Lab assignment #1: Python basics

Due Friday, September 16 2022, 5 pm

General Advice

- **Work with a partner!**
- The goals of this lab are
  - to solve some physics and math problems numerically;
  - to review basic programming concepts (such as loops, plotting, etc.) using Python;
  - to introduce the practice of writing "pseudocode" (or what we call pseudocode in this class); and
  - to learn how to time your code and test its performance.
- All our lab formats are similar in this course. We start by covering the computational and the physics background we need to do the lab.
- After reading the background below, you might want to review some of the computational physics material from PHY224 and/or PHY254 and look over some of the review material in the text. In particular, ensure that you take the time this week to learn the material in Chapters 2 & 3 as they will be expected knowledge for all the future labs. Useful review material includes:
  - Assigning variables: Sections 2.1, 2.2.1&2
  - Mathematical operations: Section 2.2.4
  - Loops: Sections 2.3 and 2.5
  - Lists & Arrays: Section 2.4
  - User defined functions: Section 2.6
  - Making basic graphs: Section 3.1

- Specific instructions regarding what to hand out are written for each question in **bold red**.

  Not all questions require a submission: some are only here to help you. When we do though, we are looking for "C^3" solutions, i.e., solutions that are **C**omplete, **C**lear and **C**oncise.

- An example of **C**larity: make sure to label all of your plot axes and include legends if you have more than one curve on a plot. Use fonts that are large enough. For example, when integrated into your report, the font size on your plots should visually be the same, or similar, as the font size of the text in your report.

- Whether or not you are asked to hand in pseudocode, you **need** to strategize and pseudocode **before** you start coding. Writing code should be your last step, not your first step.

- Test your code as you go, **not** when it is finished. The easiest way to test code is with `print()` statements. Print out values that you set or calculate to make sure they are what you think they are.

- Practice modularity. It is the concept of breaking up your code into pieces that as independent as possible form each other. That way, if anything goes wrong, you can test each piece independently. One way to practice modularity is to define external functions for repetitive tasks. An external function is a piece of code that looks like this:

  ```python
  def MultiplyAngleByTwo(argument):
      """A header that  explains the function
      INPUT:
      argument [float] is the angle in rad
      OUTPUT:
      res [float] is twice the argument"""
      res = 2.*argument
      return res
  ```

  Place them in a separate file called e.g. `MyFunctions.py`, and call and use them in your answer files with:

  ```python
  import MyFunctions as myf  # make sure file is in same folder
  DescriptiveVariableName = 4.
  myCalculationResult = myf.MultiplyAngleByTwo(DescriptiveVariableName)
  ```

Computational background

Numerical integration review. For a general first order system,

$$\frac{d\vec{x}}{dt} = \vec{F}(\vec{x}). \tag{1}$$

The simplest way to numerically integrate the system is by approximating the derivative as:

$$\frac{d\vec{x}}{dt} \approx \frac{\Delta\vec{x}}{\Delta t} = \frac{\vec{x}_{i+1} - \vec{x}_i}{\Delta t}. \tag{2}$$

where the subscript i on $\vec{x}_i$ refers to the time step. We can then rearrange eqn. (1) to read

$$\vec{x}_{i+1} = \vec{x}_i + \vec{F}(\vec{x}_i)\Delta t. \tag{3}$$

We can start with an initial $\vec{x}$, pick a Δt and implement equation (3) in a loop to calculate the new value of $\vec{x}$ on each iteration. This is called the "Euler" method, which you are expected to know from e.g. PHY224 or PHY254.

It turns out that the Euler method is unstable. The "Euler-Cromer" method, which updates the value of the derivative first, and then uses this value to update $\vec{x}$, is a small tweak that often leads to a stable result. For example, when you update the position variables x and y, use the newly updated velocities instead of the old velocities. So these lines should look like:

$$x_{i+1} = x_i + v_{x,i+1}\Delta t, \tag{4a}$$
$$y_{i+1} = y_i + v_{y,i+1}\Delta t. \tag{4b}$$

The updates for the velocities should remain as they are.

How to time the performance of your code. In Q3 we will explore how to time numerical calculations using your computer's built in stopwatch. I am going to present a crude way to do it, and there are better methods to profile code. Nonetheless, it will be enough for our purposes. Use `time.time()` as in the following example:

```
# import the "time" function from the "time" module
from time import time

start = time()  # save start time
```

```
# run your calculation
# ...

end = time()  # save end time
diff = end-start  # elapsed time (in seconds)
```

The `time()` function will not always return reliable results so it is worth rerunning tests a couple of times to check for consistency.

Physics background

Newtonian orbits. The Newtonian gravitational force keeping a planet in orbit can be approximated as

$$\vec{F}_g = -\frac{GM_S M_p}{r^2}\hat{r} = -\frac{GM_S M_p}{r^3}\left(x\hat{x} + y\hat{y}\right), \tag{5}$$

where M_S is the mass of the Sun, M_p is the mass of the planet, r is the distance between them, and $\hat{x}, \hat{y}$ the unit vectors of the Cartesian coordinate system. Using Newton's Second law ($\vec{F} = m\vec{a}$) and numerical integration, we can solve for the velocity and position of a planet in orbit as a function of time. *(Note: we have assumed the planet is much less massive than the Sun and hence that the Sun stays fixed at the centre of mass of the system).*

Using eqn. (5) and Newton's law, you can convince yourself that the equations governing the motion of the planet can be written as a set of first order equations, i.e., of the form of eqn. (1), in Cartesian coordinates as

$$\begin{aligned}\frac{dv_x}{dt} &= -\frac{GM_S x}{r^3},\\ \frac{dv_y}{dt} &= -\frac{GM_S y}{r^3},\\ \frac{dx}{dt} &= v_x,\\ \frac{dy}{dt} &= v_y.\end{aligned}$$

General relativity orbits. The gravitational force law predicted by general relativity can be approximated as

$$\vec{F}_g = -\frac{GM_S M_p}{r^3}\left(1 + \frac{\alpha}{r^2}\right)\left(x\hat{x} + y\hat{y}\right), \tag{7}$$

where α is a constant depending on the scenario. Notice this is just the Newtonian formula plus a small correction term proportional to r^{-4}. Mercury is close enough to the Sun that the effects of this correction can be observed. It results in a precession of Mercury's elliptical orbit.

Useful constants. Because we are working on astronomical scales, it will be easier to work in units larger than metres, seconds and kilograms. For distances, we will use the AU (Astronomical Unit, approximately equal to the distance between the Sun and the Earth), for mass, M_S (solar mass) and for time, the Earth year. Below are some constants you will need in these units:

- $M_S = 2.0 \times 10^{30}$ kg.
- 1 AU $= 1.496 \times 10^{11}$ m.
- $G = 6.67 \times 10^{-11}$ m^{3}kg^{-1}s^{-2} $= 39.5$ AU^3 M_S^{-1} yr^{-2}.
- $\alpha = 1.1 \times 10^{-8}$ AU^2. For the code, use $\alpha = 0.01$ AU^2 instead.
- Mass of Jupiter: $M_J = 10^{-3} M_S$.
- Jupiter's orbital radius: $a_J = 5.2 AU$.

Note that the SciPy package includes some of these constants: `https://docs.scipy.org/doc/scipy/reference/constants.html` (see the `test_constants.py` file shipped with this lab).

The Three Body Problem. We can consider the sun and two planets and see how the gravitational interactions affect the motions. This problem cannot be solved analytically. If we assume one of the planets is much more massive than the other, then we can just consider the gravitational influence of the larger planet on the smaller one and neglect the effect of the smaller object on the larger one (similar to how we treated the Sun in the 2 body problem).

Questions

1. **[40% of the lab] Modelling a planetary orbit**

   (a) Rearrange eqns. (6) to put in a format similar to eqn. (3) so that they can be used for numerical integration.
   Nothing to submit.

(b) As mentioned in the computational background section, you could try and code this up, but the code would prove reluctant to give you any satisfying answer. Instead, we will use the more stable Euler-Cromer method from now on. Write a "pseudocode" for a program that integrates your equations to calculate the position and velocity of the planet as a function of time under the Newtonian gravity force. The output of your code should include graphs of the components of velocity as a function of time and a plot of the orbit (x vs. y) in space.

Submit your pseudocode and explanatory notes.

(c) Now write a real python code from your pseudocode. We will assume the planet is Mercury. The initial conditions are:

$$x = 0.47\,\text{AU}, \quad y = 0.0\,\text{AU} \tag{8a}$$

$$v_x = 0.0\,\text{AU/yr}, \quad v_y = 8.17\,\text{AU/yr} \tag{8b}$$

Use a time step $\Delta t = 0.0001$ yr and integrate for 1 year. Check if angular momentum is conserved from the beginning to the end of the orbit. You should see an elliptical orbit. *Note: to correct for the tendency of* `matplotlib` *to plot on uneven axes, you can use one of the methods described here:*

`https://matplotlib.org/api/_as_gen/matplotlib.pyplot.axis.html`

Other note: our unit of year here is actually the Earth year, so your integration will cover several Mercury years.

Submit your code, plots, and explanatory notes.

(d) Now alter the gravitational force in your code to the general relativity form given in eqn. (7). The actual value of α for Mercury is given in the physics background, but it will be too small for our computational framework here. Instead, try $\alpha = 0.01\ \text{AU}^2$ which will exaggerate the effect. Demonstrate Mercury's orbital precession by plotting several orbits in the x, y plane that show the aphelion (furthest point) of the orbit moves around in time.

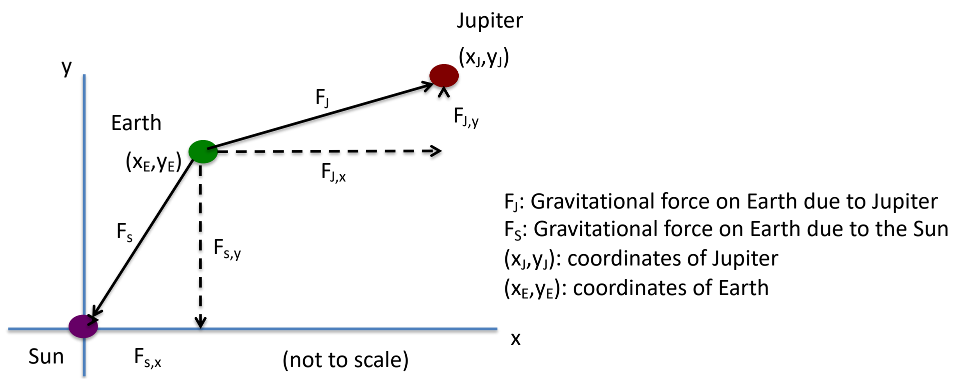
Submit your code (it can be the same file as in the previous question), plots, and explanatory notes.

2. **[40% of the lab] The three body problem.** Now lets add another planet to our system. We will consider Earth as the small planet and Jupiter as the large planet. The orbit of Earth will be determined

by the gravitational force of the Sun at the centre of the solar system and the gravitational force of Jupiter at an orbital radius of 5.2 AU. We will assume that Jupiter and the Sun are not affected by Earth's gravitational force.

(a) Write a pseudocode to add Jupiter to the system. Then alter your code from Q1c (i.e., the Newtonian gravity code) to add Jupiter and integrate the orbits for 10 (Earth) years. The figure below might be of some help. Some steps you will need to take:

- Simulate Jupiter's orbit,
- calculate the distance between Jupiter and Earth as a function of time,
- determine the net gravitational force on Earth due to both the Sun and Jupiter.



Jupiter
(x_J,y_J)
y
F_J
$F_{J,y}$
Earth
(x_E,y_E)
$F_{J,x}$
F_s
$F_{s,y}$
F_J: Gravitational force on Earth due to Jupiter
F_S: Gravitational force on Earth due to the Sun
(x_J,y_J): coordinates of Jupiter
(x_E,y_E): coordinates of Earth
x
Sun
$F_{s,x}$
(not to scale)

Use the following initial conditions for Jupiter:

$$x_J = 5.2 \text{ AU}, \quad y_J = 0.0 \text{ AU}, \tag{9a}$$
$$v_{x,J} = 0.0 \text{ AU/yr}, \quad v_{y,J} = 2.63 \text{ AU/yr}. \tag{9b}$$

Use the following initial conditions for Earth:

$$x_E = 1.0 \text{ AU}, \quad y_E = 0.0 \text{ AU}, \tag{10a}$$
$$v_{E,x} = 0.0 \text{ AU/yr}, \quad v_{E,y} = 6.18 \text{ AU/yr}. \tag{10b}$$

Submit your pseudocode, your python code, and a plot showing both Jupiter's and Earth's orbits.

(b) You should find that Jupiter does not have a big effect on Earth. Let's see what happens if Jupiter were more massive. Set Jupiter's mass to be 1000× its actual mass (i.e., if it had the same mass as the Sun; note that our approximation of Jupiter not affecting the Sun is now very wrong, but we will nonetheless retain it for the sake of argument). Run your code again with the same initial condition for Jupiter as in Q2a and plot the orbit in the x, y plane for 3 years. What happens if you run the simulation any longer?

Submit the plot of the orbit in the x, y plane, and a written description of the long-term behaviour of the Earth.

(c) Next, return Jupiter to its normal mass and replace Earth with an asteroid at an orbital distance of 3.3 AU. Plot the orbit in the x, y plane for 20 (Earth) years. Here are the initial conditions for the asteroid:

$$x_a = 3.3 \text{ AU}, \quad y_a = 0.0 \text{ AU} \tag{11a}$$

$$v_{a,x} = 0.0 \text{ AU/yr}, \quad v_{a,y} = 3.46 \text{ AU/yr} \tag{11b}$$

You should notice some perturbations in the asteroid's orbit. This is not only because the asteroid is closer to Jupiter, but also because at this orbital distance, it is in a motion resonance with Jupiter, which amplifies perturbations. Eventually, the asteroid is likely to get kicked out of the system.

Submit the plot of the orbit in the x, y plane.

3. **[20% of the lab] Timing Matrix multiplication**

Read through Example 4.3 on pages 137-138 of the text, which show you that to multiply two matrices of size $O(N)$ on a side takes $O(N^3)$ operations. So multiplying matrices that are $N = 1000$ on a side takes $O(10^9)$ operations (a "GigaFLOP").

Create two constant matrices `A` and `B` using the `numpy.ones` function. For example,

```
A = ones([N, N], float)*3.
```

will assign an $N \times N$ matrix to `A` with each entry equal to 3. Then time how long it takes to multiply the matrices to form a matrix `C`, using the code fragment in the textbook, for a range of N (from $N = 2$ to a few hundred). Print out and plot this time as a function of N and as a function of N^3. Compare this time to the time it takes numpy function `numpy.dot` to carry out the same operation. What do you notice? See `http://tinyurl.com/pythondot` for more explanations.

Submit your code, and the written answers to the questions.