6100 Takehome Final

Roberto Facey

December 7th 2020

1 Description

For the projectile project, I chose to look into a brute force method for solving each iteration. If I could calculate the extremes for each variable, I could narrow down the range for each loop. Theoretically, the projectile could only be fired between 1 and 89 degrees. If fired at zero degrees, the projectile would not move. If fire at ninety degrees the projectile would move up, but then go straight back down to its starting point.

I started by looking for the velocity needed to hit a 100 meter point at one degree. I set the timestep at 0.0001 and the final time at twenty seconds. After a couple of manual attempts, I found the required velocity was roughly 191. The second angle I looked for was 89. I kept the timestep and final time from the original problem and manually found the velocity needed. It was roughly 380.

I took the max velocity from the two extreme angles and added a bit when creating the loop. The loop for the angle was from 1 to (and including) 89. I lucked out with the final time and found that the highest final time required was around 19.43 seconds. The timestep took some messing around with. This will be discussed in the ”Issues” section.

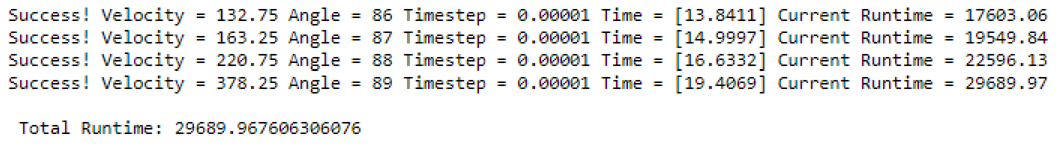
2 Issues

The first issue I found was trying to scrape the point of impact from each output. I copied and pasted the main function from the given program and added an ”if” statement to only include the most recent row once ”y” became negative.

The second issue I encountered was the timestep. I changing the timestep to 0.1 to shorten the number of iterations needed for each loop, but found that some angles were getting skipped. This issue mainly occurred in higher velocity angles. I changed it to 0.0001 to make sure all angles were captured.

The third issue I had was the velocity. Just like the timestep, the velocity needed to be broken down into smaller parts. I started with whole numbers but changed to straight to quarters (example: 38 ¿ 38.25 instead of 38 ¿ 39). This helped get a more accurate answer, but also increase the runtime for the program.

The final issue was the runtime. In the beginning of the project, I imported ”time” to see how long each iteration would take. What I was not expecting was the time between each iteration to increase toward the ends. It makes sense looking back at the project since the velocities toward the higher angles were much higher and thus more iterations were needed. Printing the time for each iteration also helped confirm that the program did not freeze or timeout.

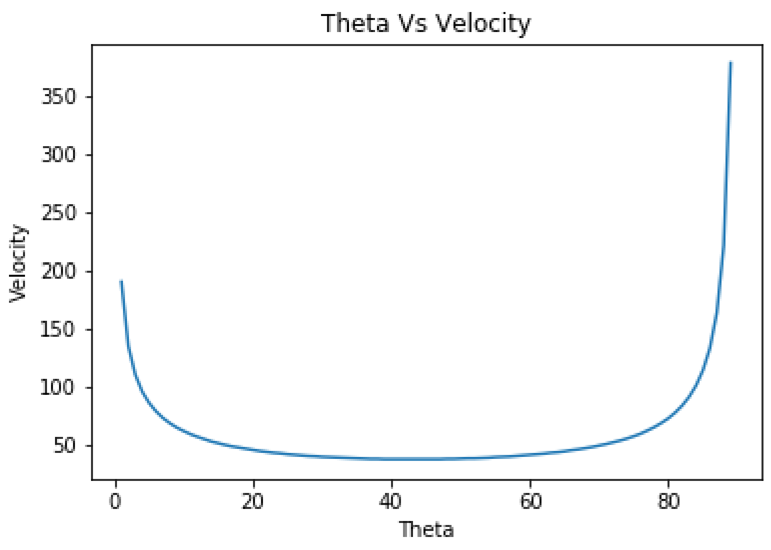


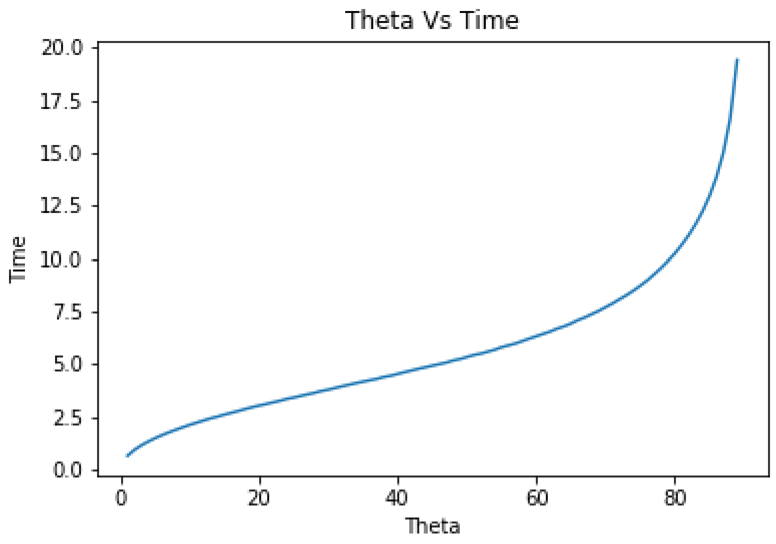
```
Success! Velocity = 132.75 Angle = 86 Timestep = 0.00001 Time = [13.8411] Current Runtime = 17603.06
Success! Velocity = 163.25 Angle = 87 Timestep = 0.00001 Time = [14.9997] Current Runtime = 19549.84
Success! Velocity = 220.75 Angle = 88 Timestep = 0.00001 Time = [16.6332] Current Runtime = 22596.13
Success! Velocity = 378.25 Angle = 89 Timestep = 0.00001 Time = [19.4069] Current Runtime = 29689.97

 Total Runtime: 29689.967606306076
```

3 Analysis Of Error

Due to the size of my steps for each loop, there didn’t seem to be much error in my results. The main issue I can see is that because my step sizes for the timestep and velocity were so small, the projectiles always landed on the outer parts of the margin of error (example: 99.005).

4 Results





5 Code

```
import numpy as np
import matplotlib.pyplot as plt
import time             # added time. Used in testing and gave me an idea of runtime

# a basic rk4 routine implemented in octave
#  based on an rk4 routine from:
# https://math.okstate.edu/people/yqwang/teaching/math4513_fall11/Notes/rungekutta.pdf

# the code has been modified to use arbitrary function names, return
# output in an array, and have vectors of an arbitrary length for
# its input



def projectile(time, p):
    x = p[0]
    z = p[1]
    vx = p[2]
    vz = p[3]

    # This is a 2trick to set the projectile constants as initial conditions
    coeff = p[4]

    drv = np.zeros(5, np.float)
    # the velocities are the derviative of position
    drv[0] = vx
    drv[1] = vz


    # calcualte the acceleration
    g = -9.8  # m/s - gravity

    ax = -coeff * vx * vx * np.sign(vx)
    az = g  - np.sign(vz) * coeff * vz*vz


    # update the change vector
    drv[2] = ax
    drv[3] = az

    # we don't actually update the projectile constants since they are constant
    drv[4] = 0

    return drv

def myrungekutta(h, t, nsteps, y0):
    # rk4 routine
    # inputs:
    #   h      - stepsize (dt)
    #   t      - starting time for the integration
    #   nsteps - number of steps to take during the integration
    #   y0     - initial conditions for the equation
    #
    #  outputs:
    #  output:  an array of the output values of system
    y = y0
    # find the number of values in the initial conditions array
    ny = len(y)
    output = np.zeros([1,ny+1], np.float)
    YCheck = 'False'

    # loop over the array and calculate the position using an rk4
    # ODE integration routine
    for i in range(nsteps):
        while YCheck == 'False':
            k1 = h*projectile(t,y)
            k2 = h*projectile(t+h/2, y+k1/2)
            k3 = h*projectile(t+h/2, y+k2/2)
            k4 = h*projectile(t+h, y+k3)
            y = y + (k1 + 2*(k2+k3) + k4)/6
            t = t + h

            output[i,0] = t
            output[i,1:] = y[:]

            if output[:,2][i] <= 0:     # added this to get only successful launches
                YCheck = 'True'

    return output

##################

##################
# integration parameters

# initial position and air resistance
x =0
z =0
coeff = 0.005

# inputs are
#  v- initial velocity of the projectile
#  theta - the angle above the ground
#  h = step size - generally below one
#  tfinal - final time in the simulation

h = float(0.0001)
tfinal = float(20.00)

# number of steps; irrelevant. choose something small
nsteps = int(tfinal / h) + 1

t = 0.0001

aResults = []
xResults = []
tResults = []

start_time = time.time()
for a in range(1, 90):
    for xv in np.arange(1, 381, 0.25):
        theta = a
        v = xv

        #v, theta, h, tfinal = input()
        vx = np.cos(theta * np.pi / 180.) * v
        vz = np.sin(theta * np.pi / 180.) * v

        # initial conditions for a circular projectile of unit size

        p0 = [x, z, vx, vz, coeff]

        o = myrungekutta(h, t, nsteps, p0)

        if o[:,1] >= 99 and o[:,1] <= 100:
            print("Success! Velocity = " + str(v) + " Angle = " + str(theta) + " Timestep =
            xResults.append(v)
            aResults.append(theta)
            tResults.append(float(o[:,0]))
            break
print("\n Total Runtime: " + str(time.time() - start_time))

with open("all_inputs.txt", "w") as file:
    for i in range(0, len(aResults)):
        file.write('\n' + str(aResults[i]) + ',' + str(xResults[i]) + ',' + str(tResults[i])

plt.title('Theta Vs Velocity')
plt.ylabel('Velocity')
plt.xlabel('Theta')
plt.plot(aResults, xResults)
plt.savefig('ThetaV')
plt.show()

plt.title('Theta Vs Time')
plt.ylabel('Time')
plt.xlabel('Theta')
plt.plot(aResults, tResults)
plt.savefig('ThetaT')
plt.show()


################################## Run second

import numpy as np

all_inputs = []

all_inputs_File = 'all_inputs.txt'

with open(tStepFile) as Curr:
```

```
with open(all_inputs_File) as Curr:
    all_inputs = [line.rstrip('\n').split(',') for line in Curr]

userInput = input("Enter the initial angle: >> ")

indexing = int(userInput)

print("The requirements for the initial angle of " + userInput + " are:")
print("Initial velocity:  " + all_inputs[indexing][1])
print("Timestep h: " + all_inputs[indexing][3])
print("TFinal:  " + all_inputs[indexing][2])
```