Lab assignment #1: Python basics

Nico Grisouard (Instructor), Alex Cabaj and Vince Pascuzzi (TAs),
Haruki Hirasawa (Marker)

Due Friday, September 14th 2018, 5 pm

The goals of this lab are to:

- solve some physics and math problems using the computer;
- review basic programming concepts using python – loops, plotting, etc.;
- introduce the practice of writing pseudocode; and
- 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. Useful review material includes:

- Assigning variables: Sections 2.1, 2.2.1-2.2.2
- 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

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. If you would like more introductory material, then go through the material on the following web page:

https://computation.physics.utoronto.ca

And of course, internet searches are your friends.

Before you jump in and start banging away on your computer:

- **FIND A PARTNER TO WORK WITH YOU!!** Read the course syllabus for more information.

- In addition, read the guidelines for submitting work with partners and on evaluation of the labs.
- Specific instructions regarding what to hand out are written for each question in the form:

  THIS IS WHAT IS REQUIRED IN THE QUESTION.

  Not every question requires an answer we want to see! Some questions are purely there to help you.
- Carefully read **all** the material in this and all the handouts. There is a lot of information to help you complete your assignment.
- We are looking for "C^3" solutions that are **C**omplete, **C**lear, and **C**oncise: concisely and clearly explain the problem, the solution and numerical methods, and what is being shown in plots.
- Make sure to label all of your plot axes and include legends if you have more than one curve on a plot.

Computational Background

Numerical integration review. For a general first order system,

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

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

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

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 velocity first, and then uses the newly updated velocity to update the positions, is a small tweak that often leads to a stable result. That is, 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.

Basic random number generation. The code

```
#import random function from random module
from random import random
randomNum = random()
```

will assign to the variable `randomNum` a random value between 0.0 and 1.0, picked from a uniform distribution (to an excellent approximation). You can shift-and-scale to choose a number on another interval, e.g.

```
randomNum = -3+random()*10.0
```

would generate a random number uniformly distributed between -3.0 and 7.0.

The code

```
#import random function from numpy.random to create random arrays
from numpy.random import random
randomNums = random(10)
print('An array of 10 random numbers.\n', randomNums)
```

will print 10 random numbers, and can be scaled as above.

How to time the performance of your code. 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"), which is feasible on a standard personal computer in a reasonable length of time (according to the text). In Question 3 we'll explore how to time numerical calculations using your computer's built in stopwatch. Note that I am going to present a crude way to it, and there are better methods to profile code. Nonetheless, it will be enough for our purposes. The trick to timing is to import the python `time` module as in the following example:

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

#save start time
start=time()

#run your calculation
for n in range(terms):
     #here are  lines indented in the for loop
     #here are more lines indented in the for loop
```

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

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 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

$$\frac{\mathrm{d}v_x}{\mathrm{d}t} = -\frac{GM_S x}{r^3} \tag{6a}$$

$$\frac{\mathrm{d}v_y}{\mathrm{d}t} = -\frac{GM_S y}{r^3} \tag{6b}$$

$$\frac{\mathrm{d}x}{\mathrm{d}t} = v_x \tag{6c}$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = v_y \tag{6d}$$

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 $= 1\ M_S$.
- 1 AU $= 1.496 \times 10^{11}$ m.
- $G = 6.67 \times 10^{-11}\ \mathrm{m^3 kg^{-1} s^{-2}} = 39.5\ \mathrm{AU^3}\, M_S^{-1}\, \mathrm{yr^{-2}}$.
- $\alpha = 1.1 \times 10^{-8}\ \mathrm{AU^2}$. For the code, use $\alpha = 0.01\ \mathrm{AU^2}$ instead.

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:

   $$x = 0.47\,\mathrm{AU}, \quad y = 0.0\,\mathrm{AU} \tag{8a}$$
   $$v_x = 0.0\,\mathrm{AU/yr}, \quad v_y = 8.17\,\mathrm{AU/yr} \tag{8b}$$

   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

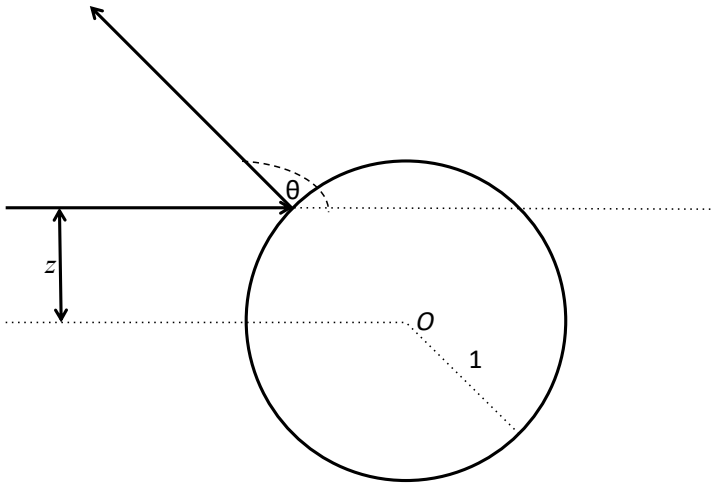

Figure 1: O is the centre of the unit circle cross section of a solid cylinder. A particle travels horizontally from the left and bounces off the cylinder through angle θ. *Note: this is only a sketch!*

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 $\alpha = 0.01$ AU^2 which will exaggerate the effect. Demonstrate Mercury's orbital precession by plotting several orbits in the x, y plane that show the perihelion (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] Distribution of scattered particles**

(a) A particle travelling horizontally from the left at height z (neglect gravity) bounces off a solid cylinder of unit radius (neglect gravity), with the particle's normal angle of reflection equal to its angle of incidence (Figure 1). The scattering angle with respect to the horizontal axis is θ as shown in Figure 1. Show that $\theta = \pi - 2\text{Arcsin}z$, and write a python function that implements this formula with z as input and θ as output. Numerically determine θ for $z = 0.25$ to make sure the result makes sense.

Now imagine that a particle coming in from the left at a randomly chosen z, with all values of z in the range $-1 < z < 1$ equally likely, scatters through the angle θ.

Do you think that all values of θ are equally likely for randomly chosen z? I.e., would you be more likely to find values in the range $170^\circ < \theta < 190^\circ$ or $90^\circ < \theta < 110^\circ$? Let's find out numerically!

NOTHING TO SUBMIT.

(b) Write a pseudocode to simulate this thought experiment of repeatedly scattering particles. The pseudocode should do the following: for each of a certain number of particles, pick a random height z, and scatter the particle (referring to the function in Q1a). Then plot a histogram of the incoming heights z and the outgoing θ, and estimate, using the numerical solution you've obtained, the relative probability of finding a particle in the range $175^\circ < \theta < 185^\circ$ versus $20^\circ < \theta < 30^\circ$. [*Hint: find the number of events in each range; the ratio of these is approximately the relative probability.*]

SUBMIT YOUR PSEUDO-CODE.

(c) Now implement this pseudocde in python. Generate the histograms described in Q2b. Is the distribution in θ uniform? Why or why not? Calculate the relative probability of finding a particle in the range $170^\circ < \theta < 190^\circ$ versus $90^\circ < \theta < 110^\circ$. Check that this number doesn't change much if you double the number of particles.

SUBMIT YOUR CODE, HISTOGRAMS, AND WRITTEN ANSWERS TO QUESTIONS.

3. **[20% of the lab]Timing Matrix multiplication** Referring to Example 4.3 in the text, 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 them to form a matrix `C` for a wide range of N from say $N = 2$ 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.