Projectile Motion: A mathematical exploration

July 6, 2024

Part I
The basics

1 Introduction to projectiles

Projectile motion is the motion of an object shot through the air only subject to the downward acceleration due to gravity. According to most models, the projectile moves at a constant velocity in one dimension whilst accelerating downwards. These velocities are independent and unrelated from each other and can only be paired and explored together as a function of time (in other words, the time links the two). As such an equation for y in terms of x can only be found if the time is rearranged as a in terms of other factors and then substituted. To that end, the range of the projectile (how far it flies in the x direction) is directly dependant on how long it is in the air.

2 Why python?

Python is a high-level langauge with simple and expressive syntax and is readable without much prerequisite knowledge. Python is a popular programming language used for a range of tasks, such as general-purpose programming, data science or web development. Personally, python is the language that I am most comfortable with, and I understand how to use the frameworks and expansive library collection to a great extent. Python has a supportive and ever-growing community with people who are active on forums and are actively developing new supporting modules every day. For this project specifically, I used a variety of libraries in order to deliver models in two styles. The math module was a necessity in order to access an accurate value for pi and trigonometric ratios (sin cos tan...). For the graphing aspect, I used matplotlib library. I used the .pyplot() function to create figures that reflect the graphs shown on the presentation and to add a legend, axis titles and to scale the size of the graph depending on the size of the displacements or time are meant to be displayed. Lastly, specifically for animation, I used the pygame library. Essentially, my animation would be a 'video' run at a certain framerate. The ball would be drawn on a specific part of the screen (according to its displacement at that time) and then the image would be coloured over in black effectively “erasing” the ball. Then a new one would be drawn in the next frame to give the illusion that the ball is moving. However, it is just hundreds of images flashing up on screen, just like normal videos function.

3 Building the plot_function function

Throughout the challenge I needed to plot trajectory graphs given certain specifications, therefore I created a function as I would need to repeatedly plot the appropriate lines using matplotlib. The function has the standard parameters of the velocity, angle, gravity and height and with those, it calculates the range of the projectile using the formula derived below. Then, in select increments, the increment (which is effectively a counter that adds very small amounts each iteration) is plotted as the x value against the y value which is calculated as a function of x (who's equation is also derived below). The list of increments and corresponding y values is then passed in the matplotlib.plot in order to create the graph.

4 Building the get_info function

For every task, a different set of information is required by the user and to help with this I built function that would help isolate this information gathering aspect of the model. For some such as task three, the initial parameters that were needed to be entered by the user could be changed given the information. For instance, in task three, the user needs to be told the minimum velocity that would be required for the projectile to pass through the target and then the user could use that information to give an appropriate figure for the velocity. Had this process been in the main file, there would be many unused variables and this was simply better coding practice. This is similar to task five where much more information had to be processed at variables like a, b and c respectively were used to calculate the high and low projectiles. The function allowed to me access only needed information.

5 Plotting r against t

The final function that was interesting to build was the plot of the magnitude of the distance of the particle of the point away from the start against the time. So, to start is very similar to task one however with a few different specifications. To get the graph to reflect the one on slide #19, the time had to end at 2.5 seconds and for the plotting to stop at that point. Therefore, using the Verlet method the time steadily increased by 0.001 seconds until the increments of time exceeded the 2.5 limit. At every time interval the value of the displacement via Pythagoras was appended to its own list with each index corresponding to the index on the time list. These were then plotted against each other.

Part II
Aspects that required problem solving

6 Graphing multiple bounces

6.1 Solution #1

Initially there were two main problems with graphing multiple bounces: the constant x direction velocity versus the variable y velocity as well as an issue with the timekeeping using the Verlet method. As is standard for the 'Verlet method', I had a variable called current_time which kept track of time and every 'cycle'. However, with the ball bouncing, when the y displacement returned to zero, it was essentially restarting the projectie launch (at least in the y proportion of the vector). However, the x displacement was still running linearly. In other words, the time in the air is different for the two components. My solution to this was to keep track of which bounce was currently underway and create a list of times where the ball hits the ground. Then for every time increment, the program checks the current time and takes away from the times_list at the index of the bounce number. As a result, the y vector has its own 'local time' compared to the x vector. From there I plotted the x and y displacements at every point, adjusting the y component of velocity by multiplying by the coefficient of restitution for every subsequent bounce. However, this solution is not very time efficient as calculating the 'local time' requires iterating through the times_list in order to find which number is the greatest that the current time is bigger than.

6.2 Solution #2

Solution #1 works perfectly well however, from a computer science point of view, it is very time inefficient. As explained, for every increment of time (which is increasing in a loop itself) another loop is enterred to find said 'local' time. After completing the project, I realised a much better way to do this. The bounces do not have to be calculated in one continuous motion and I instead made each bounce an iteration in a for loop. As I have already taken into account the start of the bounce, the end of the bounce simply ends when the y displacement hits 0 (where the floor is 0, to take into account an initial hight).

This is simply done using a while loop. Where this solution shines however, is that there can be an 'actual_time' variable initialised before the for loop and the 'bounce_time' initialised inside said loop (so that it resets to 0 every bounce). This allows me to keep track of the total airtime of the ball (for the x displacement) and the airtime for the current bounce (for the y displacement). Once again, these displacements are made into separate lists and those are inputted into the matplotlib function.

7 Animating multiple bounces

As a result of the graphing, I had an array of x and y positions at every discrete timestep. Pygame and general videos function by displaying a certain number of images that give the effect of continuous motion. I set my program to 100 fps for processing, but my discrete time interval was 0.001 seconds. If each interval corresponded to a frame, the ball would effectively be 10 times slower. As a result, I had to make move the ball move to the position for every timestep. However, I also wanted to make it scale so that no matter the fps or discrete timestep interval, the ball would 'move' at the correct 'speed'. Taking 100 fps and 0.001s as examples, I had to multiply every x-list and y-list index call by $\frac{1}{fps*timeinterval}$. If 100 and 0.001 are plugged into that formula, the correct scale multiple is outputted as long as the fraction yields in integer - if it doesn't then rounding will have to take place and make the animation less accurate. Then there was the issue for scaling the window size for all possible parameters so that the ball would always stay on the screen and/or take up most of the screen. Having a fixed window size meant that, while I was testing, the ball would fly 'offscreen' or stay in the bottom left corner moving minimally. As my computer had an optimal 'window size' I decided to scale up the changes in displacement of the ball. For instance for the x vector I used the range of the projectile after the successive bounces by using the list of bounce times and converting that to distance and using $s = ucos(\theta)t$ and same for the y vector. This would give the 'maximum displacement' the ball would take and I multiplied every other displacement by $\frac{preferredwindowsize}{maximumdisplacement}$this effectively scaled up each displacment (albeit to different amounts). Then, I had to at every point add to the displacement the radius of the ball so that it starts at the bottom right of the screen and had to add the vertical window size and the radius of the ball to every displacement in the y direction as in pygame, (0,0) is the top left of the screen. As a result of all the visual manipulation, I display the actual displacements at every point during the animation by creating a text object and assigning it to a rectangle that displays said coordinates.

8 Modelling air resistance

I created a dictionary to store each shape as its own name and to have a corresponding drag coefficient so I can access the number using the name of the object. I created three functions (so I can decide in which order things occur) to modify the displacement, velocity and acceleration of the particle respectively. These functions use the listed equations to change their respective x and y variables according to the predetermined timestep. The graphing ends when the y displacement is equal to zero.

9 Assumptions

- The projectile is dimensionless and in reality, the projectile would not be able to start from y = 0 and would hit the ground earlier therefore reducing variables such as the time of flight and range of the projectile
- Expanding on the point of the object being dimensionless, the size of the object was not considered in terms of surface area flowing through the air even in task 9. If the object is large enough, it might not have laminar flow therefore the flight time and range would have been further reduced
- The value of gravity does not vary due to the height of the projectile. Although it is negligible for relatively low elevations and projectile speeds, Newton's universal law of gravitation states that the force an object feels is inversely proportional to the square of its separation from another body. Therefore, as the projectile goes up, the value for gravity should slightly decrease, increasing the time of flight and range
- Does not take into account the movement and rotation of the Earth which would give the projectile an initial angular velocity in addition to a greater linear initial velocity. The given displacements are based on the projectiles given relative position of the Earth and not the absolute displacement of the particle in space

Part III
Equation derivations

10 Deriving slide #6 equations

10.1 Apogee y

From $v^2 = u^2 + 2as$ suvat equation where at peak vertical height so v = 0:

$0 = u^2 sin^2\theta - 2g(y_a - h)$

$y_a = \frac{u^2 sin^2\theta}{2g} + h$

10.2 Apogee x

From $v = u + at$ suvat equation where at peak vertical height so v = 0

$0 = usin\theta - gt$

$t = \frac{usin\theta}{g}$

insert into $x(t) = ucos(\theta)t$

$x_a = \frac{u^2}{g}sin\theta cos\theta$

10.3 y as a function of x

$x(t) = ucos(\theta)t$ **therefore** $t = \frac{x}{ucos\theta}$

substituting t into $y(t) = h + usin(\theta)t - \frac{1}{2}gt^2$

$y(x) = h + xtan(\theta) - \frac{g}{2u^2}(1 + tan^2x)x^2$

10.4 Range equation

Range found at the x position when the projectile hits y = 0

From $s = ut + \frac{1}{2}at^2$ suvat equation where s = -h

$-h = usin\theta t - \frac{1}{2}gt^2$

$t^2 - \frac{2usin\theta}{g} - \frac{2h}{g} = 0$

$(t - \frac{usin\theta}{g})^2 - \frac{u^2sin^2\theta}{g^2} - \frac{2gh}{g^2} = 0$

As t is not negative:

$t = \frac{usin\theta}{g} + \frac{1}{g}\sqrt{u^2sin^2\theta - 2gh}$

Factor out u from the root and simplify:

$t = \frac{u}{g}(sin\theta + \sqrt{sin^2\theta - \frac{2gh}{u^2}})$

$x(t) = ucos\theta t$ **therefore via substitution:**

$R = \frac{u^2}{g}(sin\theta cos\theta + cos\theta\sqrt{sin^2\theta + \frac{2gh}{u^2}})$

11 Working out the standard integral: $\int\sqrt{1+z^2}dz = \frac{1}{2}z(\sqrt{1+z^2}) + \frac{1}{2}ln(z+\sqrt{1+z^2})$

11.1 Needed hyperbolic function identities:

$cosh^2x \equiv 1 + sinh^2x$

$cosh^2x \equiv \frac{cosh(2x)+1}{2}$

$arcsinhx \equiv ln(x + \sqrt{x^2+1})$

$cosh(arcsinhx) \equiv \sqrt{x^2+1}$

$\int\sqrt{1+z^2}dz$

11.2 solving

let z = sinh(x), x = arcsinh(z), dz/dx = cosh(x)

$\int coshx\sqrt{1+sinh^2x}dx = \int cosh^2xdx = \frac{1}{2}\int cosh2x+1dx = \frac{1}{4}sinh2x+\frac{1}{2}x+c = \frac{1}{2}sinhxcoshx + \frac{1}{2}x + c$

substitute back in values in terms of z for x: x = arcsinh(z)

$\int\sqrt{1+z^2}dz = \frac{1}{2}z(\sqrt{1+z^2}) + \frac{1}{2}ln(z+\sqrt{1+z^2}) + c$