

Caitlyn Jones

Prof. Thomas Fai

MATH 122A: Numerical Methods and Big Data

17 September 2024

## Homework 2

### Problem 1

- A) To plot  $p(x)$  over  $x$  by evaluating  $p$ , we first need to define array `poly_coeff` filled with the coefficients of  $p(x) = x^4 - 16x^3 + 96x^2 - 256x + 256$ .

```
'''
The coefficients of  $p(x) = x^4 - 16x^3 + 96x^2 - 256x + 256$ .
'''
poly_coeff = [1, -16, 96, -256, 256]
```

Next, we need to define the variable `x_values` that will be used to evaluate  $p(x)$ , which are the values (4.00001, 4.00002, 4.00003, ..., 4.00101). The values are increasing by .00001 from 4.00001 to 4.00101.

```
'''
The x-values that will be used to evaluate  $p(x)$ .
The x-values are 4.00001, 4.00002, ..., 4.00101,
i.e. the values from 4.00001-4.00101,
increasing by .00001 each time.
'''
x_values = np.linspace(4.00001, 4.00101, 101)
```

Next, we need to evaluate  $p(x)$  at each  $x$  and save these values so that we can graph them.

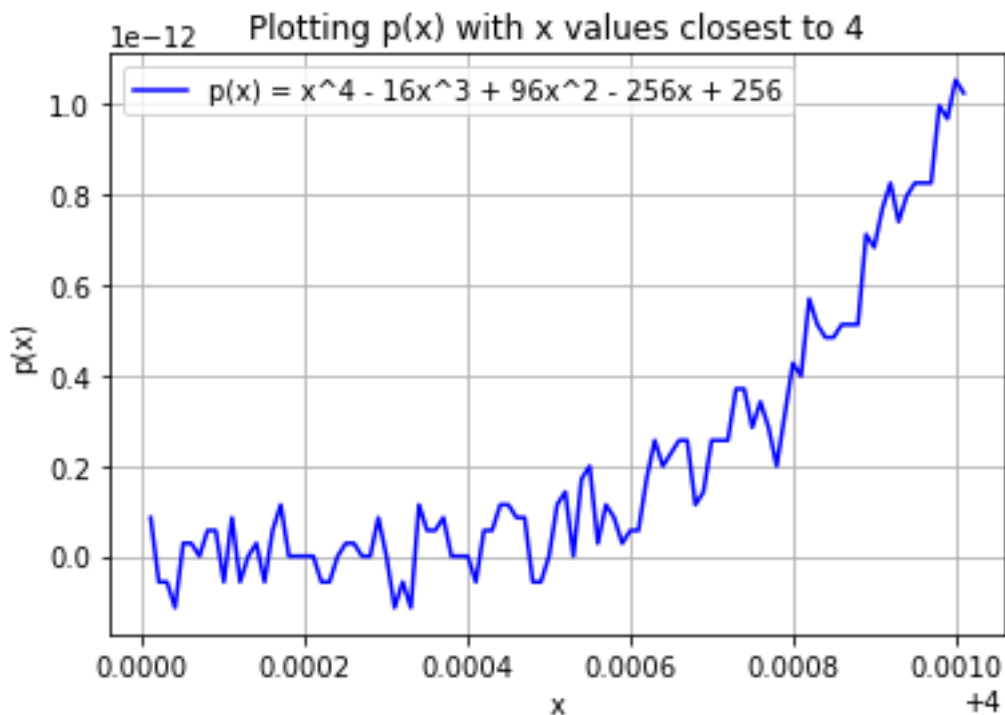
This can be done by defining a variable `p_eval` that will hold the evaluated  $p(x)$ , using `poly_coeff`, at each  $x$ , using `x_values`.

```
'''
Next, we need to evaluate  $p(x)$  at each  $x$ 
and save the values so that we can graph them.
'''
p_eval = np.polyval(poly_coeff, x_values)
```

Lastly, we want to graph  $p(x)$  over  $x$ , which can be done by using `x_values` as our x-axis and `p_eval` as our y-axis.

```
...
Lastly, we need to plot these numbers.
...

plt.plot(x_values, p_eval, label = "p(x) = x^4 - 16x^3 + 96x^2 - 256x + 256", color = "blue")
plt.xlabel("x")
plt.ylabel("p(x)")
plt.title("Plotting p(x) with x values closest to 4")
plt.legend()
plt.grid(True)
plt.show()
```



- B) The process for plotting  $p(x) = (x - 4)^4$  is very similar to part A, except we need to define  $p(x)$ . Doing so will require a method `p_of_x(x)` that will return  $p(x) = (x - 4)^4$ .

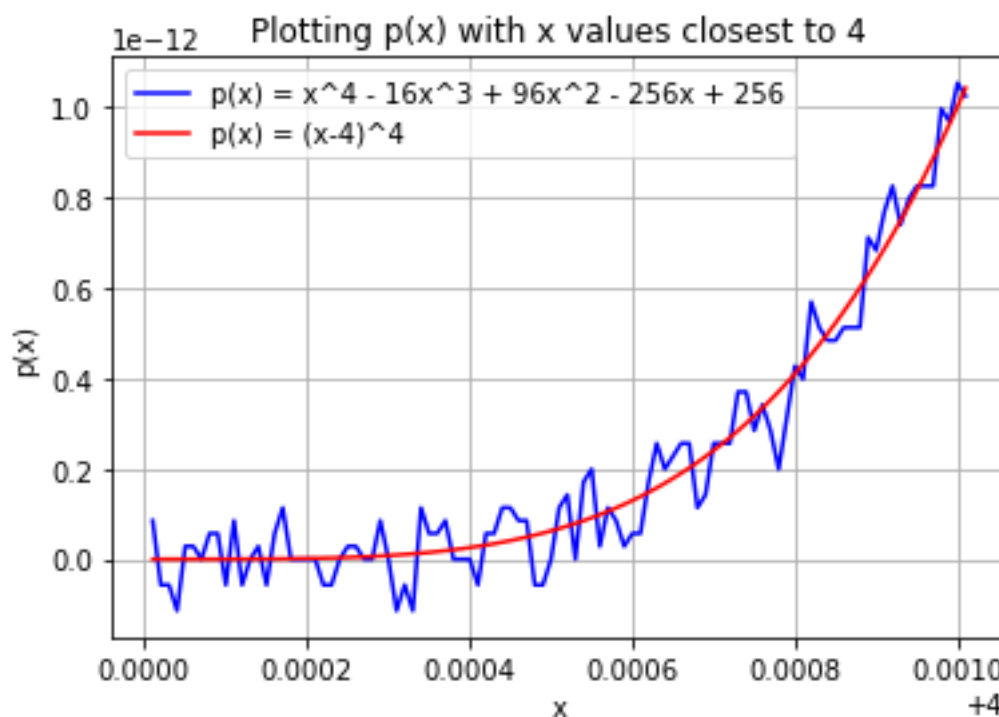
```
...
The function p(x) = (x-4)^4
...
def p_of_x(x):
    return (x-4)**4
```

We can use `x_values` defined in part A and the variable `p_of_x_eval` to evaluate  $p(x)$  at each  $x$ .

```
p_of_x_eval = p_of_x(x_values)
```

Lastly, we can add these values to the graph in part A; `x_values` will be the x-axis and `p_of_x_eval` will be the y-axis.

```
plt.plot(x_values, p_eval, label = "p(x) = x^4 - 16x^3 + 96x^2 - 256x + 256", color = "blue")
plt.plot(x_values, p_of_x_eval, label = "p(x) = (x-4)^4", color = "red")
plt.xlabel("x")
plt.ylabel("p(x)")
plt.title("Plotting p(x) with x values closest to 4")
plt.legend()
plt.grid(True)
plt.show()
```



- C) I observe that the blue graph  $p(x)_1$ , which is graphing  $p(x) = x^4 - 16x^3 + 96x^2 - 256x + 256$ , is way more detailed than the red graph  $p(x)_2$ , which is graphing  $p(x) = (x - 4)^4$ . This is due to the fact that  $p(x)_1$  is more accurate and holds more floating-point numbers than  $p(x)_2$ . The coefficients in  $p(x)_1$  allow for more floating-point numbers to be represented on the graph, as  $x$  is getting precisely evaluated due to the expansion of  $p(x)_2$ .

in  $p(x)_1$ . The lack of precision in  $p(x)_2$  or the small number of floating-point numbers in the result allows for a more general sense of progression of the graph. Though  $p(x)_1$  and  $p(x)_2$  are graphing different values, they still have the same progression of magnitude as the values increase. This is very common in arithmetic, as many floating-point operations involve numbers of similar order of magnitude instead of numbers that vary greatly in size. This is to ensure that we are not using a lot of memory/storage when recreationally computing values. In the instance of this problem, we just needed to see the progression of the graph, so graphing  $p(x)_2$  would have been sufficient enough to see the progression of magnitude. Graphing numbers that vary greatly in size requires the use and storage of more floating-point numbers which is not always needed when computing floating-point operations,

## **Problem 2**

A) For `truncate(x)`, I needed to figure out a way to return the original  $x$  up to three decimal places. To do so, I first multiplied the original number by 1000, which will move the decimal place three places to the right, and cast it as an int, which will drop the remaining numbers after the decimal. Then, I divided this number by 1000 and cast it as a float. This process will now move the decimal three places to the left, but will keep the three decimal places, as float will return a decimal number. I used the example in the assignment to make sure it works.

```
def truncate(x):
    new_int = int(x * 1000) #moves the decimal 3 places to the right and drops the other numbers because this is an int
    return float(new_int/1000) #moves the decimal place 3 to the left, leaving the first three decimal places from the original number
print(truncate(735.32567))
```

735.325

- B) To complete the next portion of this assignment, I followed the following steps:
- Compute the initial stock value of 1400 different stocks between \$0 and \$200
  - Rescale the sum from part a to that the new sum and initial index is 1000
  - For each change in stock, draw a random value between -200 and 200 **cents** and use this change in stock price to compute the new index (there are 2800 stock changes a day). Truncate the new index each time.
  - Plot the evolution over 1 day and over 22 months

To start, I needed to compute the initial stock value of 1400 different stocks. To do so, I created a variable `baseline_sum` to track the sum of these stocks and a method `initial_stock_value()` to compute the stocks and rescale the stocks. First, I calculated the 1400 different stocks with a random value between \$0 and \$200. I then added up these values to represent `baseline_sum`. Then I rescaled each of the 1400 prices so that their new sum was 1000. Then I initialized two variables; `initial_index` which is used to track the initial stock value (which is 1000) and `old_index` which will be used in part c.

```
baseline_sum = 0.0 #sum of the 1400 stocks

def initial_stock_value():
    global baseline_sum
    stocks = [random.randint(0,200) for _ in range(1400)]
    baseline_sum = float(sum(stocks))

    scale = 1000/baseline_sum #scaling number used to rescale the stocks to 1000

    price_scaled = [price * scale for price in stocks]
    #print(baseline_sum)

    return sum(price_scaled);

initial_index = initial_stock_value() #initial index = 1000
old_index = initial_index
```

To calculate the new index, I used method `calculate_change()`. To start, I calculated the stock change by randomly selecting a value between -200 and 200 **cents**.

However, since the units of the stock are in dollars, the change in stock will be a randomly selected float between -2.00 and 2.00. After receiving the `stock_change` value, I plugged that value into the equation given from the assignment, which was

$$\text{New Index} = \text{Old Index} + (\text{Change in Stock Price}) * \frac{1000}{\text{Baseline Sum}}$$

Then, I reassign

`old_index` to be `new_index` to prepare for the next iteration of this method. Lastly, I truncate and return the index.

```
def calculate_change():
    global old_index
    stock_change = random.uniform(-2.00,2.00) #stock change of [-200,200] cents which is equivalent to [-2.00,2.00] dollars
    new_index = old_index + (stock_change) * (1000/baseline_sum) #equation given in assignment
    old_index = new_index #update the old index for the next stock change

    trunc_x = truncate(old_index) #truncate the old index
    old_index = trunc_x #update with the truncated old index

    return trunc_x
```

To plot the evolution over 1 day and 22 months, I first created lists to hold the daily and 22-month stock changes, `daily_stock_changes` and `tt_month_stock_changes`.

```
daily_stock_changes = [1000] #list of the 2800 stock changes a day
tt_month_stock_changes = [1000] #list of the daily stock changes over 22 months
```

Then I created methods that will imitate the stock changes to make it easier to graph, `daily_evolution()` and `tt_month_evolution()`. For `daily_evolution()`, I called `calculate_change()` 2800 times, as there are 2800 changes daily.

```
'''
The following method imitates the 2800 a day stock changes
'''
def daily_evolution():
    for i in range (2799):
        daily_stock_changes.append(calculate_change())

    return;
```

For `tt_month_evolution()`, I first had to define a variable to compute the changes over 22 months.

```
tt_months = 2800 * 30 * 22
```

Then, I called `calculate_change()` `tt_months` times.

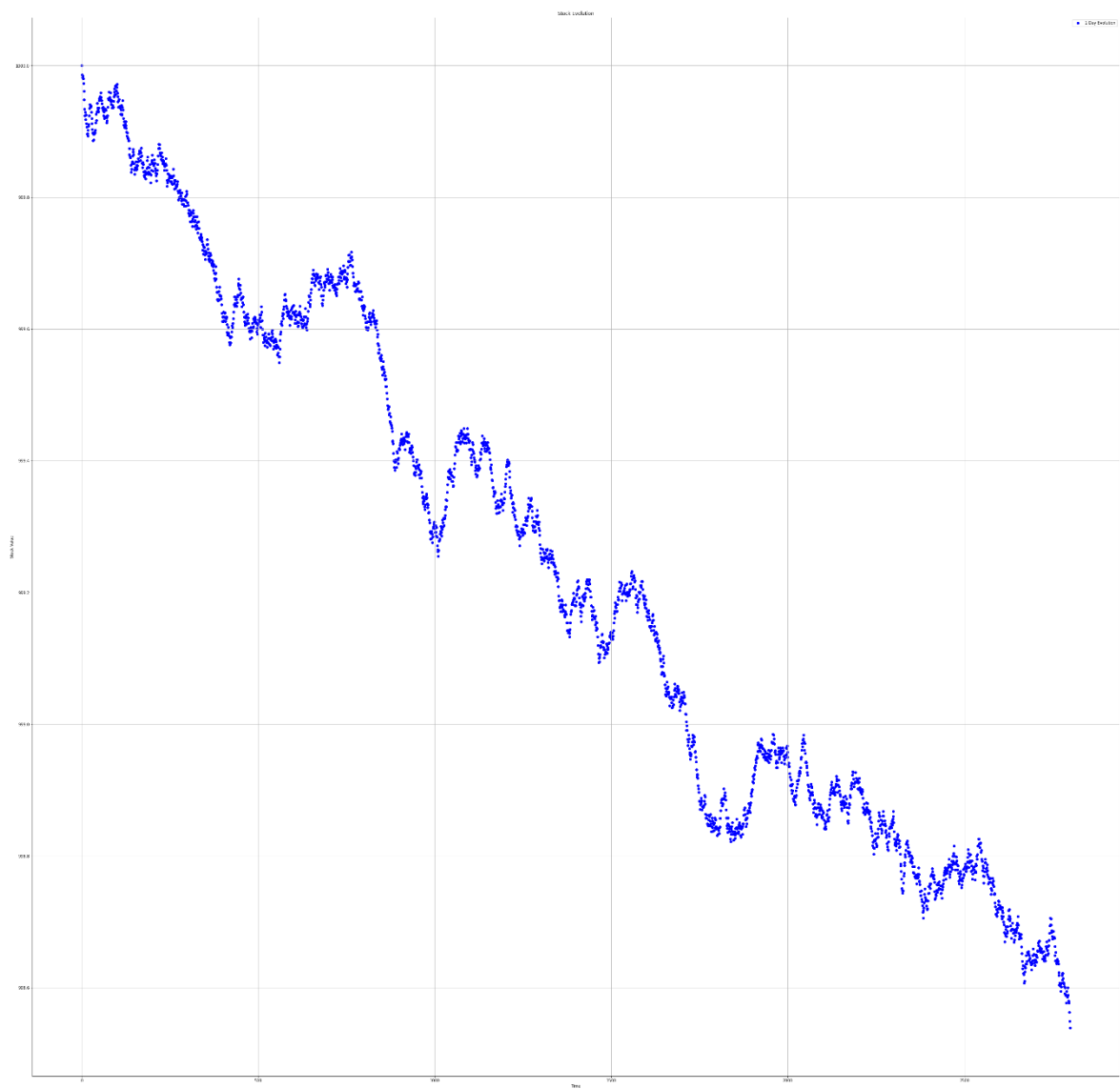
```
'''
The following method imitates the stock changes over 22 months
'''
def tt_month_evolution():
    for i in range(tt_months-1):
        tt_month_stock_changes.append(calculate_change())

    return;
```

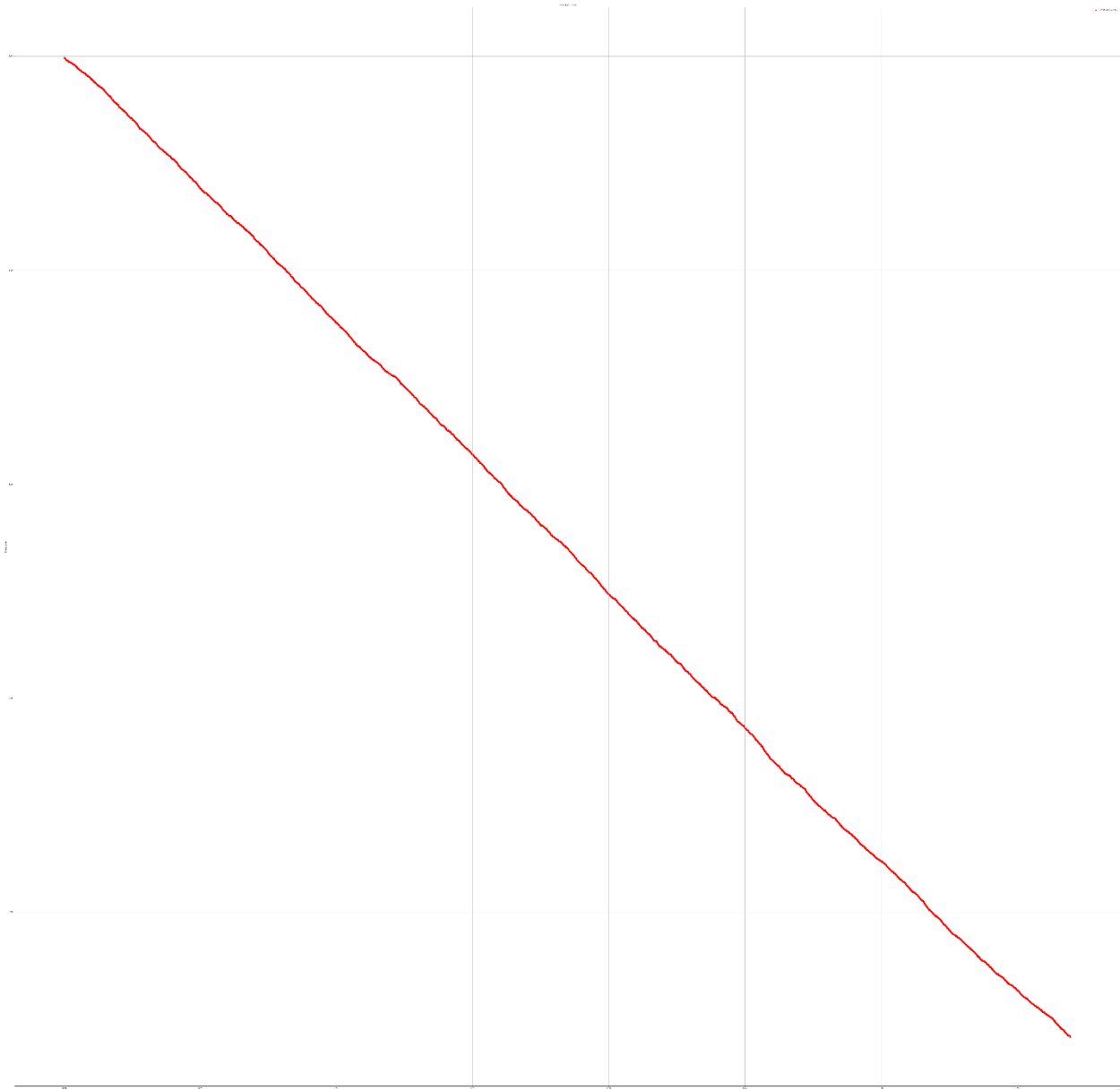
Lastly, I graphed both daily stock changes and the stock changes over 22 months.

```
plt.figure(figsize=(100,100))
plt.scatter(range(0,2800), daily_stock_changes,label = "1 Day Evolution", marker = "o", color = "blue")
plt.scatter(range(0,tt_months), tt_month_stock_changes,label = "22 Month Evolution", marker = "o", color = "red")
plt.xlabel("Time")
plt.ylabel("Stock Value")
plt.title("Stock Evolution")
plt.legend()
plt.grid(True)
plt.show()
```

Changes over one day:



Changes over 22 months:



C) I would expect on average the index to drop around .5 to 1 points a day and about 2-3 points in 22 months.

D) To modify my truncate function, I would round to 3 decimal places instead of truncating the number. To do so, I would make a new method `round_to_three(x)` that will round `x` to 3 decimal places. Again, I used the example value from the assignment to ensure that this method works.

```
def round_to_three(x):
    return round(x,3)
print(round_to_three(735.32567))
```

735.326

To graph the new evolutions, I replaced the `truncate(x)` function with the `round_to_three(x)` in the `calculate_month()` method.

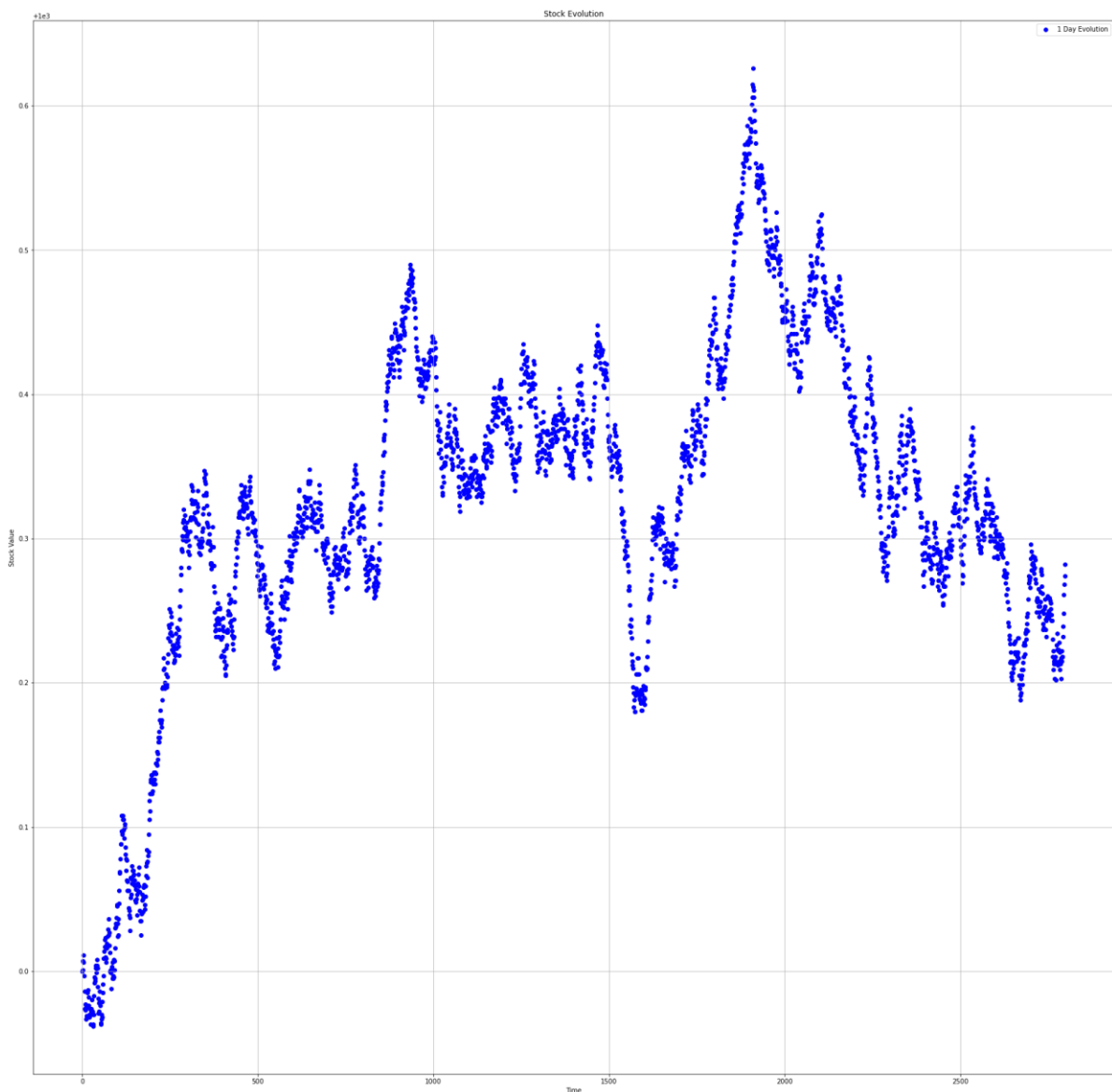
```
def calculate_change():
    global old_index
    stock_change = random.uniform(-2.00,2.00) #stock change of [-2.00,2.00] cents which is equivalent to [-2.00,2.00] dollars
    new_index = old_index + (stock_change) * (1000/baseline_sum) #equation given in assignment
    old_index = new_index #update the old index for the next stock change

    rounded = round_to_three(old_index) #truncate the old index
    old_index = rounded #update with the truncated old index

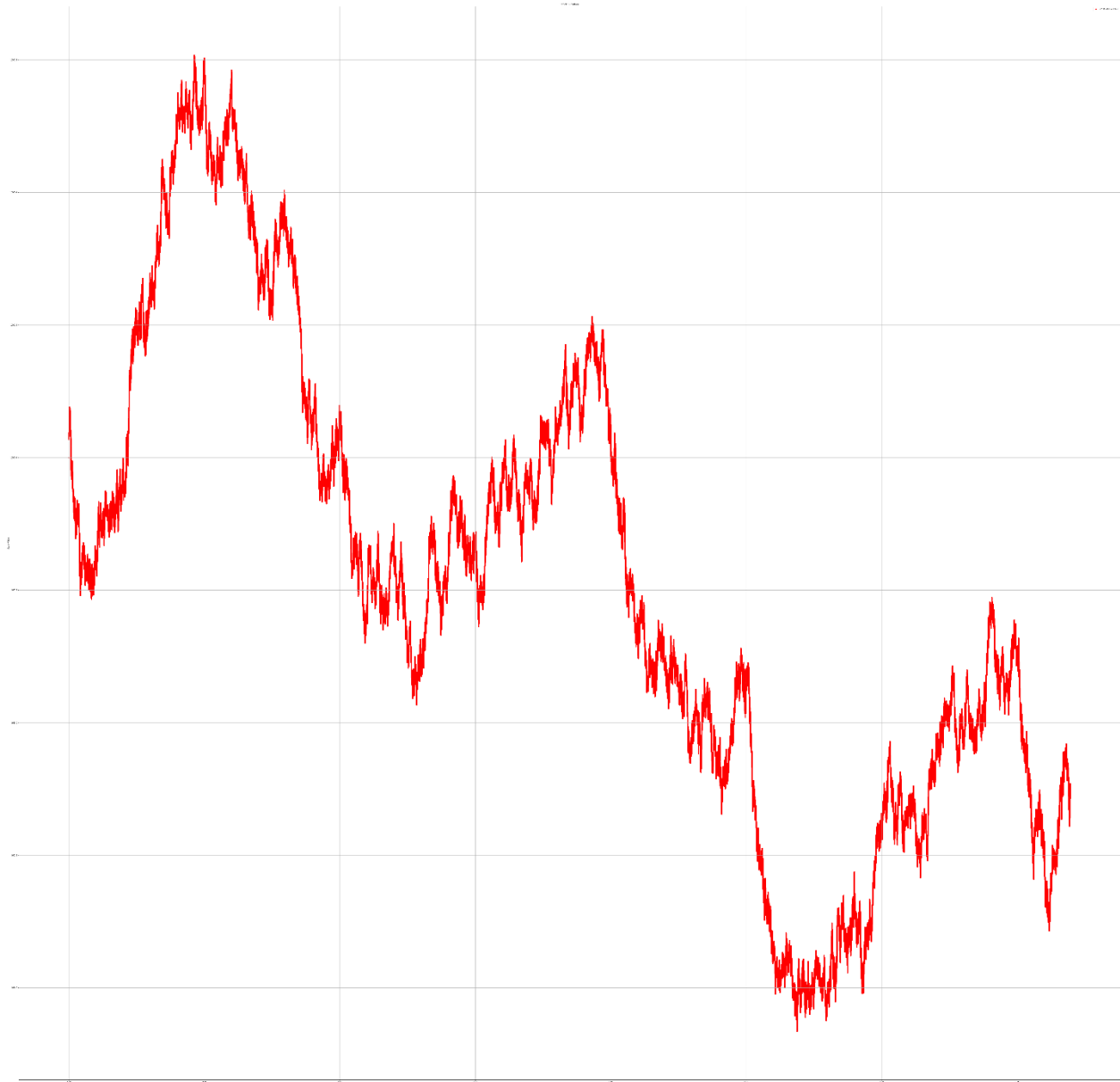
    return rounded
```

These are the new graphs for daily evolution and 22-month evolution.

Daily evolution:



22-month evolution:



These graphs show more detail and show a different magnitude of evolution. Instead of a steady decline, the values on the graphs fluctuate many times over their allotted periods of evolution.

- E) The evolution of the market during this time was a bear market, which is a market in which the stock value decreases. This is a bear market because as seen in the graphs, the evolution of the stock market over one day and 22 months declines, causing the stock prices to decline.

