

newhomework

March 20, 2021

1 Part 1 (9 points)

Julie-Ann's friends pressure her into going bungee jumping for the first time. Julie-Ann is a cautious woman, so she isn't going to jump unless she is sure it is safe to do so. She decides that she'd like you to simulate the bungee jump using Python, since you've taken a course in Computer Science, and show her the results.

1. In the first step of this part, you'll simulate a free fall in the absence of air friction.
2. In the second step, you'll include calculations for air friction.
3. In the third step, you'll add the spring forces of the bungee cord.

Rubric

Marks	Task
2.0	Loop over time steps (same for all three steps)
3.0	Free fall: Update position, velocity, time (step 1)
2.0	Air friction: Update forces, update acceleration (changes from step 1)
2.0	Bungee: Update forces (changes from step 2)
9.0	TOTAL

1.1 Step 1

Using Julie-Ann's (wearing her clothes, the harness, and bungee cord) weight of 70kg, simulate the motion of a free fall, without considering the forces of friction or the effect of the bungee. In other words (in the simulation), let's let Julie-Ann fall to her death (for science!). Only gravity will affect Julie-Ann's fall. For simplicity, use positive numbers to represent downward motion, so length will represent how far from the platform Julie-Ann has traveled.

Note: Let's assume that Julie-Ann's height is infinite, so that we can see the pattern of her fall more easily.

Create a Python function (`simulateFreeFall`), which takes 3 arguments: - `mass`: The mass of Julie-Ann (or any falling object) - `deltaT`: The length of time (in seconds) of each time interval, for the simulation - `simulationTime`: The length of time (in seconds) of the entire simulation

The function will calculate from `deltaT` and `simulationTime` how many time steps there should be. For each of these time steps, you will calculate the length (distance from the platform, in metres), velocity (in metres/s), and acceleration (which will be constant, 9.81 metres per second squared).

At each time step, record each of the following values in a Python list:

- **elapsedTime**: `deltaT` * the number of time steps that have passed
- **length**: the distance between the object (Julie-Ann) and the platform at that time step
- **velocity**: the velocity of the object (Julie-Ann) at that time step 2
- **acceleration**: the acceleration on the object at that time step

The function will return a tuple containing all four of these lists (**times**, **lengths**, **velocities**, **accelerations**).

Updating the length:

When updating the length, use the current velocity and the current elapsed time, in the formula:

$$\text{delta}D = v * \text{delta}T$$

If you use a single time step duration for *deltaT* (**deltaT**), and the current velocity for *v*, then *deltaD* will be the change in **length**, which you'll need to add to the previous **length**.

Updating the velocity:

When updating velocity, we'll calculate how the current acceleration will affect the value, using the following formula:

$$\text{delta}V = a * \text{delta}T$$

If you use a single time step duration for *deltaT* (**deltaT**), and the current **acceleration** for *a*, then *deltaV* will be the change in velocity, which you'll need to add to the previous velocity.

Updating the acceleration:

This part of the assignment does not require any modification to acceleration. JulieAnn will continue to accelerate downwards for the duration of the simulation.

Write some Python code to call your function, collecting the four input lists it returns.

- Use 70kg for Julie-Ann's **mass**, **deltaT** of 0.01 seconds, and **simulationTime** of 60 seconds.
- Chart the **length** values vs. **elapsedTime** values, with red squares for markers, using Matplotlib.
 - Be sure to include appropriate title, x-axis labels, and yaxis labels in your chart.

Sample output

```
[1]: #step 1 code and testing
```

1.2 Step 2

For the second step of this part, you will simulate another free fall, but this time you will incorporate air friction (also known as drag). Start by making a copy of your function from step 1, and rename it **simulateFallFriction**. Our function will have one new argument, in addition to the previous arguments:

- **surfaceArea**: The surface area (in m²) of the falling person/object

We'll perform this calculation by adding up all forces on our object. In this part, we have two forces:

1. the force due to gravity (we'll call this *Fweight*)
2. the force due to air friction (we'll call this *Ffriction*)

Updating the length and velocity:

Updating the **length** and the **velocity** will use the same procedure as in the previous step, but now the **acceleration** is not a constant, since it depends on air friction (which itself is dependent upon **velocity**).

Updating the acceleration:

We didn't really consider forces in the previous section, so we'll need to consider the force created by gravity, using the following formula:

$$F_{weight} = m * g$$

A simplified formula for air resistance at sea level is given below:

$$F_{friction} = -0.65 * \text{surfaceArea} * v * |v| \quad (|v| - \text{absolute value of } v)$$

For the sake of this simulation, we'll use 0.2m² for Julie's surface area.

Now, the total force ($F_{total} = F_{weight} + F_{friction}$) can be used to calculate the **acceleration**:

$$a = F_{total} / \text{mass}$$

As before, write some Python code to call your function and collect the returned lists.

- Again, plot the **length** values vs. the **elapsedTime** values in Matplotlib, using blue circles for markers.

Sample output

```
[2]: #step 2 code and testing
```

1.3 Step 3

In this part of the assignment, we will add the effect of the spring (bungee cord) on the falling object, so that Julie-Ann can see a fairly realistic simulation of her bungee jump. Copy the function from **step 2**, and rename it **simulateBungeeJumper** to start. We'll have a new argument for this function:

- **unstretchedBungeeLength**: The length (in m) of the bungee/spring

Updating the length and velocity:

Updating the **length** and the **velocity** will use the same procedure as before, but now the **acceleration** is not a constant, since it depends on air friction (which itself is dependent upon **velocity**).

Updating the acceleration:

We now have three forces acting on our falling object:

1. the force due to gravity (we'll call this *Fweight*)
2. the force due to air friction (we'll call this *Ffriction*)
3. the force due to spring resistance (we'll call this *Fspring*)

The force applied to a spring stretched by a displacement (d) is given by Hooke's Law:

$$F_{spring} = -kd \text{ (Hooke's Law)}$$

In this formula, k is the spring constant, which depends on the stretchiness of the bungee cord. For our example, we'll use $k = 21.7$ for the spring constant. In the formula, d represents how much the spring has stretched (i.e. `length`). We'll test the function with an unstretched bungee length of 30m.

As before, write some Python code to call your function and collect the returned lists. - Again, plot the length values vs. the elapsed time values in Matplotlib, using green triangles for markers.

Sample output

```
[3]: #step 3 code and testing
```

2 Part 2 (6 points)

For this part of the assignment, you will write a simulation for the spread of an infectious disease. For simplicity, we will assume that the disease has a constant recovery rate (0.15), fatality rate (0.025), and recovery rate (0.15). **Part of this program has been written for you in the cell below.**

- Your job is to fill in some of the missing functionality, as described in the two parts.

Rubric

Marks	Task
3.0	Update the values of fatalities, infected, susceptible, and recovered
3.0	Generate plot
6.0	TOTAL

```
[4]: import random
import matplotlib.pyplot as plt

def simulateDay(numFatalities, numInfected, numRecovered, numSusceptible,
    ↪ numContacts, spreadProb, deathProb, recoverProb):
    # your code for Step 1 goes here

    # determine the fatalities from yesterday's infections

    # determine the spread of the disease

    return numFatalities, numInfected, numRecovered, numSusceptible

def simulateNDays(numDays, initialInfected, numPeople, numContacts, spreadProb,
    ↪ deathProb, recoverProb):
    numFatalities = 0
    numInfected = initialInfected
```

```

numRecovered = 0
numSusceptible = numPeople - numInfected

fatalities = [numFatalities]
infected = [numInfected]
recovered = [numRecovered]
susceptible = [numSusceptible]
for day in range(numDays):
    f,i,r,s = simulateDay(numFatalities, numInfected, numRecovered,
↪numSusceptible, numContacts, spreadProb, deathProb, recoverProb)

    fatalities.append(f)
    infected.append(i)
    recovered.append(r)
    susceptible.append(s)

    numFatalities = f
    numInfected = i
    numRecovered = r
    numSusceptible = s

return fatalities, infected, recovered, susceptible

numDays = 50
f,i,r,s = simulateNDays(numDays, 1, 100, 30, 0.10, 0.025, 0.15)

# your plot from step 2 goes here

```

2.1 Step 1

Write the function `simulateDay` that simulates a single day. This function will:

- For each susceptible person (not infected, dead, or recovered), simulate if he/she becomes infected, using the following formula: “ $\text{infectionRate} = \text{spreadProb} * \text{numContacts} * \text{numInfected} / \text{numPeople}$ ”
 - **spreadProb**: In one contact with an infected person, the probability of the disease spreading
 - **numContacts**: How many times does a typical person come into contact with people in a day
 - **numInfected**: How many infected people are there in the population
 - **numPeople**: How many people are there in the population (including infected, recovered, and susceptible)
- For each infected person, simulate if he/she recovers or dies
 - **deathProb**: The probability each day of someone infected dying from the disease
 - **recoverProb**: The probability each day of someone infected recovering from the disease

As an example to illustrate how to simulate is a person dies from the disease, assuming that the `deathProb` is a number between 0 and 1:

```
rand = random.random()
if rand < deathProb:
    # this patient has died
```

The return value for this function should be a tuple containing four values:

- The number of fatalities at the end of the day
- The number of infected people at the end of the day
- The number of recovered people at the end of the day
- The number of susceptible people at the end of the day

2.2 Step 2

In the code provided, the function `simulateNDays` is called with specific parameters for the probabilities, the initial number of infected people, and the number of days to simulate. You will write code to use the return values to draw a plot of the four categories of people after each day's end. As in `simulateDay()`, this function returns four values, but these values are lists:

- A list of fatalities after each day (red square)
- A list of infected after each day (yellow triangle)
- A list of recovered after each day (green circle)
- A list of susceptible after each day (blue plus)

Draw all four sets of values on a single plot using `Matplotlib`. Use the markers and colours indicated above, beside each category of people. The resulting plot is shown in figure 4 below.

Include in your plot a legend, so that the observer can know at a glance what each shape represents. To include a legend for four plots, use the following code:

```
plt.legend(['Fatalities', 'Infected', 'Recovered', 'Susceptible'], loc='upper right')
```

Sample output