# PHYS2300: Assignment 7 - The N-Body simulation of the solar system

### Objective

To revisit Newton's gravity in three dimensions with more objects, exercise your top-down design skills, gain more experience with VPython, and learn some key numerical methods used in scientific computing.

## Newton's gravity with multiple objects

In our very first python tutorial, you developed a program to compute the gravitational attraction between two objects using Newton's law of gravity. The magnitude of the force on object 1 due to object 2 is

$$F_{12} = \frac{Gm_1m_2}{R_{12}^2}$$

where $m_1$ and $m_2$ are the masses, and $R_{12}$ is the distance between the two objects. This gave us the magnitude of the force, but not the direction. We would like to build a program that models how this force changes as the objects move through space. In three dimensions, you can write the vector force as

$$\mathbf{F}_{12} = \frac{Gm_1m_2}{|\mathbf{R}_{12}|^3}\mathbf{R}_{12}$$

$$\mathbf{R}_{12} = \mathbf{r}_2 - \mathbf{r}_1$$

$$\mathbf{r}_2 - \mathbf{r}_1 = (x_2 - x_1)\hat{\mathbf{x}} + (y_2 - y_1)\hat{\mathbf{y}} + (z_2 - z_1)\mathbf{z}$$

$$|\mathbf{R}_{12}| = \sqrt{(x_2-x_1)^2+(y_2-y_1)^2+(z_2-z_1)^2}$$

where x, y, and z are the coordinates in the 3-D system.

## Time evolution of gravitational forces

Gravity provides an acceleration on each object, which changes the velocity. In turn, this change in velocity changes the position. You can write that, with vectors, as

$$\mathbf{a}_1 = \frac{Gm_j}{|\mathbf{R}_{12}|^3}\mathbf{R}_{12}$$

$$\Delta\mathbf{v}_1 = \mathbf{a}_1\Delta t$$

$$\Delta\mathbf{x}_1 = \mathbf{v}_1\Delta t$$

where $a_1$ is the acceleration, and $\Delta x_1$ and $\Delta v_1$ are the changes in position and velocity of object 1 and $t$ is the time step or the time that elapses between each calculation. Toevolve the system, add these changes to the previous position and velocity like so:

$$\mathbf{a}_1 = \frac{Gm_2}{|\mathbf{R}_{12}|^3}\mathbf{R}_{12}$$

$$\mathbf{v}_1(t+\Delta t) = \mathbf{v}_1(t) + \mathbf{a}_1\Delta t$$
$$\mathbf{x}_1(t+\Delta t) = \mathbf{x}_1(t) + \mathbf{v}_1\Delta t$$

This is repeated for the other body of the system. This process, known as the Euler-Cromer method, is the simplest method for computing the time evolution of the system. Provided our computer is fast enough (that is, we can use a small enough time step) this is a perfectly acceptable method. Later, we will employ a refinement of this known as the Leap Frog which will improve the performance of the code.

**2-Body to N-body**

Thanks to the principle of superposition - that the net force is the sum of all of the forces on a particle - we can extend this treatment to an arbitrarily large number of particles by adding together the acceleration from each particle. Thus, the acceleration on object i due to all other objects j = 1, 2, ..., N would look like:

$$a_i = \sum_{j \neq i}^{N} \frac{Gm_2}{\left|R_{ij}\right|^3} R_{ij}$$

$$\mathbf{v}_i(t+\Delta t) = \mathbf{v}_i(t) + \mathbf{a}_i\Delta t$$
$$\mathbf{x}_i(t+\Delta t) = \mathbf{x}_i(t) + \mathbf{v}_i\Delta t$$

The j not equal to i notation indicates that we loop over all of the particles, but ignore that contribution due to the particle we are accelerating.

The pseudocode for the computation might look something like this:

```
#Declare arrays to hold the masses, initial positions, and initial velocities
of the particles
# Determine the initial values (get from user)
# Determine the timestep, and how long the user wants the simulation to run
# For each timestep:
#    For each particle:
#        For all the other particles
#        * Compute the acceleration due to gravity
#        * Use the acceleration to compute the change in velocity
#        * Use the velocity to compute the change in position
#        * Add the changes in acceleration, velocity and position to the
particle's current
#          values
#        * Report
#End the program when simulation is complete.
```

**Task1:**

Write a program to simulate a general **3-D, 2-Body system**, that requests the masses, initial positions, and velocities from the user (or input file) and computes the trajectories (or orbits) of the particles for each timestep over a specified period of time, both provided by the user (or input file). Test it with the following data for the Earth and the Sun.

$$
\begin{aligned}
\mathbf{r}_{earth} &= 1.5 \times 10^{11}\ \hat{\mathbf{x}} + 0.0\ \hat{\mathbf{y}}\ m \\
\mathbf{r}_{sun} &= 0.0\ \hat{\mathbf{x}} + 0.0\ \hat{\mathbf{y}}\ m \\
\mathbf{v}_{earth} &= 0.0\ \hat{\mathbf{x}} + 3.0 \times 10^{4}\ \hat{\mathbf{y}}\ \frac{m}{s} \\
\mathbf{v}_{sun} &= 0.0\ \hat{\mathbf{x}} + 0.0\ \hat{\mathbf{y}}\ \frac{m}{s} \\
m_{earth} &= 5.97 \times 10^{24}\ kg \\
m_{sun} &= 1.99 \times 10^{30}\ kg \\
\Delta t &= 6.3 \times 10^{4}\ s \\
T_{total} &= 3.15 \times 10^{7}\ s
\end{aligned}
$$

But first, your program should parse an input argument using **argparse** module. Your program should take the optional parameter of a file input:

***--file data.cvs  # this is the file with the 9 planets information, see canvas for file info.***

To help with the visualization, set this up using spheres in Vpython and using the rate() method in your while loop to control the simulation speed. Note that you can use different colors (and even textures, including a map of Earth!) using the materials property. For details see the Vpython help.

The key to this is the central loop, which is a nested loop over the components. For example, let's say I have created two spheres and appended them to a list called objects. In python, the loop structure could look like this:

```
for i in objects:
    i.acceleration = vector(0,0,0)
    for j in objects:
      if i != j:
        dist = j.pos - i.pos
        i.acceleration = i.acceleration + G * j.mass * dist / mag(dist)**3
  for i in objects:
    i.velocity = i.velocity + i.acceleration*dt
    i.pos = i.pos + i.velocity * dt
```

Once you have the 2-Body code working, test you program with data from the solar system below. These are taking from the Jet Propulsion Laboratory calculations of solar system positions, computed to machine accuracy for the indicated epoch, and summarized below. Take careful note of the units, recalling that the AU (or Astronomical Unit) is the average distance between the Earth and the Sun.

The file is available as an ASCII csv data file on the canvas assignment site.

Note that in both the 2 body and 9 body cases, we are going to set the Sun's initial position and velocity to 0. While your N-body code will still provide physically reasonable results, it is not optimal since it will cause the Sun to drift over the length of the simulation (while also affecting the accuracy of the planetary positions). To correct for this, we need to convert the given heliocentric coordinates to center-of-mass coordinates once they are input into the code. Recall that the center of mass (and velocity) coordinates are related to any arbitrary coordinate system by

$$\begin{aligned} M\mathbf{X_{cm}} &= \sum_{i=1}^{N} m\mathbf{r_i} \\ M\mathbf{V_{cm}} &= \sum_{i=1}^{N} m\mathbf{v_i} \end{aligned}$$

In our case, the r's and v's are given relative to the Sun at r = (0,0,0) and v = (0,0,0). You can use the relationships above to compute the center of mass for the system and subtract that position and velocity from the Sun's values and from all the other planets' values. This should keep your system from drifting and will also allow you to track the motion of the Sun that is caused by the gravitational tug of the other planets.

**Task 2: Improving the code with the Leap Frog method**

The Euler-Cromer method you are using is suitable for running the simulation for short durations with a small time step. But what if you wanted to run for millions of years or add a large number of particles? We can improve our first order method by implementing a simple second order solution called the Leap Frog method. This method is particularly well suited to orbital situations or any computation where the force calculation is independent of the velocity.

The strategy is this:

Start the calculation by computing the acceleration as before

- On the first step of the loop, compute the velocity at one half time step to determine the value at this point
- In the remaining steps, start by updating the position (and the vectors that depend on it) using this new velocity estimate
- Then continue by computing the acceleration and updating the mid-point velocity by one full time step

Mathematically, the first velocity time step looks like

$$\begin{aligned} \mathbf{a_i} &= \sum_{j \neq i}^{N} \frac{Gm_j}{|\mathbf{R}_{ij}|^3}\mathbf{R}_{ij} \\ \mathbf{v_i}\left(t + \frac{\Delta t}{2}\right) &= \mathbf{v_i}(t) + \mathbf{a_i}\frac{\Delta t}{2}. \end{aligned}$$

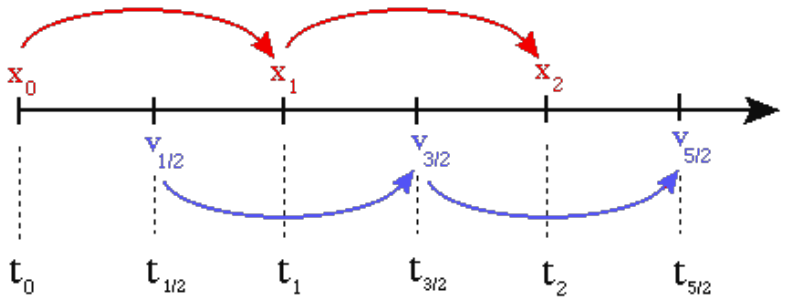
We then update the position one full time step using this midpoint velocity vi(t1/2)

$$\mathbf{x_i}(t + \Delta t) = \mathbf{x_i}(t) + \mathbf{v_i}(t_{1/2})\Delta t$$

and, using the new position, calculate the acceleration and update the velocities at the mid points:

$$\begin{aligned} \mathbf{a_i} &= \sum_{j \neq i}^{N} \frac{Gm_j}{|\mathbf{R}_{ij}|^3}\mathbf{R}_{ij} \\ \mathbf{v_i}(t_{1/2} + \Delta t) &= \mathbf{v_i}(t_{1/2}) + \mathbf{a_i}\Delta t. \end{aligned}$$

with $t_{1/2}$ denoting the velocities that are now offset by one half step. Effectively, we are using the position and velocities estimated at a step offset by one half time step, causing the calculation to "leap frog" to a solution, visualized below.




The advantage to the method is we increase the precision of the calculation from first order to second order (since we can now use a time effectively twice as large and achieve the same accuracy). However, we lose in two areas: one, we must "jump start" the algorithm by taking the first initializing half-step, and, two, we now have our positions and velocities offset by one-half time step. For our simulation that is not a problem, since we are planning on using the positions alone to update the particle motion on the screen, but if you required output of both the positions and velocities, you would need to interpolate all of your velocities back one half-step (which, from your previous work, you would know how to do!).

Below I have included some code you can use to implement the Leap Frog. The purpose of this is not to add programming complexity, but to give you some options when doing numerical calculations. For our N-Body simulator on a sufficiently fast machine, Euler-Cromer is probably enough, but implementing the Leap Frog gives you something to compare to. You could do so with some python code that looks like this:

```
for i in objects:
    i.acceleration = vector(0,0,0)
    for j in objects:
      if i != j:
          distance = j.pos - i.pos
          i.acceleration = i.acceleration + G * j.mass * distance /
mag(distance)**3
if firstStep == 0:
    for i in objects:
        i.velocity = i.velocity + i.acceleration*dt/2.0
```

```
        i.pos = i.pos + i.velocity*dt
    firstStep = 1
else:
    for i in objects:
        i.velocity = i.velocity + i.acceleration*dt
        i.pos = i.pos + i.velocity*dt
```

**To submit:**

At the end of the assignment, you will have a code that can model the **eight + one planets** in our solar system using **both** the **Euler-Cromer** and **Leap Frog** methods. Hand in this code along with clear documentation on how to run it and input the initial conditions. Also, answer the following questions:

- For fun, what happens to the Earth's orbit in the 2-body case when you increase the initial velocity to 35 km/s?
- What is the behavior of your code if you alter the time step? That is, what happens when the time step is longer than necessary? What happens if it is shorter than necessary?
- 25 Percent Extra Credit: model your solar system using both the Euler-Cromer and Leapfrog methods for the full 10 million years (or as long as you can to determine the long term behavior of your code). Describe the differences between these two methods in terms of computation time and accuracy in the conserved quantities. Note that you will have to ***"turn off" the visualization*** for this part of the assignment since that will dominate the computation time.
- (yet another) 25 Percent Extra Credit: Add an asteroid belt to your solar system using massless test particles (that is, they interact with the larger bodies but not each other). Randomly distribute their initial positions and velocities between Mars and Jupiter.