Mandatory Assignment 1

vsc879, zqd848 og jzr636

2024-03-08

Imports

```
# imports
import numpy as np
import pandas as pd
import yfinance as yf
import scipy.optimize as sco
import matplotlib.pyplot as plt
```

1

Follow the exercise sets to download daily adjusted prices for all constituents of the Dow Jones 30 index for the period from January 1st, 2000 until December 31st, 2023 from Yahoo!Finance. Remove all tickers with no continuous trading history for the entire sample (hint: you should end up with $= 27$ assets). Compute monthly returns for each of the tickers.

```
# get down jones tickers
djones_data = pd.read_excel(r'holdings-daily-us-en-dia.xlsx', skiprows=4)
djones_ticks = list(djones_data['Ticker'].dropna())

# collect adjusted prices
rawdata = pd.DataFrame(yf.download(
    tickers=djones_ticks,
    start="2000-01-01",
    end="2023-12-31"
))
adj_close_hist = rawdata['Adj Close'].copy()
adj_close_hist.columns = djones_ticks

# remove non-continuos assets
prices = adj_close_hist.dropna(axis=1, how='any')
djones_ticks = list(prices.columns)
```

```
print(f'Number of assets in cleaned dataset: {len(prices.columns)}\n')

# calc monthly returns for each asset
prices.index = pd.to_datetime(prices.index)
monthly_returns = prices.resample('ME').last().pct_change()
monthly_returns = monthly_returns.dropna()
```

```
Number of assets in cleaned dataset: 27
```

2

Compute the sample mean   and the variance-covariance matrix Σ of the monthly returns.1 Which of the   individual assets delivered the highest Sharpe ratio (assume the risk-free rate is zero) during the sample period?

```
# Compute the sample means
mu = monthly_returns.mean()

# Compute the variance-covariance matrix
Sigma = monthly_returns.cov()

# Calculate Sharpe ratio for each asset
sharpe_ratios = mu / monthly_returns.std()

# Identify the asset with the highest Sharpe ratio
max_sharpe_tick = sharpe_ratios.idxmax()
max_sharpe = sharpe_ratios.max()

print(f"The asset with the highest Sharpe ratio is {max_sharpe_tick} with a Sharpe ratio o
```

```
The asset with the highest Sharpe ratio is CSCO with a Sharpe ratio of 0.27
```

3

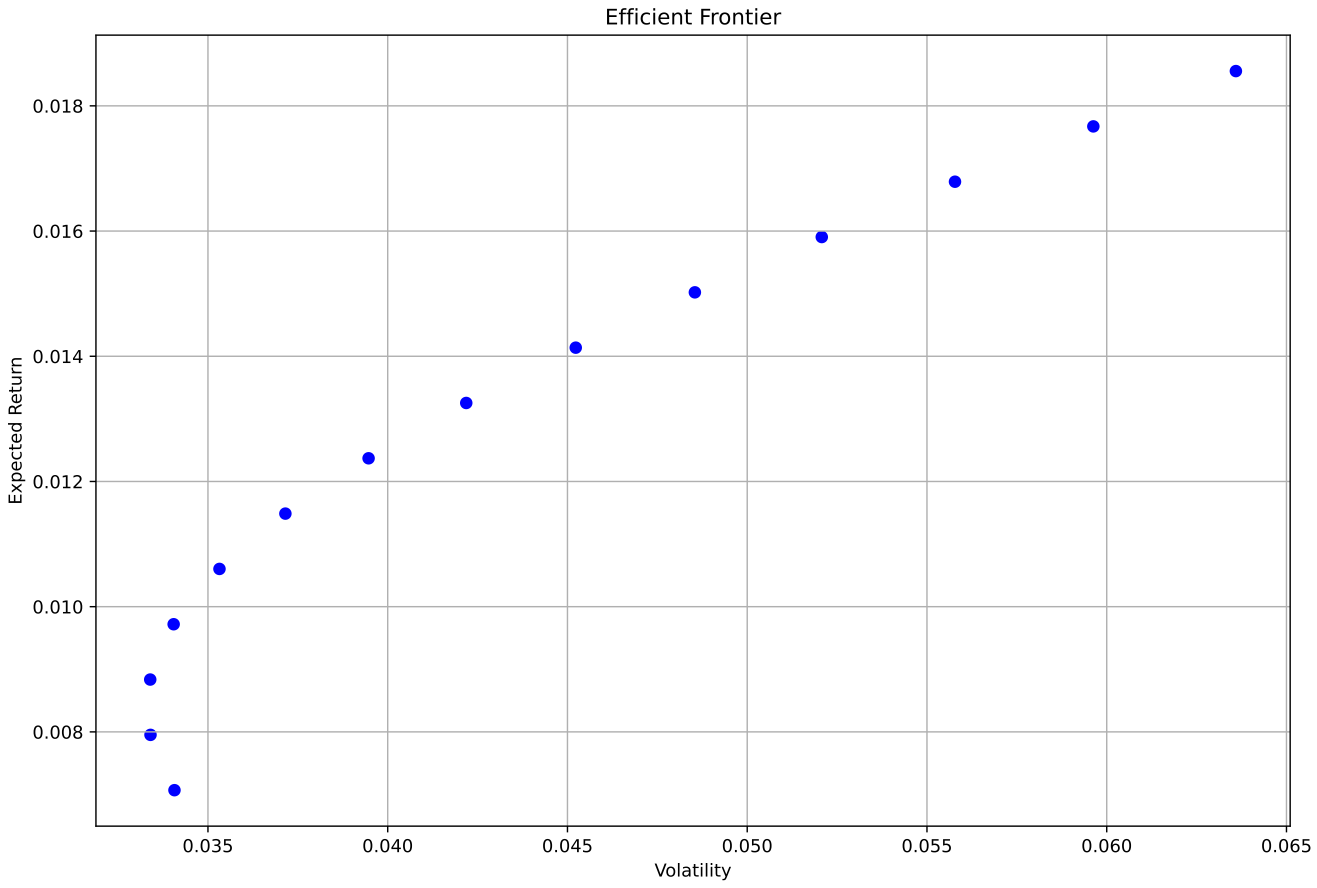
Define a function compute_efficient_frontier which takes at least two inputs: a $\times$ variance-covariance matrix Sigma_est and a vector mu_est.

```
def compute_efficient_frontier(Sigma_est, mu_est):
    N = len(mu_est)
    min_variance = lambda weights: weights.T @ Sigma_est @ weights # Calculating the weigh
    eff_portfolio = lambda weights: -mu_est @ weights # We wish to minimize, hence the neg
    constraints = [{'type': 'eq', 'fun': lambda weights: np.sum(weights) - 1}]
    bounds = tuple((0, 1) for _ in range(N)) # Weights cannot exceed 100%, therefor we set
    initial_guess = np.ones(N) / N

    min_var_result = sco.minimize(min_variance, initial_guess, method='SLSQP', bounds=bound
    omega_mvp = min_var_result.x # Extracting the weights for the minimum variance portfoli
    mvp_return = mu_est @ omega_mvp # Calculating the return

    return_constraint = {'type': 'eq', 'fun': lambda weights: mu_est @ weights - 2 * mvp_re
    eff_var_result = sco.minimize(eff_portfolio, initial_guess, method='SLSQP', bounds=bou
    omega_eff = eff_var_result.x # Extracting the weights for the minimum variance portfoli

    c_values = np.arange(-0.1, 1.21, 0.1) # Setting the given interval for values of c.
    portfolios = [(c * omega_mvp + (1 - c) * omega_eff) for c in c_values]

    df = pd.DataFrame(portfolios, columns=[f'Asset {i+1}' for i in range(N)])
    df['c'] = c_values

    return df[['c'] + [f'Asset {i+1}' for i in range(N)]]
```

4

Use the output of the function compute_efficient_frontier(Sigma_est, mu_est, ...) to visualize the theoretically optimal efficient frontier in a diagram with volatility on the x-axis and expected returns on the y-axis based on the true parameters Σ and

```
# Using the function to calculate the effeicient frontier
efficient_frontier_df = compute_efficient_frontier(Sigma, mu)

assets = [f'Asset {i+1}' for i in range(len(mu))]
expected_returns = []
volatilities = []

#Iterate over rows in the efficient_frontier_df DataFrame
for _, row in efficient_frontier_df.iterrows():
    omega_c = row[assets].values # Finding portfolio weights for current row
    expected_returns.append(np.dot(omega_c, mu)) # Expected return for the portfolio
    volatilities.append(np.sqrt(np.dot(omega_c.T, np.dot(Sigma, omega_c)))) # Volatility fo

# Plotting the figure of the efficient frontier
plt.figure(figsize=(12, 8))
plt.scatter(volatilities, expected_returns, c='blue', marker='o')

plt.title('Efficient Frontier')
plt.xlabel('Volatility')
plt.ylabel('Expected Return')
plt.grid(True)
plt.show()
```



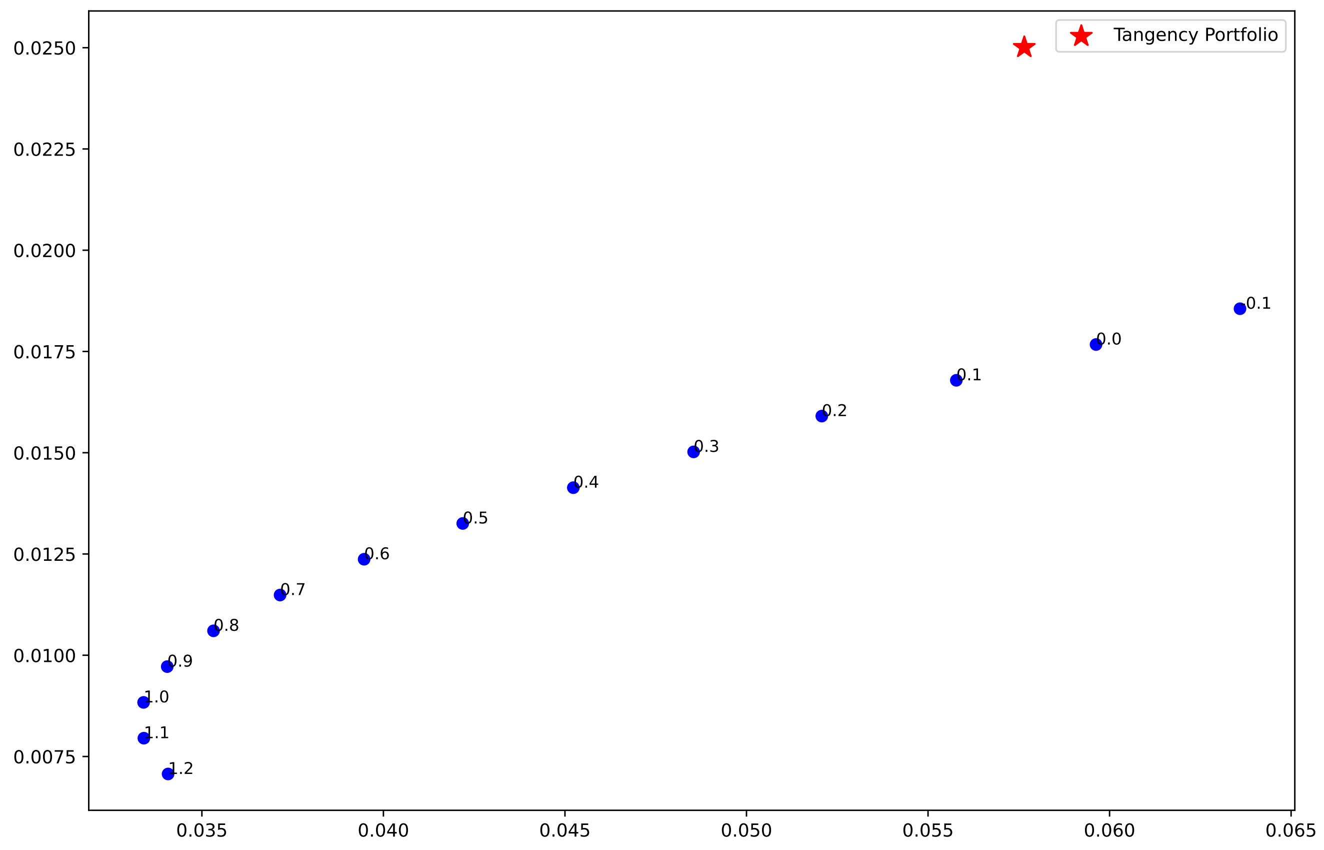
5

What are the efficient tangency portfolio weights tgc under the assumption that the risk- free rate is zero based on the true parameters   and Σ? Do the tangency portfolio weights seem like a well-balanced portfolio? What are the potential issues when implementing this portfolio in reality? What is the maximum attainable Sharpe ratio assuming the risk-free rate is zero? Should the Sharpe ratio of tgc be higher or lower than the Sharpe ratio of the individual assets?

```
# risk free asset is assumed to be zero
rf = 0

# To calculate tangency portfolio weights, we first have to find the inverse of the varian
Sigma_inv = np.linalg.inv(Sigma)
iota = np.ones(len(mu))
```

```
# Computing the tangency portfolio weights
omega_tgc = (Sigma_inv @ (mu -rf*iota)) / np.dot(iota, np.dot(Sigma_inv, mu))
sum_omega_tgc = omega_tgc.sum()
print(f" The weights of the tangency portfolio sum to: {round(sum_omega_tgc,2)}\n")

# using the eqights we can calculate the expected return and volatility of the tangency po
expected_return_tgc = np.dot(mu, omega_tgc)
volatility_tgc = np.sqrt(np.dot(omega_tgc.T, np.dot(Sigma, omega_tgc)))

# Compute the Sharpe ratio of the tangency portfolio
sharpe_ratio_tgc = expected_return_tgc / volatility_tgc

# Plotting the efficient frontier with c's as labels for each datapoint
plt.figure(figsize=(12, 8))
for i, txt in enumerate(efficient_frontier_df['c']):
    plt.scatter(volatilities[i], expected_returns[i], c='blue', marker='o')
    plt.text(volatilities[i], expected_returns[i], f'{txt:.1f}', fontsize=9)

# Plotting the tangency portfolio on the efficient frontier
plt.scatter(volatility_tgc, expected_return_tgc, c='red', marker='*', s=150, label='Tangen
plt.legend()
```

```
 The weights of the tangency portfolio sum to: 1.0
```



6

You are given the function:

```
def simulate_returns(periods=200,
    expected_returns=mu,
    covariance_matrix=Sigma):
    """
    periods (int): Number of periods
    expected_returns (array-like): Expected returns for each asset
    covariance_matrix (array-like): Covariance matrix of returns
    """
    3
    returns = np.random.multivariate_normal(expected_returns,
    covariance_matrix,
    size=periods)
    return returns
```

Explain briefly what the function is doing.

Simulation Process: The function uses the np.random.multivariate_normal method from NumPy to generate random samples from a multivariate normal distribution. This method requires the mean (expected returns), the covariance matrix, and the number of samples (size) to generate. expected_returns serves as the mean of the distribution, indicating the average return expected for each asset. covariance_matrix provides the covariances between the asset returns, which are essential for capturing the relationships between different assets' performances. periods determines how many sets of returns will be generated, with each set containing a return value for each asset.

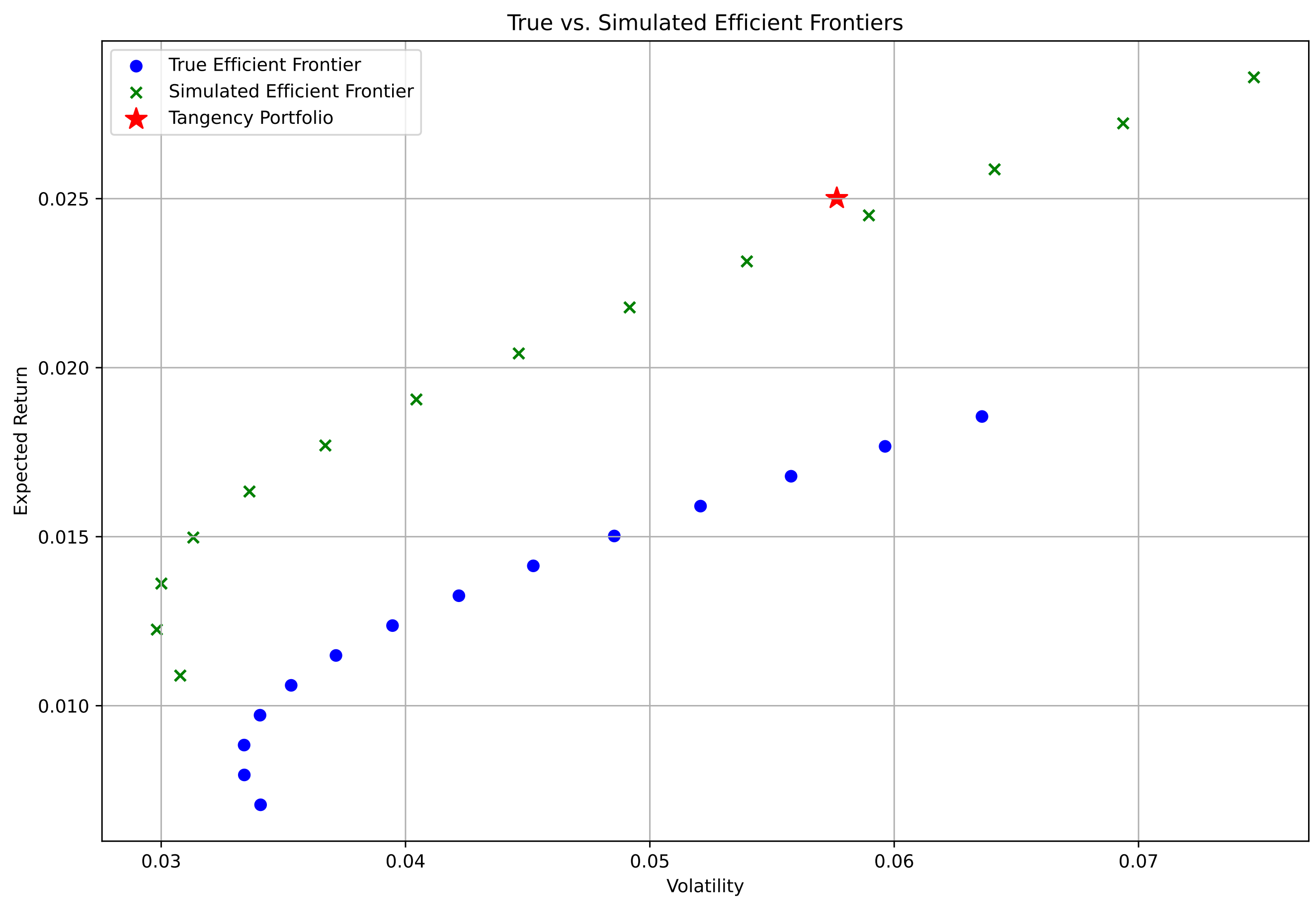
Output: The function returns a NumPy array with dimensions periods periods x N, where each row represents a set of simulated returns for all assets in one period, and N is the number of assets.

In summary, the simulate_returns function is a tool for generating synthetic asset return data based on specified expected returns and covariance among the assets. This simulated data can be used to study the properties of financial models, test investment strategies, or understand the impact of estimation uncertainty on portfolio optimization, such as the construction of efficient frontiers.

7

Use simulate_returns(periods = 200) to generate one hypothetical sample of size periods = 200. For this simulated sample, compute the sample mean, the sample variance- covariance matrix, and plug-in estimates of the mean–variance frontier again. Evaluate how close these estimates come to the "true" frontier based on exercise 3 by plotting them meaningfully in the figure from above. To be precise: We assume that the investor has the sample estimates available while the actual return distribution is determined by Σ and (which are unkown to her). Why do the simulated efficient frontiers deviate from the theoretically optimal one?

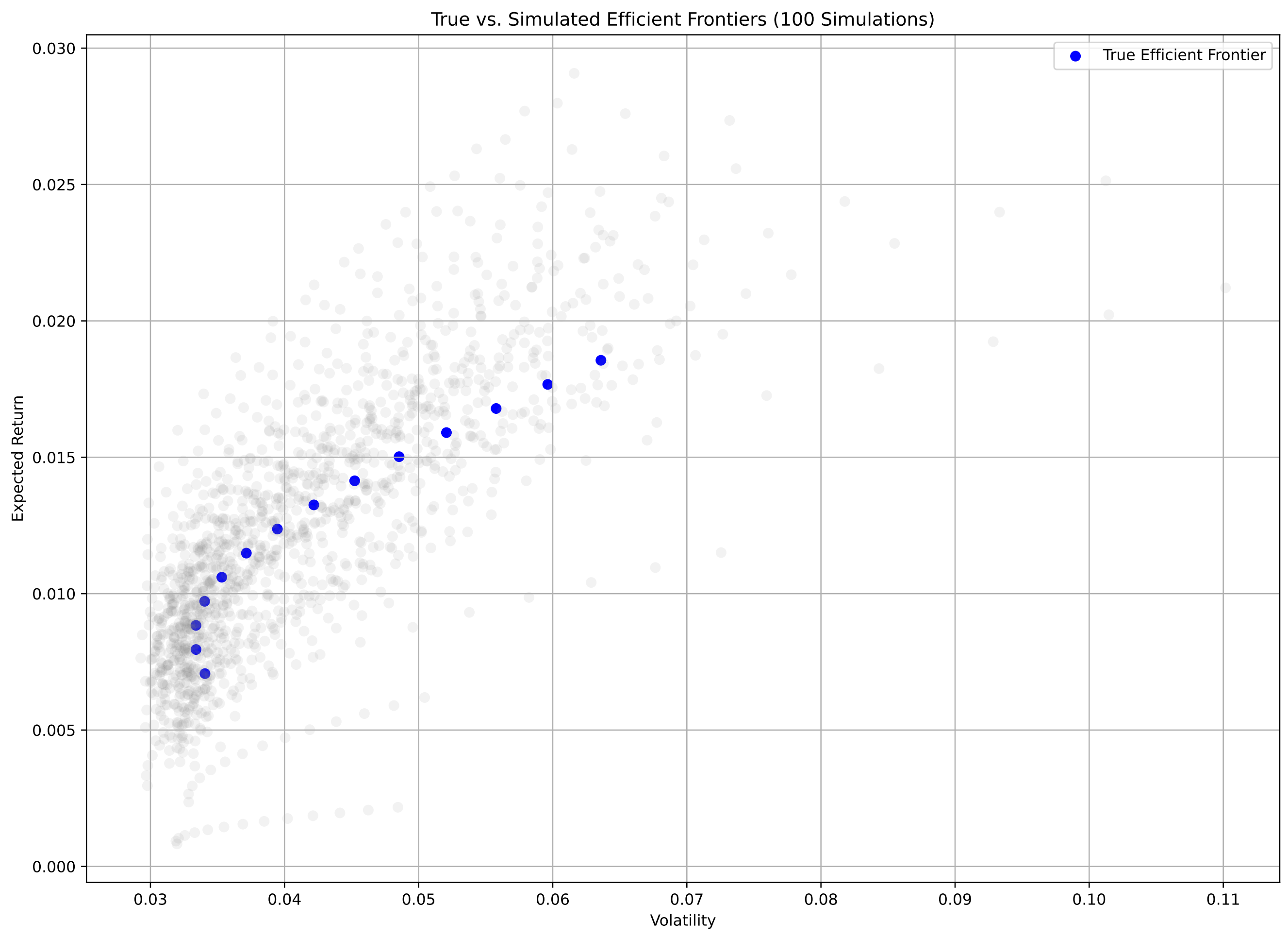
```
# Step 1: Generate simulated returns
simulated_returns = simulate_returns(periods=200, expected_returns=mu, covariance_matrix=S

# Step 2: Compute sample mean and variance-covariance matrix
sample_mu = np.mean(simulated_returns, axis=0)
sample_Sigma = np.cov(simulated_returns, rowvar=False)

# Compute the efficient frontier for the simulated data (using the sample estimates)
sample_efficient_frontier_df = compute_efficient_frontier(sample_Sigma, sample_mu)

# Compute expected returns and volatilities for the simulated efficient frontier
sample_expected_returns = []
sample_volatilities = []

for index, row in sample_efficient_frontier_df.iterrows():
    omega_c = row[assets].values  # Portfolio weights
    sample_expected_return = np.dot(omega_c, sample_mu)  # Expected return
    sample_volatility = np.sqrt(np.dot(omega_c.T, np.dot(sample_Sigma, omega_c)))  # Volat
    sample_expected_returns.append(sample_expected_return)
    sample_volatilities.append(sample_volatility)

# Plotting both the true and simulated efficient frontiers
plt.figure(figsize=(12, 8))

# True Efficient Frontier
plt.scatter(volatilities, expected_returns, c='blue', marker='o', label='True Efficient Fr

# Simulated Efficient Frontier
plt.scatter(sample_volatilities, sample_expected_returns, c='green', marker='x', label='Si

# Tangency Portfolio on the True Frontier
plt.scatter(volatility_tgc, expected_return_tgc, c='red', marker='*', s=150, label='Tangen

plt.title('True vs. Simulated Efficient Frontiers')
plt.xlabel('Volatility')
plt.ylabel('Expected Return')
plt.legend()
plt.grid(True)
plt.show()
```



8

Repeat the simulation step 100 times and visualize all simulated efficient frontiers within one figure together with the theoretically optimal one. What can you conclude from the figure? How does the result depend on the sample size periods?

```
plt.figure(figsize=(14, 10))

# Plot the true efficient frontier
plt.scatter(volatilities, expected_returns, c='blue', marker='o', label='True Efficient Fro

# Running the simulation step 100 times
for simulation in range(100):
    # Generate simulated returns
    simulated_returns = simulate_returns(periods=200, expected_returns=mu, covariance_matr

    # Compute sample mean and variance-covariance matrix
    sample_mu = np.mean(simulated_returns, axis=0)
    sample_Sigma = np.cov(simulated_returns, rowvar=False)
    # Compute the efficient frontier for the simulated data
    sample_efficient_frontier_df = compute_efficient_frontier(sample_Sigma, sample_mu)

    # Compute expected returns and volatilities for the simulated efficient frontier
    sample_expected_returns = []
    sample_volatilities = []

    for index, row in sample_efficient_frontier_df.iterrows():
        omega_c = row[assets].values  # Portfolio weights
        sample_expected_return = np.dot(omega_c, sample_mu)  # Expected return
        sample_volatility = np.sqrt(np.dot(omega_c.T, np.dot(sample_Sigma, omega_c)))  # Vo
        sample_expected_returns.append(sample_expected_return)
        sample_volatilities.append(sample_volatility)

    # Plot each simulated efficient frontier
    plt.scatter(sample_volatilities, sample_expected_returns, c='grey', alpha=0.1)

# Final plot adjustments
plt.title('True vs. Simulated Efficient Frontiers (100 Simulations)')
plt.xlabel('Volatility')
plt.ylabel('Expected Return')
plt.legend()
plt.grid(True)
plt.show()
```

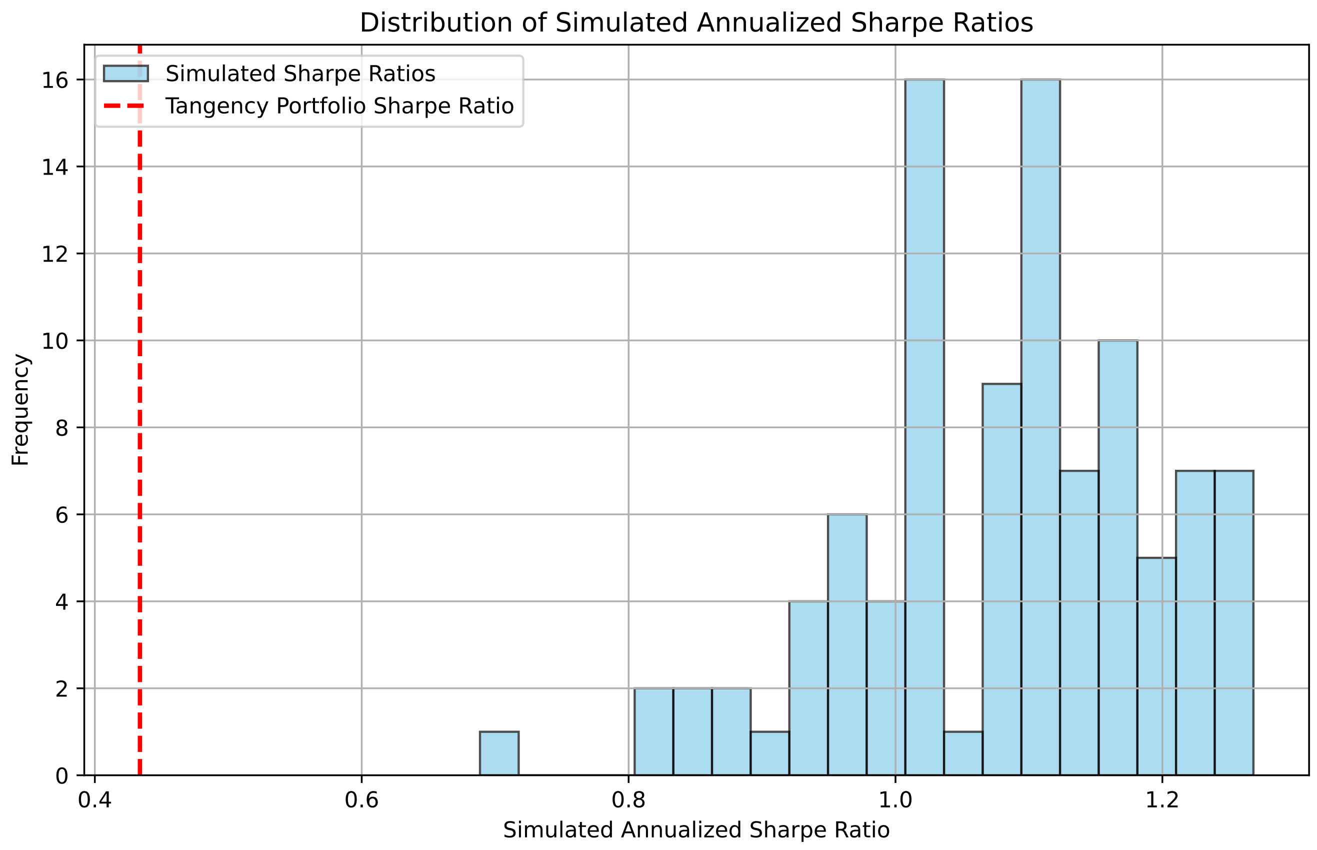


9

Compute for each of the simulated return samples the efficient tangent portfoliô (̂Σ̂,) under the assumption that the risk-free rate is zero. Then, for each of the simulated sam- ples, compute the annualized Sharpe-ratio of the efficient portfolio evaluated with the true parameters ($= \sqrt{12}$ $/\sqrt{\hat{}}$ Σ).

Visualize these Sharpe ratios in a histogram together with the Sharpe ratio of the efficient tangency portfolio based on the true parameters and Σ

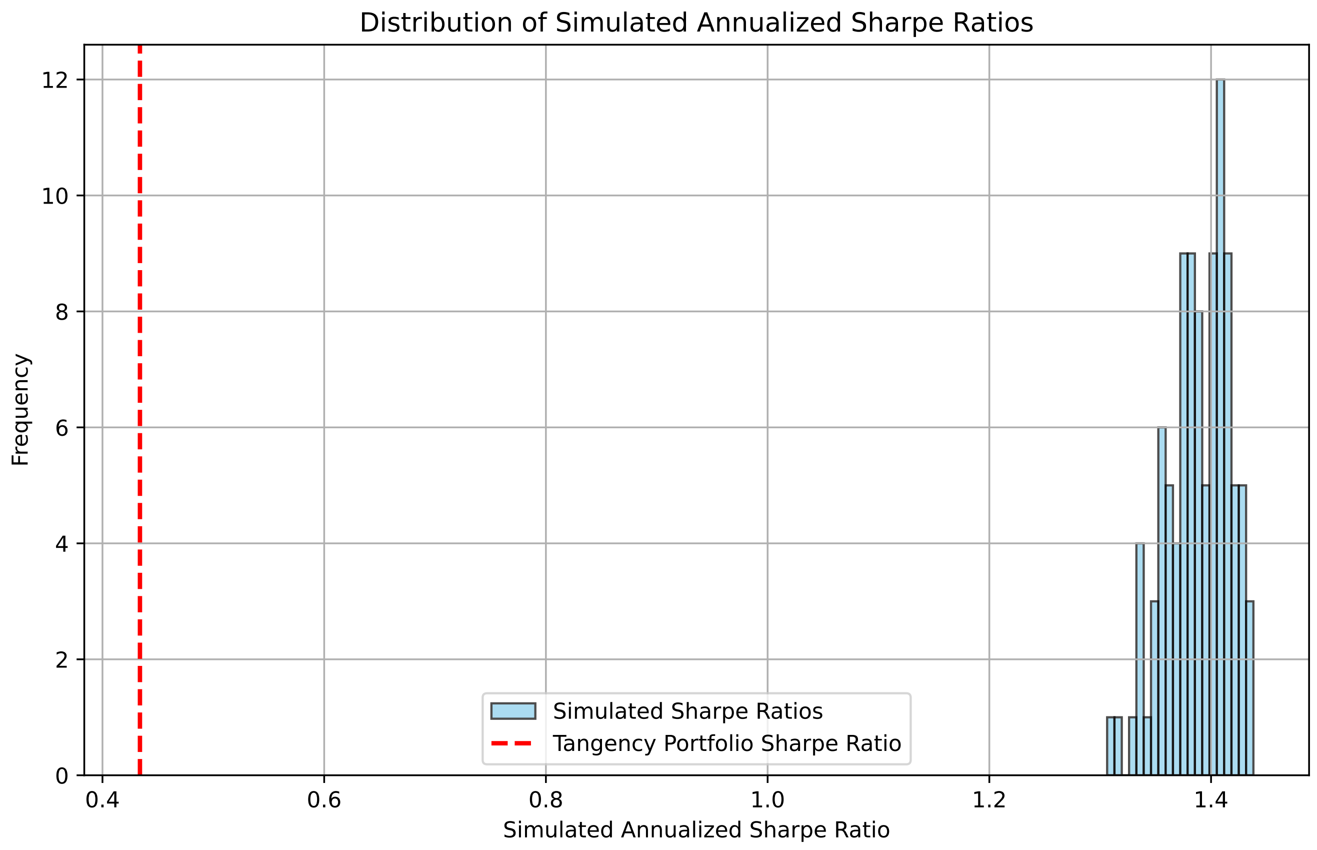
```
sharpe_ratios_simulated = []

# We run the simulation again, but in this for loop we calculate the efficient tangent por
for simulation in range(100):
    # Generate simulated returns
    simulated_returns = simulate_returns(periods=200, expected_returns=mu, covariance_matr

    # Compute sample mean and variance-covariance matrix
    sample_mu = np.mean(simulated_returns, axis=0)
    sample_Sigma = np.cov(simulated_returns, rowvar=False)

    # Calculate tangency portfolio weights
    Sigma_inv = np.linalg.inv(sample_Sigma)
    iota = np.ones(len(sample_mu))
    numerator = Sigma_inv @ (sample_mu - rf * iota)
    denominator = iota.T @ Sigma_inv @ (sample_mu - rf * iota)
    omega_tgc = numerator / denominator

    # Calculate expected return and volatility of the efficient portfolio using true param
    expected_return_efficient_portfolio = mu @ omega_tgc
    volatility_efficient_portfolio = np.sqrt(omega_tgc.T @ Sigma @ omega_tgc)

    # Calculate annualized Sharpe ratio
    sharpe_ratio = np.sqrt(12) * expected_return_efficient_portfolio / volatility_efficien

    # Append Sharpe ratio to the list
    sharpe_ratios_simulated.append(sharpe_ratio)

# Plot the distribution of simulated Sharpe ratios
plt.figure(figsize=(10, 6))
plt.hist(sharpe_ratios_simulated, bins=20, color='skyblue', edgecolor='black', alpha=0.7,
# We have the sharpe ratio of the tangent portfolio from ealier, why we can plot it in the
plt.axvline(x=sharpe_ratio_tgc, color='red', linestyle='--', linewidth=2, label='Tangency

plt.xlabel('Simulated Annualized Sharpe Ratio')
plt.ylabel('Frequency')
plt.title('Distribution of Simulated Annualized Sharpe Ratios')
plt.legend()
plt.grid(True)
plt.show()
```



10

How do the results change for larger sample sizes periods? Discuss what you can conclude from the figure. Name at least two alternative allocation strategies that could help reduce the plug-in estimates' shortfall. Show if your proposed alternatives improve upon the benchmark we simulated above. Discuss the results

```
sharpe_ratios_simulated_2 = []

# We run the simulation again, but in this for loop we calculate the efficient tangent por
for simulation in range(100):
    # Generate simulated returns
    simulated_returns = simulate_returns(periods=1000, expected_returns=mu, covariance_mat

    # Compute sample mean and variance-covariance matrix
    sample_mu = np.mean(simulated_returns, axis=0)
    sample_Sigma = np.cov(simulated_returns, rowvar=False)

    # Calculate tangency portfolio weights
    Sigma_inv = np.linalg.inv(sample_Sigma)
    iota = np.ones(len(sample_mu))
    numerator = Sigma_inv @ (sample_mu - rf * iota)
    denominator = iota.T @ Sigma_inv @ (sample_mu - rf * iota)
    omega_tgc = numerator / denominator

    # Calculate expected return and volatility of the efficient portfolio using true param
    expected_return_efficient_portfolio = mu @ omega_tgc
    volatility_efficient_portfolio = np.sqrt(omega_tgc.T @ Sigma @ omega_tgc)

    # Calculate annualized Sharpe ratio
    sharpe_ratio = np.sqrt(12) * expected_return_efficient_portfolio / volatility_efficien

    # Append Sharpe ratio to the list
    sharpe_ratios_simulated_2.append(sharpe_ratio)

# Plot the distribution of simulated Sharpe ratios
plt.figure(figsize=(10, 6))
plt.hist(sharpe_ratios_simulated_2, bins=20, color='skyblue', edgecolor='black', alpha=0.7
# We have the sharpe ratio of the tangent portfolio from ealier, why we can plot it in the
plt.axvline(x=sharpe_ratio_tgc, color='red', linestyle='--', linewidth=2, label='Tangency

plt.xlabel('Simulated Annualized Sharpe Ratio')
plt.ylabel('Frequency')
plt.title('Distribution of Simulated Annualized Sharpe Ratios')
plt.legend()
plt.grid(True)
plt.show()
```



For Exercise 8 and 9, as the sample size (periods) increases, the results tend to stabilize and converge towards more accurate estimates of the true Sharpe ratio distribution. Larger sample sizes provide more data points for estimating parameters such as expected returns and covariance matrices, which improves accuracy in the Sharpe ratio.

When adjusting for the sample size "periods", we observe that as the sample size increases, the distribution of simulated Sharpe ratios tends to narrow around the true value, indicating reduced variability and more precise estimation of portfolio performance. This suggests that larger sample sizes lead to more reliable estimates and better reflect the underlying characteristics of the asset returns.

Considering Bayesian estimation and robust optimization as potential strategies to improve plug-in estimates, we could possibly return estimation results. We haven't succeeded in implementing this in our code.

```
knitr::purl("MA1.qmd")
```