Python_A4_An_OO_approach_to_planetary_dynamics

May 10, 2022

1 SCI1022 Python Assignment 4: Object-oriented planetary dynamics

In this assignment **you have to develop object-oriented Python code** capable of simulating the motion of planets around our Sun. The simulator that you have to write involves most of the concepts that we have been working with along the unit, including, among others: programming problem-solving skills, development planning and software organization (i.e., how to encapsulate subtasks into classes and functions), data structure selection, Numpy arrays for 2D motion simulation, input from pure text files, data visualization, and object-oriented programming.

2 Main goal of the assignment

The main goal of the assignment is to confirm that the solar system will remain stable over the next 100 years. The solar system remains stable as far as all planets stay without any of the planets colliding with each other and/or not ejected from the system. Besides, we will also observe what is the effect that a (significant) variation of the mass of one of the planets causes to the stability of the system.

3 Time discretization of the solar system

This assignment builds on top of the lessons learned in the workshop entitled **2D Motion under uniform and central gravitational field**. (*Recall that the solutions of this workshop tasks are available at the Moodle page of the unit*).

In particular, the motion of the planets will be driven by the laws of Newtonian mechanics, and we will use the same approach towards time discretization of the governing equations. Thus, the simulation will be driven by a time evolution loop which, starting from a **given** initial configuration (see section below) of the solar system (i.e., initial position and velocities of the bodies in the solar system), will evolve it along time. At each time step, the time evolution loop will: 1. Update the displacement of all bodies using velocities computed in previous time step. 2. Compute the acceleration of all bodies using the displacements resulting from step 1. 3. Update the velocity of all bodies using accelerations computed in step 2.

We note that 1. has to the be coded in a different *inner* loop than 2.+ 3.

We will use `Matplotlib` in order to create an animated plot with the evolving orbits of all bodies. For those interested, click here for a revealing article on how to create animations with `Matplotlib`. In any case, you will find below code to help you in this regard.

4 A warning on the accuracy of time discretization

Although we did not mention it so far explicitly, in our simulator, we are using the so-called Euler's method to approximate the solution of a system of time-dependent Ordinary Differential Equation (ODEs). This sort of equations are ubiquitous in numerical simulations of scientific problems. Although system of ODEs are out of the scope of this unit, chances are very high that they will be introduced to you soon in other units of your degree (if they were not yet).

In order be numerically stable (i.e., to provide meaningful results), Euler's method requires a "small enough" value for the time step Δt. **For a 100 year long simulation of the solar system, $N = 10000$ time steps is a sensible choice.** Even though, you will observe in the long run some defects in the orbits generated by the simulator. These defects are caused by the error in the time discretization.

5 Why 2D and not 3D Motion? *Ecliptic plane*

For simplicity, the simulator that you have to write is though to neglect the motion of the planets along the z-axis. In other words, it is a 2D motion simulator. It turns out that most objects in our solar system roughly orbit in the so-called **ecliptic plane**. The ecliptic plane is the plane in which the Earth's orbit around the Sun is contained. Among the objects which roughly orbit within this plane we have all planets of the solar system (except Pluto). The ecliptic plane thus let us to neglect the motion along the z-axis of the planets in our simulation, and to plot their positions in this two-dimensional plane.

6 Calculating accelerations of bodies in the solar system

In our solar system, the planets are moving at a continuously varying velocity. This continuous variation of velocity is caused by the acceleration resulting from the forces that the bodies (e.g., the planets and sun) exert on each other.

Let us denote by N the number of bodies in our solar system. Let i, with $i = 1, \ldots, N$, be a unique identifier that we assign to each body. We denote as m_i the mass in kilograms of the body with identifier i, and $\vec{x_i}$ as the position vector of the body with respect to the origin of coordinates. We assume the sun to be located at the origin of coordinates. The displacement vector of body i with respect to body j (i.e., an arrow from the location of body i to the location of body j) is denoted as $\vec{x_{ij}}$. This vector is defined as $\vec{x_{ij}} = \vec{x_j} - \vec{x_i}$.

With all this notation, we can readily define the net force acting on a body with identifier i, which we denote as $\vec{F_i}$:

$$\vec{F_i} = \sum_{j \neq i} \frac{G m_i m_j}{||\vec{x_{ij}}||^3} \vec{x_{ij}},$$

with $||\vec{x_{ij}}||$ being the magnitude of $\vec{x_{ij}}$, and $G = 6.67 \cdot 10^{-11}\,\mathrm{kg}^{-1}\,\mathrm{m}^3\,\mathrm{s}^{-2}$ is the gravitational constant.

If we apply Newton's second law of movement (i.e., force is equivalent to the product of mass and acceleration), the acceleration $\vec{a_i}$ that $\vec{F_i}$ causes on body i is simply given by:

$$\vec{a_i} = \frac{\vec{F_i}}{m_i} = \sum_{j \neq i} \frac{G m_j}{||\vec{x_{ij}}||^3} \vec{x_{ij}}.$$

We note that all vector quantities (i.e., displacements, force, acceleration, etc.) actually depend on the time elapsed since the start of the simulation as the independent variable, t. Thus, e.g., $\vec{x_i}$ is actually $\vec{x_i}(t)$, $\vec{x_{ij}}$ is $\vec{x_{ij}}(t)$, and so on.

7 Initial configuration of the solar system

The initial configuration of the solar system is provided in a pure text file named `solar_system_initial_configuration.tsv`, which should be downloaded along with this Jupyter notebook from the Moodle page. This file contains real data corresponding to the configuration of our solar system on 1 May 2020. For each object in our solar system, the file reports its mass (in kilograms), its displacement (in meters) and velocity vectors (in meters/second). For each of these two vectors, the file provides its orthogonal projection into the X- and Y-axes, i.e., its components. The origin of coordinates is located at the sun. At this point we recommend you to open the file in a text editor (e.g., NotePad in Windows or TextEdit in MacOSX) so that you can familiarize with the file format.

8 Development plan, code organization, and marking

In this assignment, **it is your responsibility to decide how to split the simulator into subtasks, and encapsulate them into classes and functions.** You have considerable latitude in writing your code. There is no single correct solution. Any code that achieves the goal, and takes an object-oriented approach, will achieve high marks. In particular, to achieve full marks, your solution should: * be physically and computationally correct within the approximations given, * be logical and clearly set out, * be based on methods and functions which are concise (**they must do only one thing!**). * contain appropriate comments and **docstrings**, * be reasonably efficient.

As a hint, in the solution that the instructors devised, there were two classes (anyway, you are free to organize the software as you think it better fulfills requirements listed above):

- A class representing the objects of the solar system, called `Body`. We decided not to call it `Planet` as neither the Sun nor Pluto are actually planets. This class has attributes to store the name of the body, its mass, its current displacement and velocity, and provides methods to compute its acceleration (given all bodies of the system), and update the current displacement and velocity. The design of the interface of the latter methods is a key part of the assignment. Everything is clarified when you take into account which service the class in the next bullet points requires from Body.

- A class that drives the main simulation, called SolarSystemSimulator. This class holds a dictionary of solar system bodies indexed by name, stored in the self.bodies attribute, and provides methods to compute and internally record (attribute self.xi_history) the trajectory of all bodies, and generate an animation of the solar system motion. This latter method is clearly the hardest, so that we decided to provide code to help you out in this regard. An skeleton of SolarSystemSimulator is provided below. Do not use the visualize_animation method literally, try to understand the big picture, you might need to adapt it if necessary to your particular design.

```
# Sketch of `SolarSystemSimulator`
from IPython.display import HTML
from matplotlib import pyplot as plot
from matplotlib.animation import FuncAnimation
import matplotlib
%matplotlib inline

class SolarSystemSimulator:
    # ... Other method definition statements

    def visualize_animation(self,planet_names,num_steps_to_plot):
        fig = plot.figure()
        xmin,xmax,ymin,ymax=self.compute_xmin_xmax_ymin_ymax(planet_names)
        ax = plot.axes(xlim=(xmin, 1.2*xmax), ylim=(ymin, 1.2*ymax))

        lines=[]
        for name in planet_names:
          line, = ax.plot([], [], lw=2, label=self.bodies[name].name)
          line.set_data([], [])
          lines.append(line)

        def animate(i):
          step_increment=int(self.N/num_steps_to_plot)
          for j,name in enumerate(planet_names):
            x = self.xi_history[name][0:i*step_increment,0]
            y = self.xi_history[name][0:i*step_increment,1]
            lines[j].set_data(x, y)
          return lines

        fig.legend()
        plot.grid()
        matplotlib.rcParams['animation.embed_limit'] = 1024
        anim = FuncAnimation(fig, animate, frames=num_steps_to_plot, interval=20, blit=True)
        return HTML(anim.to_jshtml())
```

On the following code snippet illustrates the steps to generate an animation of the inner planets in the solar system during the next 100 years, using 5000 time steps, with only 200 of these steps being actually being plotted on the animation:

```
inner_planets=["Earth","Venus","Mercury","Mars"]
outer_planets_except_pluto=["Jupiter","Saturn","Uranus","Neptune"]
pluto=["Pluto"]
simulator=SolarSystemSimulator("solar_system_initial_configuration.tsv", 31536000.0*100, 10000)
simulator.compute_trajectories()
simulator.visualize_animation(inner_planets+outer_planets_except_pluto+pluto, 200)
```

8.1 Assignment tasks

Apart from the code itself, we expect the following from your work: 1. A text cell where you describe and justify in your own words the approach that you followed in order to organize your program. 2. A first set of animations in separate code cells of **100 years long simulations** of: (1) the inner planets motion; (2) the outer planets motion except Pluto; (3) Pluto and the Sun; (4) the whole solar system. 3. A second set of four animations like the ones in step 2., where the mass of Jupiter is multiplied by 10. 4. A third set of four animations like the ones in step 2., where the mass of Jupiter is multiplied by 100. 5. A description in your own words in the appropriate text cells of which is the effect of step 3. and step 4. in the stability of the system.

Note that you can create as many instances of `SolarSystemSimulator` as you like, even with different text files. In other words, using a text editor you can create new text files with the mass of Jupiter updated, and read the new files into different `SolarSystemSimulator` instances.

8.2 Expected results

In order to help you out, along with the workshop Jupyter notebook, we provide three videos with animations of the whole solar system generated by the simulator devised by the instructors. This is what you should expect from your simulation if it is physically and computationally correct.