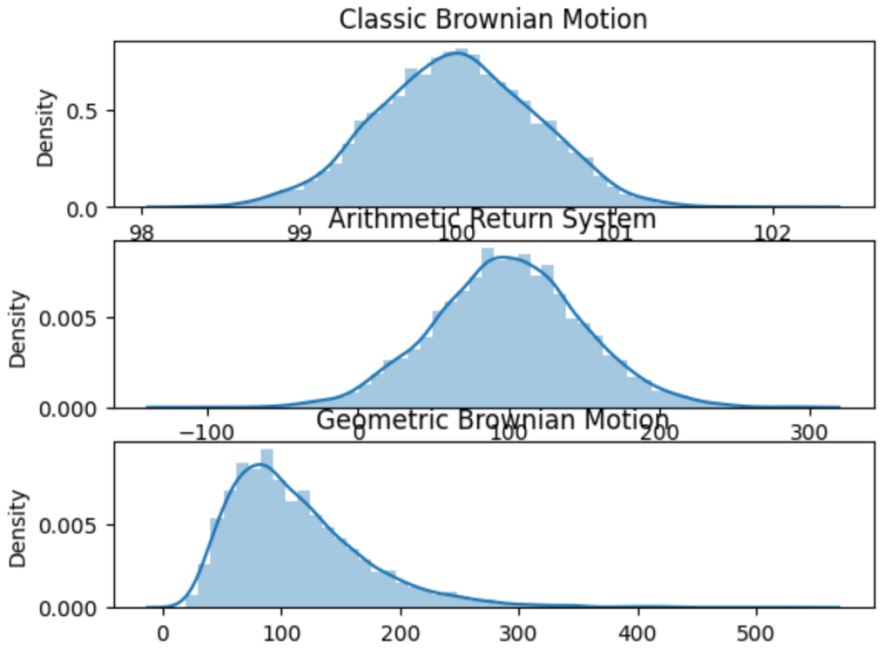
Week04

Problem1



From the plot, we can see that the calculated means are standard deviation are very close to the expected value and standard deviation for these three types of price returns. This means that our simulation works correctly.

Problem2

The function similar to return_calculate() is shown under the Problem2 section in the file Code.py.

For the arithmetic returns for all prices:

```
                Date       SPY      AAPL      MSFT      AMZN      TSLA  \
1    2/15/2022 0:00  0.016127  0.023152  0.018542  0.008658  0.053291
2    2/16/2022 0:00  0.001121 -0.001389 -0.001167  0.010159  0.001041
3    2/17/2022 0:00 -0.021361 -0.021269 -0.029282 -0.021809 -0.050943
4    2/18/2022 0:00 -0.006475 -0.009356 -0.009631 -0.013262 -0.022103
5    2/22/2022 0:00 -0.010732 -0.017812 -0.000729 -0.015753 -0.041366
..            ...       ...       ...       ...       ...       ...
244   2/3/2023 0:00 -0.010629  0.024400 -0.023621 -0.084315  0.009083
245   2/6/2023 0:00 -0.006111 -0.017929 -0.006116 -0.011703  0.025161
246   2/7/2023 0:00  0.013079  0.019245  0.042022 -0.000685  0.010526
247   2/8/2023 0:00 -0.010935 -0.017653 -0.003102 -0.020174  0.022763
248   2/9/2023 0:00 -0.008669 -0.006912 -0.011660 -0.018091  0.029957

        GOOGL      GOOG      META      NVDA  ...       PNC      MDLZ  \
1    0.007987  0.008319  0.015158  0.091812  ...  0.012807 -0.004082
2    0.008268  0.007784 -0.020181  0.000604  ...  0.006757 -0.002429
3   -0.037746 -0.037669 -0.040778 -0.075591  ... -0.034949  0.005326
4   -0.016116 -0.013914 -0.007462 -0.035296  ... -0.000646 -0.000908
5   -0.004521 -0.008163 -0.019790 -0.010659  ...  0.009494  0.007121
..        ...       ...       ...       ...  ...       ...       ...
244 -0.027474 -0.032904 -0.011866 -0.028053  ... -0.004694 -0.011251
245 -0.017942 -0.016632 -0.002520 -0.000521  ... -0.014451  0.003945
246  0.046064  0.044167  0.029883  0.051401  ... -0.000368 -0.016473
247 -0.076830 -0.074417 -0.042741  0.001443  ... -0.008469 -0.004456
248 -0.043876 -0.045400 -0.030039  0.005945  ... -0.016588 -0.007717

           MO       ADI      GILD       LMT       SYK        GM       TFC  \
1    0.004592  0.052344  0.003600 -0.012275  0.033021  0.026240  0.028572
2    0.005763  0.038879  0.009294  0.012244  0.003363  0.015301 -0.001389
3    0.015017 -0.046988 -0.009855  0.004833 -0.030857 -0.031925 -0.033380
4    0.007203 -0.000436 -0.003916 -0.005942 -0.013674 -0.004506 -0.003677
5   -0.008891  0.003243 -0.001147 -0.000673  0.008342 -0.037654 -0.002246
..        ...       ...       ...       ...       ...       ...       ...
244 -0.001277 -0.002677  0.038211  0.004134  0.002336 -0.008916 -0.005954
245  0.001066 -0.007102  0.022012  0.021826 -0.041181  0.005106 -0.009782
246 -0.008518  0.019544 -0.003590 -0.001641  0.003573  0.001451  0.008669
247 -0.001289 -0.018009 -0.004416  0.002819 -0.015526  0.004106 -0.015391
248 -0.003656  0.004275 -0.001634  0.000937 -0.014391  0.001443 -0.016619


          TJX
1    0.013237
2   -0.025984
3   -0.028763
4    0.015038
5   -0.013605
..        ...
244  0.001617
245 -0.004595
246 -0.003618
247  0.009363
248  0.005603

[248 rows x 101 columns]
```

```
/var/folders/m6/xpp1np3926ngk3wq1vyszc100000gn/T/ipykernel_36624/3032948848.py:30: PerformanceWarning: DataFrame is
highly fragmented.  This is usually the result of calling `frame.insert` many times, which has poor performance.  C
onsider joining all columns at once using pd.concat(axis=1) instead. To get a de-fragmented frame, use `newframe =
frame.copy()`
  out[vars[i]] = p2[:,i]
```

For VaR:

1) Normal distribution at 0.05:

```
0.06560156967533286
```

2) Exponentially weighted:

```
0.09138526093846899
```

3) MLE at 0.05:

```
Optimization terminated successfully.
         Current function value: -463.957934
         Iterations: 4
         Function evaluations: 16
         Gradient evaluations: 8
```

```
: array([8.84783522])
```

We first get the optimization value of 8.84783522. Then we can use it to generate the value

```
0.07325331150481068
```

4) AR(1) :

```
const     -0.000062
META.L1    0.007233
dtype: float64
```

We can see the parameter for the AR(1) model, then we can use them to generate our equation and get the results.

```
0.06588072152825065
```

5) Historic Simulation:

```
0.05590681367337082
```

In conclusion, at the 5% level, the historic simulation generates the lowest VaR, the second lowest will be the Normal distribution and its value is very close to the value of AR(1) model. The next will be the MLE fitted t Distribution and the normal distribution with an Exponentially Weighted variance (λ = 0. 94) has the highest VaR.

Problem3

For the exponentially weighted covariance with lambda = 0.94, the VaR of each portfolio as well as the total VaR is

```
Portfolio A is $ 5678.2163441848825
Portfolio B is $ 4573.269674544281
Portfolio C is $ 3849.016192226205
Portfolio Total is $ 14100.502210955368
```

Based on the results above, the portfolio A has the highest value and C has the lowest value. My method is to generate the data by using the exponentially weighted covariance with lambda = 0.94.

For the historic method, the value is

```
Portfolio A is $ 9070.10436189425
Portfolio B is $ 7351.1664714418
Portfolio C is $ 5802.65286111302
Portfolio Total is $ 18140.2087237885
```

I chose the historic method because it is very common at our field and we will see it very often. Compared with the method 1, the values are greater for all of these three portfolios.