Portfolio Optimisation using Quantum Annealing

Amey Kulkarni Devvrat Joshi

Decimal Point Analytics (Pvt. Ltd)

July 11, 2021

The optimisations described in this report have been implemented on a quantum computer using the resources provided by DWave Systems.

1 Theory

The code written is based on the following mathematical formulation. Some of the sample problems can be tried out from their Youtube channel to understand how it works. One such video is- .

1.1 Binary Portfolio Optimisation

This was the first attempt at using D'Wave resources to solve the problem of Portfolio Optimisation. This technique is fairly commonly used, as we have ascertained from many different papers on the same. We describe the Maths involved in it in the following subsections.

Problem Description

Given N stocks, their covariance matrix and average returns over a time period (computable using classical computers too since it can be done beforehand, or offline), an (optional) expected average return amount, (optional) desired number of stocks, to come up with a portfolio that satisfies these conditions to different degrees and also minimises the investment risk.

Boolean Variables

Define a boolean variable x_i for the i^{th} stock. $x_i = 1$ means that we include this stock in our portfolio, $x_i = 0$ means that we do not include this stock in our portfolio.

Objective Function

The objective function we want to minimise is the investment risk R incurred. It is defined as follows-

$$R = \frac{1}{2}\sum_{i=1}^{n} \sigma_{ii} x_i^2 + \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sigma_{ij} x_i x_j$$

where σ_{ij} is the covariance between the i^{th} and the j^{th} stocks. (σ_{ii} is the variance of the i^{th} stock.)

Fixed number of Stocks Constraint

We want to allow exactly f stocks, hence the following constraint-

$$(\sum_{i=1}^{n} x_i - f)^2 = 0$$

This gives a quadratic form equation. To satisfy this, we need exactly f many x_i set to 1.

Desired Return Constraint

We want to ensure that the Returns we get are as close to the Desired Returns as possible, therefore-

$$(\sum_{i=1}^{n} r_i x_i - \mu_p)^2 = 0$$

where r_i denotes the average returns of the i^{th} stock and μ_p is the Desired Returns.

About this Technique

Although a good first step, this technique has its own limitations, as it can only decide whether a given stock should be chosen or not, but not with what weight we should buy it. Using this technique, we will ending buying all favourable stocks in equal proportion, which may be far from optimal at times.

The next technique aims to sort out this problem.

1.2 Increased Precision Portfolio Optimisation

We aim to improve the precision of the above implementation. For doing so, we will use k qubits to represent each stock's weight in our portfolio. The p^{th} of these qubits will have power $\frac{1}{2^p}$. For example, with $k = 3$ if a portfolio has a weight $w = 0.625$ then the qubits representing it will be $101_{\frac{1}{2}} = 1 \times \frac{1}{2^1} + 0 \times \frac{1}{2^2} + 1 \times \frac{1}{2^3}$.

Variables for higher Precision

Suppose we have n stocks, and a precision of k qubits for each stocks, we would need $n \times k$ qubits. Consider the d^{th} qubit of the entire system. Let $i = \lfloor \frac{d-1}{k} + 1 \rfloor$ and $p = (d-1)\%k$ (note that here 1-indexing has been used to explain, in the code it -index based). We denote such a bit by x_{ip}. Hence, the d^{th} qubit, referred to as x_{ip} represents the p^{th} precision bit of the i^{th} stock. If this qubit is set to 1, then we add $\frac{1}{2^p}$ to the weight of the i^{th} stock, otherwise we add zero.

New Objective Function

The risk R we want to minimise is defined as follows-

$$R = \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{p=1}^{k} \sum_{q=1}^{k} \sigma_{ij} x_{ip} x_{jq} \frac{1}{2^{p+q}}$$

We are using the p^{th} qubit of the i^{th} stock and comparing it with the q^{th} qubit of the j^{th} stock to find the contribution to the risk through those specific bits.

Sum of Weights

The next constraint ensures that the sum of weights of all the stocks used adds up to 1.

$$(\sum_{i=1}^{n} \sum_{p=1}^{k} x_{ip} \frac{1}{2^p} - 1)^2 = 0$$

New Desired Return Constraint

Suppose we want the yearly returns to be as close to the desired yearly return μ_a as possible. The constraint that ensures this is as follows-

$$(\sum_{i=1}^{n}\sum_{p=1}^{k} r_i x_{ip}\frac{1}{2^p} - \mu_a)^2 = 0$$

where r_i denotes the average yearly returns (based on previous data) of the i^{th} stock.

About this Technique

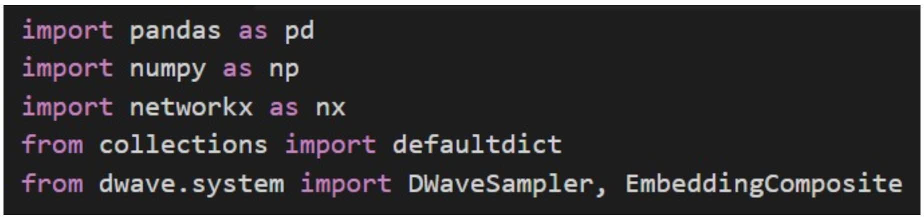
This technique is a sort of novelty, since we haven't come across it in any other papers. It can be very useful as it tells us not only which stocks we should buy to maximise our chance of profit while keeping the risk minimum, but it also tells what proportions to buy them at.

A property of the solution obtained will be that the weights will always be positive. However this does not mean that the portfolio does not give us information about whether we should sell stocks or not. This is because we can use the program to rebalance our Portfolio. This will tell us the new number of stocks we need to keep in possession optimally. If this number is higher than the previous month's, it means we need to buy the difference. If it is lower, we need to sell the difference.

2 Explanation of the Code

2.1 Imports

Figure 1 shows the list of imports required. Note that we implemented and ran our code on an IDE provided by them, the Leap IDE. You can sign up for the same from here.



```python
import pandas as pd
import numpy as np
import networkx as nx
from collections import defaultdict
from dwave.system import DWaveSampler, EmbeddingComposite
```

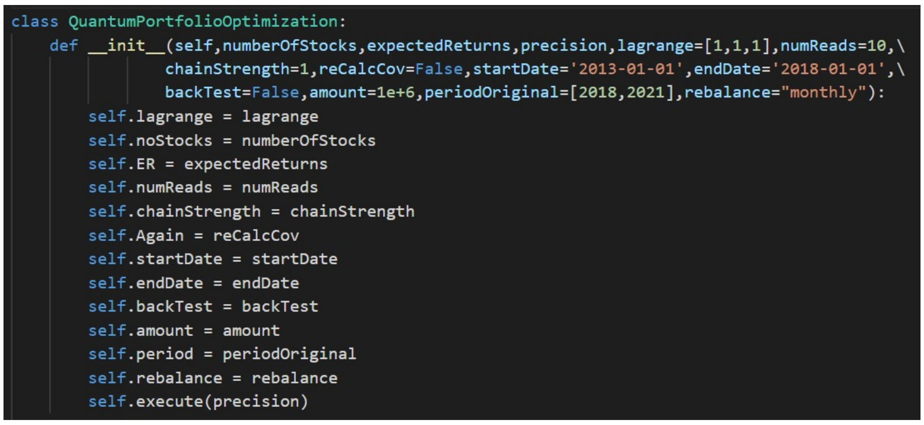
Figure 1: Set of Imports

2.2 Creating an Instance of the Class QuantumPortfolioOptimization

Figure 2 shows this.

- *lagrange* is a list of three numbers specifying the Lagrange Multipliers of the Risk Objective function, Desired Returns Constraint and Weights of Stocks Constraint respectively.

- *noStocks* is the number of stocks that are to be accounted for the portfolio optimisation from the stored data.
- *ER* is the expected/desired returns given as input by user.
- *numReads* is a quantum parameter given to the DWave quantum annealer. It is the number of times the measurement of quantum system done by quantum computer.
- *chainStength* is a quantum parameter given to th e DWave quantum annealer. It is related to the physics of quantum annealing. For more information refer: DWave Documentation Chain Stength
- *Again* is a parameter which gives the option of taking the start and end date for portfolio optimisation. The actual parameter which user gives is *reCalcCov* and if it is a boolean value.
- *startDate* and *endDate* is the interval from which you consider the data before you start the portfolio optimisation.
- *backTest* is True means we are doing backtesting and the parameters after backTest are all related to backtesting. It is a boolean value.
- *amount* is the principal amount when starting backtesting.
- *preiod* is a period of years in which backtesting is to be done. It is given as a list of two integers. Here the first integer is the starting year and the last integer is one less than the ending year.
- *rebalance* can take values in "yearly", "monthly", "daily". It is the period between rebalancing.

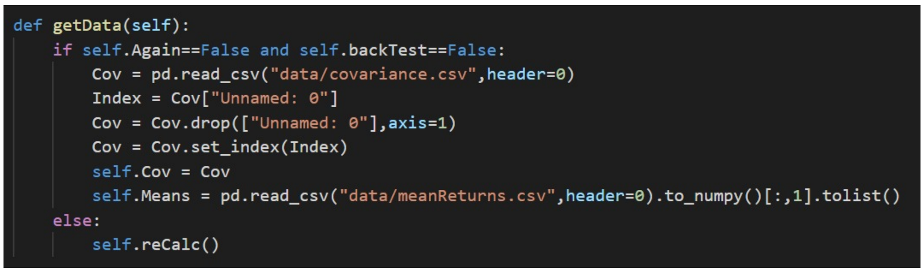


```
class QuantumPortfolioOptimization:
    def __init__(self,numberOfStocks,expectedReturns,precision,lagrange=[1,1,1],numReads=10,\
            chainStrength=1,reCalcCov=False,startDate='2013-01-01',endDate='2018-01-01',\
            backTest=False,amount=1e+6,periodOriginal=[2018,2021],rebalance="monthly"):
        self.lagrange = lagrange
        self.noStocks = numberOfStocks
        self.ER = expectedReturns
        self.numReads = numReads
        self.chainStrength = chainStrength
        self.Again = reCalcCov
        self.startDate = startDate
        self.endDate = endDate
        self.backTest = backTest
        self.amount = amount
        self.period = periodOriginal
        self.rebalance = rebalance
        self.execute(precision)
```

Figure 2: Creating an Instance of the Class

2.3 Reading the Data

Figure 3 shows the function that reads the Covariance matrix. Not that we have already made such a CSV using this Google Colab Notebook

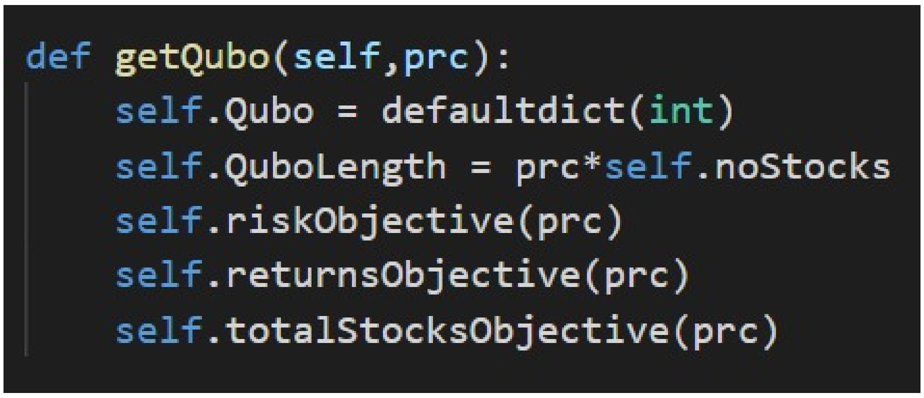


```python
def getData(self):
    if self.Again==False and self.backTest==False:
        Cov = pd.read_csv("data/covariance.csv",header=0)
        Index = Cov["Unnamed: 0"]
        Cov = Cov.drop(["Unnamed: 0"],axis=1)
        Cov = Cov.set_index(Index)
        self.Cov = Cov
        self.Means = pd.read_csv("data/meanReturns.csv",header=0).to_numpy()[:,1].tolist()
    else:
        self.reCalc()
```

Figure 3:

2.4 The Function *getQubo*

Figure 4 is the function that creates the matrix *self.Qubo* that is finally sent to the annealer. Please refer this video to understand how to create such a matrix that is fed to annealer. Creating the matrix is the most important step of the algorithm.



```python
def getQubo(self,prc):
    self.Qubo = defaultdict(int)
    self.QuboLength = prc*self.noStocks
    self.riskObjective(prc)
    self.returnsObjective(prc)
    self.totalStocksObjective(prc)
```

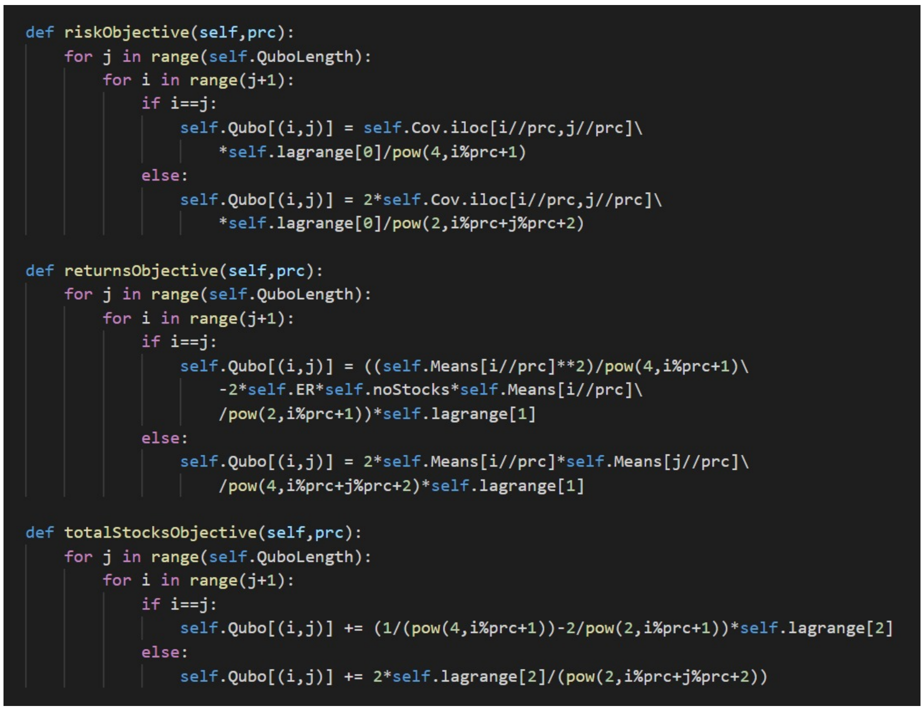
Figure 4: *getQubo*

2.5 Risk, Desired Returns and Weights of Stocks

Figure 5 shows how to manipulate the matrix *Qubo* (as explained in the video above) according to our objective function and constraints.

2.6 Executing on D'Wave Quantum Computer

dWaveExecute, as shown in Figure 6, is a function which actually creates a corresponding embedding to the QUBO created above. Then this embedding is sent to the D'Wave Leap cloud where our QUBO is executed on their quantum annealer. We receive the response from the D'Wave cloud and then we use the response to get the desired results in *results* function.



```python
def riskObjective(self,prc):
    for j in range(self.QuboLength):
        for i in range(j+1):
            if i==j:
                self.Qubo[(i,j)] = self.Cov.iloc[i//prc,j//prc]\
                    *self.lagrange[0]/pow(4,i%prc+1)
            else:
                self.Qubo[(i,j)] = 2*self.Cov.iloc[i//prc,j//prc]\
                    *self.lagrange[0]/pow(2,i%prc+j%prc+2)

def returnsObjective(self,prc):
    for j in range(self.QuboLength):
        for i in range(j+1):
            if i==j:
                self.Qubo[(i,j)] = ((self.Means[i//prc]**2)/pow(4,i%prc+1)\
                    -2*self.ER*self.noStocks*self.Means[i//prc]\
                    /pow(2,i%prc+1))*self.lagrange[1]
            else:
                self.Qubo[(i,j)] = 2*self.Means[i//prc]*self.Means[j//prc]\
                    /pow(4,i%prc+j%prc+2)*self.lagrange[1]

def totalStocksObjective(self,prc):
    for j in range(self.QuboLength):
        for i in range(j+1):
            if i==j:
                self.Qubo[(i,j)] += (1/(pow(4,i%prc+1))-2/pow(2,i%prc+1))*self.lagrange[2]
            else:
                self.Qubo[(i,j)] += 2*self.lagrange[2]/(pow(2,i%prc+j%prc+2))
```

Figure 5: Objective and Constraints

2.7 Recalculating the Covariance Matrix

reCalc takes the historical closing price data of all the stocks and only calculates the covariance matrix and the average returns of stocks from the data between the *startDate* and *endDate*. Figure 7.

2.8 Back Test

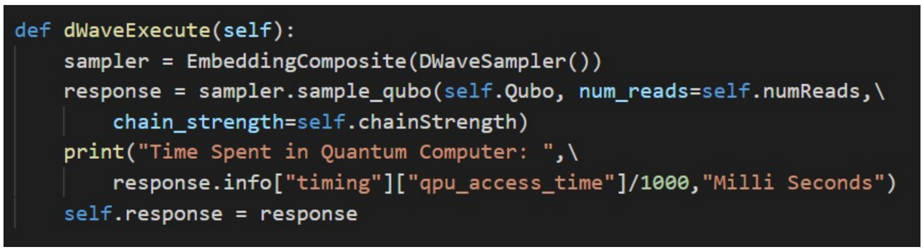
doBackTest does the backtesting based on the the parameters given for back-testing. There are three options: "yearly", "monthly" and "daily". For every rebalancing, the weights, volatility and actual returns are recalculated. The function also compares our portfolio with equal weights portfolio. Figure 8.

2.9 Executing the Functions

The *execute* function is a method which first gets the data, then creates the QUBO, runs on the D'Wave computer and gets the results. Figure 9.

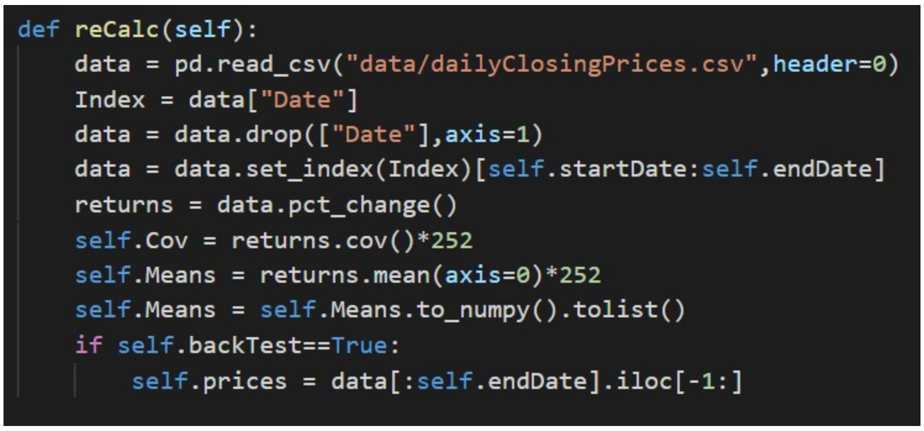
2.10 Helper Functions

The *executeTest* is a helper function of *doBackTesting* function which every-time updates the data, QUBO and executes on the D'Wave computer. It also calculates the number of shares allocated to each stock. Figure 10.



```python
def dWaveExecute(self):
    sampler = EmbeddingComposite(DWaveSampler())
    response = sampler.sample_qubo(self.Qubo, num_reads=self.numReads,\
        chain_strength=self.chainStrength)
    print("Time Spent in Quantum Computer: ",\
        response.info["timing"]["qpu_access_time"]/1000,"Milli Seconds")
    self.response = response
```

Figure 6:



```python
def reCalc(self):
    data = pd.read_csv("data/dailyClosingPrices.csv",header=0)
    Index = data["Date"]
    data = data.drop(["Date"],axis=1)
    data = data.set_index(Index)[self.startDate:self.endDate]
    returns = data.pct_change()
    self.Cov = returns.cov()*252
    self.Means = returns.mean(axis=0)*252
    self.Means = self.Means.to_numpy().tolist()
    if self.backTest==True:
        self.prices = data[:self.endDate].iloc[-1:]
```

Figure 7:

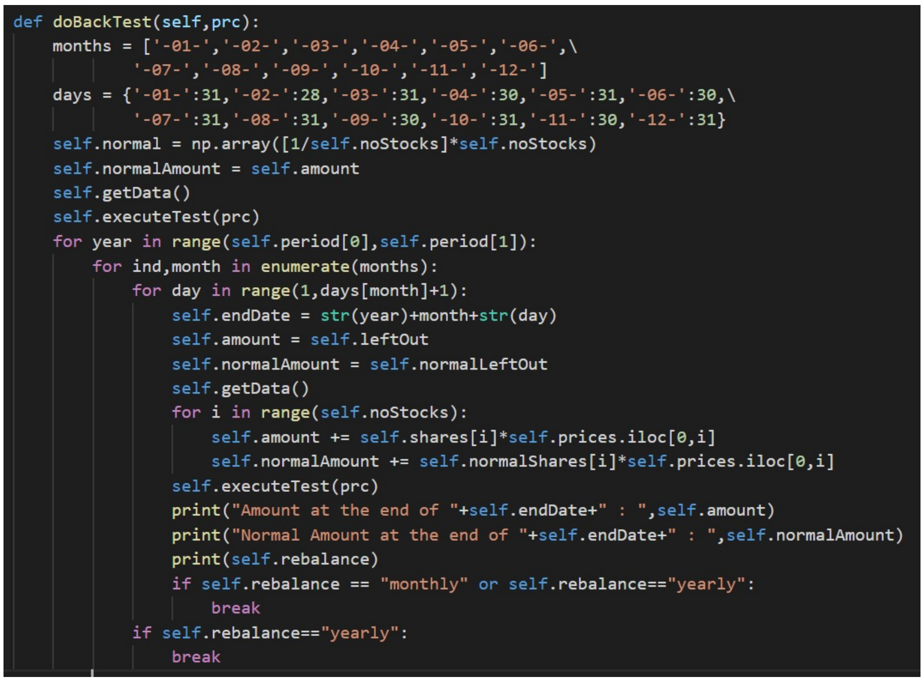
2.11 Running the Code

This Figure 11. shows the example for running the code.

3 Results

We have done extensive backtesting using the above code. Generally speaking, we have always outperformed the *dumb portfolio*, that is, the portfolio in which we simply allocate resources between available assets equally. Some of the variants we have tried out during backtesting are the following-

1. Used different combinations of stocks and precision bits.
2. Used different time intervals for the testing phase and the data reading phase.
3. Added rebalancing (selling and buying new stocks) at the end of the a given time interval.
4. Used different rebalancing intervals to find out which suits best.
5. Used different stocks. At times we tested using the leading stocks in the world, sometimes we used stocks from the tech industry and other times we NIFTY-50 stocks. This ensured that our model was viable in different markets.



```python
def doBackTest(self,prc):
    months = ['-01-','-02-','-03-','-04-','-05-','-06-',\
            '-07-','-08-','-09-','-10-','-11-','-12-']
    days = {'-01-':31,'-02-':28,'-03-':31,'-04-':30,'-05-':31,'-06-':30,\
            '-07-':31,'-08-':31,'-09-':30,'-10-':31,'-11-':30,'-12-':31}
    self.normal = np.array([1/self.noStocks]*self.noStocks)
    self.normalAmount = self.amount
    self.getData()
    self.executeTest(prc)
    for year in range(self.period[0],self.period[1]):
        for ind,month in enumerate(months):
            for day in range(1,days[month]+1):
                self.endDate = str(year)+month+str(day)
                self.amount = self.leftOut
                self.normalAmount = self.normalLeftOut
                self.getData()
                for i in range(self.noStocks):
                    self.amount += self.shares[i]*self.prices.iloc[0,i]
                    self.normalAmount += self.normalShares[i]*self.prices.iloc[0,i]
                self.executeTest(prc)
                print("Amount at the end of "+self.endDate+" : ",self.amount)
                print("Normal Amount at the end of "+self.endDate+" : ",self.normalAmount)
                print(self.rebalance)
                if self.rebalance == "monthly" or self.rebalance=="yearly":
                    break
            if self.rebalance=="yearly":
                break
```

Figure 8:

6. Tested out during different periods in history, such as just after some company incurred a high loss, during a recession, demonetisation, etc.

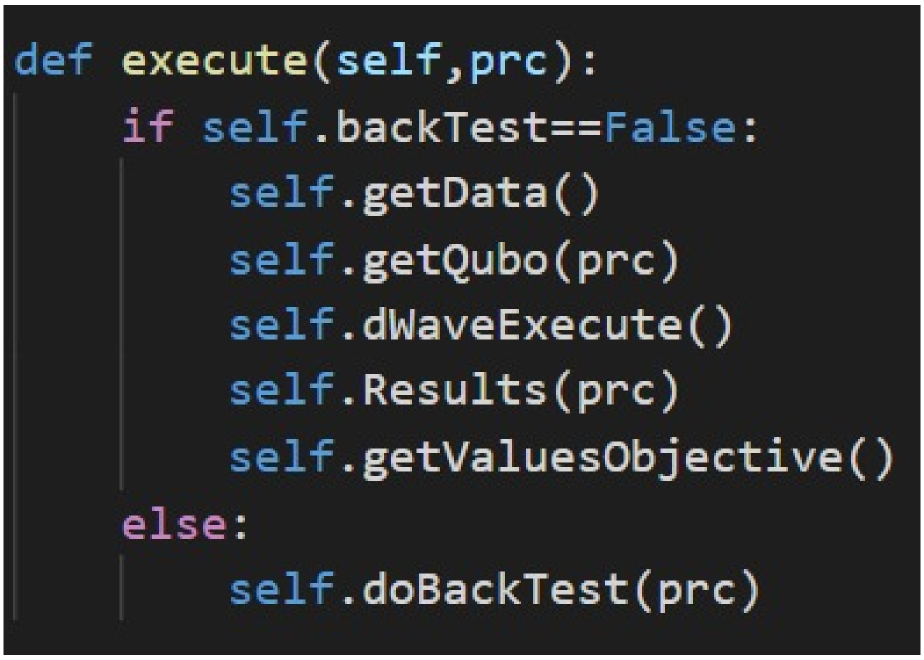
The time required for each of these tests is consistently between 16-18 ms, with negligible changes with increase in the number of stocks or precision. Classical versions of the underlying algorithm we used involves matrix inversion, which is typically a very time intensive operation. It will take a few to do so for, say, 10000 dimensional matrix, while quantum annealers can do it in much lesser time.

4 Future Work

1. We have implemented Markowitz's Portfolio Optimisation Technique. One can out some portfolio optimisation algorithm with quantum annealing.

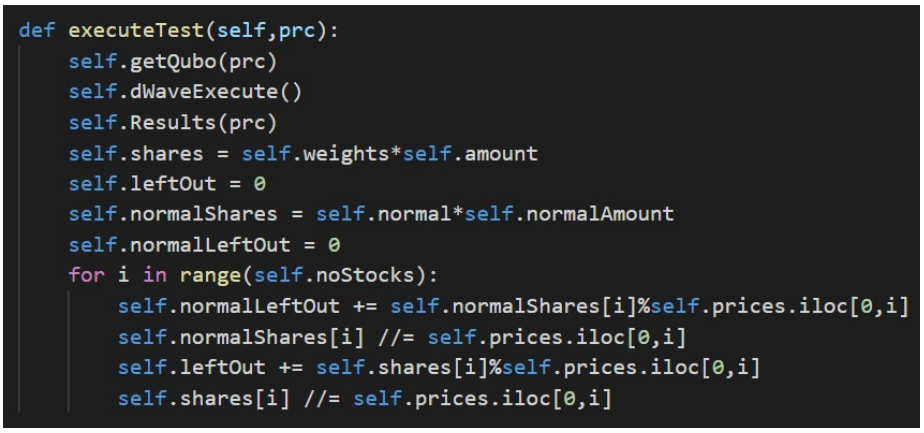
2. Build an interface that lets users interactively do portfolio optimisation using quantum annealer.

3. Using quantum annealer for prediction in finance using machine learning. Loss function in machine learning is also an optimisation problem and quantum computing using annealing is good at it!



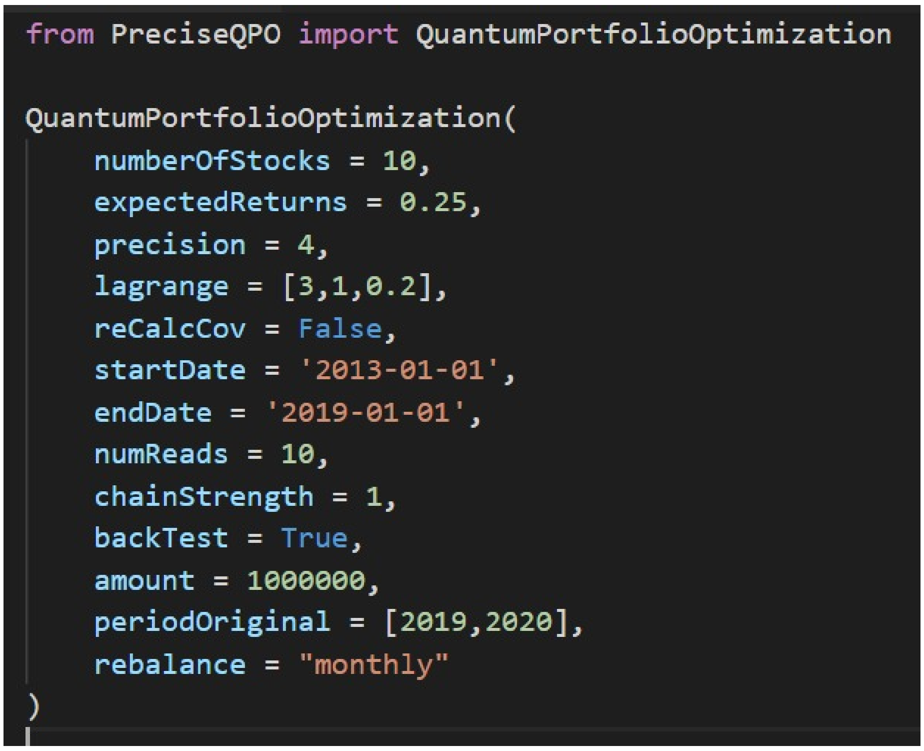
```python
def execute(self,prc):
    if self.backTest==False:
        self.getData()
        self.getQubo(prc)
        self.dWaveExecute()
        self.Results(prc)
        self.getValuesObjective()
    else:
        self.doBackTest(prc)
```

Figure 9:



```python
def executeTest(self,prc):
    self.getQubo(prc)
    self.dWaveExecute()
    self.Results(prc)
    self.shares = self.weights*self.amount
    self.leftOut = 0
    self.normalShares = self.normal*self.normalAmount
    self.normalLeftOut = 0
    for i in range(self.noStocks):
        self.normalLeftOut += self.normalShares[i]%self.prices.iloc[0,i]
        self.normalShares[i] //= self.prices.iloc[0,i]
        self.leftOut += self.shares[i]%self.prices.iloc[0,i]
        self.shares[i] //= self.prices.iloc[0,i]
```

Figure 10:



```python
from PreciseQPO import QuantumPortfolioOptimization

QuantumPortfolioOptimization(
    numberOfStocks = 10,
    expectedReturns = 0.25,
    precision = 4,
    lagrange = [3,1,0.2],
    reCalcCov = False,
    startDate = '2013-01-01',
    endDate = '2019-01-01',
    numReads = 10,
    chainStrength = 1,
    backTest = True,
    amount = 1000000,
    periodOriginal = [2019,2020],
    rebalance = "monthly"
)
```

Figure 11: