110770716

Darren Kong

AMS326

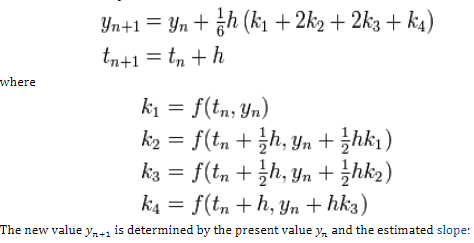
Homework 3

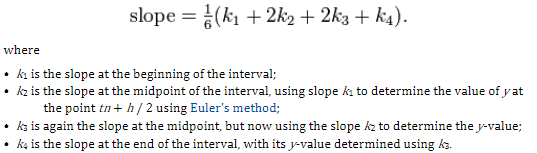
**Problem 3.1**

This problem is rather straight forward. The first thing I did was to set up constants v\_0, v\_B1, v\_B2, v\_B3, and a. They are 14, 7, 14, 21, and 7777 respectively.

Then I created a function called **w(x)**, which was simply the expression and returns the result of the expression. This represents the flow of water in the river. The next function was **f(x, y, vb)**. This takes in three parameters x, y, and vb, and returns the value of the expression.

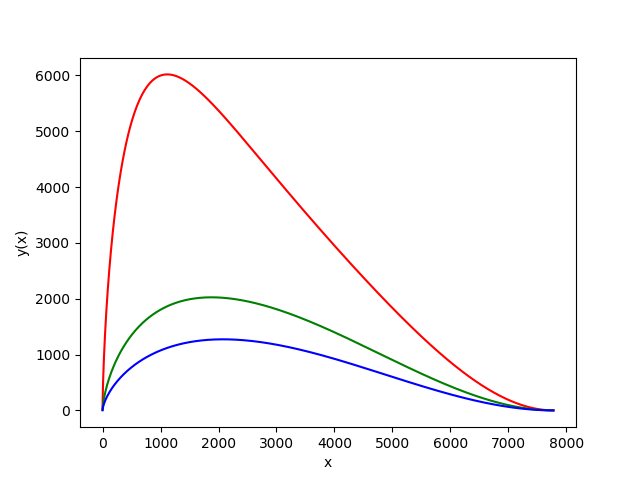
The next function is **runge\_kutta(f, x0, y0, h)**. The method takes in a function f, which was defined above. It also takes in values x0 and y0 which are the initial values given in the problem, . It also takes in one more parameter h, which is the step size specified, -.1. The function also takes another parameter **vb**, which is the boat speed. The function approximates the next values for x and y based on the parameters. In other words, the formulas are shown below:





The entire **runge\_kutta()** method is thrown into a loop. The loop simply records the x and y values generated from each iteration of the runge\_kutta method and returns both the x and y values in two respective lists called **xplot1** and **yplot1** respectively. This is then plotted onto a graph represented by the **red** line.

The above steps are then repeated for different values of boat speed **v\_B#**. The **green line** represents the boat speed of 14 and the **blue line** represents the boat speed of 21.

A sample of the resultant graph is shown below.

**Problem 3.2**

The original code for the was written in python, but it came to my attention that the program would take too long to run within the time limit and take too much memory. Therefore, I opted to rewrite the majority of the code in C for better performance.

The first function that was written was **getRandom(min, max)**. It returns a random double value that is between the values specified by **min** and **max**.

The second function is **intersectLineCircle2(left, d)**. It is a rather clever function that incorporated an idea that was discussed between me and **Farhan Ahmed**. Basically, since we are using a disc/circle as the “needle”, we simply only care about the horizontal boundary values of the circle. There is no need to care about the vertical y-values. We generate a random **left** value by using **getRandom()**. Then we know the diameter of the circle by the given parameter **d**. We subtract **d** from left to get **right**. Then we simply floor both **left** and **right** and subtract floors from each other. This will give us the amount of intersections with vertical lines that the circle crosses. We floor the two values since our vertical lines are on integer based x’s. Intuitively, this method works because of the subtraction yields the integer difference between the two values.

The third function is **buffonDisc(nlines, d, tosses)**. The value of **nlines** contains both the range of the bounding x-values as well as the amount of vertical lines. A loop is done for **tosses** amount of times where **intersectLineCircle2()** is called and a running sum of intersects is recorded. Finally, the value of the intersects is divided by the number of tosses to get the probability which is printed out.