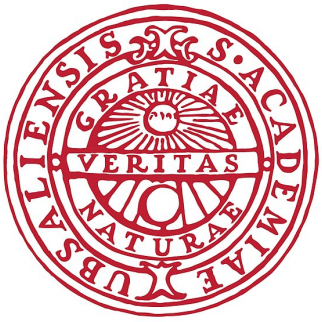
Monte-Carlo Methods in Finance



UPPSALA UNIVERSITET

Inference theory I (1MS035)

Authors:

Glöckner Edward,
Lennerstrand Sofia

Uppsala
December 8, 2022

Contents

1 Project Description

In this project two aspects of how one can use statistics in finance will be covered. Both, using Monte-Carlo methods in order to simulate stock prices and optimizing portfolio allocation. *Part 1* will cover simulation of Amazon stock prices and *Part 2* will cover optimization and allocation of stocks in a portfolio, compared to a uniform allocation. Hypothesis testing is essential for both parts to investigate the value of the two methods.

2 Introduction

2.1 Part 1

In finance, the price of a stock can often be modelled as following Brownian motion. Brownian motion is a Markov Stochastic Process in which the movement of the random variable in question follows the Wiener's Process.

In this project, Python will be used in order to Monte-Carlo simulate the 'future' unknown price of Amazon stock utilizing Brownian motion. The simulated prices will then be compared with the true value of the stock.

2.2 Part 2

Modern portfolio theory suggests diversification of a portfolio is an important aspect in regards to risk and expected returns. Diversification, ruffly meaning spreading ones asset across multiple industries, countries and financial instruments is however not that simple to achieve. This is due to the fact that individual stocks in a portfolio can have a correlation with another stock in the same portfolio, leading to the portfolio not being as diversified as one may believe.

In this project, random portfolios are generated and the weight of the individual stocks are Monte-Carlo simulated. Thus yielding a large number of possible allocations. To find the optimal allocation we choose the allocation which have the highest sharpe ratio.

The optimal portfolio is then tested on a before unseen testing set, and compared to a uniform allocation. To test if the portfolios sharp ratio have the same distribution or not, a paired t test was performed. The paired t test will test whether the mean difference for each portfolio before and after optimization is zero or not.

3 Method

3.1 Part 1

The stock prices of Amazon from the beginning of 2018 to mid 2020 (2020-06-02) for each day was used as a training set. Initially the return values of the stock each day compared to the previous day was found, here called Δx. The price was then simulated half a year worth of trading days, 121 days, after 2020-06-02, under the assumption that Δx is log normal distributed. The stock values was modelled as a Stochastic Markov Process. If the previous assumptions are made, one can apply the Weiner's Process which describes the movement of a random variable that follows the Markov Process.

Historical stock price data of Amazon between 2017-11-30 and 2022-11-29 was downloaded from Yahoo finance. In figure (1) the data is visualized:

Figure 1: Amazon stock prices.

The generalized Weiner's Process can be described using the following equation

$$\Delta x = a \cdot dt + b \cdot \epsilon\sqrt{t} \tag{1}$$

where $a \cdot dt$ is the drift rate which here corresponds to the mean of Δx, and $b \cdot \epsilon\sqrt{t}$ is the standard deviation of Δx.

The Weiner's Process was used to model the daily returns of the stock price which the natural logarithm of it was assumed to be normal distributed.

$$\frac{\Delta S}{S} \sim \Phi\left(\mu \cdot \Delta t, \sigma^2 \cdot \Delta\sqrt{t}\right)$$

Using Itô's lemma the distribution above is transformed into,

$$\frac{\ln S_{t+1}}{\ln S_t} \sim \Phi\left(\mu - \frac{\sigma^2}{2} \cdot \Delta t, \sigma^2 \cdot \Delta\sqrt{t}\right)$$

which yields the model of the stock price,

$$S_{t+1} = S_t \cdot e^{\mu - \frac{\sigma^2}{2}dt + \sigma \cdot \epsilon\sqrt{dt}}$$

ϵ above represents the stochastic component of Brownian Motion which means that the Δx, or return, for each day is stochastic. The training set is used to calibrate the μ and σ, and the testing set is used to validate if our model has any value for forecasting future prices.

Our hypotheses is as follows:

$$\begin{aligned} H_0 : \mu &= \mu_0 \\ \text{vs} & \\ H_1 : \mu \neg &= \mu_0 \end{aligned} \tag{2}$$

where μ is the expectation value of our sample. The confidence method will be used to determine if our null hypotheses should be rejected or not. To do this, a 99% confidence interval will be calculated for the simulated values, and then it can be checked if the correct value lies in that interval. This can be formulated as:

$$\begin{aligned} &\text{Calculate } I_\mu \\ &\text{If } \mu_0 \notin I_\mu \text{ then reject } \mu_0 \end{aligned} \tag{3}$$

3.2 Part 2

The stock prices of 18 different stocks during a 5 year period, from 2017 to 2022 was used for this project. 80 % of the data was used as training set where the optimal allocation of the stocks was calculated, and the rest was used as a testing set validating if the allocation had a significant impact or not. 15 different portfolios was randomly generated, and can be seen in table (1).

Portfolios				
	Stock 1	Stock 2	Stock 3	Stock 4
Portfolio 1	KO	CVS	FDX	HP
Portfolio 2	BAC	CVS	IBM	AMZN
Portfolio 3	BAC	ET	FDX	NKE
Portfolio 4	BAC	COST	ET	XOM
Portfolio 5	FDX	BRK-B	GM	TSLA
Portfolio 6	TSLA	NKE	BRK-B	ET
Portfolio 7	GS	COST	RTX	XOM
Portfolio 8	GOOG	CVS	FDX	COST
Portfolio 9	COST	BRK-B	CVS	GS
Portfolio 10	GM	CVS	GS	TSLA
Portfolio 11	IBM	BRK-B	ET	NKE
Portfolio 12	IBM	TSLA	PFE	GS
Portfolio 13	NKE	BAC	GOOG	HP
Portfolio 14	AMZN	HP	PFE	FDX
Portfolio 15	RTX	BRK-B	COST	AMZN

Table 1: Randomly generated portfolios

After this the covariance-matrix was calculated for each portfolio, and an example of portfolio 1 can be visualized in figure (2):

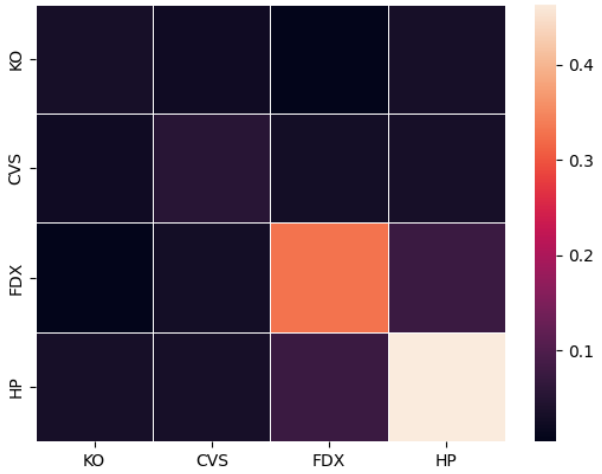


Figure 2: Covariance matrix of portfolio 1

note that the diagonal elements are the individual stocks' variance.

One import measure of a portfolios stock is the sharpe ratio, denoted as SR, given by equation (4):

$$\mathrm{SR} = \frac{R_p - R_f}{\sigma_p} \tag{4}$$

where R_p is the return of the portfolio, R_f is the risk-free rate, and σ_p is the standard deviation of the portfolio's excess returns. For simplicity, the sharpe ratio is often assumed to follow a normal distribution, and this assumption will be used throughout the project. A discussion of the distribution will be provided in the discussion section.

After this, the portfolios was simulated using a Monte-Carlo method. Random allocations was assigned to the portfolio, and the returns for the training set was calculated. After a large number of simulations using different allocations, the one allocation that had the highest sharpe ratio was selected.

After the optimal allocation was calculated, it is time to test the portfolio versus a uniformed allocated portfolio using the test set. Note that the program has not been in touch with the testing set yet, since the optimal portfolio was calibrated using the training set. The sharpe ratio of the testing set using the optimal allocation, as well as the sharpe ratio using a uniform

allocation was calculated. Now a paired sample is obtained, with the sharpe ratio of the testing set before and after the optimized allocation was applied. This is then used in a paired t test to check if the mean difference for each portfolio before and after optimization is zero or not.

Our hypotheses is as follows:

$$\begin{aligned} H_0 : \mu_d \ &= 0 \\ \text{vs} \qquad & \\ H_1 : \mu_d \neg &= 0 \end{aligned} \tag{5}$$

where μ_d is the difference of the sharpe ratio before and after the optimized portfolio.

4 Results

4.1 Part 1

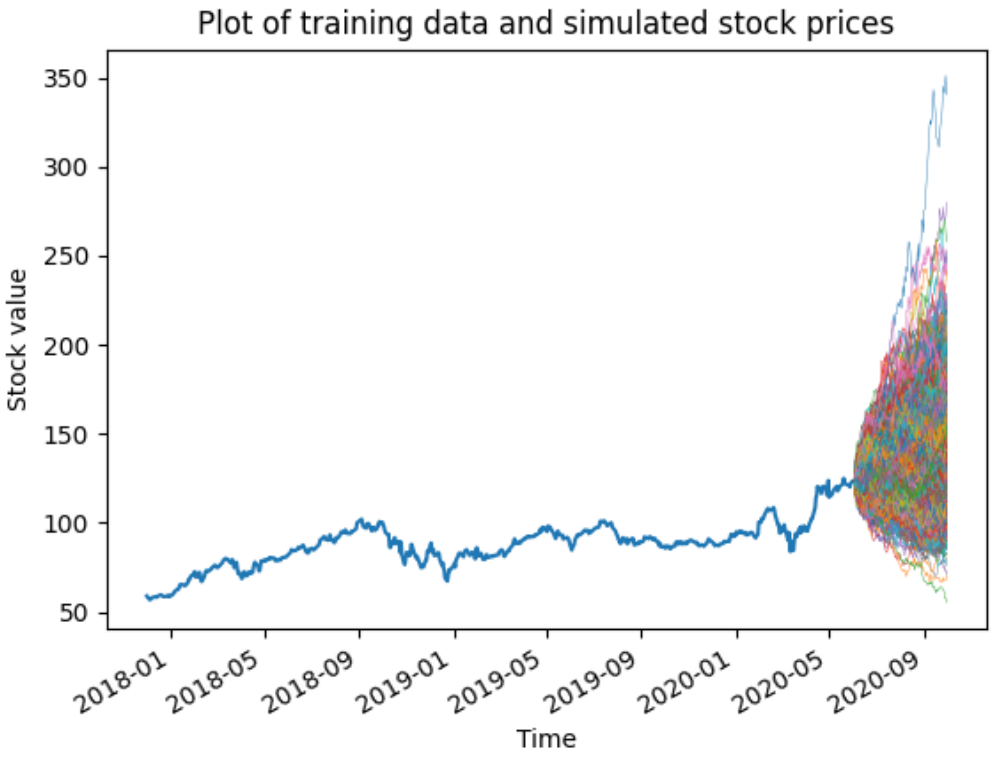


Figure 3: Tree diagram over the training data and the Monte Carlo simulated stock prices.

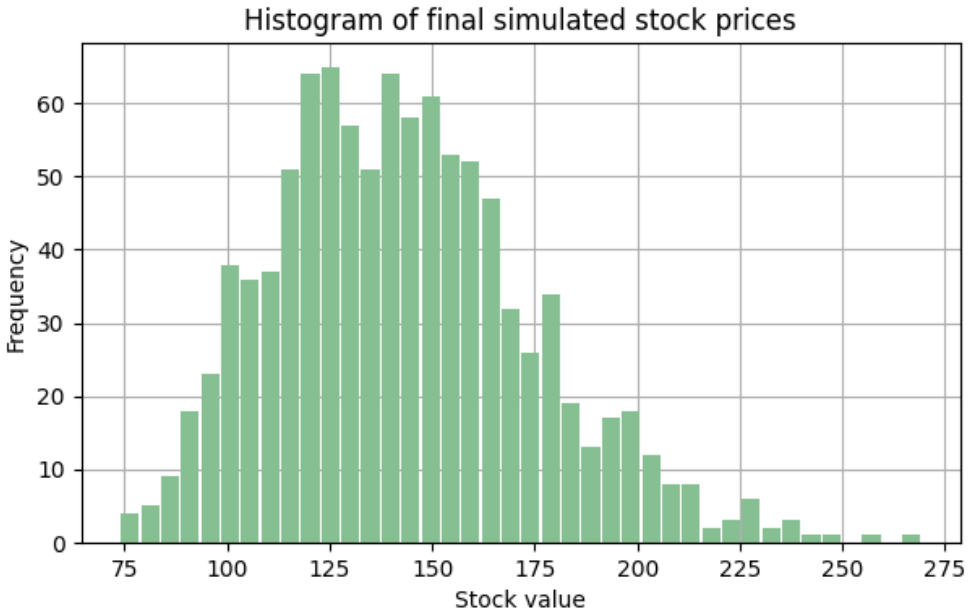


Figure 4: Histogram over the final Monte Carlo simulated stock prices and their frequency.

The correct final stock value,

$$154.9$$

and the confidence interval for the final simulated stock values,

$$[139.8, 145.1]$$

To confirm normal distribution a QQ-plot was made of the final stock values. It can be noted that the final values more specifically follow a Fat-tail distribution.

Figure 5: Final stock values Fat-tailed distributed

4.2 Part 2

Table (2) shows the optimal allocation.

Optimal allocation of stocks in portfolio				
	Stock 1	Stock 2	Stock 3	Stock 4
Portfolio 1	0.0174	0.0469	0.0104	0.9251
Portfolio 2	0.9096	0.0307	0.0444	0.0151
Portfolio 3	0.0159	0.0662	0.9149	0.0028
Portfolio 4	0.0410	0.0108	0.8650	0.0830
Portfolio 5	0.0429	0.0195	0.0308	0.9066
Portfolio 6	0.9033	0.0448	0.0018	0.0499
Portfolio 7	0.0646	0.0272	0.0000	0.9080
Portfolio 8	0.0039	0.0442	0.9269	0.0248
Portfolio 9	0.0260	0.0571	0.0126	0.9040
Portfolio 10	0.0518	0.0031	0.0152	0.9298
Portfolio 11	0.0457	0.0098	0.9353	0.0090
Portfolio 12	0.0093	0.9463	0.0356	0.0086
Portfolio 13	0.0240	0.0114	0.0145	0.9499
Portfolio 14	0.0018	0.9230	0.0304	0.0446
Portfolio 15	0.9296	0.0061	0.0625	0.0015

Table 2: Optimal portfolio allocation with respect to sharpe ratio

Table (3) illustrates the returns of the portfolio running the testing set, before and after the optimized allocation.

Sharpe ratio of portfolio before and after optimal allocation			
Sharpe ratio	Uniform	Optimal	Performance
Portfolio 1	-0.001900	-0.000749	60.54 %
Portfolio 2	-0.002565	-0.002134	16.81 %
Portfolio 3	-0.001803	-0.001594	11.60 %
Portfolio 4	-0.002425	-0.001740	28.25 %
Portfolio 5	-0.001979	-0.001784	9.83 %
Portfolio 6	-0.002022	-0.001786	11.67 %
Portfolio 7	-0.002627	-0.001616	38.47 %
Portfolio 8	-0.002258	-0.001620	28.26 %
Portfolio 9	-0.002811	-0.002175	22.61 %
Portfolio 10	-0.002087	-0.001768	15.27 %
Portfolio 11	-0.001928	-0.001731	10.20 %
Portfolio 12	-0.002327	-0.001775	23.70 %
Portfolio 13	-0.001528	-0.000723	52.70 %
Portfolio 14	-0.001777	-0.000739	58.39 %
Portfolio 15	-0.002458	-0.002309	6.05 %

Table 3: Returns of optimized portfolio versus uniformed allocated portfolio

The results was then passed into a paired t-test, which yield the results:

Paired t-test

data: before and after

$t = -6.2945$, *df* $= 14$, *p-value* $= 1.973e-05$

alternative hypothesis: true mean difference is not equal to 0

95 percent confidence interval:

-0.0007375850 -0.0003626817

sample estimates:

mean difference

-0.0005501333

5 Discussion

5.1 Part 1

By applying the confidence method as a hypothesis test, one can see that the correct final value of the stock is not found within the calculated 95% confidence interval of the simulated values. This means that H_0, thus μ_0 must be rejected.

To improve the hypothesis test, and thus improving the simulation of the stock price, an idea would be to make the simulation dynamic. If the reference μ value was to be recalculated from not just the training data but from the new simulated data after a certain time, say every month, one can assume that the simulated μ value would be closer to the real stock value.

To be noted from figure (5) is that the final values of the simulated stock values don't look like a perfect normal distribution. The distribution observed, where events that deviate from the mean by five or more standard deviations have a higher probability compared to a normal distribution, is commonly referred to as a fat-tailed distribution. The QQ-plot illustrates that rare events are more likely to be found than in a normal distribution, which is to be expected in a psychological dynamical system such as the stock market. This new information also indicates that our initial assumption of log normal distributed Δx is wrong.

5.2 Part 2

As we can observe from our paired t-test, we reject our null hypotheses H_0 and say that the true mean difference is not equal to 0. We can also see from out test that the optimal allocation increases the sharpe ratio on average with 0.0005501333, compared to the uniform allocation. This difference is relatively not large, and normally investors look for a sharpe ratio above 1.5 when making investing decisions. It should be noted however, that the stocks in the portfolios where randomly assigned, and one could greatly increase the sharpe ratio using an initial screening of which stocks should be added to the portfolio. This portfolio optimizer should rather be used as an optimizer of an already decided on portfolio, and not as a trading strategy alone.

In this project we assumed that the sharpe ratio could be modelled as a normally distributed random variable. From observations this is however not always the case, since the returns are not always normally distributed. Papers have been published trying to improve the model using for example

a fat tail distribution, or a t distribution.

As in part 1 of this project, exploring a dynamically allocated portfolio could potentially yield an even larger increase in the sharpe ratio. As the expected value and the volatility of the stocks making up the portfolio changes over time, the optimal portfolio allocation also changes, but this is not handled in the current version of this report.

6 Appendix

6.1 Part 1

6.1.1 main.py

```
from functions import *
import scipy

import datetime
# Data of the stock
ticker = "AMZN"
#file_path = "Data/AMZN.csv"
file_path = "Brownian-Motion-Monte-Carlo-Stocks\Data\AMZN.csv"

# Reads the data into a dataframe
data = read_csv(file_path, ticker)
train, test = split_timeseries(data)
train_returns = log_returns(train)
start_val = train["Close"].iloc[len(train)-1]

days_sim = 121
mc_sim = MC(train, start_val, train_returns, nofsim=1000, days_sim=days_sim)

#plot_tree(train, days_sim, mc_sim)
#plot_histogram(final_values)
S, S_interval_1, S_interval_2, expected_delta_x =
conf_interval(start_val, train, 5000)
plot_conf(S, S_interval_1, S_interval_2, expected_delta_x)
final_values = final_values(mc_sim)
mean_final_value = final_values.mean()
final_date = mc_sim.last_valid_index()
correct_final_value = test.iloc[days_sim]["Close"]
print(type(final_values))

print("Mean simulated value: ", mean_final_value)
print("Correct final value: ", correct_final_value)

interval = conf_interval_values(final_values)

#plot_dataframe(train)
```

```
plot_tree(train,days_sim,mc_sim)
print("Interval", interval)
plot_histogram(final_values)
```

6.1.2 functions.py

```
from scipy.stats import norm
from scipy import stats as st
import pandas as pd
import numpy as np
from datetime import datetime
import matplotlib.pyplot as plt


def read_csv(file_path, tick):
    """
    Reads a file_path for a gvin ticker, and returns a dataframe
    with date and closing price
    @params:
        file_path: path to the destination of the file
        tick: ticker of the stock

    @returns:
        temp: pandas dataframe with closing price and date of the stock

    """
    temp = pd.read_csv(file_path, parse_dates=[0], index_col=0)
    temp.rename(columns={"Close" : tick})
    temp.drop(["Open", "High", "Low", "Adj Close", "Volume"], axis = 1,
    inplace = True)
    return temp


def split_timeseries(timeSeries):
    """
    @params:

    @returns:

    Splits a dataframe into a training set containing 80% of the data,
    and a testing set containing
```

```
    """
    Train = timeSeries[0 : round(len(timeSeries)*0.5)]
    Test = timeSeries[round(len(timeSeries)*0.5) : len(timeSeries)]
    return Train, Test


def log_returns(dataframe):
    """
    @params:
        dataframe: pandas dataframe of our stock.
    @returns:
        diff: logaritmic difference.
    """
    diff = np.log(dataframe).diff().dropna()
    return diff


def MC(train, start_val, dataframe, nofsim, days_sim):
    """
    @params:
        dataframe: pandas dataframe of our stock
    @returns:


    """
    #Parameters
    mu = dataframe.mean()
    var = dataframe.var()
    drift = mu - 0.5*var
    std = dataframe.std()
    days = np.arange(days_sim)

    #Variables
    epsilon = norm.ppf(np.random.rand(len(days), nofsim))
    delta_x = drift.values + std.values*epsilon
    sim_values = np.zeros_like(delta_x)
    sim_values[0] = start_val

    #Simulation
    for t in range(1, len(days)):
        sim_values[t] = sim_values[t-1]*np.exp(delta_x[t])
```

```
    index_values = [i for i in range(0, len(days))]
    column_index = [i for i in range(0, nofsim)]
    new_dataframe = pd.DataFrame(sim_values)

    dates_train = pd.to_datetime(train.index.values)
    dates_sim = pd.date_range(str(dates_train[-1]), periods = days_sim,
    freq="D")
    new_dataframe.index = dates_sim

    return new_dataframe


def plot_tree(train, days_sim, mc_sim):
    dates_train = pd.to_datetime(train.index.values)
    dates_sim = pd.to_datetime(mc_sim.index.values)


    train.loc[dates_train[0] : dates_train[-1], "Close"].plot()

    for i in range(0, len(mc_sim.columns)):
        mc_sim.loc[dates_sim[0] : dates_sim[-1], i].plot(lw=0.3)
    plt.xlabel("Time")
    plt.ylabel("Stock value")
    plt.title("Plot of training data and simulated stock prices")
    plt.show()

def plot_dataframe(df):
    df.plot()
    plt.show()

def final_values(mc_sim):
    last_values = mc_sim.iloc[-1,:]
    return last_values

def plot_histogram(values):
    values.hist(bins=40, grid=True, figsize=(7,4), color = "#86bf91",
    zorder=2, rwidth=0.9)
    plt.xlabel("Stock value")
    plt.ylabel("Frequency")
    plt.title("Histogram of final simulated stock prices")
```

```
    plt.show()

def conf_interval(start_val, dataframe, nofsim):
    """
    @params:
        dataframe: pandas dataframe of our stock
    @returns:


    """
    #Parameters
    mu = dataframe.mean()
    var = dataframe.var()
    drift = mu - 0.5*var
    std = dataframe.std()
    days = np.arange(252)

    #Variables
    epsilon = norm.ppf(np.random.rand(len(days), nofsim))
    delta_x = drift.values + std.values*epsilon
    delta_x_interval_1 = np.zeros(len(days))
    delta_x_interval_2 = np.zeros(len(days))
    expected_delta_x = np.zeros(len(days))

    #Defining confidence intervals
    for t in range(1, len(days)):
        delta_x_interval_1[t] = drift*(t-days[0]) +
        std *1.96*np.sqrt(t - days[0])
        delta_x_interval_2[t] = drift*(t-days[0]) +
        std *-1.96*np.sqrt(t - days[0])

    S = np.zeros_like(delta_x)
    S_interval_1 = np.zeros_like(delta_x_interval_1)
    S_interval_2 = np.zeros_like(delta_x_interval_2)
    S_interval_1[0] = start_val
    S_interval_2[0] = start_val
    S[0] = start_val
    expected_delta_x[0] = start_val

    #Simulation of confidence interval
    for t in range(1, len(days)):
```

```
        S[t] = S[t-1]*np.exp(delta_x[t])
        S_interval_1[t] = start_val*np.exp(delta_x_interval_1[t])
        S_interval_2[t] = start_val*np.exp(delta_x_interval_2[t])
        expected_delta_x[t] = expected_delta_x[t-1]*np.exp(mu.values)

    return S,S_interval_1,S_interval_2,expected_delta_x


def plot_conf(S,S_interval_1,S_interval_2,expected_delta_x):
    plt.figure(figsize=(12.2,4.5))
    color = 'black'
    plt.plot(S)
    plt.plot(S_interval_1,color=color)
    plt.plot(S_interval_2,color=color)
    plt.plot(expected_delta_x,color=color)
    plt.title('Price of Amazon stock 1 year from now: 95% Confidence Interval')
    plt.xlabel('Time',fontsize=18)
    plt.ylabel('Price',fontsize=18)
    plt.show()


def conf_interval_values(last_values):
    last_values = last_values.to_numpy()
    interval = st.t.interval(alpha = 0.99, df = len(last_values)-1,
    loc = last_values.mean(), scale = st.sem(last_values))
    return interval
```

6.2 Part 2

6.2.1 main.py

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import random

from functions import *

def data_manipulation():
    pass

def simulate(portfolios, path):
    results = []
    for i in range(0, len(portfolios)):
        # Create our portfolio and prepare for simulation
        current_portfolio = portfolios[i]
        data = [read_csv(path + tick + ".csv", tick) for tick in
        current_portfolio]
        portfolio = merge_dataframes(data)
        train, test = split_timeseries(portfolio)
        weights_uniform = uniform_weights(portfolio)

        # Calculate
        rtrns_train = returns_timeseries(train)
        rtrns_test = returns_timeseries(test)
        covar_matrix = variance_matrix(rtrns_train)

        #plot_heatmap(covar_matrix)
        mc_train = monte_carlo_simulation(Portfolio = train,
        varianceMatrix= covar_matrix)
        optimal_portfolio = optimal_sharpe_ratio(mc_train)
        print(optimal_portfolio)
        results.append(run_live(rtrns_test, test, optimal_portfolio))

    return results

def run_live(rtrns_test, test, optimal_portfolio):
    covari_matrix_test = variance_matrix(rtrns_test)
```

```
    rf = 0.01

    # We start with the uniform weights distribution
    weights_uniform = uniform_weights(test)
    individ_rtrns_uniform = test.resample("D").last().pct_change().mean()
    rtrns_uniform = np.dot(weights_uniform, individ_rtrns_uniform)
    var_uniform = covari_matrix_test.mul(weights_uniform,
    axis=0).mul(weights_uniform, axis=1).sum().sum()
    sd_uniform = np.sqrt(var_uniform)
    ann_sd_uniform = sd_uniform*np.sqrt(250)
    sharpe_ratio_uniform = (rtrns_uniform - rf)/ann_sd_uniform

    # Now we calculate for the optimal allocation
    weights_optimal = [optimal_portfolio[i] for i in range(2,
    len(optimal_portfolio))]
    individ_rtrns_optimal = test.resample("D").last().pct_change().mean()
    rtrns_optimal = np.dot(weights_optimal, individ_rtrns_optimal)
    var_optimal = covari_matrix_test.mul(weights_optimal,
    axis=0).mul(weights_optimal, axis=1).sum().sum()
    sd_optimal = np.sqrt(var_optimal)
    ann_sd_optimal = sd_optimal*np.sqrt(250)
    sharpe_ratio_optimal = (rtrns_optimal - rf)/ann_sd_optimal

    return (sharpe_ratio_uniform, sharpe_ratio_optimal)

def print_results(sharpe_ratio_uniform, sharpe_ratio_optimal):
    print("Sharpe ratio uniform: ", sharpe_ratio_uniform)
    print("Sharpe ratio optimal: ", sharpe_ratio_optimal)
    performance = 100 * (sharpe_ratio_optimal -
    sharpe_ratio_uniform)/abs(sharpe_ratio_uniform)
    print("Performance: ", performance, " %")


def main():
    tickers = ["AMZN", "BAC", "BRK-B", "COST", "CVS", "ET",
               "FDX", "GM", "GOOG", "GS", "HP", "IBM",
               "KO", "NKE", "PFE", "RTX", "TSLA", "XOM"]

    portfolios = [] # A list of all different portfolios we are going to test

    for i in range(0, 15):
```

```
        new_portfolio = random.sample(tickers,4)
        portfolios.append(new_portfolio)

    path = "/Users/edwardglockner/Library/CloudStorage/OneDrive-
    Uppsalauniversitet/Fristående Kurser/Inferensteori
    I/PortfolioOptimization/Data/"
    results = simulate(portfolios, path)

    for i in range(0, len(results)):
        print_results(results[i][0], results[i][1])



if __name__ == "__main__":
    main()
```

6.2.2 functions.py

```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
import seaborn as sns

# Here all the functions will be declared


def read_csv(file_path, tick):
    """
    @params:

    @returns:

    Reads a file_path for a given ticker, and returns a dataframe
    with a column: close.
    """
    temp = pd.read_csv(file_path, parse_dates=[0], index_col=0)
    temp.rename(columns={"Close" : tick}, inplace = True)
    temp.drop(["Open", "High", "Low", "Adj Close", "Volume"], axis = 1,
    inplace = True)
    return temp
```

```
def merge_dataframes(args):
    """
    @params:

    @returns:

    Merges dataframes into one dataframe.
    """
    Portfolio = args[0]
    for i in range(1, len(args)):
        Portfolio = pd.merge(Portfolio, args[i], on = "Date")
    return Portfolio

def split_timeseries(timeSeries):
    """
    @params:

    @returns:

    Splits a dataframe into a training set containing 80% of the data,
    and a testing set containing
    """
    Train = timeSeries[round(len(timeSeries)*0.5) : round(len(timeSeries)*0.8)]
    Test = timeSeries[round(len(timeSeries)*0.8) : len(timeSeries)]
    return Train, Test

def uniform_weights(timeSeries):
    """
    @params:

    @returns:

    """
    numAssets = len(timeSeries.columns)
    return [1/numAssets for i in range(0, numAssets) ]

def plot_timeSeries(timeSeries):
    """
    @params:

    @returns:
```

```
    """
    timeSeries.plot()
    plt.show()

def returns_timeseries(timeSeries):
    """
    @params:

    @returns:

    """
    return timeSeries.pct_change()

def returns_allocated(returns_timeseries, weights):
    """
    @params:

    @returns:

    """
    return returns_timeseries.dot(weights)

def plot_returns(returns):
    """
    @params:

    @returns:

    """
    returns.plot(colormap= "jet", title = "sdfsdf")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.show()

def histogram_returns(returns):
    """
    @params:

    @returns:
```

```
    """
    returns.hist(bins=40, grid=True, figsize=(7,4), color='#86bf91',
    zorder=2, rwidth=0.9)
    plt.xlabel("Returns")
    plt.ylabel("Frequency")
    plt.title("sdfsf")
    plt.show()

def variance_matrix(returns):
    """
    @params:

    @returns:

    """
    return returns.cov()*252

def variance_timeseries(VarianceMatrix, Weights):
    """
    @params:

    @returns:

    """
    return np.transpose(Weights)@VarianceMatrix@Weights

def volatility_timeseries(VarianceMatrix, Weights):
    """
    @params:

    @returns:

    """
    return np.sqrt(variance_timeseries(VarianceMatrix, Weights))

def monte_carlo_simulation(Portfolio, varianceMatrix, numSimulations=20000):
    """
    @params:

    @returns:
```

```
    """
    PortfolioReturns = []
    PortfolioWeights = []
    PortfolioVolatility = []
    numAssets = len(Portfolio.columns)
    individual_returns = Portfolio.resample("D").last().pct_change().mean()
    for port in range(numSimulations):
        weights = np.random.random(numAssets)
        weights = weights/np.sum(weights)
        PortfolioWeights.append(weights)
        returns = np.dot(weights, individual_returns)
        PortfolioReturns.append(returns)
        var = varianceMatrix.mul(weights, axis=0).mul(weights,
        axis=1).sum().sum()
        sd = np.sqrt(var)
        ann_sd = sd*np.sqrt(250)
        PortfolioVolatility.append(ann_sd)

    data = {"Returns" : PortfolioReturns, "Volatility" : PortfolioVolatility}
    for counter, symbol in enumerate(Portfolio.columns.tolist()):
        data[symbol + " weight"] = [w[counter] for w in PortfolioWeights]


    return pd.DataFrame(data)

def optimal_sharpe_ratio(PortfolioVersions):
    """
    @params:

    @returns:

    """
    rf = 0.01
    optimal_risky_portfolio = PortfolioVersions.
    iloc[((PortfolioVersions["Returns"] -
    rf)/PortfolioVersions["Volatility"]).idxmax()]
    return optimal_risky_portfolio

def calculate_sharpe_ratio(portfolio):
    """
    @params:
```

```
    @returns:

    """
    rf = 0.01
    return (portfolio["Returns"]-rf)/portfolio["Volatility"]

def plot_heatmap(covariance_matrix):
    ax = sns.heatmap(covariance_matrix, linewidth=0.5)
    plt.show()
```

6.2.3 stats.r

```
before <-c(-0.001900, -0.002565, -0.001803, -0.002425, -0.001979, -0.002022,
-0.002627, -0.002258, -0.002811, -0.002087, -0.001928, -0.002327, -0.001528,
-0.001777, -0.002458)
after <-c(-0.000749, -0.002134, -0.001594, -0.001740, -0.001784, -0.001786,
-0.001616, -0.001620, -0.002175, -0.001768, -0.001731, -0.001775, -0.000723,
-0.000739, -0.002309)

t.test(before, after, paired = TRUE, alternative="less")
```