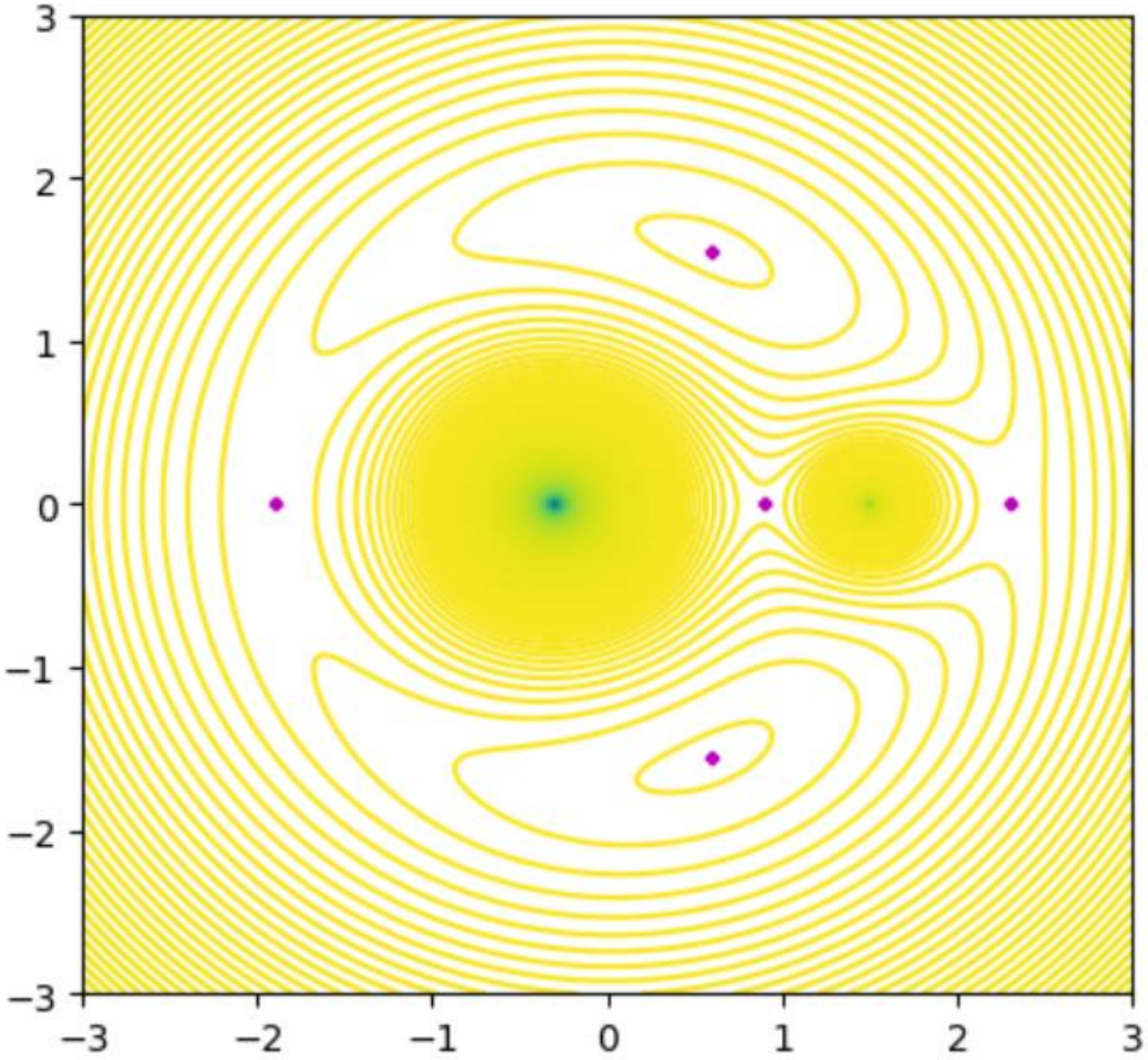
Contour field plot for a collinear three body system (Note the two equilibrium points in between)



Contour field plot for a three-body system positioned at the vertices of equilateral triangle



Contour field plot for planet star system



Contour field plot for restricted three body system

(Note the points marked in purple. These are known as the Lagrange Points i.e. locations in space where the gravitational forces of two large bodies, like the Sun and Earth, create a balance with the centripetal force required for a smaller object to move with them. This allows objects to orbit the Sun at the same speed as a planet, while remaining in the same position relative to both.)