Lab assignment #1: Python basics

Instructor: Nicolas Grisouard (nicolas.grisouard@utoronto.ca)

Due Friday, September 18th 2020, 5 pm

Room 1 (groups #1–20) Mohamed Shaaban (Marker, m.shaaban@mail.utoronto.ca),
URL: https://gather.town/xg1R1WVWH3wfgdCY/phy407-ms
PWD: phy407-2020-MS

Room 2 (groups #21–40) Mikhail Schee (mikhail.schee@mail.utoronto.ca),
URL: https://gather.town/ZdDgdR4j2gGZIqvc/MS-PHY407
PWD: phy407-2020-MS

Room 3 (groups #>40) Nicolas Grisouard (nicolas.grisouard@utoronto.ca),
URL: https://gather.town/SluBq0pSNaWQycc4/phy407-NG
PWD: phy407-2020-NG

General Advice

- **Work with a partner!**
- Read this document and do its suggested readings to help with the pre-labs and labs. The goals of this lab are
  - to solve some physics and math problems using the computer;
  - to review basic programming concepts using python – loops, plotting, etc.;
  - to introduce the practice of writing "pseudocode" (or rather, 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. 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

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.

- Specific instructions regarding what to hand out are written for each question in the form:

THIS IS WHAT IS REQUIRED IN THE QUESTION.

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:

```
def MyFunc(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:

```
import MyFunctions as mf  # make sure file is in same folder
ZehValyou = 4.
ZehDubble = my.MyFunc(ZehValyou)
```

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 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 do 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 2nd 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

$$\frac{dv_x}{dt} = -\frac{GM_S x}{r^3}, \tag{6a}$$

$$\frac{dv_y}{dt} = -\frac{GM_S y}{r^3}, \tag{6b}$$

$$\frac{dx}{dt} = v_x, \quad \text{and} \tag{6c}$$

$$\frac{dy}{dt} = v_y. \tag{6d}$$

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.
- $G = 6.67 \times 10^{-11}$ 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.

Note that the `SciPy` package includes some of these constants[1] (see `test_constants.py` shipped with this lab).

Logistic map. For definiteness, let us consider the evolution of a population of insects with time: during year p, n_p insects could potentially breed, each of them producing on average μ new insects the next year. Namely, we have the recursive law

$$n_{p+1} = \mu n_p. \tag{8}$$

In μ is a constant, we therefore have

$$n_p = \mu^p n_0, \tag{9}$$

with n_0 the initial population. That is, we have exponential growth (if $\mu > 1$; decay otherwise) of the insect population, which is what would happen if there were no predators and they could enjoy an infinite amount of resources.

To take into account a finite amount of resources, we need to consider that as the population grows, resources diminish and μ decreases. For simplicity, let us assume that this decrease is a linear function of n_p, i.e.,

$$\mu = r\left(1 - \frac{n_p}{n_m}\right), \tag{10}$$

where n_{max} is the number of insects that will consume the entirety of resources available and make it impossible for the population to renew itself the next year, and r is the maximum reproduction rate, which is a constant. We then have

$$n_{p+1} = r\left(1 - \frac{n_p}{n_m}\right) n_p, \tag{11}$$

or, introducing $x_p = n_p / n_{max} \in [0, 1]$,

$$x_{p+1} = r\left(1 - x_p\right) x_p. \tag{12}$$

This equation above is referred to as a logistic map. It may seem simple and harmless, but do not be fooled...

[1] https://docs.scipy.org/doc/scipy/reference/constants.html

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:

   $$x = 0.47\,\text{AU}, \quad y = 0.0\,\text{AU} \tag{13a}$$
   $$v_x = 0.0\,\text{AU/yr}, \quad v_y = 8.17\,\text{AU/yr} \tag{13b}$$

   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 $\alpha = 0.01$ 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] Bifurcation and chaos**, adapted from a problem by Philippe Depondt (Sorbonne Université).

   (a) Write a pseudocode to compute and plot, as a function of the number of years p, the evolution of a population of insects described by eqn. (12).

SUBMIT YOUR PSEUDOCODE.

(b) For modularity's sake, write a python function which, given an initial normalized population x_0, a maximum reproduction rate r, and a total number of years p_{max}, outputs an array or list of values x_p, one for each p.

NOTHING TO SUBMIT.

(c) With $x_0 = 0.1$ and $r = 2$, plot the evolution of the population over $p_{max} = 50$. Then, plot the evolution of the population over $p_{max} = 50$ for a few values of r between 2 and 4, all in one figure. Do you see anything interesting?

SUBMIT YOUR PLOT OF x_p FOR A FEW VALUES OF r, AND YOUR COMMENTS ON THE SOLUTIONS. YOU WILL SUBMIT THE CODE LATER.

(d) On the same script, code the bifurcation diagram. That is, you will create a plot where the horizontal axis represents values of r (the bifurcation parameter), and the vertical axis shows values of x_p occurring for a given r.

To do so, keep $x_0 = 0.1$, increase r from 2 to 4 in increments of 0.1, and use $p_{max} = 2000$. However, for $r < 3$, only plot the last 100 x_p's, and only the last 1000 x_p's otherwise. Once you are confident that it works, reduce the r increments to e.g. 0.005.

In order for the diagram to look good, plot each x_p as a small dot, instead of joining the points with lines. And if you want to play with colours, try and make sure that it also looks nice. A safe choice would look draw inspiration from the following command.

```
plt.plot(x, 'k.', markersize=0.1)
```

The `'k'` enforces the colour black, the dot `'.'` is to use dot markers (not to be confused with dotted lines `':'`), and the marker size can be adjusted to optimize aesthetics.

How do you connect this bifurcation diagram with the previous plot x_p vs. p? Comment on how the notions of "period doubling" and "chaos" fit into that narrative, and give an approximate value of r beyond which the system can be considered chaotic.

SUBMIT YOUR PLOT AND YOUR WRITTEN ANSWERS TO THE QUESTIONS ABOVE.

(e) Re-do the previous figure, this time for $3.738 \leq r \leq 3.745$ in increments of 10^{-5}. Comments?

SUBMIT YOUR ANSWER TO THE QUESTION (NO NEED TO SUBMIT YOUR PLOT, UNLESS YOU FIND IT USEFUL TO ANSWER), AND YOUR CODE SO FAR.

(f) One of the essential features of chaos is the sensitivity to initial conditions. Choose a value of r that you are confident will lead to chaotic behaviour. Then, choose two initial populations, $x_0^{(1)} = x_0$ and $x_0^{(2)} = x_0 + \epsilon$, with ϵ a random number such that $|\epsilon| \ll x_0$, and iterate the logistic map for several iterations. Plot the two population evolution histories on the same plot.

Note: how many iterations? The answer can depend on the initial conditions; as you answer this question and the next, you will know if you have too many or on the contrary,

if you could increase it. In any case, clearly state in your report which numbers you chose and why.

SUBMIT YOUR PLOT, AND A BRIEF EXPLANATION OF HOW YOU CHOSE YOUR PARAMETERS.

(g) Now, plot the difference $\delta = |x_p^{(2)} - x_p^{(1)}|$, using a semi-log scale for the vertical axis (`plt.semilogy(...)`). You should see the spread grow more or less exponentially, until it saturates, because the population is bounded.

Note: this saturation time tells you how many iterations were sufficent in the previous question. You may go back and adjust.

Before this saturation, the spread δ can therefore be modelled as $ae^{\lambda p}$, where λ is called the "Lyapunov exponent" which measures how fast two almost identical systems end up diverging. Give an estimate of the numerical value of λ in this system.

Note: you could code some kind of fit... or you could just plot $ae^{\lambda p}$ on top of the previous semi-log plot of δ and try a few different values of a and λ, until you get a good eyeball agreement.

SUBMIT YOUR PLOT, AND YOUR CODE. IF ALL OF YOUR CODE FITS ON ONE SCRIPT (INCLUDING THE ONE FOR PARTS (A)–(E)), YOU ONLY HAVE TO SUBMIT IT ONCE.

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.