Problem Set 2

SFHS club whose name I still don't know!

October 27th 2016

1 Ball trajectory

Our goal in this problem will be to numerically approximate the trajectory of a ball after it is kicked from the ground, until it hits the ground again. As usual, let's start with a simpler problem and build up from there!

We will use lists to represent the position of the ball in different points in time. For instance, the ball starts at $(S_x, S_y, S_z) = (0, 0, 0)$. We will also use (v_x, v_y, v_z) to denote velocity, (a_x, a_y, a_z) to denote acceleration, and (F_x, F_y, F_z) to denote force. As usual in physics, we will use the metric system of units in this problem set.

1.1 Playing soccer in vacuum

Assume, for now, that the soccer ball is in a vacuum, which will make the numerical approximations easier. The trick here will be progressing in very small time steps and estimating how much the ball would travel in each one of those time steps if its velocity remained constant. Then, for the next time step, you should update the velocity of the ball according to the laws of physics and let it travel a little bit again! If your time steps are small enough, this will result in a fairly accurate trajectory.

Laws of motion in vacuum:

- $(F_x, F_y, F_z) = (0, 0, -mg)$
- $(a_x, a_y, a_z) = (0, 0, -g)$

Planning the code

Let's first define some fundamental variables that will be used in your script. Choosing the time step is important, since it will determine the trade-off between accuracy and speed of your code. Play with different values of initial velocities!

```
dt = 0.1    # A time step of 0.1s
S = [0,0,0]  # Initial position of the ball
v = [10,5,15]    # Initial velocity in the directions x,y,z in m/s
g = 9.78    # Gravity acceleration in m/s^2
t = 0   # Time variable that will be iterated
m = 0.43    # Mass of a soccer ball in kg
```

Progressing one time step at a time

Let's make the ball travel its first step. First, update the velocity vector by taking the acceleration into account. In our approximation, the change in velocity at each time step is equivalent to $dv = dt \cdot a$. Remember that acceleration is defined as $a = \Delta v / \Delta t$.

Then, update the position of the ball using a similar reasoning: since $v = \Delta S / \Delta t$, we will use the approximation $dS = v \cdot dt$.

```
t = t + dt  # Adds a time step to the current time
v[2] = v[2] + dt*-g    # Updating the z velocity according to laws of physics
s[0] = s[0] + dt*v[0]    # Updating the x position using x velocity
s[1] = s[1] + dt*v[1]    # Updating the y position using y velocity
s[2] = s[1] + dt*v[1]    # Updating the z position using z velocity
```

Now let's progress one time step at a time, until the ball hits the ground again (that is, until $S_z = 0$). Try to do it with a while loop! Your objective is to store the position of the ball at each time step in a list whose elements are the positions (S_x, S_y, S_z) (that is, a list of lists).

1.2 Playing soccer with friction

Let's assume a slightly more realistic view of the world. We can approximate drag by a force that depends on the square of velocity and acts in the opposite direction: $F_{drag} = -k \cdot V^2$

The constant k depends on several factors, including the density of air, and the area of the ball. Use the following approximation:

$$k = \frac{1}{2} \rho C_d A \tag{1}$$

Here, $C_d = 0.47$ is a drag coefficient (approximately 0.47 for a sphere), $\rho = 1.225 kg/m^3$ is the density of air, and $A = 0.038 m^2$ is the cross sectional area of the ball.

Now we will have to use the mass of the soccer ball in order to determine the acceleration caused by drag and gravity combined. Use $F = m \cdot a$ to obtain (a_x, a_y, a_z) and then proceed to updating the velocity and the position of the ball using the same approximation tricks we used before.

Laws of motion with friction:

- $(F_x, F_y, F_z) = (-kV_x^2, -kV_y^2, -mg - kV_z^2)$
- $(a_x, a_y, a_z) = ?$

1.3 Play around with your script

Can you determine how far the ball travels after being kicked with different initial velocities? Try [10, 5, 15] and [5, 1, 20], for example. Also, can you determine the maximum height of the ball during its trajectory? In the future,

we will learn about ways to plot the trajectory of the ball, so make sure you save your script!

Finally, compare the results you obtain for the vacuum case and the case with drag.

2 Column Cipher

In cryptography, a column cipher, also called "Columnar transposition", works by writing the message to encode in rows of a fixed length, and then extracting the content column by column and concatenating them. So the message THIS IS A COLUMN CYPHER with rows of length 5 would be written:

THISI
SACOL
UMNCY
PHERX

and then be sent as "TSUPHAMHICNESOCRILYX". Note that you first need to remove spaces and pad the message with extra "X" characters to make it a multiple of the number of columns. In this exercise you will encode the message This message is very secret with a column cipher with rows of length 3. It is similar in spirit to the Caesar cipher exercise but of higher difficulty so allow some time or keep it for a second pass.

2.1

As mentioned in the introduction, the first pre-processing step is to remove spaces and use all upper-case letters. Convert the message to this format.

2.2

Just for this message, explore interactively how many "X" characters we need to add to the end of the message so that its length is a multiple of 3? Add them. (We will build a generic algorithm to compute the number of "X"s at the next question.)

2.3

Let's now devise a general expression to compute the number of padding "X" characters that will work for any message. Make sure that if there will be 2 characters on the last line, your computation returns 1 and that if there will not be any partial line, the algorithm returns 0.

The number of letters on the last incomplete line is len(message) % 3 because the modulo operator % gives the remainder when dividing by an integer. What does % do to negative integers in Python?

2.4

The first would-be column contains the characters at index 0, 3, 6, etc. Use slicing to extract these characters from the padded message without the need to actually make the 2D array of characters displayed in the introduction.

2.5

Extract the second and third columns using slicing and produce the encoded message. The pattern shown here and at the previous question will be used to build in the subsequent questions a generic function that encodes any message.

2.6

Re-use the code from previous questions to build a function encodecolumn() which takes 2 arguments (the message to encode and the number of rows) and returns the encoded message. Run the code below to test your function by encoding the original message and decoding the encoded message from question 5.

2.7

This question gets you to experiment with the random package of the standard library. The cipher strategy given above is very weak because its key, that is the amount of information the decoder needs in order to decode the message is only 1 integer, necessarily smaller than the length of the message. To make it (a little bit) more secure, we could decide that instead of putting the columns back together from left to right, the order is a random permutation of all the column indices. Once some chooses a row length to try, there are still row length! possibilities. For example, assuming that we use rows of length 5, the message THIS IS A COLUMN CYPHER:

01234
THISI
SACOL
UMNCY
PHERX

being converted to TSUPHAMHICNESOCRILYX assumes that the list of column indices is 01234. But we could have decided to go in reverse order for example, assuming a column order 43210, which would have led to ILYXSOCRICNEHAMHTSUP. Modify your encodecolumn function above, computing a permutation of the column numbers instead of simply grabbing the columns in the original order. To generate these permutations, experiment below with the shuffle function of the random package from the standard library. The permutation used will need to be returned by the encoding function to be given to the trusted decoder.