PHY407 Fall 2015: Computational Lab 1
Python Programming Introduction

In this lab we'll solve a couple of interesting[1] physics problems and at the same time review some basic concepts using python – things like loops, plotting, loops etc.. We'll also learn a bit about how to time your code for testing and understanding how to estimate the number of operations your code is doing.

We'll 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 material from PHY224 and/or PHY254 and look over Chapters 2 and 3 of 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

Newman's text assumes that you are using Python 3 but the U of T Physics python package UTTPD is based on Python 2. The differences are described in Appendix B starting on p. 510; the ones we deal with relate to integer division and printing. 1) In Python 2, division of two integers i and j, when written `i/j` in your code, results in the integer part of i/j, but in Python 3 it results in the simple quotient. 2) In Python 3, the `print` statement is a function but in Python 2 it is a standalone command.

Computational Background

- **Numerical integration review (relevant for Question 2):** Consider a system of n first-order time-dependent ordinary differential equations for coordinates $u_i = u_i(t),\ i = 1, \ldots, n$:

$$\frac{du_i}{dt} \equiv \dot{u}_i = R_i(u_1, \ldots, u_n),\ j = 1, \ldots, n, \tag{1}$$

  where the R_i are functions of the u_j and the dot indicates a time derivative (for later reference, a double dot indicates a second derivative with respect to time). The simplest way to numerically integrate the system is by approximating the derivative as:

$$\dot{u}_i = \frac{du_i}{dt} \approx \frac{\Delta u_i}{\Delta t} = \frac{u_i(t + \Delta t) - u_i(t)}{\Delta t},\ i = 1, \ldots, n \tag{2}$$

[1] At least Paul thinks they're interesting.

and then rearranging the system to read:

$$u_i(t+\Delta t) \approx u_i(t) + R_i[u_1(t), \ldots, u_n(t)]\Delta t,\ i = 1, \ldots, n \tag{3}$$

The numerical procedure is to start with an initial set of $u_i(t=0)$, then use (3) to update to the next time step. Indicating the time by superscripts, $u_i^k = u_i(t_k)$ at time $t_k = k\Delta t$ can be updated as

$$u_i^{k+1} = u_i^k + R_i(u_1^k, \ldots, u_n^k)\Delta t. \tag{4}$$

This is called the "Euler" method, which you've used before in PHY224 and PHY254. Later in the course we'll change the time that R_i is evaluated at and the approximation in (2) to come up with other integration methods.

- **How to time the performance of your code (relevant for Question 3):** 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. 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

- **A relativistic particle on a spring (for Question 2)**: In relativistic mechanics, Newton's second law for one dimensional motion on the x axis for a particle experiencing a force $F(x,t)$ is

$$\dot{p} = F, \tag{5}$$

where the dot indicates a time derivative and the relativistic momentum is

$$p = \frac{m\dot{x}}{\sqrt{1-\frac{\dot{x}^2}{c^2}}}, \tag{6}$$

where m is the particle mass and c is the speed of light.

Now consider a relativistic particle on a linear spring with $F = -kx$. It can be shown that the relativistic spring's energy

$$E = \frac{mc^2}{\sqrt{1 - \dot{x}^2/c^2}} + \frac{1}{2}kx^2 \tag{7}$$

is conserved. For small velocities, $|\dot{x}/c| \ll 1$, this implies that $\frac{1}{2}m\dot{x}^2 + \frac{1}{2}kx^2$ is conserved, as you'd expect for a classical Newtonian spring. You can rearrange (7) to show that the speed $|\dot{x}|$ is less than c even if you start with a very large initial displacement $x(t = 0)$ which would give rise to a huge initial acceleration. We'll explore this in Question 2.

From (5)–(6) you can show (but don't need to show for this assignment) that the equation of motion for the relativistic spring is

$$m\ddot{x} = -kx\left(1 - \frac{\dot{x}^2}{c^2}\right)^{3/2}.$$

To turn this into a system that you can integrate numerically, define the equivalent first order ODE system using the notation in (1): define $u_1 \equiv x, u_2 \equiv \dot{x}$ and thus

$$\begin{aligned} \dot{u}_1 &= u_2, \\ \dot{u}_2 &= -\frac{k}{m}u_1\left(1 - \frac{(u_2)^2}{c^2}\right)^{3/2} \end{aligned}$$

This is an interesting nonlinear ODE system that is hard to solve exactly but easy to solve numerically. The classical mass-on-spring system can be recovered for small velocities with $|u_2/c| \to 0$.

Lab Instructions

For grading purposes, you only need to hand in solutions to the following parts. Ensure that your codes are well commented and readable by an outsider:

- Q1a: Do not submit solution. Q1b-c: Submit pseudocode, python code, plots, and answers to questions.
- Q2: Submit pseudocode, Python code, plots, and written answers to questions.
- Q3: Submit plots and written answers to questions.

Distribution of scattered particles (40% of lab mark)

1. (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.

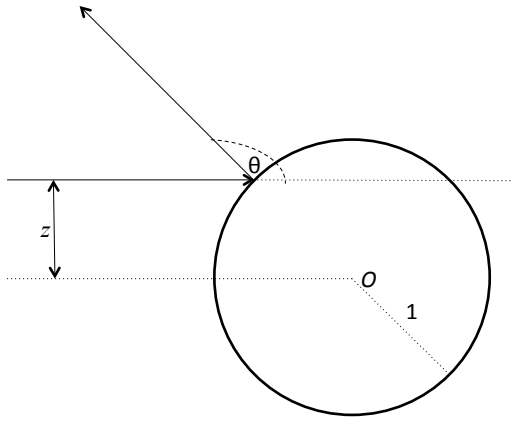

Figure 1: O is the centre of the unit circle cross section of a solid cylinder. A particle travels horizontally from the left and bounces off the cylinder through angle θ.

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!

(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 Q1). 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 $170^\circ < \theta < 190^\circ$ versus $90^\circ < \theta < 110^\circ$. [*Hint: find the number of events in each range; the ratio of these is approximately the relative probability.*]

(c) Now implement this pseudocde in python. Generate the histograms described in Q1b. 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. [*Hints: The following code:*

```
#import random function from random module
from random import random
#...
#Generate a value between 0 and 1 (uniform distribution)
randomNum = random()
```

will assign a random value between 0.0 and 1.0, uniformly distributed, to the variable `randomNum`. *The* `pylab` *function* `pylab.hist(zArray,100)` *will plot a histogram of the* `numpy` *array* `zArray` *with 100 bins. Your output will be more compact if you use the* `pylab.subplot` *to put more than one plot in a figure. Use* `pylab.saveas` *to save your output in pdf or jpg format.*]

Relativistic mass on a spring (40% of lab mark)

2. (a) Write a pseudocode to integrate the equation of motion for a relativistic particle on a spring, separately plotting its position and velocity, using the Euler method.

   Now implement this pseudocode in python and do the following:

   (b) Solve the system, including plots of the position and velocity for $k = 12\,\mathrm{N\,m^{-1}}$, $m = 1\,\mathrm{kg}$, starting with $x(t=0) = 1\,\mathrm{m}$ and from rest with $\dot{x}(t=0) = 0\,\mathrm{m/s}$, over the range $0 \leq t \leq 10\,\mathrm{s}$. Use a time step of $0.001\,\mathrm{s}$ here and in the rest of the question (this is a very short time step in order to compensate for the fact that the Euler method is unstable). Plot the relativistic and classical (non-relativistic) solutions on top of each other. Are relativistic effects important in this problem?

   (c) **Revised September 16.** For the classical particle on a spring $m\ddot{x} = -kx$, what initial displacement $x_c > 0$ for a particle initially at rest would lead to a speed equal to c at $x = 0$, using the same parameters as Q2b? *Hint: it's about 87,000 km.* Repeat Q2b but for $x = x_c$ and for $x = 10x_c$, and using the same timestep of $\Delta t = 0.001\,\mathrm{s}$. Does the period of the oscillation increase or decrease? How does the time series of the velocity change? Explain qualitatively the changes you see. For $x(t=0) = 10x_c$ compare the period of the oscillation to $4x_c/c$; why is $4x_c/c$ a pretty good estimate of the period, more or less independently of k and m [you can estimate periods from your graphs or numerically]?

Timing matrix multiplication (20% of lab mark)

3. Referring to Example 4.3 in the text, create two constant matrices `A` and `B` using the `numpy.ones` function (e.g. `A = ones([N,N],float)*3.0` will assign an `N`×`N` matrix to `A` with each entry equal to 3.0). 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? There's a nice discussion of this at `http://tinyurl.com/pythondot`.