UCF Physics PHZ 3150: Introduction to Numerical Computing

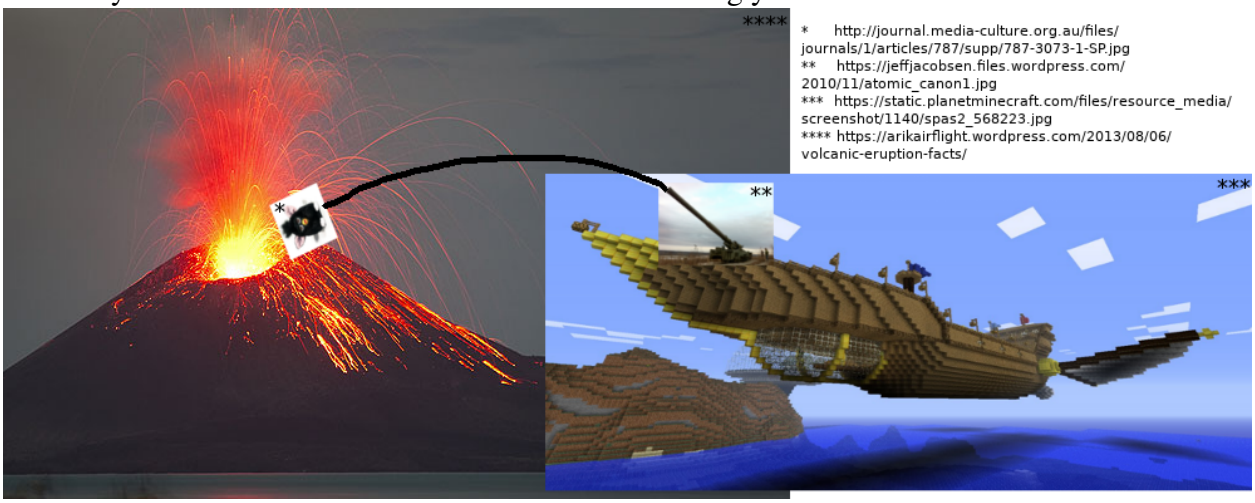
Spring 2019 Final Project

Due Wednesday 24 April 2019 10:00 am

Project designed, tested, and written by Zacchaeus Scheffer, 2019. This is a draft of the final project, for your comment and thoughts.

1 Overview

The Furbies^{TM}have begun to awaken. Humanity holds on by a thread as you and your team round up as many Furbies as possible. In order to dispose of them, you must return them to the fiery depths from which they were forged, Krakatoa, which has recently become active. However, it seems that nearing Krakatoa alerts the Furbies. You think this because the last party you sent up Krakatoa never returned. You will just have to make due with the modest supply of airships and cannons that were entrusted to you by the remaining world superpowers. Your plan is to fly as close as you can to Krakatoa and launch the Furbies using your cannons.



There are, however, a few problems. The cannons were not made to launch Furbies, so you will need to write code to calculate the trajectory of the Furby. Additionally, after Krakatoa became active, the weather got worse, so if you want to get as many Furbies as possible to their (final) destination, you will need to take wind resistance into account. Once you have access to this data, you will need to implement a PID algorithm (`https://en.wikipedia.org/wiki/PID_controller`) to control the cannon and direct it accurately.

Each cannon has a serial number. It can be controlled my importing the airship library and creating an instance of the Cannon class initialized with the correct serial number:

```
import airship
cannon = airship.Cannon(1234)
```

The Package is well documented, so you should be able to figure out everything you need from that. However, be sure **NOT** to try and set any of the cannon's variables by assignment as this will cause the encoders to malfunction (**NEVER** do `cannon.vel = [number]`, etc.). Use

`cannon.adjust()` instead.

2 Assignment

0) Keep a log and keep all text files (programs, mainly) under Git.
1) **Preliminaries:** In your main project file:

- Import the Cannon module included in this assignment.
- Take control of cannon # 1337 by initializing `cannon` as an instance of the `Cannon` class included in the `airship` module with serial number 1337.
- After importing matplotlib.pyplot, do:
  `from mpl_toolkits.mpl3d import Axes3D`
  Initialize a pyplot figure and a 3D subplot:
  `fig = plt.figure()`
  `ax = fig.add_subplot(111, projection='3d')`
  You will use the subplot axis (`ax`) to plot by sending it to functions.
- Set the variable `pos_target` to the tuple (-50, 1.234, 150). This is the current position of Krakatoa in cylindrical (z, ϕ, ρ) coordinates relative to you in your airship (at the origin).
- Set the variable `v_wind` to the tuple (-1.5, -1, 2). This is the current wind velocity in Cartesian (z, y, x) coordinates.

2) **Circles:** Write a helper function named `xycircle(center, rad, res=100)` that will be used later to generate circles in your 3D plot which will represent the cannon and target's (Krakatoa's) positions in your plot. Generate the z, y, and x positions positions of a circle parallel to the xy plane, centered at the point `center` with the radius `rad`. `center` will be a tuple of the (z, y, x) center of the circle. (The z values of the circle should all be the z value of the center of the circle). Make the circle have `res` number of points. You want the circle to close in on itself, so make sure your parameter runs from 0 to 2π inclusive. Return a single 2D array of shape (3, `res`) where each row has the values for a particular coordinate (rows in z, y, x order).

Print the output of `xycircle((0, 0, 0), 2, 5)`.

3) **Setup:** Write a function `plotsetup(pos_target, ax, rad_cannon=8, radii_target=(3, 7, 15))` that plots the the cannon and target and sets the appropriate z, y, and x limits on the plot. Set the z, y, and x limits of `ax` (a 3D subplot object) to be $\pm$ the maximum of the absolute value of z and ρ coordinates of the target. (If ρ is 10 and z is -20, then each of the limits should range from -20 to +20). Plot a circle of radius `rad_cannon` around the center of the cannon on `ax` (the cannon is at the origin) using points obtained from `xycircle`. Do the same for each radius in `target_radii` except centered at the target position. Do not rely on radii_target having three elements. The position of the target, `pos_target`, is given in

cylindrical coordinates (z, ϕ, ρ). Remember, `xycircle` wants a center in Cartesian coordinates. The Cartesian coordinates of `pos_target` can be found by:

$$\begin{aligned} z =& z \\ y =& \rho \sin(\phi) \\ x =& \rho \cos(\phi) \end{aligned}$$

(Hint: check the order that ax.plot() wants its arguments.) Color your lines however you wish.

Use `plotsetup` to plot your cannon and your target, Krakatoa, (which you assigned to a variable earlier). Feel free to adjust `rad_cannon` and `radii_target` to your liking, but make sure the former is a number and the latter is a tuple of at least two numbers. Save the plot.

4) **Trajectory** Write a function `calctrajectory(cannon, v_wind, time=10, steps=1000, k=.003)` that calculates the trajectory of a Furby. Find this using one of the two methods below. The second of these methods is worth many more points and takes into account wind resistance (which is important to maximize Furby destruction).

The zenith angle, `theta`, the azimuthal angle, `phi`, and the magnitude of the velocity can be obtained from `cannon` (Hint: `help(cannon)`). The components of a vector in Cartesian coordinates can be found from its spherical coordinates by the following equations:

$$\begin{aligned} v_z &= v \cos(\theta) \\ v_y &= v \sin(\theta) \sin(\phi) \\ v_x &= v \sin(\theta) \cos(\phi) \end{aligned}$$

After you implement one of the following methods, call `calctrajectory` with your cannon and the current wind velocity (which you assigned to a variable in the beginning). Plot the trajectory on your subplot object. Save the plot.

i) **Without Wind** Find the z, y, & x components of the initial velocity of the Furby. The position of the Furby at a given time, can be found from its initial position, initial velocity, and gravitational acceleration ($g = 9.8$ [m/s^2]).

$$\begin{aligned} z(t) &= z(0) + v_z t - \frac{1}{2} g t^2 \\ y(t) &= y(0) + v_y t \\ x(t) &= x(0) + v_x t \end{aligned}$$

The time should range from 0 to `time` seconds and have `steps` values. The Furby starts at the origin ($x(0) = y(0) = z(0) = 0$). Return the resulting array. (`k` and `v_wind` won't be used in this method)

ii) **With Wind** This problem becomes a lot more interesting with wind resistance. You may have solved a problem involving wind resistance in one dimension, but the problem becomes much

more difficult in three dimensions because the force the wind exerts on the Furby in the x direction is a function of the velocity in the z and y direction as well as in the x direction. For this reason, you will need to employ **numerical integration** to solve for the trajectory. In the following, I will present the equations you will use as **vector equations**. Check out the Vectors section of the Appendix for tips and tricks for vectors in Python.

To solve this problem we will use the drag equation (`https://en.wikipedia.org/wiki/Drag_equation`):

$$F_{drag} = \frac{1}{2}\rho|\vec{v}_{af}|\vec{v}_{af}C_D A$$

Here, $\vec{v}_{af}$ will be the velocity vector of the air relative to the Furby:

$$\vec{v}_{af} = \vec{v}_{air} - \vec{v}_{Furby}$$

and $|\vec{v}_{af}|$ is the magnitude of $\vec{v}_{af}$. The velocity vector of the air is the velocity vector of the wind. Note that `v_wind` may be a tuple or a numpy array, so if you want to do vector operations like in the Vectors section of the Appendix, you may need to convert `v_wind` to a numpy array. You can look at the Wikipedia page to learn about the other terms, but they will be constant across Furbies, so we will absorb them in to one constant in the end. We can solve for the acceleration of the Furby using Newton's second law:

$$\begin{aligned} m\vec{a} &= F_{gravity} &&+ F_{drag} \\ m\vec{a} &= m\vec{g} &&+ \frac{1}{2}\rho|\vec{v}_{af}|\vec{v}_{af}C_D A \\ \vec{a} &= \vec{g} &&+ k|\vec{v}_{af}|\vec{v}_{af} \end{aligned}$$

(You will just need the last line). Here, we have absorbed many constants like the Furby's mass into a constant k which you found earlier (this is the `k` passed to your function). $\vec{g}$ is the vector of gravitational acceleration

$$\vec{g} = (-9.8, 0, 0)[\text{ms}^{-2}]$$

Now, we can apply numerical integration. For this we will iterate through our time steps and find the position, velocity, and acceleration of the Furby at the current time step in terms of those values at the previous step. It might be easier for you to construct arrays of those three quantities, but only the position array must be returned. The recursion relationships you will use are:

$$\begin{aligned} \vec{r}_{i+1} &= \vec{r}_i + \vec{v}_i * \delta t + \frac{1}{2}\vec{a}_i \delta t^2 \\ \vec{v}_{i+1} &= \vec{v}_i + \vec{a}_i \delta t \\ \vec{a}_{i+1} &= \vec{g} + k|(\vec{v}_{af})_i|(\vec{v}_{af})_i \end{aligned}$$

The initial position is the origin, (0, 0, 0), the initial velocity can be found from the cannon as shown above, and the initial acceleration is just $\vec{g}$. The time step is:

$$\delta t = \frac{\text{total time}}{\text{\# of steps}}$$

5) **Landing** Write a function `landingpos(traj, z_target)` that takes an array of positions, `traj`, and the z component of the target's position, `z_target`, and returns the azimuthal

angle of the landing position and the distance in the xy plane from the cannon to the location where the Furby first goes below `z_target` level. Find the index of `traj` where the z value first goes below `z_target`. The x and y values at this index will give you the distance in the xy plane of the Furby from the cannon at this point by:

$$\rho = \sqrt{x^2 + y^2}$$

This xy distance will henceforth be referred to as ρ. Those same x and y values can be used to find the azimuthal angle ϕ of that point. ϕ is measured from the positive x-axis and goes counter-clockwise (looking down) about the z axis. Normally, if you try to take the arc-tangent, you get a value from $-\pi/2$ to $\pi/2$, but we want values from $-\pi$ to π. You could try to solve this problem by checking the sign of x and y to correct accordingly, but this is messy so I suggest you use arc-tan2 (the new and improved): `https://en.wikipedia.org/wiki/Atan2`. (Hint: check numpy). Return a tuple containing the ϕ and ρ of the landing location (in that order).

Print the output of calling `landingpos` with the trajectory calculated in question # 4 and the z value of Krakatoa's position (`pos_target`).

6) **PID iteration** Write a function `piditer(error, i, dt, kp, ki, kd)` which takes in an array of errors at each time step, a current time step index, a time step length, and PID k values and returns a PID output value u_i. (Error values after the `i` index will not be calculated yet, so be sure not to use them.) The output value will be:

$$u_i = P_i + I_i + D_i$$

P_i is the proportional term:

$$P_i = K_P e_i$$

where e_i is the error at the `i`th index and K_P is the proportional gain passed to your function (`kp`). I_i is the integral term. Because we are working with discrete values, the integral term will be approximated by a Riemann sum:

$$I_i = K_I \delta t \sum_{j=0}^{i} e_j$$

where δt is the time step length (`dt`) and K_I is the integral gain (`ki`). D_i is the derivative term. Once again, we are working with discrete values so we will approximate a derivative by a difference quotient:

$$D_i = K_D \frac{e_i - e_{i-1}}{\delta t}$$

where K_D is the derivative gain (`kd`). Note that the derivative doesn't make sense when `i` $= 0$ because there is no "change" in the error (there is no e_{-1}), so if that happens, set the derivative term (D_i) to zero. Return u_i.

Print the result of `piditer(np.arange(10), 5, 1, 1.5, 1.5, 1.5)`.

7) **PID controller** Write a function `pid(cannon, ax, pos_target, v_wind=np.zeros(3), steps=50, k_phi=(1.7,1.5,.00125),`

k_vel=(.25,1.5,.005)) that acts as a **p**roportional-**i**ntegral-**d**erivative controller for the cannon (https://en.wikipedia.org/wiki/PID_controller). Create an array to hold error values which has two rows and steps columns. The first row is for the error in ϕ_l (azimuthal angle about the cannon base that the Furby lands) and the second row is for the error in ρ (distance in the xy plane from the cannon to the Furby when it reaches the z value of the target). Additionally, make an array of power signals, also with two rows and steps columns. Make the first row the power sent to the ϕ_c (azimuthal angle about the cannon base that the cannon is aimed) motor and the second row the power sent to the v (initial velocity) motor and the of the cannon. Try no to mix up the angle of the cannon, ϕ_c, and the angle of the landing, ϕ_l.

Now for each step, do the following:

- Use calctrajectory to calculate the trajectory for the Furby.
- Plot the trajectory on ax. If i gives the step number, set the color of the line to be plt.cm.viridis(i/steps). This will give each trajectory you plot a color from purple to yellow tending towards yellow as the step number increases. If you want to use a different color map, check out https://matplotlib.org/examples/color/colormaps_reference.html, but make sure you can tell which lines correspond to earlier and later steps.
- Use landingpos to find the ϕ_l and ρ position that the Furby landed.
- Calculate the error in ϕ_l and ρ and insert these values into your error array. Make sure you calculate your error as the target value minus the actual (landing) value. If the error in ϕ_l is greater than π, subtract 2π from the ϕ error. If the error in ϕ_l is less than $-\pi$, add 2π to the ϕ_l error. (This way the cannon tries to move the smaller angle.)
- Use piditer to find the power to supply to the ϕ_c and v motors. Find the phipower using the error in ϕ_l, the step number, and the values in k_phi as your PID gains (the values are in (kp, ki, kd) order). Find the velpower using the error in ρ the step number, and PID gains in k_rho. Insert these powers into your power array. The time step for both of these calculations is the time it takes for the cannon to adjust (Hint: ask the cannon how long it takes to adjust nicely).
- adjust the cannon using the phipow and velpow you calculated above.

Return a tuple containing the error array and the power array (in that order).

Call pid with your cannon, your subplot, the current position of Krakatoa (pos_target), and the current wind velocity (v_wind). Save your plot.

8) **Final Report** Use the arrays returned from calling pid and make a (2D) plot for the error in ϕ_l and ρ (two lines on one plot) as well as a plot of the power sent to the ϕ_c and v motors of your cannon. You may need to clear your figure before trying to plot. (After clearing your figure, just use plt.plot() instead of your subplot object. You aren't passing it around or using a 3D plot, so no need to use a subplot object)

3 Appendix

3.1 Vector equations (and Python)

For our uses, vectors will represent a quantity with a magnitude in each Cartesian direction:

$$\vec{a} = (a_z, a_y, a_x)$$

We can multiply vectors by numbers to get a new vector:

$$k\vec{a} = k(a_z, a_y, a_x) = (ka_z, ka_y, ka_x)$$

Additionally, we can add vectors together, so if we have that one vector is the sum of two other vectors:

$$\vec{a} = \vec{b} + \vec{c}$$

then that vector's components are the sum of the other two's like so:

$$\vec{a} = (a_z, a_y, a_x) = (b_z, b_y, b_x) + (c_z, c_y, c_x) = (b_z + c_z, b_y + c_y, b_x + c_x)$$

Lastly, you can take the magnitude (or norm) of a vector (turns a vector into a number).

$$|\vec{a}| = |(a_z, a_y, a_x)| = \sqrt{a_z^2 + a_y^2 + a_x^2}$$

Now for the motivation for this notation. Consider the following equation:

$$\vec{a} = \vec{g} + k|\vec{v}_{af}|\vec{v}_{af}$$

This vector equation really represents 3 equations:

$$a_z = g_z + k|\vec{v}_{af}|(v_{af})_z$$
$$a_y = g_y + k|\vec{v}_{af}|(v_{af})_y$$
$$a_x = g_x + k|\vec{v}_{af}|(v_{af})_x$$

However, Python (and especially numpy) was made to deal with these sort of computations, so you could calculate it as 3 normal equations, or as one vector equation:

```
# Python exapmle
k    = .00123          # just some number for the sake of illustration
g    = np.array([-9.8, 0, 0])
v_af = v_air - v_furby # v_air & v_baloon are numpy arrays of length 3

# as three normal equations
a = np.ones(3)
nrmv = np.linalg.norm(v_af)
a[0] = g[0] + k * nrmv * v_af[0]
a[1] = g[1] + k * nrmv * v_af[1]
a[2] = g[2] + k * nrmv * v_af[2]

# as one vector equation
a = g + k * np.linalg.norm(v_ab) * v_af
```

Either of these methods works fine. WARNING: This does not work with tuples or lists. If you want to do operations like this, make sure your vectors are stored as numpy arrays.