Predicting Stock Price Using Brownian Motion

Dennis Jonathan/BM2019B/23101910027

Brownian Motion is a type of movement which was first observed by physician by the name of Robert Brown in the year of 1827. Britanica stated that the idea of **Brownian Motion** in physics describes particles that are placed in a given medium which has no preferred direction for random oscillation, over a period of time, the particles will be spread evenly in that medium. Essentially, particles which has **Brownian Motion** has a tendency to undergo some small and randomly occuring fluctuations.

That brings us to the world of mathematics, and to be exact, the world of simulations. The characteristic of this motion turns out to be incredibly useful for simulations, especially the random movement aspect of it. **Brownian Motion** is defined as the existance of a probability distribution over a set of continuous function $B : \mathbb{R}_{\geq 0} \to \mathbb{R}$ such that:

1. $B(0) = 0$
2. Stationary for $0 \leq s \leq t$, $B(t) - B(s) \sim N(0, t - s)$
3. Independent increment

For this exercise, we will be using two variants of the motion, the first is called as **Brownian Motion with Drift** and the second is called as **Geometric Brownian Motion**. Both simulations will be used on a stock of our choice, for now we will be simulating the **Closing Price** of Apple.inc with ticker **AAPL**. Our data will be pulled straight from Yahoo Finance.

1 Pre-Requisites

1.1 Installing Some Libraries

First of all, we will be installing some libraries which were not available on **Deepnote** (the site which I used to make this notebook), which are `openpyxl` and `xlrd`. We will also need to install `pandas_datareader` to pull our stock prices from the afore mentioned source. To install a library in Python, one need to use `!pip install *name*`. If you are using a **Jupyter Notebook**, there is absolutely no need to install the first two libraries.

[1]:
```
# Installing libraries
!pip install openpyxl
!pip install xlrd
!pip install pandas_datareader
```

1.2 Importing the Necessary Libraries

After installing the libraries, we will need to import them to our notebook. In general, there are four kinds of libraries we will be using, those are: - Libraries for data processing and mathematics (`pandas` and `numpy`) - Libraries to scrape the data (`pandas_datareader`) - Libraries to utilize the datetime capability of Python (`datetime` and `dateutil`) - Libraries to plot our graph (`matplotlib` and `seaborn`)

The last one (`warnings`) is completely optional. As we are not using all the functions in the library, we only need to import those we actually use.

[2]:
```
# Importing the libraries
import pandas as pd
import numpy as np
from pandas_datareader.data import DataReader

# Importing Datetime libraries
from datetime import date
from dateutil.relativedelta import relativedelta

# Importing plotting libraries
import matplotlib.pyplot as plt
import seaborn as sns

# Preventing warning pop ups
import warnings
warnings.filterwarnings('ignore')

# Matplotlib settings
%matplotlib inline
```

1.3 Pulling in the Stock Data from Yahoo Finance

One can actually download a `.csv` file straight from Yahoo Finance, but that will not be as fun as scraping the site straight from our notebook. First, you can decide the period of data we want to pull. After that, we will take today's date and call it `end`, then substract it by the period we have previously decided and call it `start`.

Pulling the data is relatively easy, using `DataReader()`, we just need to input the ticker, the data source, the start and end date. Because we will only need the **Closing Price**, we need to add `['close']` just after the function and add the method `.to_frame()` to make it into a nice and neat dataframe.

```
[3]: # Specify start date and end date
     period_year = 5
     end = date.today()
     start = end - relativedelta(years = period_year)

     # Specify stocks that you want to get and data source
     ticker = ['AAPL']
     data_source = 'yahoo'

     # Read the data
     df = DataReader(ticker[0], data_source, start, end)['Close'].to_frame()
     df.drop_duplicates(keep = 'first', inplace = True)
```

After pulling the data, it is always a good omen to fix our column names. I found that the output dataframe above usually has either the ticker as the column name or "Close", thus we can easily rename them into lower case "close" using `.rename()`.

We will see the change as well as a glimpse of the first 5 of our dataset using `.head()` method for a dataframe.

```
[4]: # Renaming our column
     df.rename(columns = {ticker[0]:'close', 'Close':'close'}, inplace = True)

     # Glimpse of the data
     df.head()
```

```
[4]:                 close
     Date
     2017-03-31  35.915001
     2017-04-03  35.924999
     2017-04-04  36.192501
     2017-04-05  36.005001
     2017-04-07  35.834999
```

After seeing the first five entries, we can also observe that the index for each entry correspond to the closing price for that particular date. This is good news as we do not need to change the index of our dataframe anymore since stock price is a time series data.

Next up, we can also see the dimension of our data. This will be useful later on as we will need the number of rows to decide the interval for our simulation.

```
[5]: # Previewing the data dimension
     print('The data has {p1} rows and {p2} columns'.format(p1 = df.shape[0], p2 = df.
      ↪shape[1]))
```

```
The data has 1217 rows and 1 columns
```

As we can see, our dataset has 1217 rows and 1 column, which is the closing price. The number of rows could certainly change if the period in the previous cells is also changed.

1.4 Converting the Closing Prices into Geometric Return

We can also calculate the **Geometric Return** of the closing price. This will be extremely useful for the **Geometric Brownian Motion** simulation. The formula of a **Geometric Return** is

$$R_t = ln\left(\frac{S_t}{S_{t-1}}\right) \tag{1}$$

Where R_t is the return for time t, S_t is the stock price at time t and S_t is the stock price at the time $t-1$.

To get the **Geometric Return** we can use the `np.log()` function and dividing our closing price at time i with that of time $i-1$. We can loop through the entries, but it is recommended to explore opportunities to vectorise this calculation. After that is done, we will create a new column called `return` and assign the **Geometric Return** there. Because we cannot get the return for the first entry, we can slice it off our dataset.

[6]:
```
# Finding the geometric return of the closing price
temp = np.array([0])
for i in range(1, len(df)):
    temp = np.append(temp, np.log(df['close'][i]/df['close'][i-1]))

# Appending the return to the dataframe
df['return'] = temp

# Slicing off the first entry
df = df[1:]
```

2 Brownian Motion with Drift

We have now reached the first simulation, which is **Brownian Motion with Drift**. This simulation follows a pattern which is

$$X(t) = \mu t + \sigma B(t) \tag{2}$$

Where $X(t)$ is the value for time t. μ is the **average real stock price** which is also called as the **Drift** variable. σ is the **standard deviation of the real stock price**. Last but not the least, $B(t)$ is the Brownian Motion at that particular time.

Essentially, what makes this different from a regular **Brownian Motion** is the presence of the **Drift** variable. The **Drift** variable acts as the gradient or trend, thus a positive value will see an increase and a negative will see a decrease. regular **Brownian Motion** does not have this, thus it fluctuates only around $y = 0$. There are several assumptions for this type of motion, those are:

1. $X(0) = X_0$
2. Stationary for $0 \leq s \leq t$, $X(t) - X(s) \sim N(\mu\Delta t, \sigma^2\Delta t)$ where $\Delta t = (t - s)$
3. Independent increment

2.1 Brownian Motion with Drift Algorithm

To create the **Brownian Motion with Drift** algorithm, we will use the second assumption. Moving the variables around will result in the equation and assuming that $s = (t-1)$

$$X(t) = X(t-1) + N(\mu\Delta t, \sigma^2\Delta t) \tag{3}$$

where $\Delta t = [t-(t-1)]$. Thus if we set a point and call it X_0, we can iterate through all the data and simulate the movement of the stock price. In order to make this work, we will also need to find the values for μ and σ as mentioned above. After doing that, we can also repeat the simulation as much as we want, therefore we also need a metric to measure the performance of each repetition. For this algorithm, we are going to use **Mean Absolute Percentage Error** or **MAPE**. **MAPE** has a formula of

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{\text{Actual} - \text{Prediction}}{\text{Actual}}\right|$$

Then using **MAPE**, we can choose the repetition with the best fit to our actual data and return it back to us.

[7]:
```
def bm_drift(data, period = 1, seed = False, rep = 1):

    # Checking if the seed parameter and setting the random seed if necessary
    if seed  == True:
        np.random.seed(0)

    # Creating an empty dataframe to store the repetition and an array to store␣
↪the MAPE for the repetition
    back = pd.DataFrame()
    mape = np.array([])

    # Iterating through the repetition
    for i in range(rep):

        # Creating an array of the intervals
        index = np.linspace(0., period, len(data))

        # Determining the X0, mu, and sigma
        X = np.array([data[0]])
        mu = np.mean(data)
        sigma = np.std(data)

        # Iterating through the values of the dataset
        for j in range(1, len(data)):
            # Calculating the delta
            dt = index[j] - index[j-1]
            # Calculating the increment and adding it to the previous index
            increment = np.random.normal(loc = mu * dt, scale = sigma * np.
↪sqrt(dt))
            temp = X[-1] + increment

            # Appending the new X to the array
            X = np.append(X, temp)

        # Finding the MAPE for the repetition and appending it back to the array
        MAPE = np.mean(np.abs((data - X)/data))
        mape = np.append(mape, MAPE)

        # Storing the iteration of each repetition to the dataframe
        back['iter_{iteration}'.format(iteration = i)] = X

    # Resetting the storage dataframe's index to the data index (date)
    back = back.set_index(data.index)

    # Finding the iteration with the lowest MAPE
    minimum = np.where(mape == mape.min())[0][0]

    # Printing and returning the iteration with the lowest MAPE
    print('This is iteration {it} with MAPE {ma}%'.format(it = minimum, ma =␣
↪mape[minimum] * 100))
    return back['iter_{iteration}'.format(iteration = minimum)]
```

The function `bm_drift()` takes in several parameters, namely: - `data` : The stock data we are trying to simulate. - `period` : The period of the stock data, measured in years. - `seed`: The value for `Numpy`'s random seed. This is to ensure replicability. - `rep`: The number of repetitions.

2.2 Using the Brownian Motion with Drift for the Geometric Return

We will now use the function we have made to simulate the movement of our **Geometric Return** or the data in `df['return']`. In theory, this should work but the result might not be good, but we will see it for our eyes first before saying anything. To ensure replicability, we will set `seed = True`. We will repeat the simulation for 1000 times thus `rep = 1000` and we will need to set the `period = period_year` (it uses the variabel above in **Section 1.3.**).

[8]:
```
# Using bm_drift to simulate return
drift_return = bm_drift(df['return'], period = period_year, seed = True, rep =␣
↪1000)
```

```
This is iteration 492 with MAPE 222.9345193994934%
```

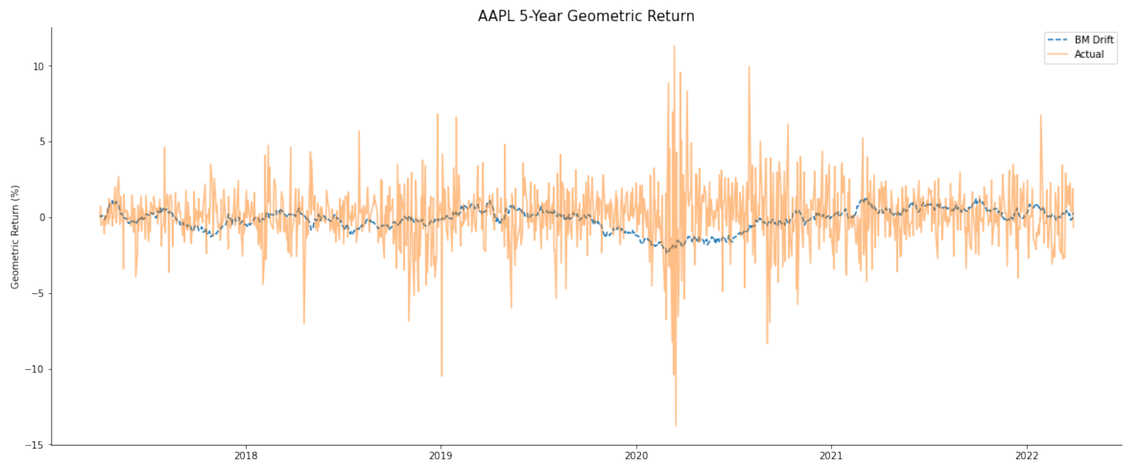
The result is less than impressive as the iteration with the best **MAPE** is still way too high which is actually more 200%. We can also observe the result using the plot relative to the actual returns.

```
[9]: # Setting the figure size
plt.figure(figsize = (20, 8))

# Plotting our data
plt.plot(drift_return * 100, linestyle = 'dashed', label = 'BM Drift')
plt.plot(df['return'] * 100, label = 'Actual', alpha = 0.5)

# Plot settings
plt.title('{stock} {period}-Year Geometric Return'.format(stock = ticker[0],␣
 ↪period = period_year),fontsize = 15)
plt.ylabel('Geometric Return (%)')
plt.legend()
sns.despine()

# Showing the plot
plt.show()
```



The plot gives an indication that the algorithm tried to follow the change in the **Geometric Return**, but it is too rapid and too volatile to follow it perfectly.

2.3 Using the Brownian Motion with Drift for the Closing Price

Next, we will use the function to simulate the closing price of the stock. This in theory should work better as one of the uses of **Brownian Motion with Drift** is to simulate actual prices not returns. We will use the same parameters as the section above.

[10]:
```
# Using bm_drift to simulate the closing price
drift_close = bm_drift(df['close'], period = period_year, seed = True, rep =␣
 ↪1000)
```

```
This is iteration 447 with MAPE 21.63495021750128%
```

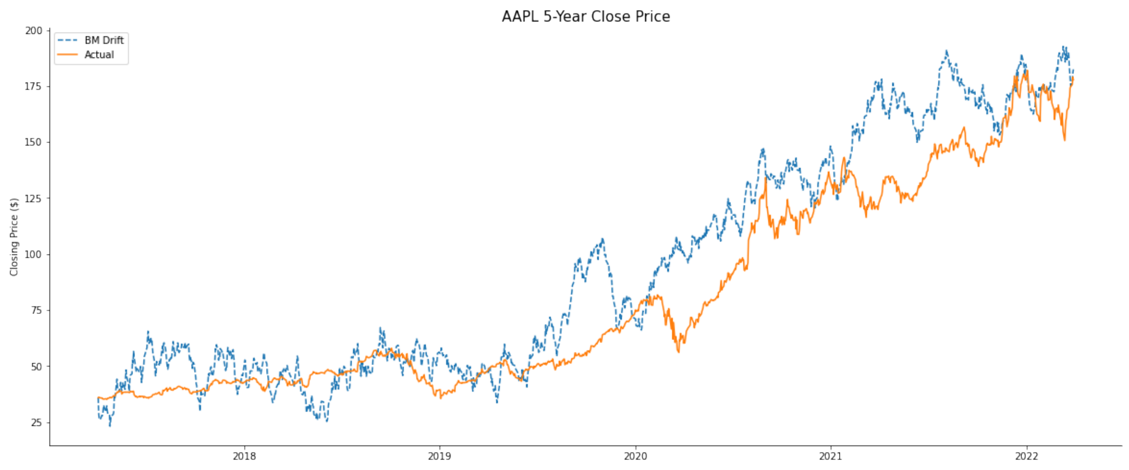
Using the function, we found out that the 447th iteration yields the lowest **MAPE**, which is roughly 21.6%. We can also check the result by plotting it relative to the actual closing price and see for ourselves whether the simulation works.

[11]:
```
# Setting the figure size
plt.figure(figsize = (20, 8))

# Plotting the data
plt.plot(drift_close, linestyle = 'dashed', label = 'BM Drift')
plt.plot(df['close'], label = 'Actual')

# Plot settings
plt.title('{stock} {period}-Year Close Price'.format(stock = ticker[0], period =␣
 ↪period_year),fontsize = 15)
plt.ylabel('Closing Price ($)')
plt.legend()
sns.despine()

# Showing the plot
plt.show()
```



Unlike the plot for the returns, the closing price simulation is a lot closer to its actual closing price. There are times where it tends to overshoot a little and times where it does not change quickly enough to the movement of the actual price.

3 Geometric Brownian Motion

Geometric Brownian Motion can be described as **Brownian Motion with Drift** but a lot more sensitive to changes. In the previous section, we have seen that **Brownian Motion with Drift** have a tendency to not change as rapidly, thus this simulation method tries to "fix" the issue. The model itself is defined as:

$$S(t) = e^{X(t)} \tag{4}$$

Where $S(t)$ is the value of the simulation at time t and $X(t)$ is actually the value of the **Brownian Motion with Drift** at the same time t.

3.1 Geometric Brownian Motion Algorithm

So you may ask, what makes this different from **Brownian Motion with Drift**. First of all, we will not be using **Equation 4** to actually model the value, instead we will be using either:

$$S(t) = S(t-1) \cdot e^{N(\mu\Delta t, \sigma^2 \Delta t)} \tag{5}$$

or the equation

$$S(t) = S(t-1) \cdot e^{\left[\left(\mu - \frac{\sigma^2}{2}\right)\Delta t + \sigma N(0, \Delta t)\right]} \tag{6}$$

Where $\Delta t = [t - (t-1)]$. What makes it also different is how we get μ and σ. Several people has interpretations of the exact method, but the common denominator is that we are using the **Geometric Return** to find both variables.

$$\mu = \frac{\bar{R}}{\Delta t} + \frac{\sigma^2}{2} \tag{7}$$

$$\sigma = \frac{S}{\sqrt{\Delta t}} \tag{8}$$

Where $\bar{R}$ is the average **Geometric Return** of the stock over the period and S is the standard deviation of the **Geometric Return**. We will also use **MAPE** to find the repetition which is closest to our actual stock price. And just like **Brownian Motion with Drift**, we will also need to find S_0.

```
[12]: def geo_bm(data, tar, ret = 'return', period = 1, seed = False, rep = 1):

          # Checking if the seed parameter and setting the random seed if necessary
          if seed  == True:
              np.random.seed(0)

          # Creating an empty dataframe to store the repetition and an array to store␣
       ↪the MAPE for the repetition
          back = pd.DataFrame()
          mape = np.array([])

          # Iterating through the repetition
          for i in range(rep):

              # Creating an array of the intervals
              index = np.linspace(0., period, len(data))

              # Creating an array with the first value
              S = np.array([data[tar][0]])

              for j in range(1, len(data)):

                  # Calculating the delta, sigma, and mu
                  dt = index[j] - index[j-1]
                  sigma = np.std(data[ret]) / np.sqrt(dt)
                  mu = np.mean(data[ret]) / dt + 0.5 * sigma**2

                  # Finding the increment and multiplying the value with the previous S
                  increment = np.exp((mu - sigma**2 / 2) * dt + sigma * np.random.
       ↪normal(0, np.sqrt(dt)))
                  temp = S[-1] * increment

                  # Appending the new S to the array
                  S = np.append(S, temp)

              # Finding the MAPE for the repetition and appending it back to the array
              MAPE = np.mean(np.abs((data[tar] - S)/data[tar]))
              mape = np.append(mape, MAPE)

              # Storing the iteration of each repetition to the dataframe
              back['iter_{iteration}'.format(iteration = i)] = S

          # Resetting the storage dataframe's index to the data index (date)
          back = back.set_index(data.index)
```

```
    # Finding the iteration with the lowest MAPE
    minimum = np.where(mape == mape.min())[0][0]

    # Printing and returning the iteration with the lowest MAPE
    print('This is iteration {it} with MAPE {ma}%'.format(it = minimum, ma =␣
↪mape[minimum] * 100))
    return back['iter_{iteration}'.format(iteration = minimum)]
```

The function `geo_bm()` takes in several parameters, namely: - `data` : The stock data we are trying to simulate, this expects a **DataFrame**. - `tar` : The target column of our simulation in string. - `ret` : The geometric return for our target. - `period` : The period of the stock data, measured in years. - `seed`: The value for `Numpy`'s random seed. This is to ensure replicability. - `rep`: The number of repetitions.

3.2 Using the Geometric Brownian Motion for the Geometric Return

We can also use the **Geometric Brownian Motion** to simulate the **Geometric Return** of our stock, but the result should be sub-optimal. We are going to set `data = df` as our data, `tar = 'return'` as our target, `ret = 'return'` as our **Geometric Return**, and the same `period`, `seed`, and `rep` as the what we did for **Brownian Motion with Drift**.

[13]:
```
# Using the function to simulate return
geo_return = geo_bm(df, 'return', ret = 'return', period = period_year, seed =␣
↪True, rep = 1000)
```

```
This is iteration 681 with MAPE 100.69088254238696%
```

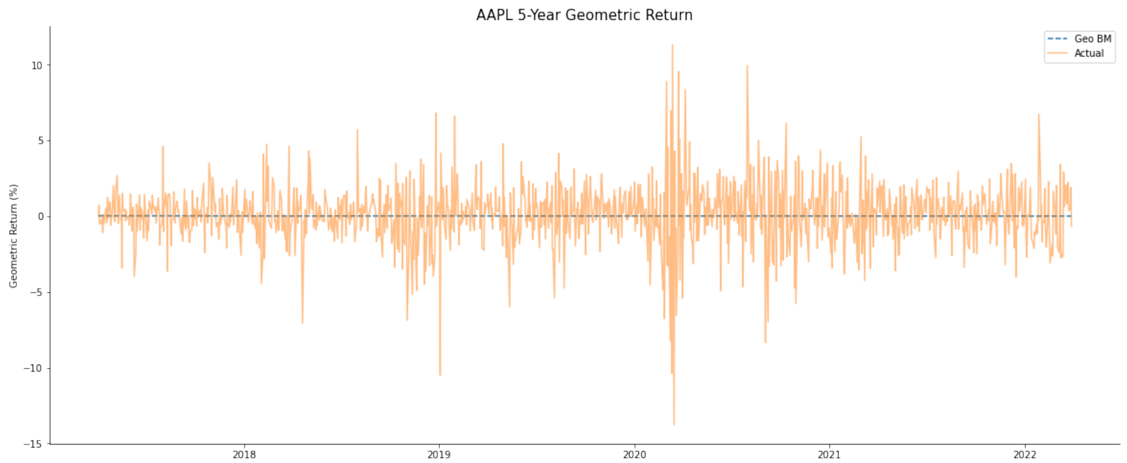
As we can see, we now have a **Geometric Brownian Motion** which has a **MAPE** of around 100%. To check this, we will also plot the result relative to the actual return we have calculated previously.

[14]:
```
# Setting the figure
plt.figure(figsize = (20, 8))

# Plotting the data
plt.plot(geo_return * 100, linestyle = 'dashed', label = 'Geo BM')
plt.plot(df['return'] * 100, label = 'Actual', alpha = 0.5)

# Plot settings
plt.title('{stock} {period}-Year Geometric Return'.format(stock = ticker[0],␣
↪period = period_year),fontsize = 15)
plt.ylabel('Geometric Return (%)')
plt.legend()
sns.despine()

# Displaying the plot
plt.show()
```



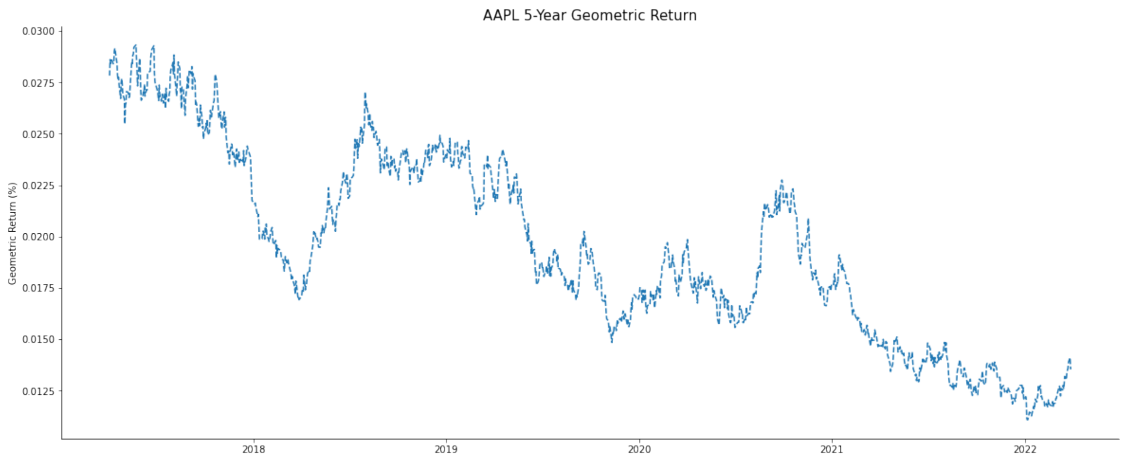
The result is quite disappointing, but that is not really the truth. The only reason why our simulation resulted in what appeared to be a straight line is because the fluctuation of the return is so high that we cannot use this simulation to actually model the return. To prove this, we will plot the simulated values individually.

```
[15]: # Setting the figure
plt.figure(figsize = (20, 8))

# Plotting the data
plt.plot(geo_return * 100, linestyle = 'dashed', label = 'Geo BM')

# Plot settings
plt.title('{stock} {period}-Year Geometric Return'.format(stock = ticker[0],␣
 ↪period = period_year), fontsize = 15)
plt.ylabel('Geometric Return (%)')
sns.despine()

# Displaying the plot
plt.show()
```



As we can see, the simulated return is so minuscule compared to the actual return, it appeared as a straight line in the previous figure, but it is not actually a straight line at all. We can also see that there is indeed no negative value. This could theoretically happen when S_0 is positive and since we only iterate with the formula as our pattern, the value of e^x will never be negative, thus we cannot actually use **Geometric Brownian Motion** to predict stock return because stock return could yield a negative value.

3.3 Using the Geometric Brownian Motion for the Closing Price

So now we are going to use the function to simulate the **Closing Price**. This should yield a better simulated set of values compared to the section before. We are going to use exactly the same parameter as what we did in **Section 3.2.** with one exception, the target is now set as `tar = 'close'`.

[16]:
```
# Using the function to simulate the closing price
geo_close = geo_bm(df, 'close', ret = 'return', period = period_year, seed =␣
↪True, rep = 1000)
```

```
This is iteration 171 with MAPE 9.767060987041855%
```

We can see that for our 1000 iterations, the best iteration is iteration 171 with **MAPE** of roughly 9.77%. This by itself is a lot better than that of **Brownian Motion with Drift** (21.6%). We can check the figure to see whether our simulation is indeed correct.

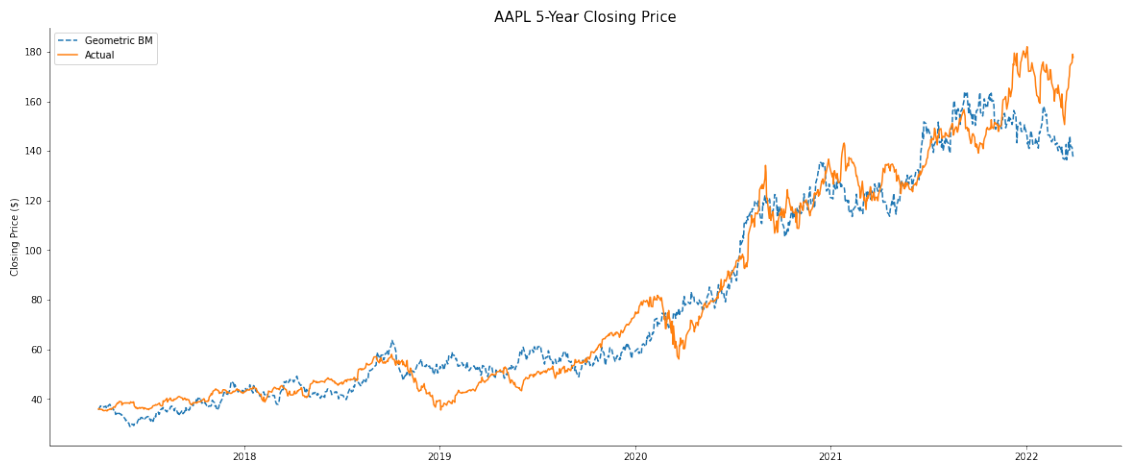
[17]:
```
# Setting up the figure
plt.figure(figsize = (20, 8))

# Plotting the data
plt.plot(geo_close, linestyle = 'dashed', label = 'Geometric BM')
plt.plot(df['close'], label = 'Actual')

# Plot settings
plt.title('{stock} {period}-Year Closing Price'.format(stock = ticker[0], period␣
↪= period_year), fontsize = 15)
```

```python
plt.ylabel('Closing Price ($)')
plt.legend()
sns.despine()

# Displaying the plot
plt.show()
```



From the graph above, it is clear to see that our simulation is a lot closer to the actual **Closing Price**. When compared to the result of **Brownian Motion with Drift**, it also changes much more rapidly and has less of the overshooting issue.

4 Conclusion

To conclude this exercise, it is clear to see that both of the **Browinian Motion** simulations can actually model **Closing Price** of a stock well, with the **Geometric Brownian Motion** a lot better than **Brownian Motion with Drift**. When simulating **Geometric Return**, it is clear that former suffers a lot more than latter, especially due to very-very small movements and the lack of negative values. It is actually not recommended to model **Geometric Return** with either, but if you should choose one, **Brownian Motion with Drift** will probably result in a better simulation.

Some suggestion to improve the simulation would be the utilisation of vectorisation in the computation of both $X(t)$ and $S(t)$. Using for loops can take a lot longer especially with bigger datasets. There are tutorials of doing **Geometric Brownian Motion** with vectorisation in the internet, such as this one.

References

[1] ASX Portofolio. (2021). *Simulating Geometric Brownian Motion in Python*. Retrieved April 1, 2022, from https://asxportfolio.com/options-stochastic-calculus-simulating-gbm

[2] Britanica. (n.d.). *Brownian motion | physics*. Encyclopedia Britannica. Retrieved April 1, 2022, from https://www.britannica.com/science/Brownian-motion

[3] Liden, J. (2018). *Stock Price Predictions using a Geometric Brownian Motion*. Uppsala Universitet. Retrieved April 1,2022, from https://www.diva-portal.org/smash/get/diva2:1218088/FULLTEXT01.pdf

[4] Maulidya, V., et.al. (2020). *Prediksi Harga Saham Menggunakan Geometric Brownian Motion Termodifikasi Kalman Filter dengan Konstrain*. Indonesian Journal of Applied Mathematics Retrieved April 1, 2022, from https://journal.itera.ac.id/index.php/indojam/article/view/307/111

[5] Reddy, K. & Clinton, V. (2016). *Simulating Stock Prices Using Geometric Brownian Motion: Evidence from Australian Companies*. Australasian Accounting, Business and Finance Journal, 10(3). https://doi.org/10.14453/aabfj.v10i3.3