CQF | INSTITUTE

Certificate in Quantitative Finance

Exam 1 Report

June 2022 Program

Name: Xiaodong Yang

Date: 24th August 2022

Contents

1 Products, Strategies and Pricing

1.1 Solve the following problems

Table 1: Summary Table

Questions	Solutions	
1.1(a)	Put Option Value:	1.80
1.1(b)	Forward Payoff:	$-K_1 + K_2$
1.1(c)	Future Price:	9.7872
1.2	Minimum Profit:	0.7503

(a) The binomial tree of the put option is shown in Figure 1. The probability of underlying price up can be solved based risk neutral. The option price is 1.80.

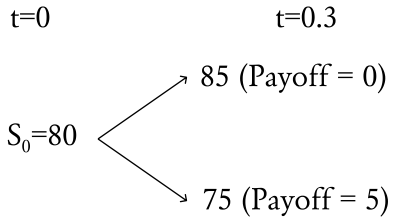


Figure 1: Binomial Tree of Put Option

The percentage of up is:

$$u = \frac{S_u}{S_0} = 1.0625$$

The percentage of down is:

$$v = \frac{S_v}{S_0} = 0.9375$$

The probability of upside is:

$$p' = \frac{1 - v + r\delta t}{u - v} = 0.6333$$

The value of the European put option can be written as:

$$V = \frac{1}{1 + r\delta t}\left(p'V^+ + (1 - p')V^-\right)$$

(b) The company holds two forward contracts. For the first contract, the company holds a long position that it will buy 10 million Japanese Yen on January 1st, 2010. Assuming the spot price is defined as K_1 and the forward price is defined as S_T. Hence, the payoff of the first forward contract is $S_T - K$. For another contract, the spot price is defined as K_2 the company hold a short position and its payoff is $K_2 - S_T$. Therefore, the net profit is $-K_1 + K_2$.

This strategy can be used to hedge the foreign exchange risk from the volatility of the Japanese Yen in the future. In this case, it's not a perfect hedge since the maturity of two forwards does not match each other. Hence, the company need to take the foreign exchange risk from July 1st 2009 to September 1st 2009. The conditions of a perfect hedge include the same underlying (Japanese Yen), the same maturity, and the same amount of money.

(c) The Equation 1 shows the valuation formula of futures. The future price is 9.7872.

$$K = (S_0 + C - D)e^{\delta t} \tag{1}$$

S_0: the price of the future at $t = 0$
C: the storage cost of the future
D: the dividends of the future
δt: the time step

1.2 European Option - Arbitrage

The equation that expresses put-call parity is:

$$C + Ke^{-rt} = P + S \tag{2}$$

The price of call option can be calculated by Equation 2, which is equal to -0.7503. Assuming the lowest price of call option is equal to 0 in extreme situation, the profit is 0.7503 at least if we hold a call option. Hence, the minimum profit is 0.7503. The arbitrage portfolio can be constructed by short a put option and underlying, and long risk-free assets with value of Ke^{-rt}.

1.3 Pricing Options by Binomial Method

The Binomial method is employed to price the call option. The underlying price has two paths: up or down in every time step. And interest rate and volatility of asset need to be used in this method.

The up factor is defined as:

$$u = 1 + \sigma\sqrt{\delta t}$$

The down factor is defined as:

$$v = 1 - \sigma\sqrt{\delta t}$$

The probability of underlying price up is defined as:

$$p' = \frac{1}{2} + \frac{r\sqrt{\delta t}}{2\sigma}$$

The value of the call option can be written as:

$$V = \frac{1}{1 + r\delta t}(p'V^+ + (1 - p')V^-)$$

1.3.1 Option Values with Different Volatilities

The underlying price, strike price, interest rate, the maturity, and time step are given, they are 100, 100, 0.05, 1, 4 respectively. In this part, the range of volatility of underlying is set as from 0.01 to 1.

Then, we price the options with different volatility and plot it in Figure 2. From Figure 2, it indicates that the call option price with a higher volatility are more expensive than those with lower volatilities. Higher volatility means call option holders take more risk. Hence, they need to more compensation.

Python code is as follows:

```
spot, strike, rate, sigma, time, steps = 100, 100, 0.05, 0.2, 1, 4
vol_vec = np.arange(0.01,1,0.04)
V_vol = []
for i in range(len(vol_vec)):
    Value = binomial_option(spot,strike,rate,vol_vec[i],time,steps,output=2)[0,0]
    V_vol.append(Value)
```

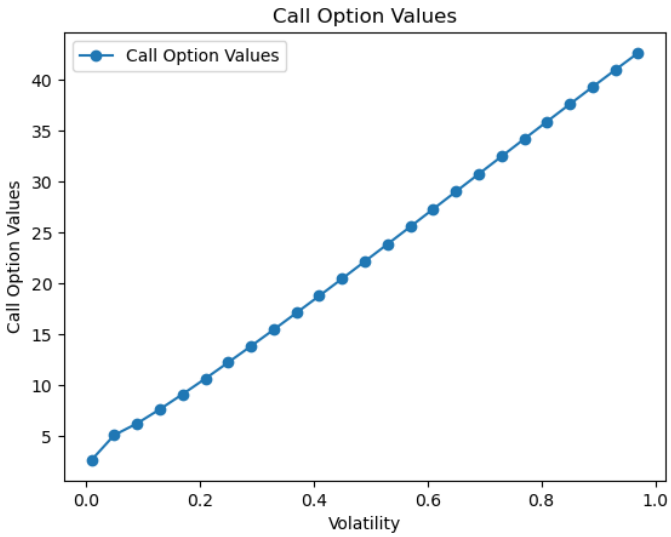


Figure 2: Call Option Values (Different Volatilities)

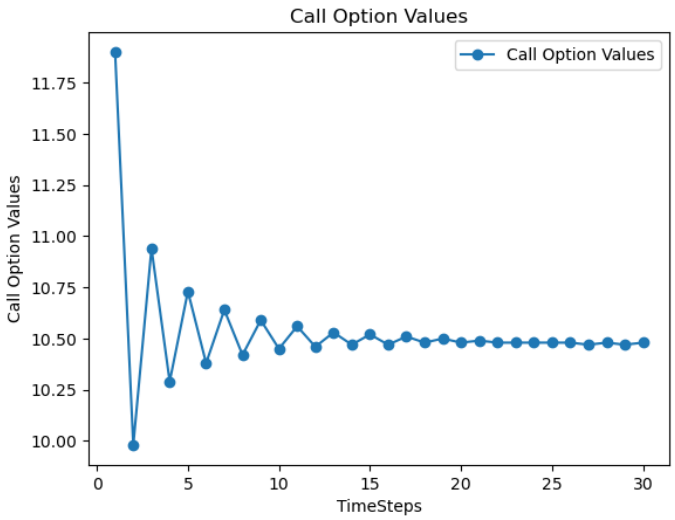


Figure 3: Call Option Values (Different Time Steps)

1.3.2 Option Values with Different Time Steps

In this part, the fixed volatility is 0.2. We price the call options with different time steps from 1 to 30. The call option values are shown in Figure 3. It discloses that the time steps are smaller, the option valuation is closer to the actual value. From Figure 3, the option valuation is stable when the time steps is more than 20.

Python code is as follows:

```
step_vec = np.arange(1,31)
V_tstep = []
for i in range(len(step_vec)):
    Value = binomial_option(spot, strike, rate, sigma, time, step_vec[i], output=2)[0,0]
    V_tstep.append(Value)
```

2 Portfolio Optimisation

Table 2: Summary Table

Questions	Solutions	
2.1	Covariance:	$\begin{pmatrix} 0.0049 & 0.00168 & 0.0063 & 0.00546 \\ 0.00168 & 0.0144 & 0.01512 & 0.01248 \\ 0.0063 & 0.01512 & 0.0324 & 0.04212 \\ 0.00546 & 0.01248 & 0.04212 & 0.0676 \end{pmatrix}$
2.2	Weights of Optimal Portfolio:	$\begin{pmatrix} 0.0587 \\ 0.7590 \\ -0.3195 \\ 0.5018 \end{pmatrix}$
2.2	Standard Deviation:	0.1323
2.3	Weights of Optimal Portfolio:	$\begin{pmatrix} 0.9054 \\ 0.8291 \\ -1.3746 \\ 0.6401 \end{pmatrix}$
2.3	Expected Return:	0.0336
2.3	Standard Deviation:	0.0258

2.1 Covariance Matrix

The vector of asset expected returns is given by:

$$\mu = \begin{pmatrix} 0.04 \\ 0.08 \\ 0.12 \\ 0.15 \end{pmatrix}$$

The vector of asset standard deviation is given by:

$$\sigma = \begin{pmatrix} 0.07 \\ 0.12 \\ 0.18 \\ 0.26 \end{pmatrix}$$

The correlation matrix of 4 assets is given by:

$$\rho = \begin{pmatrix} 1 & 0.2 & 0.5 & 0.3 \\ 0.2 & 1 & 0.7 & 0.4 \\ 0.5 & 0.7 & 1 & 0.9 \\ 0.3 & 0.4 & 0.9 & 1 \end{pmatrix}$$

The covariance matrix can be written as:

$$\Sigma = \sigma^T \rho \sigma = \begin{pmatrix} 0.0049 & 0.00168 & 0.0063 & 0.00546 \\ 0.00168 & 0.0144 & 0.01512 & 0.01248 \\ 0.0063 & 0.01512 & 0.0324 & 0.04212 \\ 0.00546 & 0.01248 & 0.04212 & 0.0676 \end{pmatrix}$$

Python code for this part is as follows:

```
mu_vec = np.array([0.04, 0.08, 0.12, 0.15]).reshape(4,1)
sigma_vec = np.array([0.07, 0.12, 0.18, 0.26]).reshape(4,1)
corr_mat = np.array([[1, 0.2, 0.5, 0.3],
                     [0.2, 1, 0.7, 0.4],
                     [0.5, 0.7, 1, 0.9],
                     [0.3, 0.4, 0.9, 1]])
cov_mat = sigma_vec * corr_mat * sigma_vec.reshape(1,4)
```

2.2 Optimisation

The portfolio selection problem is generally defined as a minimization of risk subject to a return constraint. Two reasons for this convention are: a return objective seems intuitively easier to formulate than a risk objective; and risks are easier to control than returns. In this project, the objective is to minimise the half of variance of the portfolio, and there are two constraints: the total weights of the portfolio is equal to 1 and the expected return of the portfolio is 10%. The 'w' is defined as the individual weight of the portfolio.

The objective:

$$\underset{\mathrm{w}}{Min} \frac{1}{2} \mathrm{w}^T \Sigma \mathrm{w}$$

Subject to constraints:

$$\mathrm{w}^T \mathbf{1} = 1$$

$$\mathrm{w}^T \mu = 0.1$$

The Lagrange function is be created with two Lagrange miltipliers λ and γ:

$$L = \frac{1}{2} \mathrm{w}' \Sigma \mathrm{w} + \lambda(m - \mathrm{w}'\mu) + \gamma(1 - \mathrm{w}'\mathbf{1})$$

The first order derivative is:

$$\frac{\partial L}{\partial \mathrm{w}} = \Sigma \mathrm{w} + \lambda \mu + \gamma \mathbf{1}$$

Then, setting the first order derivative is equal to 0, we get:

$$\Sigma \hat{\mathrm{w}} + \lambda \mu + \gamma \mathbf{1} = 0$$

We plug $\hat{w}$ to the constraints:

$$\mu'\Sigma^{-1}(\lambda\mu+\gamma\mathbf{1})=\lambda\mu'\Sigma^{-1}\mu+\gamma\mu'\Sigma^{-1}\mathbf{1}=m$$

$$\mathbf{1}'\Sigma^{-1}(\lambda\mu+\gamma\mathbf{1})=\lambda\mathbf{1}\Sigma^{-1}\mu+\gamma\mathbf{1}'\Sigma^{-1}\mathbf{1}=1$$

We define the following scalars:

$$\begin{cases} A=\mathbf{1}'\Sigma^{-1}\mathbf{1} \\ B=\mu'\Sigma^{-1}\mathbf{1}=\mathbf{1}'\Sigma^{-1}\mu \\ C=\mu'\Sigma^{-1}\mu \end{cases}$$

The constraint equations can be written as:

$$\begin{cases} C\lambda+B\gamma=m \\ B\lambda+A\gamma=1 \end{cases}$$

The λ and γ can be solved:

$$\Rightarrow \begin{cases} \lambda=\dfrac{Am-B}{AC-B^2} \\ \gamma=\dfrac{C-Bm}{AC-B^2} \end{cases}$$

We plug λ and γ to $\hat{w}$:

$$\hat{\mathrm{w}}=\frac{1}{AC-B^2}\Sigma^{-1}[(A\mu-B\mathbf{1})m+(C\mathbf{1}-B\mu)]$$

Python code for this part is as follows:

```
one_vec = np.array([1, 1, 1, 1]).reshape(4,1)
A = one_vec.reshape(1,4)@np.linalg.inv(cov_mat)@one_vec
B = one_vec.reshape(1,4)@np.linalg.inv(cov_mat)@mu_vec
C = mu_vec.reshape(1,4)@np.linalg.inv(cov_mat)@mu_vec
Lambda = (A*m-B)/(A*C-B**2)
Gamma = (C-B*m)/(A*C-B**2)
w_hat = np.linalg.inv(cov_mat)@(Lambda*mu_vec+Gamma*one_vec)
sigma_m = np.round(np.sqrt((A*m**2-2*B*m+C)/(A*C-B**2)),4)
mu_m = np.round(w_hat.reshape(1,4)@mu_vec,4)
```

The solutions are as follows:

$$\hat{\mathrm{w}}=\begin{pmatrix} 0.0587 \\ 0.7590 \\ -0.3195 \\ 0.5018 \end{pmatrix}$$

The variance of the portfolio is follows:

$$\sigma_\pi^2(m) = \frac{Am^2 - 2Bm + C}{AC - B^2} = 0.0175$$

or

$$\sigma_\pi^2 = w'\Sigma w = 0.0175$$

In Figure 4, the green marker is the optimal portfolio and its expected return and standard deviation are 0.1 and 0.1325 respectively. The red line is the efficient frontier.

2.3 Less Constrained Optimisation

In this part, there is only one constraint, which means that minimum variance of portfolio can be calculated given a specific return. The Markowitz Efficient Frontier is created by those portfolios with the minimum variance.

The Objective:

$$\underset{\mathrm{w}}{Min}\, \frac{1}{2}\mathrm{w}^T\Sigma\mathrm{w}$$

Subject to constraints:

$$\mathrm{w}^T\mathbf{1} = 1$$

The Lagrange function is be created with two Lagrange miltipliers γ:

$$L = \frac{1}{2}\mathrm{w}'\Sigma\mathrm{w} + \gamma(1 - \mathrm{w}\mathbf{1})$$

The first order derivative is:

$$\frac{\partial L}{\partial \mathrm{w}} = \Sigma\mathrm{w} + \gamma\mathbf{1}$$

Then, seeting the first order derivative is equal to 0, we get:

$$\Sigma\hat{\mathrm{w}} + \gamma\mathbf{1} = 0$$

$$\hat{\mathrm{w}} = \Sigma^{-1}(-\gamma\mathbf{1})$$

Plug w to the constaints:

$$\mathbf{1}'\Sigma^{-1}(-\gamma\mathbf{1}) = 1$$

We get γ and $\hat{\mathrm{w}}$:

$$\gamma = -\frac{1}{\mathbf{1}'\Sigma^{-1}\mathbf{1}}$$

$$\hat{\mathrm{w}} = \frac{1}{AC - B^2}\Sigma^{-1}[(A\mu - B\mathbf{1})m + (C\mathbf{1} - B\mu)]$$

The solutions are as follows:

$$\hat{\mathrm{w}}_G = \begin{pmatrix} 0.9054 \\ 0.8291 \\ -1.3746 \\ 0.6401 \end{pmatrix}$$

The expected return of minimum variance portfolio is:

$$\mu_G = 0.0336$$

The standard deviation of the portfolio is follows:

$$\sigma_G = 0.0258$$

In Figure 5, the blue marker is the minimum variance portfolio with the expected return of 0.0336 and a standard deviation of 0.0256. And the red line is the Markowitz Efficient Frontier. The returns of all portfolios on this efficient frontier are not less than 0.0336.

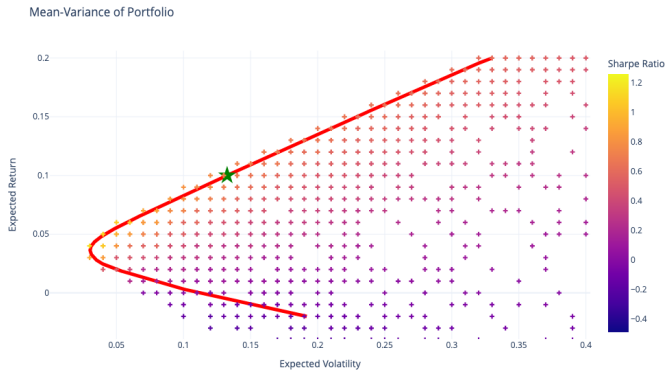


Figure 4: Mean-Variance of Portfolio

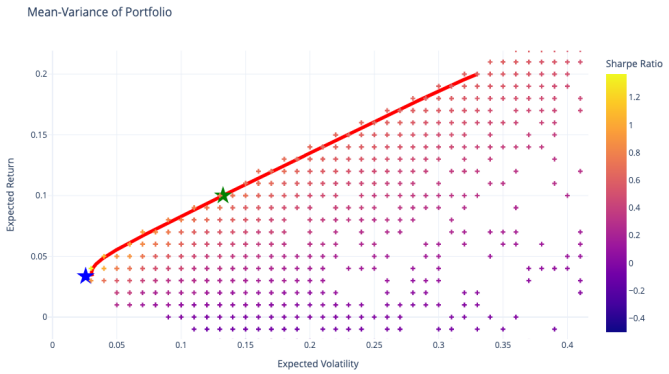


Figure 5: Mean-Variance of Portfolio

Python code for this part is as follows:

```
Gamma_g = np.round((-1)/(one_vec.reshape(1,4)@np.linalg.inv(cov_mat)@one_vec),4)
wg_hat = np.round(np.linalg.inv(cov_mat)@(-Gamma_g*one_vec),4)
mu_g = np.round(wg_hat.T@mu_vec,4)
sigma_g = np.round(np.sqrt(wg_hat.reshape(1,4)@cov_mat@wg_hat),4)
```

3 Empirical Value at Risk

Table 3: Summary Table

Questions	Solutions	
3.3	Breach Percentage(Method 1):	2.76%
3.3	Breach Percentage(Method 2):	2.25%

3.1 Calculate 99%/10 Day VaR using Sample Standard Deviation

10-day volatility can be calculated by:

$$\sigma_{\text{10-day}} = \sqrt{10 \times \sigma^2_{1-day}}$$

10-day return is defined as simple average of the sample multiply by constant:

$$\mu_{\text{10-day}} = 10 \times \mu = 0.0044$$

10-day VaR can be written as:

$$\text{VaR} = \mu_{\text{10-day}} + \sigma_{\text{10-day}} \times \text{Factor}$$

The VaR (99%/10-day) is a vector with length of 988.

Python codes for this part are as follows:

```
sigma_21D = pd.Series(df['Returns']).rolling(window=21, center=False).std().dropna()
sigma_10D = np.sqrt(10*(sigma_21D)**2)
mu_10D = np.mean(df['Returns'])*10
VaR_1 = norm.ppf(1-0.99, mu_10D, sigma_10D)
```

3.2 Calculate 99%/10 Day VaR using GARCH Model

The GARCH(1,1):

$$\sigma^2_n = \omega + \alpha\mu^2_{n-1} + \beta\sigma^2_{n-1}$$

The ω, α, and β are given, the values are 0.000001, 0.047, and 0.9466 respectively. The expected variance can be calculated by:

$$E(\sigma^2_n) = \omega + \alpha\mu^2_{n-1} + \beta\sigma^2_{n-1}$$

10-day volatility is from GARCH (1,1) Model. VaR (99%/10-day) can be get by the same as above approach, which contains 988 elements.

Python codes for this part are as follows:

```
# GARCH(1,1) function
def garch(omega, alpha, beta, ret):
    var = []
    for i in range(len(ret)):
        if i==0:
            var.append(omega/np.abs(1-alpha-beta))
        else:
            var.append(omega + alpha * ret[i-1]**2 + beta * var[i-1])

    return np.array(var)
omega, alpha, beta = 0.000001, 0.047, 0.9466
# Verifty variance values
var = garch(omega,alpha,beta,df['Returns'][-989:])
sigma_10d = np.sqrt(var[-988:]*10)
VaR_2 = norm.ppf(1-0.99, mu_10D, sigma_10d)
```

3.3 Calculate the percentage of VaR breaches for both measures

In summary, Figure 8 shows that the comparison between VaR calculated by sample standard deviation and GARCH (1,1) Model and realised returns (10-day). The breach is defined as the realised returns less than VaR. In this project, the percentages of breach are 2.76% and 2.25% respectively for two different methods.

Value-at-risk (VaR) defined as the worst loss that might be expected from holding a security or portfolio over a given period of time (usually 10 days for the purpose of regulatory capital reporting), given a specified level of confidence. VaR is a threshold in normal distribution of returns, which is can be calculated by expected returns, volatility, and critical value at specific confidence level. In Part C, volatility is modelling by two different approaches. Therefore, the VaR breaches are dependent on the level of VaR (Volatility). From Figure 8, when the volatility of realised returns is very high, its VaR is likely to be high, which increases the probability that the realised returns is below the VaR.

Another conclusion is that the VaR breaches are independent in time. The changes of the prices of the financial asset are stochastic processes. In other words, it's a Wiener process, which means the prices of underlying is independent in a extreme short time interval. Hence, the VaR breaches are independent in time.

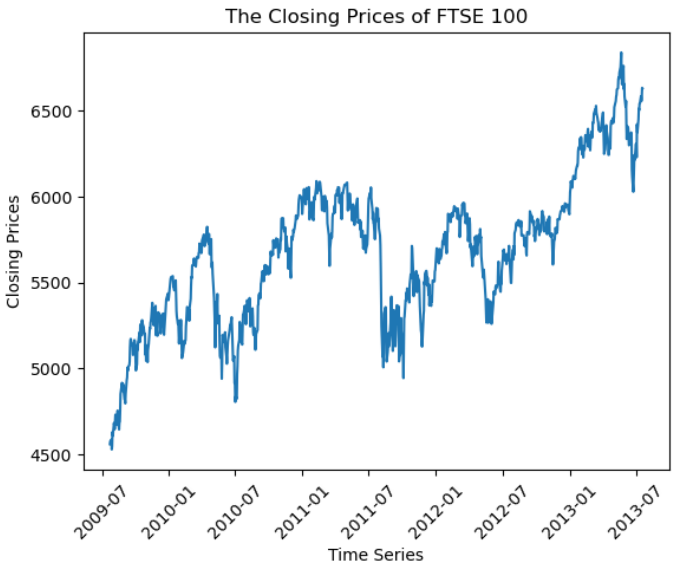


Figure 6: The Closing Prices of FTSE 100

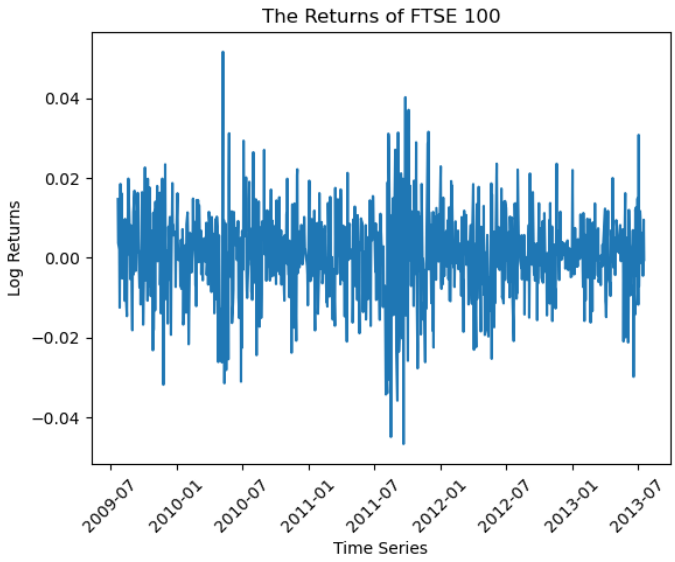


Figure 7: The Returns of FTSE 100

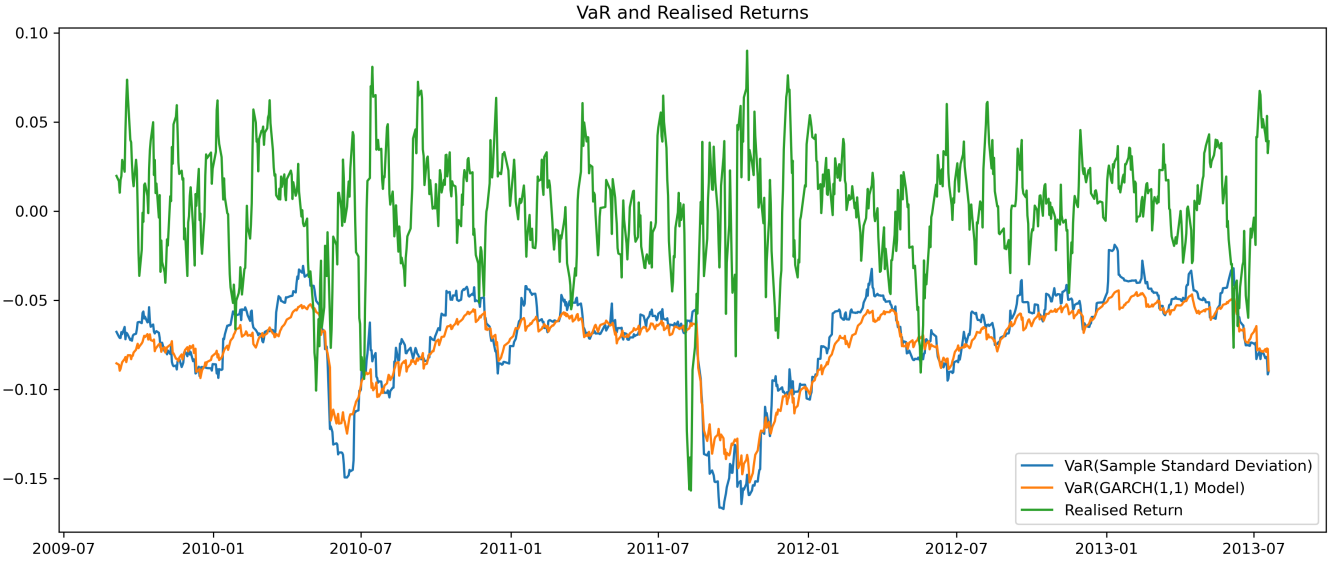


Figure 8: VaR with different methods and Realised Returns

Python codes for this part are as follows:

```
realised_return = []

for i in range(len(data)-10):
    i_1 = i + 10
    returns = np.log(data['Closing Price'][i_1]/data['Closing Price'][i])
    realised_return.append(returns)

pct1=np.round(sum(realised_return[-978:]<VaR_1[:-10])/len(realised_return[-978:]),4)
pct2=np.round(sum(realised_return[-978:]<VaR_2[:-10])/len(realised_return[-978:]),4)
```

4 Appendix (Python Code)

Table of Contents

1 A. Products, Strategies and Pricing

In [161]:

```
# Import math functions from NumPy
import numpy as np
import pandas as pd
from numpy import *
from numpy.linalg import multi_dot

# Import plotting functions from helper
from helper import plot_asset_path, plot_probability, plot_binomial_tree

# Import matplotlib for visualization
import matplotlib
import matplotlib.pyplot as plt

from scipy.stats import norm
from scipy.optimize import minimize
from tabulate import tabulate

# # Plot settings
# plt.style.use('dark_background')
# matplotlib.rcParams['figure.figsize'] = [24.0, 8.0]
# matplotlib.rcParams['font.size'] = 10
# matplotlib.rcParams['lines.linewidth'] = 2.0
# matplotlib.rcParams['grid.color'] = 'black'

from helper import plot_var

import cufflinks as cf
cf.set_config_file(offline=True, dimensions=((1000,600)))

import matplotlib.dates as mdate

# Import plotly express for EF plot
import plotly.express as px
px.defaults.template, px.defaults.width, px.defaults.height = "plotly_white", 1000, 600

import warnings
warnings.filterwarnings('ignore')
```

executed in 12ms, finished 20:52:09 2022-08-21

$$u = 1 + \sigma\sqrt{\delta t}$$

$$v = 1 - \sigma\sqrt{\delta t}$$

$$p' = \frac{1}{2} + \frac{r\sqrt{\delta t}}{2\sigma}$$

$$V = \frac{1}{1 + r\delta t}(p'V^{+} + (1 - p')V^{-})$$

In [2]:

```
def binomial_option(spot, strike, rate, sigma, time, steps, output=0):

    """
    binomial_option(spot, strike, rate, sigma, time, steps, output=0)

    Function for building binomial option tree for european call option payoff.

    Parameters
    ----------
    spot        int or float   - spot price
    strike      int or float   - strike price
    rate        float          - interest rate
    sigma       float          - volatility
    time        int or float   - expiration time
    steps       int            - number of time steps
    output      int            - [0: price, 1: payoff, 2: option value, 3: option delta]

    Returns
    ----------
    out : ndarray
    An array object of price, payoff, option value and delta as specified by the output parameter

    """

    # define parameters
    ts = time/steps                                 # ts is time steps, dt
    u = 1 + sigma*sqrt(ts)                          # u is up factor
    v = 1 - sigma*sqrt(ts)                          # v is down factor
    p = 0.5 + rate*sqrt(ts)/(2*sigma)               # p here is risk neutral probability (p') - for ease of use
    df = 1/(1+rate*ts)                              # df is discount factor

    # initialize arrays
    px = zeros((steps+1, steps+1))                  # price path
    cp = zeros((steps+1, steps+1))                  # call intrinsic payoff
    V = zeros((steps+1, steps+1))                   # option value
    d = zeros((steps+1, steps+1))                   # delta value

    # binomial loop
    for j in range(steps+1):
        for i in range(j+1):
            px[i,j] = spot * power(v,i) * power(u,j-i)
            cp[i,j] = maximum(px[i,j] - strike,0)

    for j in range(steps+1, 0, -1):
        for i in range(j):
            if (j == steps+1):
                V[i,j-1] = cp[i,j-1]                # terminal payoff
                d[i,j-1] = 0                        # terminal delta
            else:
                V[i,j-1] = df*(p*V[i,j]+(1-p)*V[i+1,j])
                d[i,j-1] = (V[i,j]-V[i+1,j])/(px[i,j]-px[i+1,j])

    results = around(px,2), around(cp,2), around(V,2), around(d,4)

    return results[output]
```

executed in 8ms, finished 03:16:25 2022-08-21

In [3]:

```
spot, strike, rate, sigma, time, steps = 100, 100, 0.05, 0.2, 1, 4
```

executed in 2ms, finished 03:16:26 2022-08-21

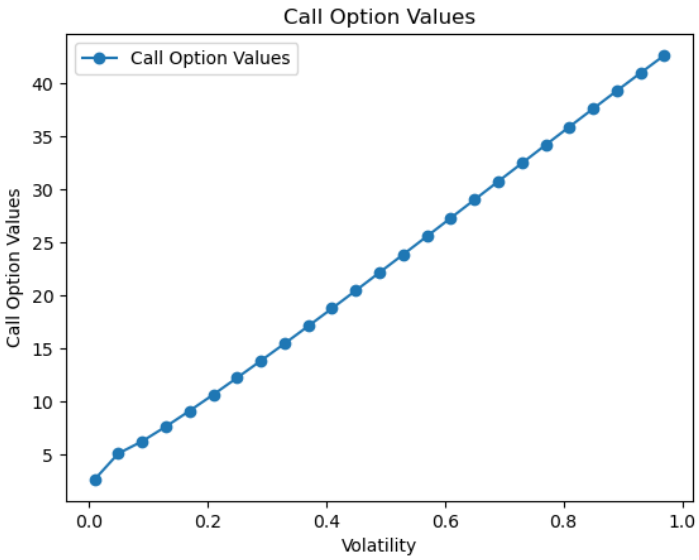
In [4]:

```
vol_vec = np.arange(0.01,1,0.04)
V_vol = []
for i in range(len(vol_vec)):
    Value = binomial_option(spot, strike, rate, vol_vec[i], time, steps, output=2)[0,0]
    V_vol.append(Value)
```

executed in 8ms, finished 03:16:26 2022-08-21

In [5]:
```
plt.plot(vol_vec, V_vol, 'o-', label='Call Option Values')
plt.title('Call Option Values')
plt.xlabel('Volatility')
plt.ylabel('Call Option Values')
plt.legend()
plt.show()
```

executed in 189ms, finished 03:16:27 2022-08-21

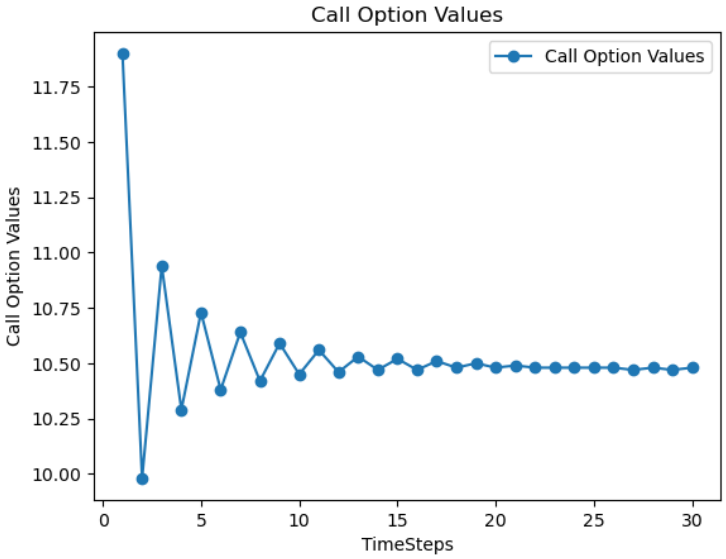


In [6]:
```
step_vec = np.arange(1,31)
V_tstep = []
for i in range(len(step_vec)):
    Value = binomial_option(spot, strike, rate, sigma, time, step_vec[i], output=2)[0,0]
    V_tstep.append(Value)
```

executed in 59ms, finished 03:16:27 2022-08-21

In [7]:
```
plt.plot(step_vec, V_tstep, 'o-', label='Call Option Values')
plt.title('Call Option Values')
plt.xlabel('TimeSteps')
plt.ylabel('Call Option Values')
plt.legend()
plt.show()
```

executed in 168ms, finished 03:16:28 2022-08-21

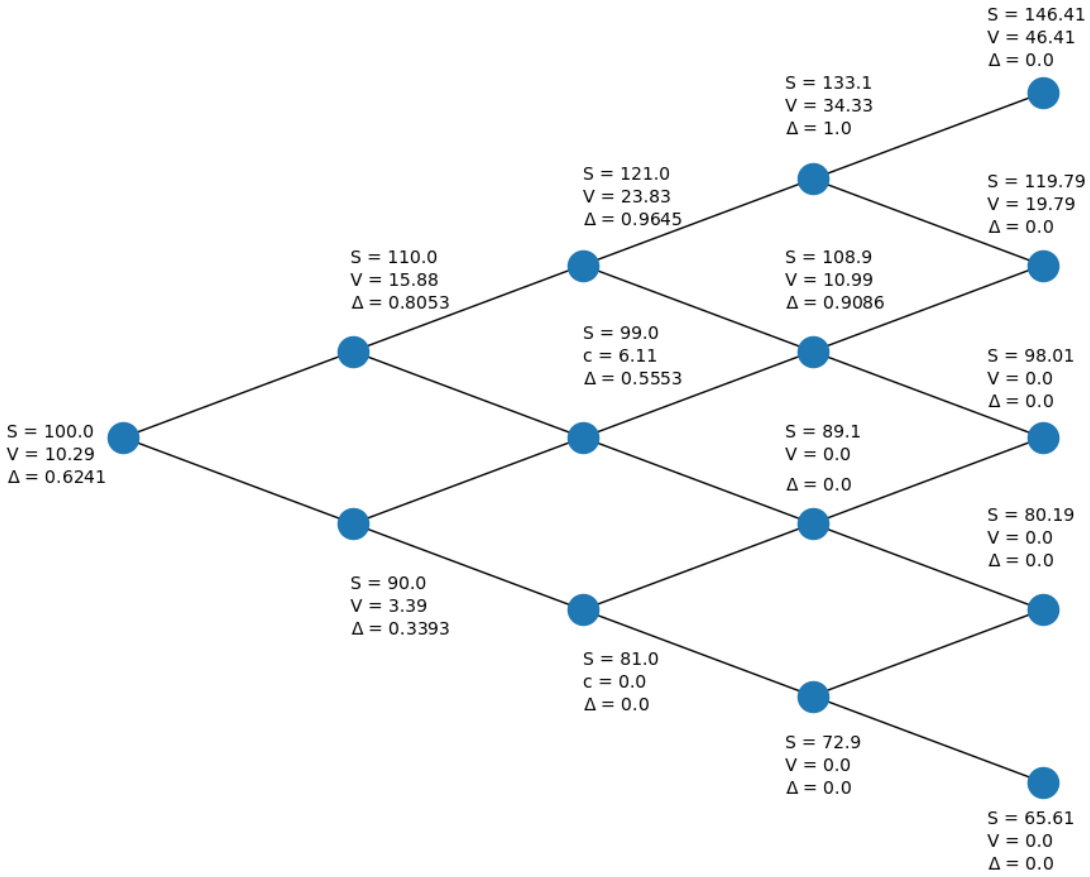


In [8]:
```
px = binomial_option(spot, strike, rate, sigma, time, steps, output=0)
cp = binomial_option(spot, strike, rate, sigma, time, steps, output=1)
opx = binomial_option(spot, strike, rate, sigma, time, steps, output=2)
delta = binomial_option(spot, strike, rate, sigma, time, steps, output=3)
```

executed in 2ms, finished 03:16:28 2022-08-21

In [12]:
```
# Plot a 4-Step Binomial Tree
plot_binomial_tree(px[0,0], px, opx, delta)
plt.show()
```

executed in 486ms, finished 03:16:30 2022-08-21



2 B. Portfolio Optimisation

2.1 Covariance Matrix

In [13]:
```
mu_vec = np.array([0.04, 0.08, 0.12, 0.15]).reshape(4,1)
sigma_vec = np.array([0.07, 0.12, 0.18, 0.26]).reshape(4,1)
corr_mat = np.array([[1, 0.2, 0.5, 0.3],
                     [0.2, 1, 0.7, 0.4],
                     [0.5, 0.7, 1, 0.9],
                     [0.3, 0.4, 0.9, 1]])
m = 0.1
```

executed in 4ms, finished 03:16:31 2022-08-21

Covariance Matrix:

$$\Sigma = \sigma^T \rho \sigma$$

In [14]:
```
cov_mat = sigma_vec * corr_mat * sigma_vec.reshape(1,4)
cov_mat
```

executed in 5ms, finished 03:16:32 2022-08-21

Out[14]:
```
array([[0.0049 , 0.00168, 0.0063 , 0.00546],
       [0.00168, 0.0144 , 0.01512, 0.01248],
       [0.0063 , 0.01512, 0.0324 , 0.04212],
       [0.00546, 0.01248, 0.04212, 0.0676 ]])
```

2.2 Optimisation - Tangency Portfolio

2.2.1 Solve optimization by Lagrangian method.

The Objective:

$$\underset{\mathrm{w}}{Min}\ \frac{1}{2}\mathrm{w}^T\Sigma w$$

Subject to constraints:

$$\mathrm{w}^T\mathbf{1} = 1$$
$$\mathrm{w}^T\mu = 0.1$$

$$L = \frac{1}{2}w'\Sigma w + \lambda(m - w'\mu) + \gamma(1 - w'\mathbf{1})$$

$$\frac{\partial L}{\partial w} = \Sigma w + \lambda\mu + \gamma\mathbf{1}$$

Set: $\Sigma\hat{w} + \lambda\mu + \gamma\mathbf{1} = 0$

$$\Rightarrow \hat{w} = \Sigma^{-1}(\lambda\mu + \gamma\mathbf{1})$$

$$\frac{\partial^2 L}{\partial w^2} = \frac{\partial}{\partial w}(\Sigma w - \lambda\mu - \gamma\mathbf{1}) = \Sigma > 0$$

Plug w to the constaints:

$$\mu'\Sigma^{-1}(\lambda\mu + \gamma\mathbf{1}) = \lambda\mu'\Sigma^{-1}\mu + \gamma\mu'\Sigma^{-1}\mathbf{1} = m$$

$$\mathbf{1}'\Sigma^{-1}(\lambda\mu + \gamma\mathbf{1}) = \lambda\mathbf{1}\Sigma^{-1}\mu + \gamma\mathbf{1}'\Sigma^{-1}\mathbf{1} = 1$$

We define the following scalars:

$$\begin{cases} A = \mathbf{1}'\Sigma^{-1}\mathbf{1} \\ B = \mu'\Sigma^{-1}\mathbf{1} = \mathbf{1}'\Sigma^{-1}\mu \\ C = \mu'\Sigma^{-1}\mu \end{cases}$$

$$AC - B^2 > 0$$

$$\begin{cases} C\lambda + B\gamma = m \\ B\lambda + A\gamma = 1 \end{cases}$$

$$\Rightarrow \begin{cases} \lambda = \dfrac{Am - B}{AC - B^2} \\ \gamma = \dfrac{C - Bm}{AC - B^2} \end{cases}$$

$$\hat{w} = \frac{1}{AC - B^2}\Sigma^{-1}[(A\mu - B\mathbf{1})m + (C\mathbf{1} - B\mu)]$$

2.2.2 Standard deviation of optimal portfolio.

In [15]:
```
one_vec = np.array([1, 1, 1, 1]).reshape(4,1)
```
executed in 2ms, finished 03:16:34 2022-08-21

In [16]:
```
A = one_vec.reshape(1,4)@np.linalg.inv(cov_mat)@one_vec
B = one_vec.reshape(1,4)@np.linalg.inv(cov_mat)@mu_vec
C = mu_vec.reshape(1,4)@np.linalg.inv(cov_mat)@mu_vec
```
executed in 5ms, finished 03:16:35 2022-08-21

In [17]:
```
Lambda = (A*m-B)/(A*C-B**2)
Gamma = (C-B*m)/(A*C-B**2)
```
executed in 3ms, finished 03:16:35 2022-08-21

In [257]:
```
w_hat = np.round(np.linalg.inv(cov_mat)@(Lambda*mu_vec+Gamma*one_vec),4)
print('====================================')
print('The Weights of Optimal Portfolio:')
print(w_hat)
print('====================================')
```
executed in 7ms, finished 10:48:32 2022-08-22

```
====================================
The Weights of Optimal Portfolio:
[[ 0.0587]
 [ 0.759 ]
 [-0.3195]
 [ 0.5018]]
====================================
```

$$\sigma_\pi^2(m) = \frac{Am^2 - 2Bm + C}{AC - B^2}$$

$$\sigma_\pi^2 = w'\Sigma w$$

In [263]:
```
sigma_m = np.round(np.sqrt((A*m**2-2*B*m+C)/(A*C-B**2)),4)
print('================================================')
print('The Volatility of Optimal Portfolio:', sigma_m)
print('================================================')
```
executed in 4ms, finished 10:50:07 2022-08-22

```
================================================
The Volatility of Optimal Portfolio: [[0.1325]]
================================================
```

In [262]:
```
sigma_mv = np.round(np.sqrt(w_hat.reshape(1,4)@cov_mat@w_hat),4)
print('================================================')
print('The Volatility of Optimal Portfolio:', sigma_mv)
print('================================================')
```

executed in 5ms, finished 10:50:01 2022-08-22

```
================================================
The Volatility of Optimal Portfolio: [[0.1325]]
================================================
```

In [267]:
```
mu_m = np.round(w_hat.reshape(1,4)@mu_vec,4)
print('======================================')
print('The Return of Optimal Portfolio:', mu_m)
print('======================================')
```

executed in 4ms, finished 10:52:09 2022-08-22

```
======================================
The Return of Optimal Portfolio: [[0.1]]
======================================
```

In [269]:
```
Sharpe_Ratio = np.round(mu_m/sigma_m,4)
print('================================================')
print('The Sharpe Ratio of Optimal Portfolio:', Sharpe_Ratio)
print('================================================')
```

executed in 4ms, finished 10:53:02 2022-08-22

```
================================================
The Sharpe Ratio of Optimal Portfolio: [[0.7547]]
================================================
```

2.2.3 On a graph identifying this optimal portfolio.

In [23]:
```
def portfolio_simulation(mu_vec,cov_mat,numofportfolio):

    # Initialize the lists
    rets = []; vols = []; wts = []

    # Simulate 5,000 portfolios
    for i in range(numofportfolio):

        # Generate random weights
        weights = (2*(random.random(len(mu_vec)))-1)[:, newaxis]

        # Set weights such that sum of weights equals 1
        weights /= np.sum(weights)

        rets.append((weights.T @ mu_vec)[:, newaxis])
        vols.append(sqrt(multi_dot([weights.T, cov_mat, weights])))
        wts.append(weights.flatten())

    # Create a dataframe for analysis
    portdf = pd.DataFrame({
        'port_rets': array(rets).flatten(),
        'port_vols': array(vols).flatten(),
        'weights': list(array(wts))
    })

    portdf['sharpe_ratio'] = portdf['port_rets'] / portdf['port_vols']

    return round(portdf,2)
```

executed in 7ms, finished 03:16:43 2022-08-21

In [24]:
```
# Create a dataframe for analysis
numofportfolio = 5000
numofasset = len(mu_vec)
temp = portfolio_simulation(mu_vec,cov_mat,numofportfolio)
temp
```

executed in 212ms, finished 03:16:44 2022-08-21

Out[24]:

	port_rets	port_vols	weights	sharpe_ratio
0	0.41	1.03	[-0.5978535980496537, -3.1958738608865986, 0.9...	0.40
1	0.11	0.16	[0.15096477059271765, 0.17734367177554344, 0.3...	0.69
2	0.11	0.15	[0.12428963960293983, 0.37792423984527285, 0.0...	0.71
3	0.17	0.29	[-1.259816940916555, 1.1311150899064435, 1.169...	0.61
4	0.07	0.10	[0.3442371420483194, 0.4612092424008677, 0.291...	0.75
...	...	...	...	...
4995	0.19	0.73	[2.117405575158758, -3.2231796840271603, -1.41...	0.25
4996	0.07	0.14	[0.8591620734786002, -0.6244988011673591, 0.80...	0.54
4997	0.01	0.14	[0.8351521663855488, 0.4907262490675793, 0.386...	0.09
4998	0.10	0.15	[0.5010079655063576, -0.15093768735108729, 0.3...	0.65
4999	0.09	0.11	[0.2527629867584861, 0.34189813802528285, 0.43...	0.75

5000 rows × 4 columns

In [25]:
```
# Verify the above result
temp.describe().T
```
executed in 23ms, finished 03:16:52 2022-08-21

Out[25]:

	count	mean	std	min	25%	50%	75%	max
port_rets	5000.0	0.124500	1.580056	-28.45	0.05	0.10	0.14	86.79
port_vols	5000.0	0.680162	4.344617	0.03	0.13	0.20	0.36	220.97
sharpe_ratio	5000.0	0.457404	0.326832	-0.49	0.34	0.57	0.68	1.26

In [26]:
```
# Import optimization module from scipy
import scipy.optimize as sco
```
executed in 3ms, finished 03:16:54 2022-08-21

In [27]:
```
# Define portfolio stats function
def portfolio_stats(weights):

    weights = array(weights)[:,newaxis]
    port_rets = (weights.T @ mu_vec)[:, newaxis]
    port_vols = sqrt(multi_dot([weights.T, cov_mat, weights]))

    return array([port_rets, port_vols, port_rets/port_vols]).flatten()
```
executed in 4ms, finished 03:16:54 2022-08-21

In [28]:
```
def min_volatility(weights):
    return portfolio_stats(weights)[1]
```
executed in 10ms, finished 03:16:56 2022-08-21

In [29]:
```
# Define initial weights
initial_wts = numofasset * [1./numofasset]
initial_wts
```
executed in 4ms, finished 03:16:57 2022-08-21

Out[29]:
```
[0.25, 0.25, 0.25, 0.25]
```

In [30]:
```
# Each asset boundary ranges from 0 to 1 bounds
bnds = tuple((-1,1) for x in range(numofasset))
bnds
```
executed in 5ms, finished 03:16:58 2022-08-21

Out[30]:
```
((-1, 1), (-1, 1), (-1, 1), (-1, 1))
```

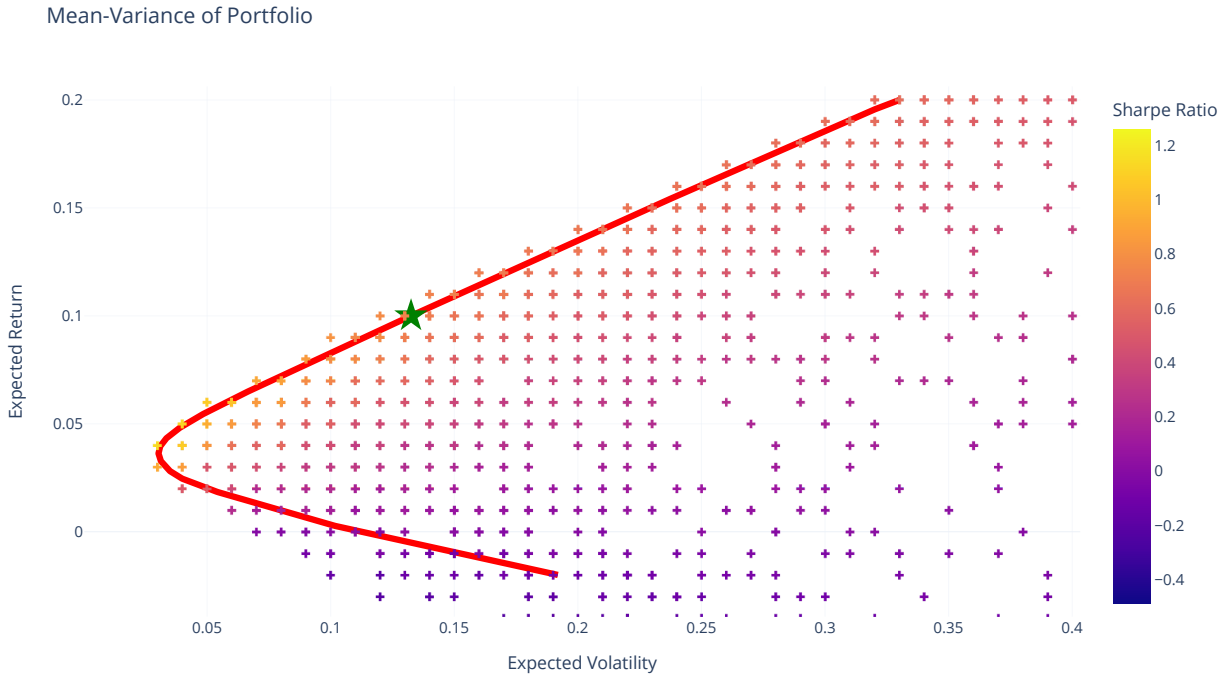
In [31]:
```
# Efficient frontier params
targetrets = linspace(-0.02,0.20,1000)
tvols = []

for tr in targetrets:

    ef_cons = ({'type': 'eq', 'fun': lambda x: portfolio_stats(x)[0] - tr},
               {'type': 'eq', 'fun': lambda x: sum(x) - 1})

    opt_ef = sco.minimize(min_volatility, initial_wts, method='SLSQP', bounds=bnds, constraints=ef_cons)

    tvols.append(opt_ef['fun'])

targetvols = array(tvols)
```
executed in 7.07s, finished 03:17:05 2022-08-21

In [32]:
```
# Dataframe for EF
efport = pd.DataFrame({
    'targetrets' : around(targetrets[14:],2),
    'targetvols': around(targetvols[14:],2),
    'targetsharpe': around(targetrets[14:]/targetvols[14:],2)
})
```
executed in 4ms, finished 03:17:05 2022-08-21

In [35]:

```
import plotly.graph_objects as go

fig = go.Figure()

# Plot simulated portfolio
fig = px.scatter(
    temp, x='port_vols', y='port_rets', color='sharpe_ratio',
    labels={'port_vols': 'Expected Volatility', 'port_rets': 'Expected Return','sharpe_ratio': 'Sharpe Ratio'},
    title="Mean-Variance of Portfolio"
     ).update_traces(mode='markers', marker=dict(symbol='cross'))

# Plot Efficient Frontier
fig.add_trace(
    go.Line(
        x=targetvols,
        y=targetrets,
        name = 'Efficient Frontier'
    )).update_traces(line_color='Red', line_width=5)

# Plot Optimal Portfolio
fig.add_scatter(
    mode='markers',
    x=[0.1325],
    y=[0.10],
    marker=dict(color='Green', size=20, symbol='star'),
    name = 'Optimal Portfolio').update(layout_showlegend=False)

fig.show()
```

executed in 663ms, finished 03:17:51 2022-08-21



2.3 Optimisation - Global Minimum Variance Portfolio

2.3.1 Solve optimization by Lagrangian method.

The Objective:

$$\underset{\mathrm{w}}{Min}\ \frac{1}{2}\mathrm{w}^T\Sigma w$$

Subject to constraints:

$$\mathrm{w}^T\mathbf{1} = 1$$

$L = \frac{1}{2}w'\Sigma w + \gamma(1 - w\mathbf{1})$

$\frac{\partial L}{\partial w} = \Sigma w + \gamma\mathbf{1}$

Set: $\Sigma\hat{w} + \gamma\mathbf{1} = 0$

$\Rightarrow \hat{w} = \Sigma^{-1}(-\gamma\mathbf{1})$

Plug w to the constaints:

$\mathbf{1}'\Sigma^{-1}(-\gamma\mathbf{1}) = 1$

$$\gamma = -\frac{1}{\mathbf{1}'\Sigma^{-1}\mathbf{1}}$$

$$\hat{w} = \frac{1}{AC - B^2}\Sigma^{-1}[(A\mu - B\mathbf{1})m + (C\mathbf{1} - B\mu)]$$

In [281]:
```
Gamma_g = np.round((-1)/(one_vec.reshape(1,4)@np.linalg.inv(cov_mat)@one_vec),4)
print('====================================================')
print('The Gamma of Minimum Variance Portfolio:', Gamma_g)
print('====================================================')
```

executed in 5ms, finished 10:58:56 2022-08-22

```
====================================================
The Gamma of Minimum Variance Portfolio: [[-0.0007]]
====================================================
```

In [276]:
```
wg_hat = np.round(np.linalg.inv(cov_mat)@(-Gamma_g*one_vec),4)
print('========================================')
print('The Return of Minimum Variance Portfolio:')
print(wg_hat)
print('========================================')
```

executed in 4ms, finished 10:57:08 2022-08-22

```
========================================
The Return of Minimum Variance Portfolio:
[[ 0.9054]
 [ 0.8291]
 [-1.3746]
 [ 0.6401]]
========================================
```

2.3.2 The return and standard deviation of optimal portfolio.

In [282]:
```
mu_g = np.round(wg_hat.T@mu_vec,4)
print('====================================================')
print('The Return of Minimum Variance Portfolio:', mu_g)
print('====================================================')
```

executed in 5ms, finished 10:59:37 2022-08-22

```
====================================================
The Return of Minimum Variance Portfolio: [[0.0336]]
====================================================
```

In [283]:
```
sigma_g = np.round(np.sqrt(wg_hat.reshape(1,4)@cov_mat@wg_hat),4)
print('========================================================')
print('The Volatility of Minimum Variance Portfolio:', sigma_g)
print('========================================================')
```

executed in 5ms, finished 11:00:08 2022-08-22

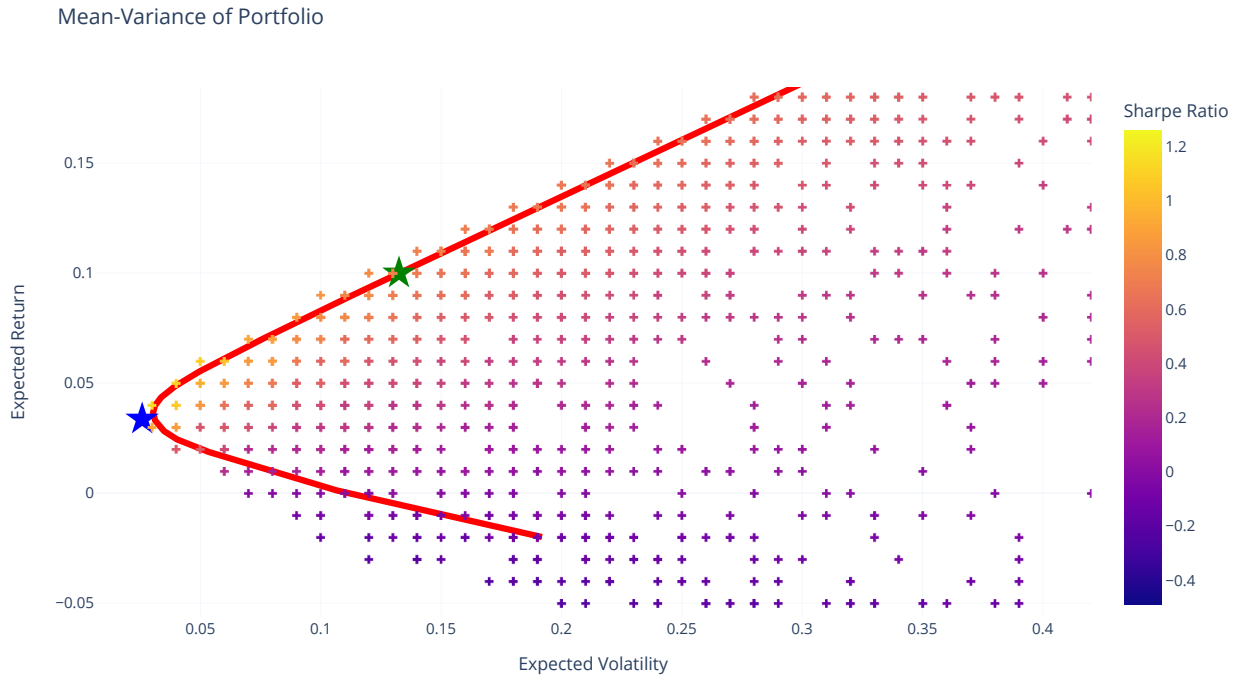
```
========================================================
The Volatility of Minimum Variance Portfolio: [[0.0258]]
========================================================
```

2.3.3 On a graph identifying and name this optimal portfolio.

In [43]:

```
# Plot simulated portfolio
fig = px.scatter(
    temp, x='port_vols', y='port_rets', color='sharpe_ratio',
    labels={'port_vols': 'Expected Volatility', 'port_rets': 'Expected Return','sharpe_ratio': 'Sharpe Ratio'},
    title="Mean-Variance of Portfolio"
     ).update_traces(mode='markers', marker=dict(symbol='cross'))

# Plot Efficient Frontier
fig.add_trace(
    go.Line(
        x=targetvols,
        y=targetrets,
        name = 'Efficient Frontier'
    )).update_traces(line_color='Red', line_width=5)

# Plot Optimal Portfolio
fig.add_scatter(
    mode='markers',
    x=[0.1325],
    y=[0.10],
    marker=dict(color='Green', size=20, symbol='star'),
    name = 'Optimal Portfolio').update(layout_showlegend=False)

# Plot Optimal Portfolio
fig.add_scatter(
    mode='markers',
    x=[0.0258],
    y=[0.0336],
    marker=dict(color='Blue', size=20, symbol='star'),
    name = 'Minimum Variance Portfolio').update(layout_showlegend=False)

fig.show()
```

executed in 114ms, finished 03:24:00 2022-08-21



3 C. Empirical Value at Risk

3.1 Calculate the 99%/10day VaR using a sample standard deviation.

In [542]:

```
import math
from scipy.stats import norm

# Load locally stored data
data = pd.read_excel('/Users/antoneyoung/Jupyter Notebook/CQF Python Lab/FTSE100.xlsx', index_col=0, parse_dates=True)

# Check values
data.head()
```

executed in 136ms, finished 09:20:02 2022-08-23

Out[542]:

Date	Closing Price
2009-07-22	4493.73
2009-07-23	4559.80
2009-07-24	4576.61
2009-07-27	4586.13
2009-07-28	4528.84

In [546]:

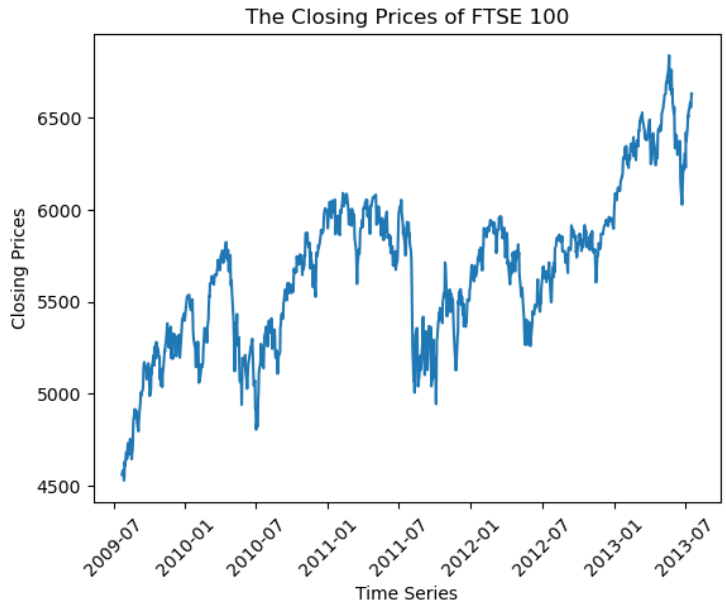
```
data['Returns'] = data.pct_change()
df = data.copy()
df = df.dropna()
```

executed in 8ms, finished 09:21:04 2022-08-23

In [247]:

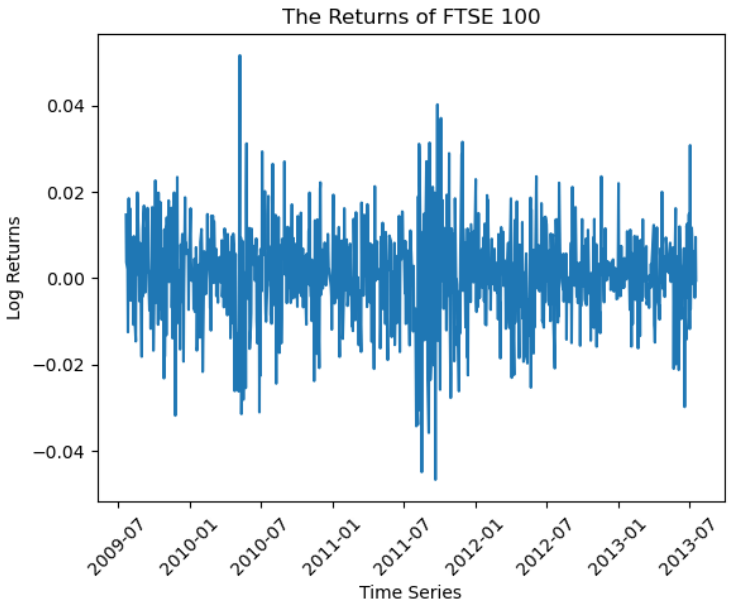
```
plt.plot(df.index, df['Closing Price'])
plt.xlabel('Time Series')
plt.ylabel('Closing Prices')
plt.title('The Closing Prices of FTSE 100')
plt.xticks(rotation=45)
plt.show()
```

executed in 202ms, finished 22:14:51 2022-08-21



In [246]:
```
plt.plot(df['Returns'])
plt.xlabel('Time Series')
plt.ylabel('Log Returns')
plt.title('The Returns of FTSE 100')
plt.xticks(rotation=45)
plt.show()
```

executed in 192ms, finished 22:14:23 2022-08-21



3.1.1 The rolling 21-day sample standard deviation.

In [451]:
```
sigma_21D = pd.Series(df['Returns']).rolling(window=21, center=False).std().dropna()
```

executed in 5ms, finished 21:32:48 2022-08-22

3.1.2 10-day volatility

In [443]:
```
sigma_10D = np.sqrt(10*sigma_21D**2)
```

executed in 2ms, finished 21:30:57 2022-08-22

3.1.3 99%/10-day VaR

In [454]:
```
mu_10D = np.mean(df['Returns'])*10
print('====================================================')
print('The Expected Return (10-day):', mu_10D)
print('====================================================')
```

executed in 4ms, finished 21:35:39 2022-08-22

```
====================================================
The Expected Return (10-day): 0.0044473291971980815
====================================================
```

In [455]:
```
VaR_1 = norm.ppf(1-0.99, mu_10D, sigma_10D)
```

executed in 3ms, finished 21:35:46 2022-08-22

3.2 Calculate the 99%/10day VaR using a GARCH.

$$\sigma^2 = 0.000001 + 0.047u_{t-1}^2 + 0.9466\sigma_{t-1}^2$$

$$h_t = \frac{\omega}{1-\beta} + \alpha(r_{t-1}-\mu)^2 + \alpha\beta(r_{t-2}-\mu)^2 + \alpha\beta^2(r_{t-3}-\mu)^2 + \cdots$$

In [457]:
```
omega, alpha, beta = 0.000001, 0.047, 0.9466
```

executed in 2ms, finished 21:37:04 2022-08-22

In [458]:
```
# GARCH(1,1) function
def garch(omega, alpha, beta, ret):

    var = []
    for i in range(len(ret)):
        if i==0:
            var.append(omega/np.abs(1-alpha-beta))
        else:
            var.append(omega + alpha * ret[i-1]**2 + beta * var[i-1])

    return np.array(var)
```

executed in 3ms, finished 21:37:08 2022-08-22

In [459]:
```
# Verifty variance values
var = garch(omega,alpha,beta,df['Returns'][-989:])
sigma_10d = np.sqrt(var[-988:]*10)
VaR_2 = norm.ppf(1-0.99, mu_10D, sigma_10d)
```

executed in 7ms, finished 21:37:42 2022-08-22

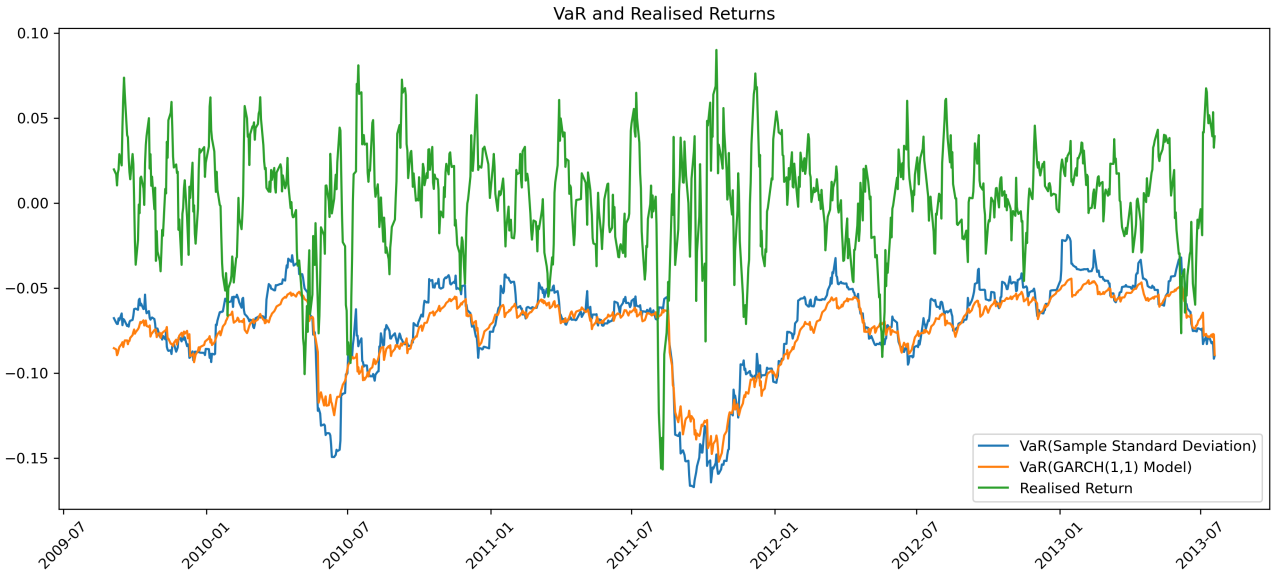
3.3 The percentage of VaR breaches for both measures

In [572]:
```
realised_return = []

for i in range(len(data)-10):
    i_1 = i + 10
    returns = np.log(data['Closing Price'][i_1]/data['Closing Price'][i])
    realised_return.append(returns)
```

executed in 14ms, finished 09:58:27 2022-08-23

In [580]:
```
plt.figure(dpi=300, figsize=(15,6))
plt.plot(df[-978:].index, VaR_1[-988:-10], label='VaR(Sample Standard Deviation)')
plt.plot(df[-978:].index, VaR_2[-988:-10], label='VaR(GARCH(1,1) Model)')
plt.plot(df[-978:].index, realised_return[-978:], label='Realised Return')
plt.xticks(rotation=45)
plt.title('VaR and Realised Returns')
plt.legend()
plt.show()
```

executed in 577ms, finished 10:04:03 2022-08-23



In [576]:
```
percentage1 = np.round(sum(realised_return[-978:]<VaR_1[:-10])/len(realised_return[-978:]),4)
print('==================================')
print('The percentage of breaches:', percentage1)
print('==================================')
```

executed in 4ms, finished 09:58:50 2022-08-23

```
==================================
The percentage of breaches: 0.0276
==================================
```

In [577]:
```
percentage2 = np.round(sum(realised_return[-978:]<VaR_2[:-10])/len(realised_return[-978:]),4)
print('==================================')
print('The percentage of breaches:', percentage2)
print('==================================')
```

executed in 4ms, finished 09:58:55 2022-08-23

```
==================================
The percentage of breaches: 0.0225
==================================
```

END