PIC16B Group Project: Astrophysical Simulation of Gravitational Dynamics

Sparsh Vashist, James Freedman

Link to GitHub Repository (public): https://github.com/sparsh463/PIC16B_Project

I: Project Overview

For this project, we created and optimized a simulation and accompanying visualization of gravitational dynamics in Earth's solar system. Beginning with a script for computing the changing acceleration of bodies governed by the Newtonian laws of gravitation:

$$\vec{F}_{21} = -G\frac{m_1 m_2}{|\vec{r}_{21}|^2}\hat{r}_{21}$$

where

- $\vec{F}_{21}$ is the force applied on object 2 exerted by object 1,
- G is the gravitational constant,
- m_1 and m_2 are respectively the masses of objects 1 and 2,
- $|\vec{r}_{21}| = |\vec{r}_2 - \vec{r}_1|$ is the distance between objects 1 and 2, and
- $\hat{r}_{21} = \frac{\vec{r}_2 - \vec{r}_1}{|\vec{r}_2 - \vec{r}_1|}$ is the unit vector from object 1 to object 2.

We used astropy to create a simulation of the celestial bodies in the solar system, generating continuous position, force, and acceleration arrays. With a given simulation run time, the arrays were visualized using Plotly to generate an interactive animation of the orbits and changing parameters of the bodies throughout a run.

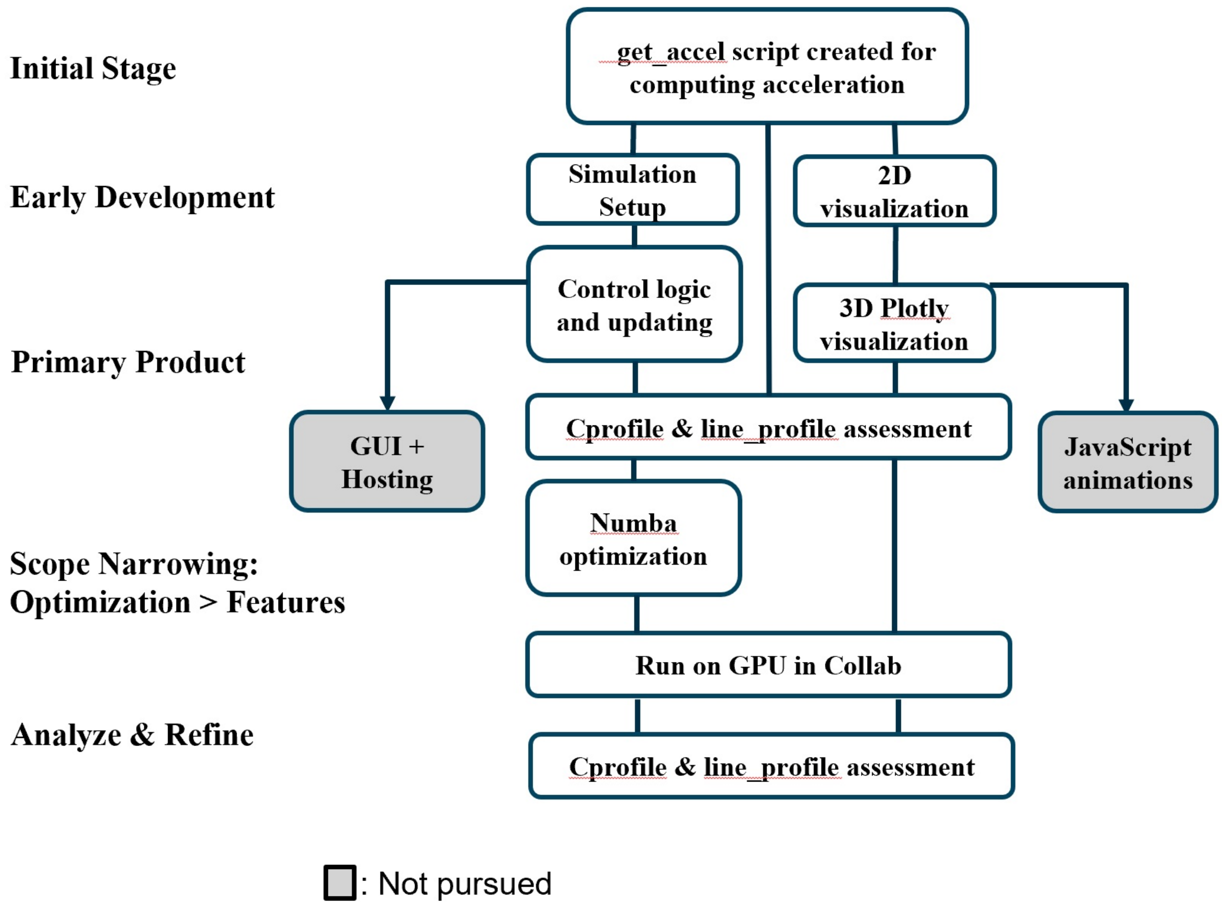
A major component of our project was focused on optimization techniques and profiling for the overall project. We ran cProfile and line-by-line assessments of the program, and were able to find bottlenecks, such as the matrix multiplication of the changing accelerations, and the visualization with Plotly. Using techniques from the course, we used numba to optimize the calculations within an explicit for-loop, and use just-in-time compilation to significantly reduce the run time for the simulation.

There are many possible future directions for our product - one line of work can be focused on improving accessibility, through hosting and adding GUI features to the simulation and visualization. Additionally, testing the program on more complex celestial systems and verifying with empirical orbit data could be promising to further refine the accuracy of the simulation.

II: Project Development Flowchart

Below is a flowchart describing the project development process. There were several areas where we could have focused our efforts on that we did not decide to pursue, and we attempt to record them here.

N-Body Simulation Project Flowchart



III: Technical Components

A: Simulation Architecture

The core function behind this N-body simulation is get_accel, which takes a matrix of space positions and a vector of masses, and returns the accelerations in a matrix, as seen below:

```
In [ ]: def get_accel(R, M):

    softening_parameter = 0.0001
    """
    Compute the gravitational accelerations of masses.
    R is an N x 3 matrix of space positions, units of [cm].
    M is a length N vector of masses, units of [g].
    Returns an N x 3 matrix of accelerations, units of [cm/s^2].
    """
    # get N x 1 matrices for position
    X = R[:, 0:1]
    Y = R[:, 1:2]
    Z = R[:, 2:3]

    # compute deltas (N x N) (all pairwise particle separations: r_j - r_i)
    DX = X.T - X
    DY = Y.T - Y
    DZ = Z.T - Z

    # compute 1/R^3 for each pair
    IR3 = (DX**2 + DY**2 + DZ**2 + softening_parameter**2)**(-1.5)

    # gravitational constant
    G = 6.67259e-8 # [cm^3/g/s^2]

    # accelerations
    AX = (DX*IR3)@M
    AY = (DY*IR3)@M
    AZ = (DZ*IR3)@M
    A  = G * np.array((AX, AY, AZ)).T
```

```
    return A
```

Using the above base function, a primary technical component during the project was to integrate it into a multi-step simulation, returning acceleration vectors that could in turn update position matrices for the bodies in the simulation. Much of the script involves the proper setup of the objects to be used in the simulation loop - improving the modularity of this setup phase is also an interesting direction for future effort.

The simulation phase involved calculating and updating the position (X), velocity (V), and acceleration for the bodies in the simulation, iterating through the updates powered by the base get_accel function. Then, we save the data for each step into a pandas dataframe.

In [ ]:

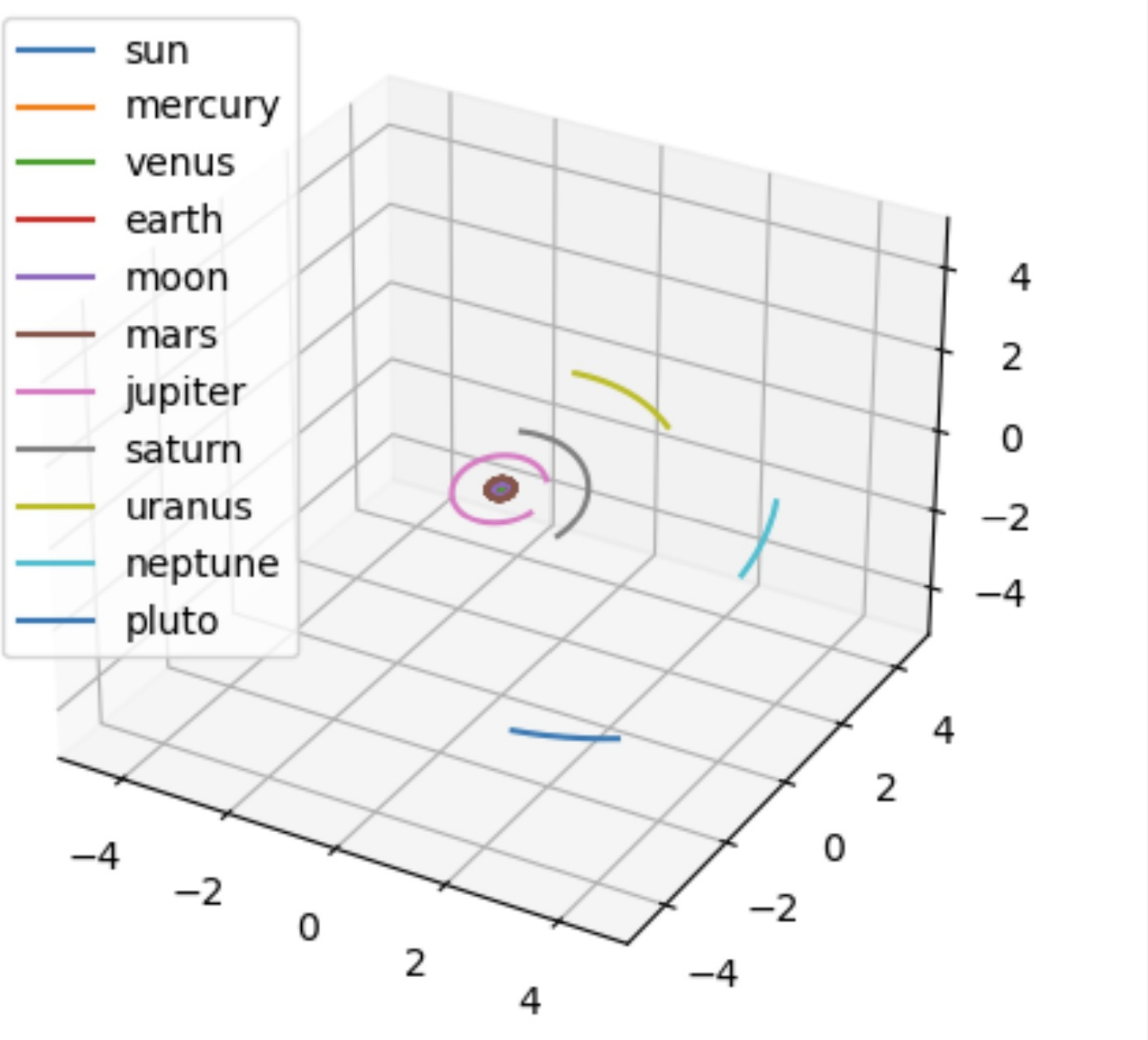
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from astropy import units as u
from astropy.time import Time
from astropy.coordinates import get_body_barycentric_posvel
import accel_python as accel ## Acceleration script
from mpl_toolkits.mplot3d import Axes3D
from tqdm import tqdm
import plotly.express as px

yr = 3.15576e7     # [s]     year

# Celestial Objects List
objects = ['sun', 'mercury', 'venus', 'earth', 'moon', 'mars', 'jupiter', 'saturn', 'uranus', 'neptune', 'pluto

# Initialising an Array of Positions, Velocities and Mass
X = np.ones((len(objects), 3))  # [cm]
V = np.ones((len(objects), 3))  # [cm]
M = np.array([1.989e33, 3.285e26, 4.867e27, 5.9742e27, 7.348e25, 6.4e26, 1.89e30, 5.68e29, 8.68e28, 1.024e29, 1

# Setting a start time for the simulation
# (astropy uses this time to determine pos, vel)
time = Time('2023-10-15 00:00')

# Getting barycentric position and velocities and filling the X,V arrays.
for i, body in enumerate(objects):
    """Populate X and V arrays with pos and vel values of each solar
    system object using astropy's get_body_.... function"""

    pos, vel = get_body_barycentric_posvel(body, time, ephemeris='jpl')
    X[i, :] = pos.xyz.to(u.cm).value  # [cm]
    V[i, :] = vel.xyz.to(u.cm/u.s).value

# Integration parameters
n_step = 1000
dt = 0.01 * yr  # [s]
acc = accel.get_accel(X, M)

## Creating a multi-dimensional array to store pos, vel after every n-step
X_values = np.ones((n_step, len(objects), 3))
V_values = np.ones((n_step, len(objects), 3))

# Create a list to store the data (to be later converted to a Pandas DF)
df_list = []

# Integrating this trial
for i in tqdm(range(n_step)):

    """Calcualting X and V for each n_step"""

    # First Step of Leapfrog, Updating Velocities
    V += acc * dt / 2.

    # Updating Positions
    X += V * dt

    # Updating Accelerations
    acc = accel.get_accel(X, M)

    # 2nd Step of Leapfrog, Update Velocities
    V += acc * dt / 2.

    ## Populating the X_values and V_values with X and V values of a particualr n-step
    X_values[i, :, :] = X
    V_values[i, :, :] = V

    # Append Data for this n_step to the Data List
    for j in range(len(objects)):
        df_list.append([objects[j], M[j], i, X_values[i, j, 0], X_values[i, j, 1], X_values[i, j, 2], V_values[i
```

```
# Creating a DataFrame from the data list
df = pd.DataFrame(df_list, columns=['object', 'mass', 'n_steps', 'x_pos', 'y_pos', 'z_pos', 'x_vel', 'y_vel', ':
df.to_csv("Simulation_Results.csv")
```

B: Plotly Visualization

A significant component of this project was creating and improving the visualization output for a simulation fun of N steps. The initial visualization showed the computed orbits but was a static display instead of animation of planetary motion.

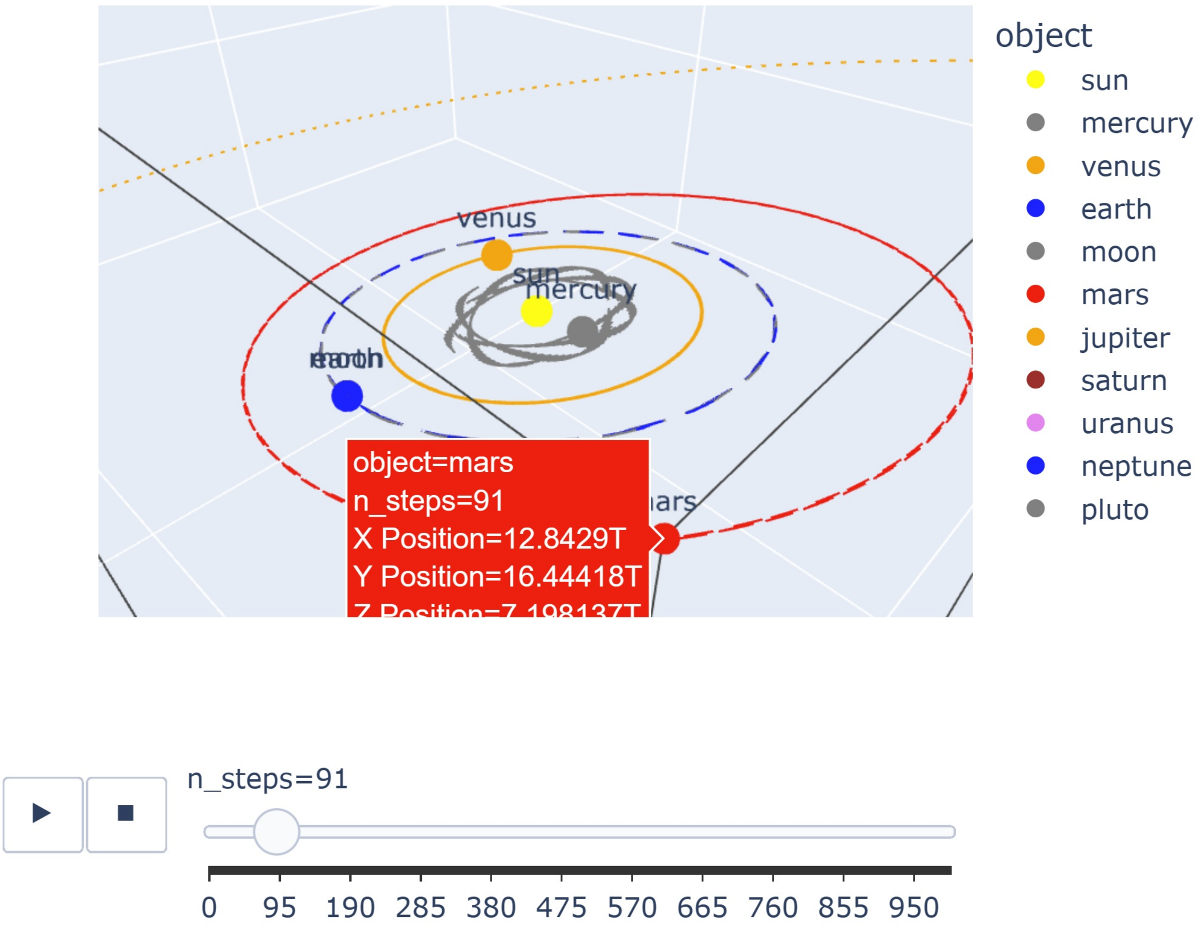
In [ ]:
```
"Initial Visualization with MatPlotLib"

fig = plt.figure()
ax = plt.axes(projection='3d')

ax.axes.set_xlim3d(left= -5e14, right= 5e14)
ax.axes.set_ylim3d(bottom= -5e14, top= 5e14)
ax.axes.set_zlim3d(bottom= -5e14, top= 5e14)

for i in range(len(objects)):
    ax.plot3D(X_values[:,i,0],X_values[:,i,1], X_values[:,i,2], label = objects[i])
plt.legend()
plt.show()
```



Using Plotly, we are able to display the motion of the system throughout the length of the simulation run with interactive displays of their position, mass, and velocity.

In [ ]:
```
# import df from CSV for the writeup
import plotly.express as px
df = pd.read_csv("Simulation_Results.csv")

"Making an Interactive 3D Plot with Plotly"

## Defining Color Map

color_map = {
```

```
    'sun': 'yellow',
    'mercury': 'gray',
    'venus': 'orange',
    'earth': 'blue',
    'moon': 'gray',
    'mars': 'red',
    'jupiter': 'orange',
    'saturn': 'brown',
    'uranus': 'violet',
    'neptune': 'blue',
    'pluto':'gray'
}

## Adding the Interactive 3D scatter Plot
fig = px.scatter_3d(df, x='x_pos', y='y_pos', z='z_pos', color='object', animation_frame='n_steps', text='objec
                    title="Solar System Simulation", labels={'x_pos': 'X Position', 'y_pos': 'Y Position', 'z_p
                    range_x=[-1e15, 1e15], range_y=[-1e15, 1e15], range_z=[-1e15, 1e15],
                    color_discrete_map=color_map)

# Adding lines to connect positions in each frame
for obj in df['object'].unique():
    obj_data = df[df['object'] == obj]
    line_trace = px.line_3d(obj_data, x='x_pos', y='y_pos', z='z_pos')

    # Set the line style (e.g., dash pattern) and line color based on the color_map
    line_trace.update_traces(line=dict(dash='dot', color=color_map[obj]))

    fig.add_traces(line_trace.data)

# Showing the interactive plot
fig.update_traces(textposition='top center')  # Adjust the position of the text labels
fig.show()
```



While the visualization is user friendly and impressive, as we will see more from optimization, it unfortunately is quite slow, and difficult to optimize. One significant observation made from the cProfile analyses of performance was that the internal Plotly methods consume a very large amount of the overall program run time. Due to this, a promising area for improvement would be exploring alternative visualization techniques, such as using JavaScript animations. However, as these were outside the scope of PIC16B, we chose to focus our efforts on optimizing the simulation computing using numba.

C: Optimization

A major goal of this project was to improve run time speed and performance throughout the code. To do this, we began by running tests on the code to asses performance - here are the tests we ran on our initial code using cProfile:

In [ ]:
```
if __name__ == '__main__':
    profiler = cProfile.Profile()
    profiler.enable()

    main()  # Call the main function of your script

    profiler.disable()
    profiler.dump_stats('output_python.pstats')  # Save stats to a file
```

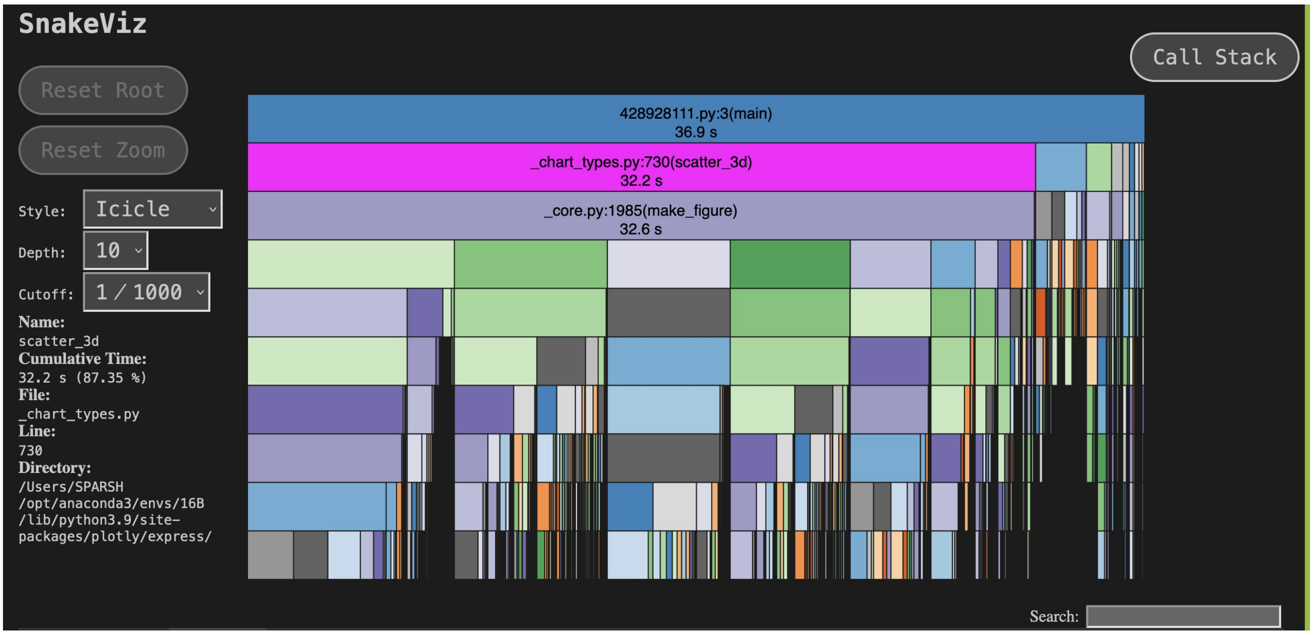
The output informed us of two main bottlenecks in our code. The looping get_accel function constantly computing updated accelerations for the simulation, and the visualization using Plotly. Without pursuing advanced visualization techniques, we dedicated our efforts to optimizing the get_accel function using numba.

We re-wrote get_accel using an explicit for loop for the matrix calculations, and then used the @njit decorator to optimize the code with just-in-time compilation. As we see after our analysis of the cProfile assessment of the numba-optimized code, we did indeed significantly improve performance through these steps:

In [ ]:
```
@njit(parallel=True, cache=True)
def get_accel(R, M):
    """
    Compute the gravitational accelerations of masses.
    R is an N x 3 matrix of space positions, units of [cm].
    M is a length N vector of masses, units of [g].
    Returns an N x 3 matrix of accelerations, units of [cm/s^2].

    """
    N = R.shape[0]
    A = np.zeros_like(R)
    G = 6.67259e-8  # [cm^3/g/s^2]
    eps = 1e-3  # Softening factor to prevent division by zero

    for i in range(N):
        for j in range(N):
            if i != j:
                dx = R[j, 0] - R[i, 0]
                dy = R[j, 1] - R[i, 1]
                dz = R[j, 2] - R[i, 2]
                inv_r3 = ((dx**2 + dy**2 + dz**2 + eps**2)**(-1.5))
                A[i, 0] += M[j] * dx * inv_r3
                A[i, 1] += M[j] * dy * inv_r3
                A[i, 2] += M[j] * dz * inv_r3

    A *= G
    return A
```

After we run the same CProfile diagnostics (see the git repo for outputs), we determined that the numba optimization resulted in a significant improvement in overall performance of the simulation by optimizing the interior computation function get_accel.

We were able to improve to 0.00373s for the simulation with Numba compared to 0.01569s for the simulation beforehand, a 426% improvement.



ncalls	tottime	percall	cumtime	percall	filename:lineno(function)
1001	0.00373	3.726e-06	0.004073	4.069e-06	accel_numba.py:4(get_accel)
2	6.458e-06	3.229e-06	7.499e-06	3.75e-06	_pylab_helpers.py:100(get_active)

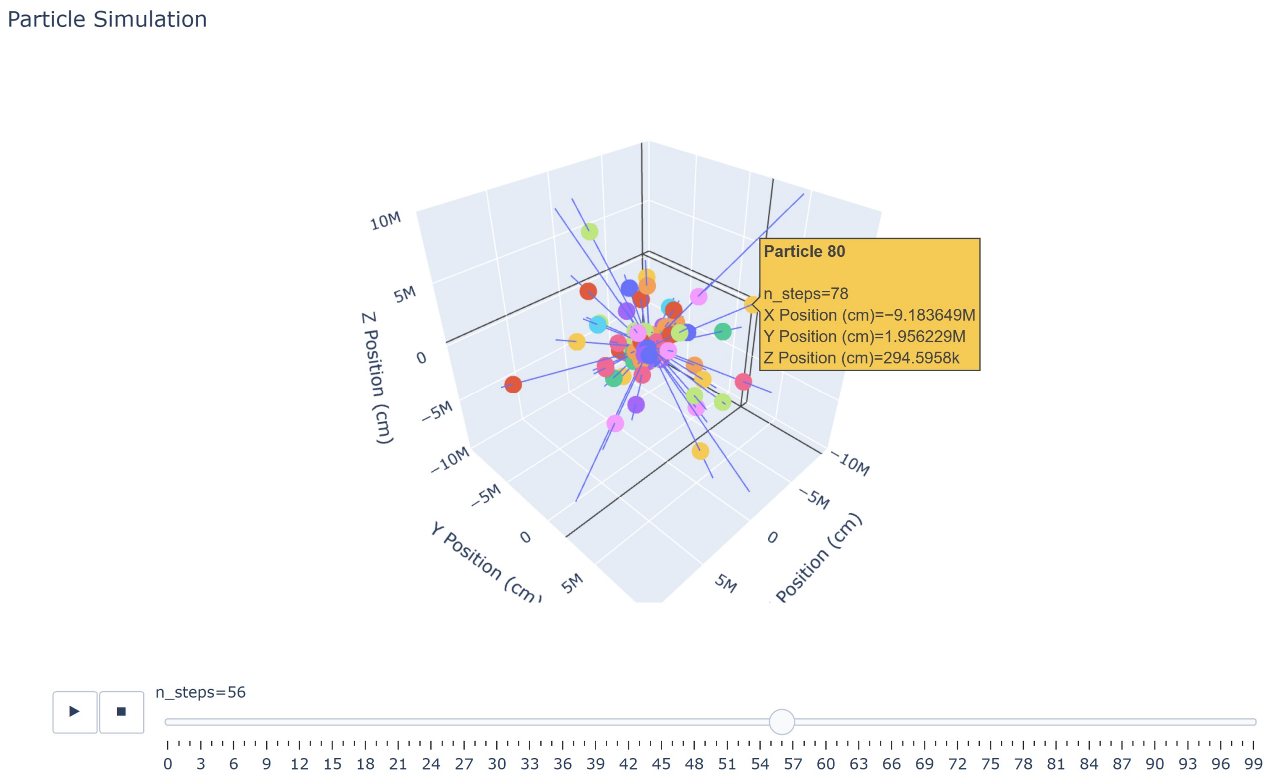
D: Extending to Many Bodies

After optimizing the get_accel function, a natural step with a more efficient inner function was to expand the simulation from a relatively simple solar system model to a chaotic, complex system. Here we initialize and visualize a system with 100 objects each starting from a stationary position, and the resulting gravitational effects. We are attaching a screenshot for this writeup but the code is available to generate the interactive visualization on the git repo linked at the top.

In [ ]:

```
def create_nbody_sim(N =10, mass=1e8, radius=1e5, n_step=1000,
                     dt=0.01, create_vis=True, accel_type ='numba'):
    import matplotlib.pyplot as plt
    import numpy as np
    import pandas as pd
    from numba import njit
    from mpl_toolkits.mplot3d import Axes3D
    from tqdm import tqdm
    import plotly.express as px

    if accel_type=='numba':
        import accel_numba as accel ## Acceleration script
    elif accel_type == 'python':
        import accel_python as accel


    yr = 3.15576e7
    N = N
    # Mass of each particle
    M_particle = np.ones(N)*mass # [g]

    # Setting up Initial Parameters
    radius = radius
    theta = np.random.uniform(0, 2 * np.pi, size=N)
    phi = np.random.uniform(0, np.pi, size=N)

    X = np.zeros((N, 3), dtype=np.float64)
    X[:, 0] = radius * np.sin(phi) * np.cos(theta)
    X[:, 1] = radius * np.sin(phi) * np.sin(theta)
    X[:, 2] = radius * np.cos(phi)

    V = np.zeros((N, 3), dtype=np.float64)  # Stationary particles, so initial velocities are zero

    #  Integration parameters
    n_step = n_step
    dt = dt*yr # [s]

    # Create a multi-dimensional array to store pos, vel, and potential energy after every n-step
    X_values = np.ones((n_step, N, 3))
    V_values = np.ones((n_step, N, 3))

    acc = accel.get_accel(X, M_particle)

    # Create a list to store the data (to be later converted to a Pandas DF)
    df_list = []

    # Integrating this trial
    for i in tqdm(range(n_step)):
        """Calculating X, V, and PE for each n_step"""

        # First Step of Leapfrog, Updating Velocities
        V += acc * dt / 2.

        # Updating Positions
        X += V * dt

        # Updating Accelerations and Potential Energy
        acc = accel.get_accel(X, M_particle)


        # 2nd Step of Leapfrog, Update Velocities
        V += acc * dt / 2.

        # Populating the X_values, V_values, PE_values, KE_values with X, V, PE, and KE values of a particular i
        X_values[i, :, :] = X
        V_values[i, :, :] = V
```

```
        # Append Data for this n_step to the Data List
        for j in range(N):
            df_list.append([f'Particle {j+1}', M_particle, i, X_values[i, j, 0], X_values[i, j, 1], X_values[i,
                          V_values[i, j, 0], V_values[i, j, 1], V_values[i, j, 2]])

    # Creating a DataFrame from the data list
    df = pd.DataFrame(df_list, columns=['particle', 'mass', 'n_steps', 'x_pos', 'y_pos', 'z_pos', 'x_vel', 'y_v

    if create_vis==True:

        # 3D Plot
        fig = plt.figure()
        ax = plt.axes(projection='3d')

        ax.axes.set_xlim3d(left=-100 * radius, right=100 * radius)
        ax.axes.set_ylim3d(bottom=-100 * radius, top=100 * radius)
        ax.axes.set_zlim3d(bottom=-100 * radius, top=100 * radius)

        for j in range(N):
            ax.plot3D(X_values[:, j, 0], X_values[:, j, 1], X_values[:, j, 2], label=f'Particle {j+1}')
        plt.legend()
        plt.title("Particle Simulation - 3D Plot")
        plt.xlabel("X Position (cm)")
        plt.ylabel("Y Position (cm)")
        ax.set_zlabel("Z Position (cm)")
        plt.show()

        # Adding the Interactive 3D scatter Plot
        fig = px.scatter_3d(df, x='x_pos', y='y_pos', z='z_pos', color='particle', animation_frame='n_steps',
                            hover_name='particle', hover_data={'particle': False},
                            title="Particle Simulation",
                            labels={'x_pos': 'X Position (cm)', 'y_pos': 'Y Position (cm)', 'z_pos': 'Z Positio
                            range_x=[-100 * radius, 100 * radius], range_y=[-100 * radius, 100 * radius], range_

        # Adding lines to connect positions in each frame
        for j in range(N):
            particle_data = df[df['particle'] == f'Particle {j+1}']
            line_trace = px.line_3d(particle_data, x='x_pos', y='y_pos', z='z_pos')

            fig.add_traces(line_trace.data)

        # Showing the interactive plot
        fig.update_traces(textposition='top center')  # Adjust the position of the text labels
        fig.update_layout(scene=dict(zaxis=dict(range=[-100 * radius, 100 * radius])))
        fig.show()

create_nbody_sim(N = 100, mass=1e8, radius=1e5, n_step=1000, dt=0.001, create_vis=True)
```



IV: Conclusion

This project was an interesting and challenging foray into the world of simulation and optimization. From the initial mathematical computing steps related to the acceleration functions, to the simulation architecture and visualization phases, and then the optimization and assesment of our code, we were able to develop a usable and interesting product for modeling gravitational dynamics.

There are many possible future directions that one could take with this project, including hosting for wider accessibility, as well as further pursuing an advanced visualization with JavaScript. We don't have to worry much about ethical considerations, other than ensuring that our abstractions of complex processes hold up relatively well compared to real world data.