Modeling the S & P 500 With Various Methods

Andrew Struthers

March 2022

1 Introduction

The stock market is frequently targeted by mathematicians and non-mathematicians alike. Most people's goal in life is to make a bunch of money, and the stock market provides that glimmer of hope a lot of us need. Many have tried to model the stock market to try and get an advantage in buying or selling of indexes to increase their overall portfolio value. These models find some level of success, but most of the time they'd be better off just buying a share of something and forgetting about it for years.

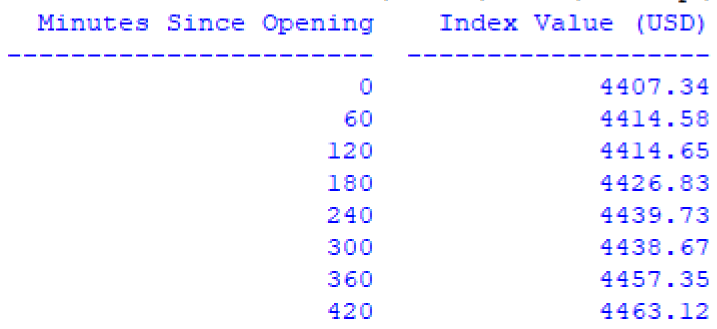
Luckily for me, I haven't learned from any of their lessons. I will be attempting to model then S & P 500, one of the most famous indexes on the stock market. The S & P 500 is a portfolio of the top 500 performing companies on the publically traded market, tied into one index. The S & P 500 value is a pretty good indication of the market and the economy as a whole, so many models try and take a stab at predicting this. My dataset comes from a Python API called YFinance. YFinance is an API built off of the old Yahoo Finance tool that was widely popular and used in a number of applications. I will be taking a sample of the price of the S & P 500 once every hour, each hour, for the duration of an arbitrary day. I will then attempt to fit a curve to this and predict the next hour's price. This project will start, however, by trying to fit various models to the S & P 500 prices and seeing if any of them actually represent the index. The dataset consists of 8 evenly spaced values from the hourly price of the S & P 500. I am using the arbitrary date of March 17th, 2022 to gather data. Since the data I am gathering is the culmination of the entire stock value at the end of each hour, the data should be spread out enough to get some meaningful variation, as well as providing me enough space between datapoints to see if my approximations are accurate throughout an hour.

The functional relationship between the domain and range is the value of the index changes as a function of time. There are many other factors that actually determine the value of an index, but for the sake of this model the analyzed relationship is strictly time in minutes since the market opened as the independent variable and price in USD as the dependent variable. Numerical differentiation of this dataset will tell us the rate in which the value of the stock increased or decreased throughout the hour between consecutive datapoints.

I look forward to hopefully building a model that can accurately predict the value of the S & P 500 at a given point throughout the day. This outcome is rather unlikely, so another takeaway from making this model will be building code to do analysis on real world data with the goal of furthering my code beyond this project to build a model throughout the day on live data. If I can achieve either one of these goals, I will be very happy with this project. I am attempting to create a symbolic representation of the function through the known values using the Divided Difference method to form a Lagrange polynomial. Since this is an interpolating polynomial, it will be continuous and differentiable on the known interval. I should be able to take this polynomial and find the symbolic derivative of it, as well as compute values using the derivatives.

2 Analysis

As stated above, I will build a Lagrange Polynomial based off of 8 evenly spaced points shown in Figure 1: These points are the points I will be using to build the Lagrange Polynomial, but for



```
Minutes Since Opening    Index Value (USD)
------------------------  --------------------
                       0              4407.34
                      60              4414.58
                     120              4414.65
                     180              4426.83
                     240              4439.73
                     300              4438.67
                     360              4457.35
                     420              4463.12
```

Figure 1: Raw x and y Values from Market

the Taylor approximation I will be using $x_0 = 240$ because that is the centerpoint of my data, and from Lab 4 we found that using the centerpoint of the known data for a Taylor Polynomial provides more accurate information on the known bounds. I will also be using $x_0 = 360$ to see if

I can build an approximation from the second to last hour before closing that accurately predicts the value in an hour. This will be my proof of concept that my model can predict an hour in advanced, which will be the basis of my real-time prediction code. I have concerns about rounding error and using too many datapoints to the point of my computer not being able to keep up with the computations. Python isn't the fastest computational language to exist, so I am worried that with all these datapoints (or more if I shorten the interval) that I will be asking Python to do too much. Some limitations of my two estimates will be the ability to predict points too far in the future, since both approximation methods have been known to be inaccurate over large x values outside of the known bounds. I will also not be able to predict the next day's values based off of the current day because there is after-market and pre-market trading that goes on.

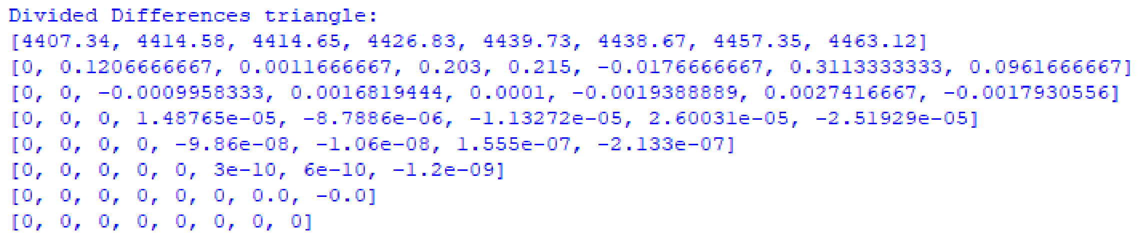
3 Predictions

I predict that, in order for my Taylor approximation to have any shot at being accurate, the Lagrange Polynomial will need to be a true interpolating polynomial on the known datapoints and not suffer from any rounding error. I fear that the rounding limitations Python has set in place will prevent my approximation from being accurate.

4 Computer Program

My derived Lagrange Polynomial ended up being

$$\begin{aligned} L(x) = 4407.34 + 0.121(x) - 0.000996(x)(x-60) + 1.48765e-05(x)(x-60)(x-120) - 9.86e \\ - 08(x)(x-60)(x-120)(x-180) + 3e-10(x)(x-60)(x-120)(x-180)(x-240) \end{aligned} \tag{1}$$



```
Divided Differences triangle:
[4407.34, 4414.58, 4414.65, 4426.83, 4439.73, 4438.67, 4457.35, 4463.12]
[0, 0.1206666667, 0.0011666667, 0.203, 0.215, -0.0176666667, 0.3113333333, 0.0961666667]
[0, 0, -0.0009958333, 0.0016819444, 0.0001, -0.0019388889, 0.0027416667, -0.0017930556]
[0, 0, 0, 1.48765e-05, -8.7886e-06, -1.13272e-05, 2.60031e-05, -2.51929e-05]
[0, 0, 0, 0, -9.86e-08, -1.06e-08, 1.555e-07, -2.133e-07]
[0, 0, 0, 0, 0, 3e-10, 6e-10, -1.2e-09]
[0, 0, 0, 0, 0, 0, 0.0, -0.0]
[0, 0, 0, 0, 0, 0, 0, 0]
```

Figure 2: Divided Difference Graph

This isn't the full form, because a lot of the terms ended up being multiplied by 0 due to rounding error in the computer. The divided differences triangle can be seen in Figure 2. It can be seen that the last few terms were 0, which should not have happened. This ended up with a divided difference plot that looked like Figure 3. It can be seen that it doesn't accurately hit the last three points,

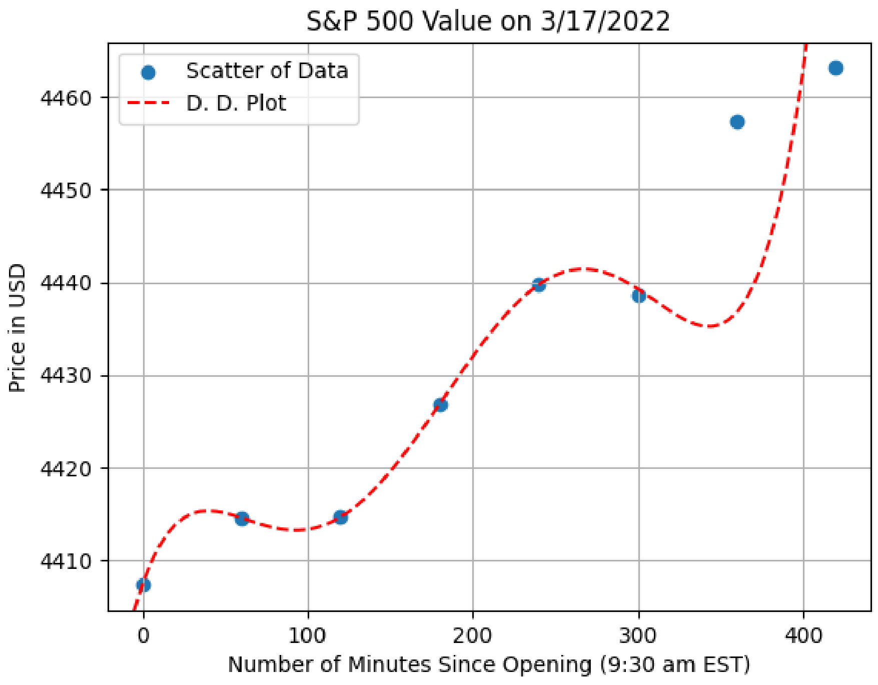
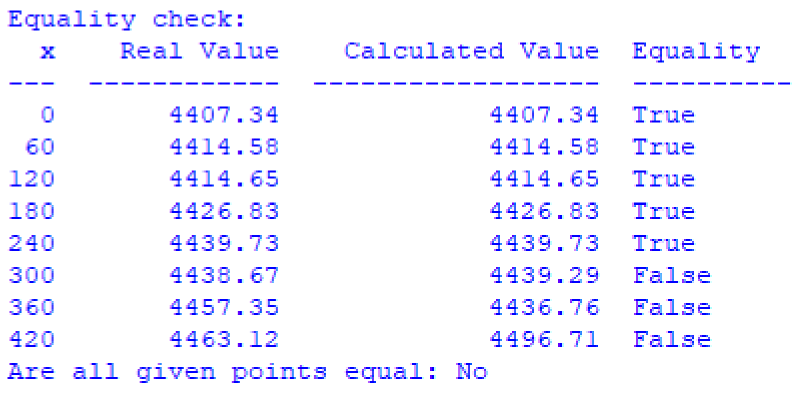


Figure 3: Divided Difference Graph

further shown in Figure 4. This discrepancy is most likely due to the round-off error mentioned and visible in Figure 2.



```
Equality check:
  x    Real Value    Calculated Value   Equality
---  ------------  ------------------  ----------
  0       4407.34             4407.34  True
 60       4414.58             4414.58  True
120       4414.65             4414.65  True
180       4426.83             4426.83  True
240       4439.73             4439.73  True
300       4438.67             4439.29  False
360       4457.35             4436.76  False
420       4463.12             4496.71  False
Are all given points equal: No
```

Figure 4: Divided Difference Check

Even though the Lagrange Polynomial wasn't perfectly complete, it still interpolated the majority of the given points. Using this polynomial, I formed a 3rd order and 5th order Taylor polynomial at $x_0 = 240$ and $x_0 = 360$. As seen in Figure 5, each third order polynomial was fairly accurate at estimating the function around the given x_0 value, but did not predict far into the past or future very accurately. The only exception was the 3rd order Taylors with $x_0 = 360$. This matched the

Lagrange Polynomial almost perfectly after $x = 360$, but because the Lagrange wasn't accurate at this point due to rounding errors, it wasn't the most meaningful result.

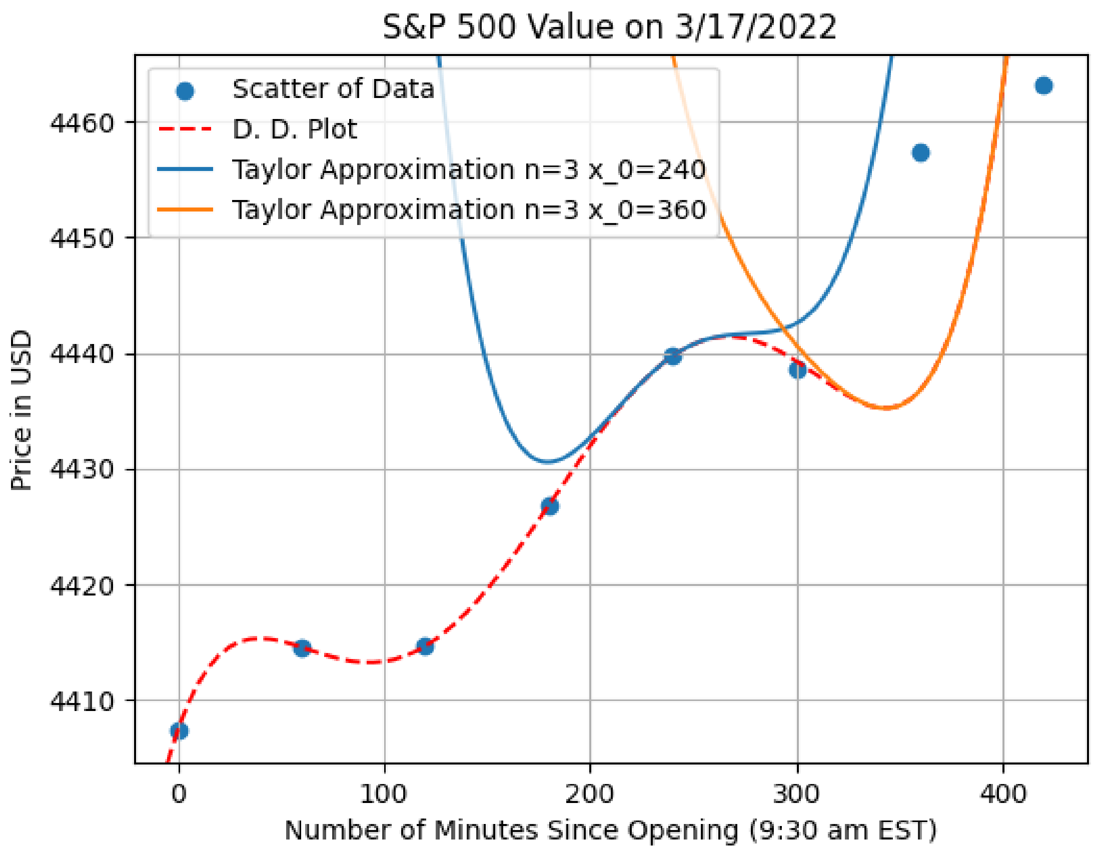


Figure 5: Taylor Polynomial $n = 3$

An unexpected thing happened when I formed the Taylor Polynomial of order 5. As seen in Figure 6, regardless of the x_0 value give, the Taylor Polynomial ended up perfectly matching the Lagrange Polynomial. I do not know why this behaviour is happening, my only guess is that since I formed the Taylor based off of the Lagrange, and the Lagrange was only accurate at five of the eight points given, the order of polynomial required more information than the Lagrange polynomial had to give. This doesn't make too much sense, but ultimately the result was useless compared to just using the Lagrange.

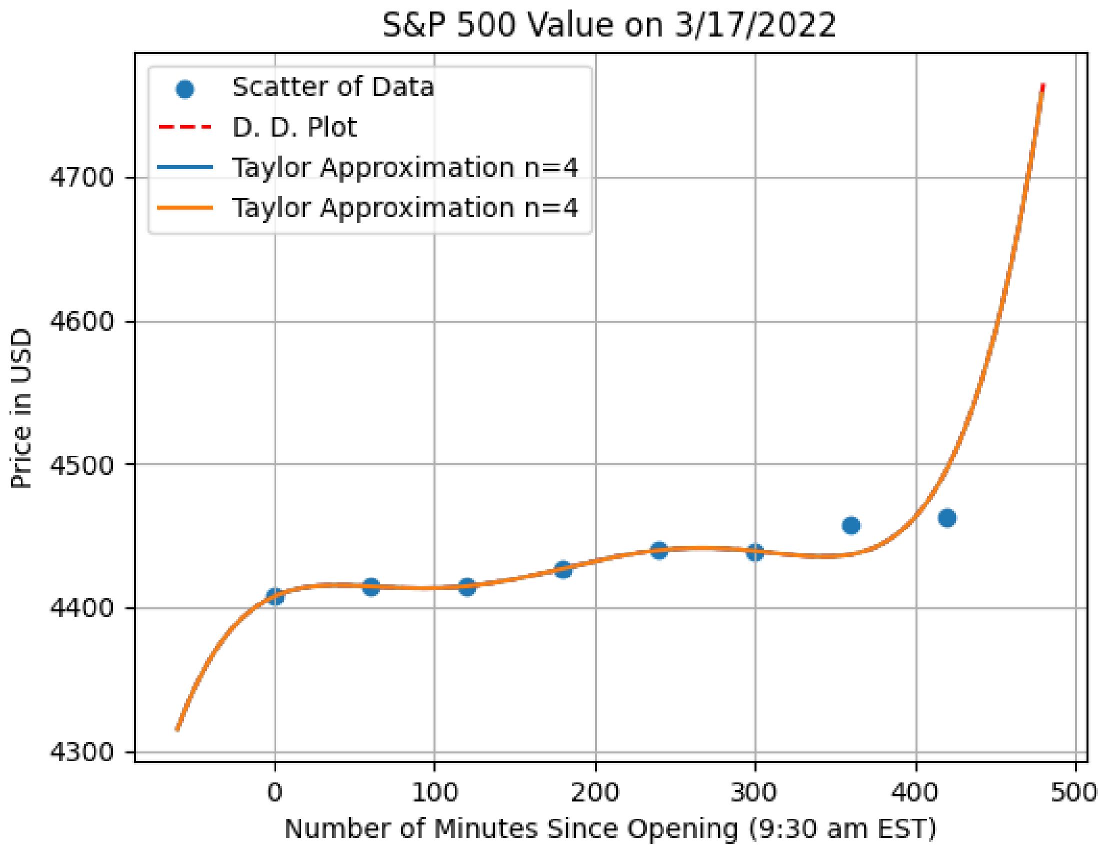
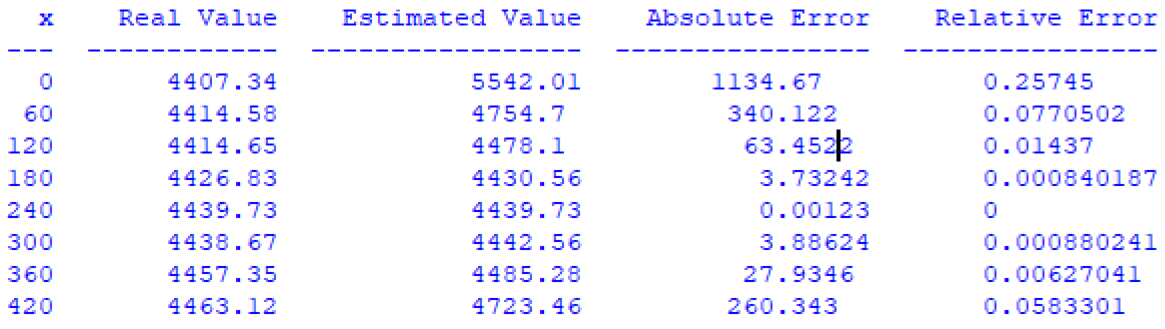


Figure 6: Taylor Polynomial $n = 3$

5 Results

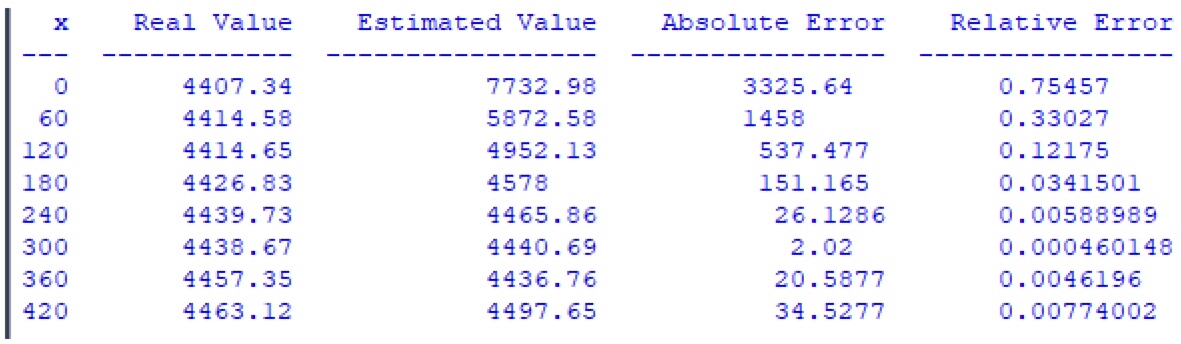
The approximation results for the Lagrange Polynomial can be seen in Figure 4. In Figure 7 and 8, the calculated error for the given values with both Taylor approximations can be found.



```
  x    Real Value    Estimated Value    Absolute Error    Relative Error
---  ------------  -----------------  ----------------  ----------------
  0       4407.34            5542.01        1134.67         0.25745
 60       4414.58            4754.7          340.122        0.0770502
120       4414.65            4478.1           63.4522       0.01437
180       4426.83            4430.56           3.73242      0.000840187
240       4439.73            4439.73           0.00123      0
300       4438.67            4442.56           3.88624      0.000880241
360       4457.35            4485.28          27.9346       0.00627041
420       4463.12            4723.46         260.343        0.0583301
```

Figure 7: 3rd Order Taylor Approximation with $x_0 = 240$

Since the goal was to try and predict the stock value at $x = 420$, we can analyze each prediction at that point. For the Taylor centered at $x_0 = 240$, the real value was $4463.12 and the approximation was $4723.46, giving a relative error of 5.8%. This seems pretty good, especially



```
  x     Real Value     Estimated Value     Absolute Error     Relative Error
---   ------------   -----------------   ----------------   ----------------
  0        4407.34             7732.98          3325.64           0.75457
 60        4414.58             5872.58          1458              0.33027
120        4414.65             4952.13            537.477         0.12175
180        4426.83             4578               151.165         0.0341501
240        4439.73             4465.86             26.1286        0.00588989
300        4438.67             4440.69              2.02          0.000460148
360        4457.35             4436.76             20.5877        0.0046196
420        4463.12             4497.65             34.5277        0.00774002
```

Figure 8: 3rd Order Taylor Approximation with $x_0 = 360$

considering how far away the x_0 value was. Looking at the rest of the approximation, we can see that the maximum error throughout the estimation was at $x = 0$, relatively 26%. Compared to the average error of the estimate centered at $x_0 = 360$, we can see that the estimate from the midpoint of the dataset was indeed more accurate at predicting the function as a whole, like I predicted. However, looking explicitly at $x = 420$, we can see that the relative error of the polynomial centered at $x_0 = 360$ was only 0.77%.

With these observations from the Taylor polynomial, the proof of concept for predicting the stock value one hour in advanced using the most recent point in the dataset seems to hold. This is exciting news because I can now form the basis for a real-time stock analyzer that is within 1% accurate, at least as far as this sample data is concerned. I can now start working towards automating a buying and selling algorithm that takes the Taylor polynomial at a given time and predicts what the value will be in an hour. If the predicted value is favorable, I would probably want to buy. If the predicted value is not favorable, I can sell. After a transaction takes place, I can gather more data and rebuild the model for the next hour.

Moving on, we can look at at the numerical derivative estimation compared to the actual Lagrange derivative, we can see some interesting results. This is the Lagrange derivative plotted at the known values. With this we can see the average rate of change over each hour in our dataset, which is interesting because that number is the percent the stock value increased over the one hour interval. We can also compare this to the numerical approximations of the derivative seen in Figure 10:

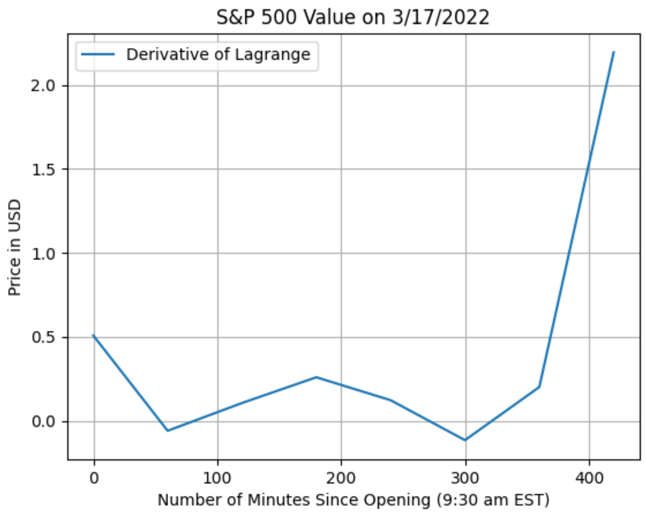


Figure 9: Lagrange Derivative

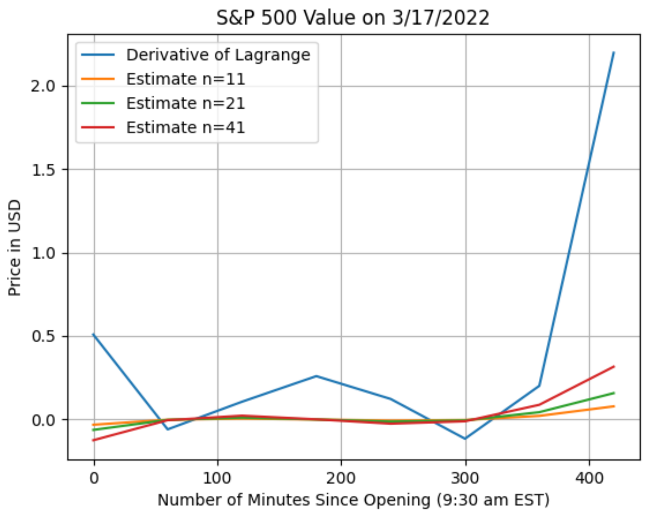


Figure 10: Numerical Derivatives

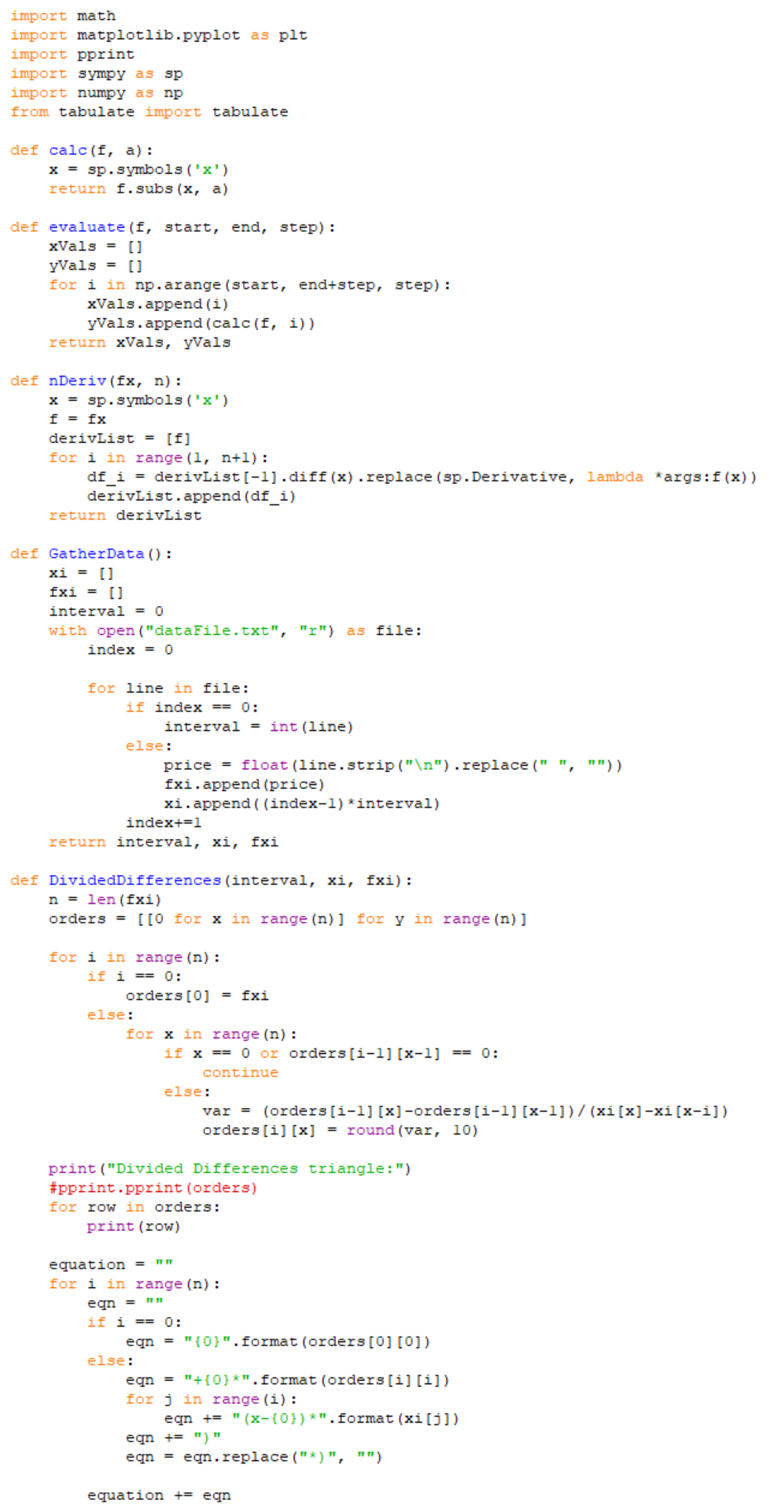
The numerical derivatives honestly give a better curve for the derivative than the plotted Lagrange derivative. Doing error analysis, we can see that the estimation with $n = 41$ was the most accurate because it had the smallest average error compared to the Lagrange polynomial. Regardless, they all follow generally the same trend, except around $x = 0$. The Lagrange derivative is decreasing, whereas the numerical approximations are increasing.

I believe a lot of the results don't make absolute sense due to the Lagrange Polynomial not being a true interpolating polynomial on all the points given. The roundoff error that messed with the initial approximation persisted throughout the entire modeling process, leading to some unexpected results.

With the combination of the Lagrange derivative and the Taylor polynomial, I can really dial in a predictive model. The Lagrange derivative will tell me the rate of change between the known datapoints, so I can analyze the derivative to know whether the rate of the value of the index is increasing or not. This is particularly useful for identifying spikes or large drops. If the derivative is increasing and the value is increasing, then I can predict the value in the next hour is going to skyrocket and I should definitely be buying. Both of these approximations will help me to build a predictive stock buying and selling machine.

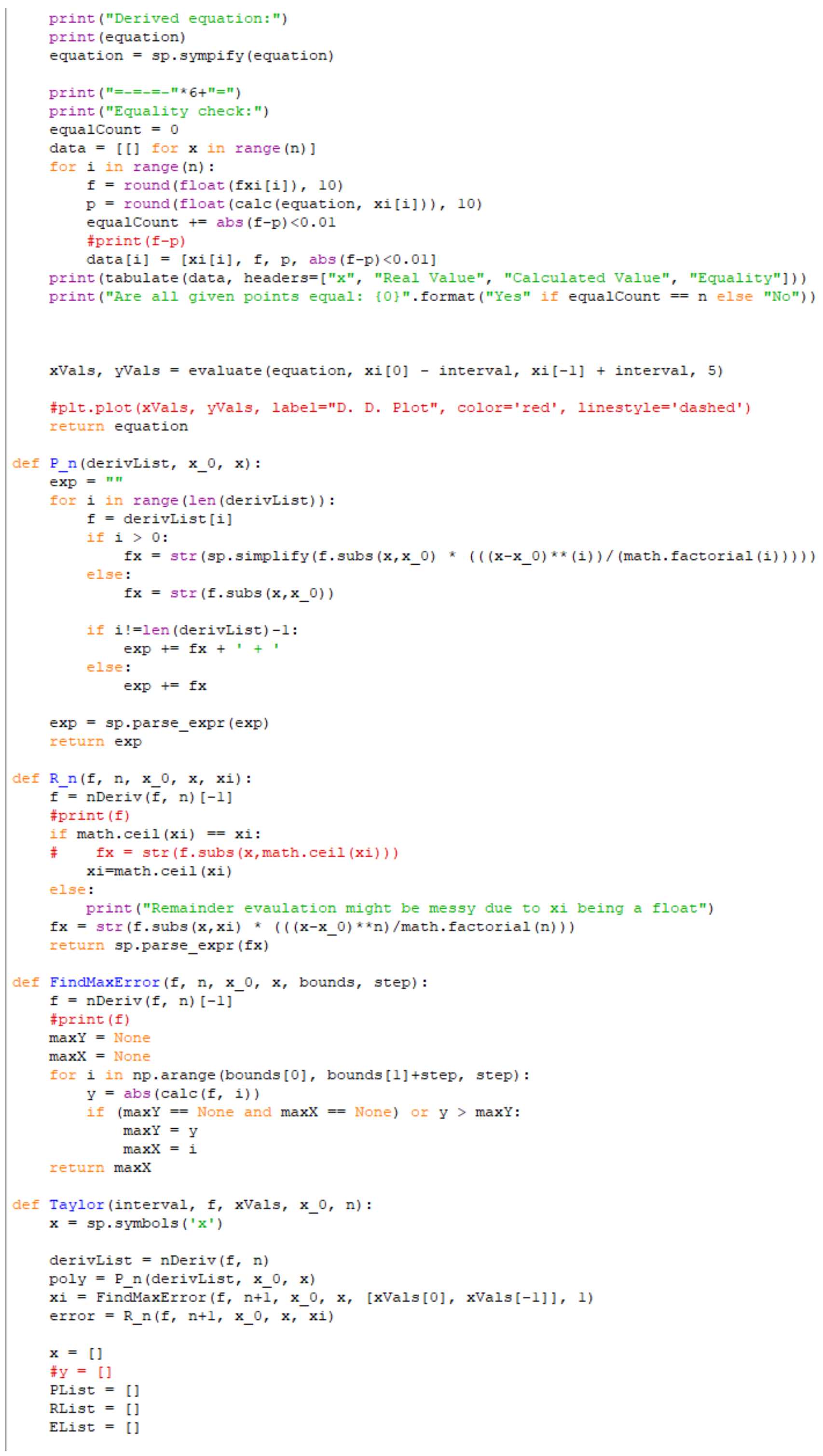
6 Code

Below are screenshots of the code used to run these calculations.



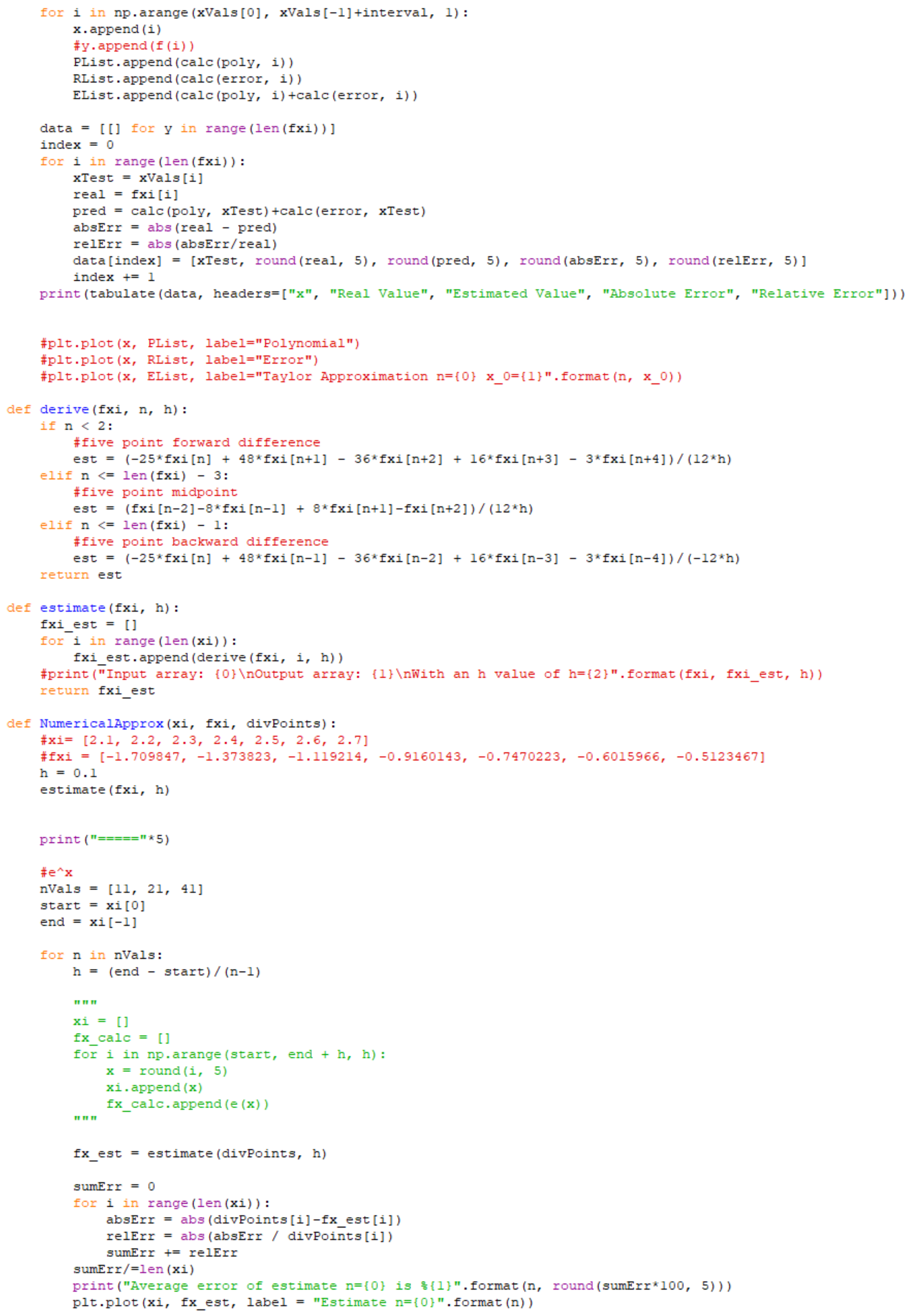
```python
import math
import matplotlib.pyplot as plt
import pprint
import sympy as sp
import numpy as np
from tabulate import tabulate

def calc(f, a):
    x = sp.symbols('x')
    return f.subs(x, a)

def evaluate(f, start, end, step):
    xVals = []
    yVals = []
    for i in np.arange(start, end+step, step):
        xVals.append(i)
        yVals.append(calc(f, i))
    return xVals, yVals

def nDeriv(fx, n):
    x = sp.symbols('x')
    f = fx
    derivList = [f]
    for i in range(1, n+1):
        df_i = derivList[-1].diff(x).replace(sp.Derivative, lambda *args:f(x))
        derivList.append(df_i)
    return derivList

def GatherData():
    xi = []
    fxi = []
    interval = 0
    with open("dataFile.txt", "r") as file:
        index = 0

        for line in file:
            if index == 0:
                interval = int(line)
            else:
                price = float(line.strip("\n").replace(" ", ""))
                fxi.append(price)
                xi.append((index-1)*interval)
            index+=1
    return interval, xi, fxi

def DividedDifferences(interval, xi, fxi):
    n = len(fxi)
    orders = [[0 for x in range(n)] for y in range(n)]

    for i in range(n):
        if i == 0:
            orders[0] = fxi
        else:
            for x in range(n):
                if x == 0 or orders[i-1][x-1] == 0:
                    continue
                else:
                    var = (orders[i-1][x]-orders[i-1][x-1])/(xi[x]-xi[x-i])
                    orders[i][x] = round(var, 10)

    print("Divided Differences triangle:")
    #pprint.pprint(orders)
    for row in orders:
        print(row)

    equation = ""
    for i in range(n):
        eqn = ""
        if i == 0:
            eqn = "{0}".format(orders[0][0])
        else:
            eqn = "+{0}*".format(orders[i][i])
            for j in range(i):
                eqn += "(x-{0})*".format(xi[j])
            eqn += ")"
            eqn = eqn.replace("*)", "")

        equation += eqn
```

Figure 11: Python Code 1



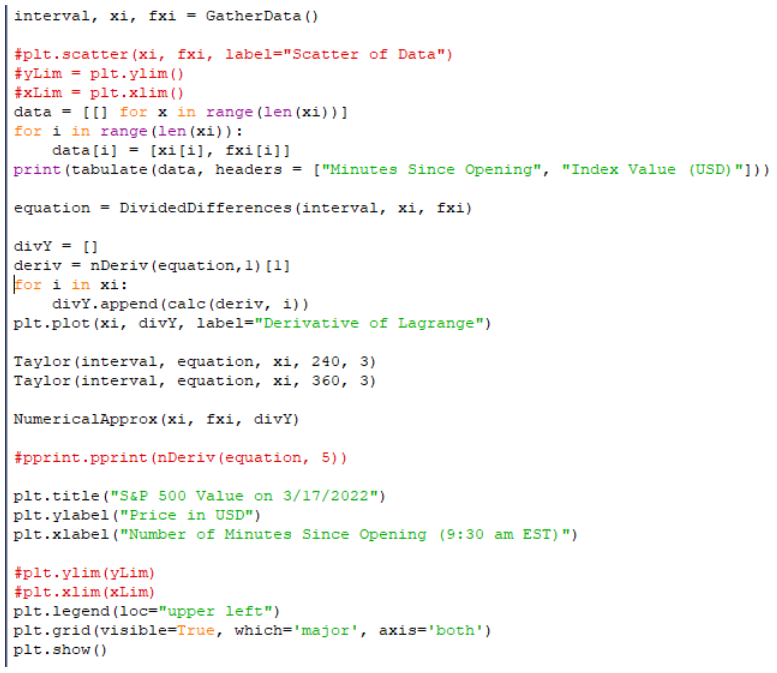
```python
    print("Derived equation:")
    print(equation)
    equation = sp.sympify(equation)

    print("=-=-=-"*6+"=")
    print("Equality check:")
    equalCount = 0
    data = [[] for x in range(n)]
    for i in range(n):
        f = round(float(fxi[i]), 10)
        p = round(float(calc(equation, xi[i])), 10)
        equalCount += abs(f-p)<0.01
        #print(f-p)
        data[i] = [xi[i], f, p, abs(f-p)<0.01]
    print(tabulate(data, headers=["x", "Real Value", "Calculated Value", "Equality"]))
    print("Are all given points equal: {0}".format("Yes" if equalCount == n else "No"))



    xVals, yVals = evaluate(equation, xi[0] - interval, xi[-1] + interval, 5)

    #plt.plot(xVals, yVals, label="D. D. Plot", color='red', linestyle='dashed')
    return equation

def P_n(derivList, x_0, x):
    exp = ""
    for i in range(len(derivList)):
        f = derivList[i]
        if i > 0:
            fx = str(sp.simplify(f.subs(x,x_0) * (((x-x_0)**(i))/(math.factorial(i)))))
        else:
            fx = str(f.subs(x,x_0))

        if i!=len(derivList)-1:
            exp += fx + ' + '
        else:
            exp += fx

    exp = sp.parse_expr(exp)
    return exp

def R_n(f, n, x_0, x, xi):
    f = nDeriv(f, n)[-1]
    #print(f)
    if math.ceil(xi) == xi:
    #    fx = str(f.subs(x,math.ceil(xi)))
        xi=math.ceil(xi)
    else:
        print("Remainder evaulation might be messy due to xi being a float")
    fx = str(f.subs(x,xi) * (((x-x_0)**n)/math.factorial(n)))
    return sp.parse_expr(fx)

def FindMaxError(f, n, x_0, x, bounds, step):
    f = nDeriv(f, n)[-1]
    #print(f)
    maxY = None
    maxX = None
    for i in np.arange(bounds[0], bounds[1]+step, step):
        y = abs(calc(f, i))
        if (maxY == None and maxX == None) or y > maxY:
            maxY = y
            maxX = i
    return maxX

def Taylor(interval, f, xVals, x_0, n):
    x = sp.symbols('x')

    derivList = nDeriv(f, n)
    poly = P_n(derivList, x_0, x)
    xi = FindMaxError(f, n+1, x_0, x, [xVals[0], xVals[-1]], 1)
    error = R_n(f, n+1, x_0, x, xi)

    x = []
    #y = []
    PList = []
    RList = []
    EList = []
```

Figure 12: Python Code 2



```python
    for i in np.arange(xVals[0], xVals[-1]+interval, 1):
        x.append(i)
        #y.append(f(i))
        PList.append(calc(poly, i))
        RList.append(calc(error, i))
        EList.append(calc(poly, i)+calc(error, i))

    data = [[] for y in range(len(fxi))]
    index = 0
    for i in range(len(fxi)):
        xTest = xVals[i]
        real = fxi[i]
        pred = calc(poly, xTest)+calc(error, xTest)
        absErr = abs(real - pred)
        relErr = abs(absErr/real)
        data[index] = [xTest, round(real, 5), round(pred, 5), round(absErr, 5), round(relErr, 5)]
        index += 1
    print(tabulate(data, headers=["x", "Real Value", "Estimated Value", "Absolute Error", "Relative Error"]))


    #plt.plot(x, PList, label="Polynomial")
    #plt.plot(x, RList, label="Error")
    #plt.plot(x, EList, label="Taylor Approximation n={0} x_0={1}".format(n, x_0))

def derive(fxi, n, h):
    if n < 2:
        #five point forward difference
        est = (-25*fxi[n] + 48*fxi[n+1] - 36*fxi[n+2] + 16*fxi[n+3] - 3*fxi[n+4])/(12*h)
    elif n <= len(fxi) - 3:
        #five point midpoint
        est = (fxi[n-2]-8*fxi[n-1] + 8*fxi[n+1]-fxi[n+2])/(12*h)
    elif n <= len(fxi) - 1:
        #five point backward difference
        est = (-25*fxi[n] + 48*fxi[n-1] - 36*fxi[n-2] + 16*fxi[n-3] - 3*fxi[n-4])/(-12*h)
    return est

def estimate(fxi, h):
    fxi_est = []
    for i in range(len(xi)):
        fxi_est.append(derive(fxi, i, h))
    #print("Input array: {0}\nOutput array: {1}\nWith an h value of h={2}".format(fxi, fxi_est, h))
    return fxi_est

def NumericalApprox(xi, fxi, divPoints):
    #xi= [2.1, 2.2, 2.3, 2.4, 2.5, 2.6, 2.7]
    #fxi = [-1.709847, -1.373823, -1.119214, -0.9160143, -0.7470223, -0.6015966, -0.5123467]
    h = 0.1
    estimate(fxi, h)


    print("====="*5)

    #e^x
    nVals = [11, 21, 41]
    start = xi[0]
    end = xi[-1]

    for n in nVals:
        h = (end - start)/(n-1)

        """
        xi = []
        fx_calc = []
        for i in np.arange(start, end + h, h):
            x = round(i, 5)
            xi.append(x)
            fx_calc.append(e(x))
        """

        fx_est = estimate(divPoints, h)

        sumErr = 0
        for i in range(len(xi)):
            absErr = abs(divPoints[i]-fx_est[i])
            relErr = abs(absErr / divPoints[i])
            sumErr += relErr
        sumErr/=len(xi)
        print("Average error of estimate n={0} is %{1}".format(n, round(sumErr*100, 5)))
        plt.plot(xi, fx_est, label = "Estimate n={0}".format(n))
```

Figure 13: Python Code 3



```
interval, xi, fxi = GatherData()

#plt.scatter(xi, fxi, label="Scatter of Data")
#yLim = plt.ylim()
#xLim = plt.xlim()
data = [[] for x in range(len(xi))]
for i in range(len(xi)):
    data[i] = [xi[i], fxi[i]]
print(tabulate(data, headers = ["Minutes Since Opening", "Index Value (USD)"]))

equation = DividedDifferences(interval, xi, fxi)

divY = []
deriv = nDeriv(equation,1)[1]
for i in xi:
    divY.append(calc(deriv, i))
plt.plot(xi, divY, label="Derivative of Lagrange")

Taylor(interval, equation, xi, 240, 3)
Taylor(interval, equation, xi, 360, 3)

NumericalApprox(xi, fxi, divY)

#pprint.pprint(nDeriv(equation, 5))

plt.title("S&P 500 Value on 3/17/2022")
plt.ylabel("Price in USD")
plt.xlabel("Number of Minutes Since Opening (9:30 am EST)")

#plt.ylim(yLim)
#plt.xlim(xLim)
plt.legend(loc="upper left")
plt.grid(visible=True, which='major', axis='both')
plt.show()
```

Figure 14: Python Code 4

References

R.L. Burden and J.D. Faires. *Numerical Analysis*. Cengage Learning, 2010. ISBN 9781133169338. URL https://books.google.com/books?id=Dbw8AAAAQBAJ.

J.F. Epperson. *An Introduction to Numerical Methods and Analysis*. Wiley, 2013. ISBN 9781118626238. URL https://books.google.com/books?id=01U5W5hzvCoC.