University of Crete
Department of Physics

Undergraduate Thesis

Quantum algorithms for computational finance applications

Michail Koumpanakis

AM:4840

1.4.2020

Supervisors : Georgios P. Tsironis and Georgios Neofotistos

Acknowledgements

First of all, I would like to thank my supervisors Prof. Georgios P. Tsironis and Georgios Neofotistos for their guidance, constructive comments and support. I would also like to thank Dr. Georgios Barmparis for contributing to my solid computational foundation and coding experience that was needed in order to complete this thesis. I would also like to thank my family and friends for their support.
Special thanks to Danai for her continuing support and for being my muse all the time.

Abstract

Recently there has been increased interest on quantum algorithms and how they are applied to real life problems. But is this interest justified? As an example, researchers have tried to apply quantum algorithms to solve linear systems of equations faster. These algorithms are considered to be more efficient and perform better than classical algorithms, in general. Among other fields, the field of finance presents real life problems, such as portfolio optimization and options pricing, which may exploit the efficiency of quantum algorithms for their solution.

The purpose of this thesis is to apply both classical and quantum algorithms in two important financial problems namely portfolio optimization and options pricing. We utilize advanced quantum algorithms such as the Variational Quantum Eigensolver (VQE) and the Quantum Amplitude Estimation (QAE).

VQE is a hybrid classical-quantum algorithm that is applied for optimization problems e.g. molecule simulations, and optimization problems. In this thesis I am applying the VQE algorithm to solve linear systems of equations in the framework of the Markowitz portfolio optimization model.

QAE is a method used in quantum computing to measure probabilities of desired states and is a generalization of Grover's search algorithm. I apply QAE to options pricing and I compare it to the classical Black-Scholes Merton model for pricing European call and put options.

I have used IBM's Quantum Experience platform to run the software developed and compare the performance of the aforementioned quantum and classical algorithms.

Contents

Chapter 1

Introduction

Unlike classical computers that rely on 0 and 1 for operations (that is, a bit), quantum computing takes advantage of its quantum nature and introduces a new kind of bit, the qubit, where 0 and 1 can exist in a superposition state. Quantum algorithms theoretically outperform classical algorithms.

In this thesis we apply certain quantum algorithms on finance applications, specifically portfolio optimization and options pricing. The first being one of the most important problems in finance.

The second, prices derivatives contracts that had a payoff depending on the underlying assets price at the exercise time. Pricing derivatives contracts depend on the solution of complex mathematical equations that are computational heavy to solve. Scientists have suggested quantum algorithms that make the above process faster.

The first algorithm that we will use is the **Variatonal Quantum Eigensolver (VQE)**, which is a method originally designed to approximate the ground energy of a molecule faster than a classical algorithm would do. This method has applications in the field of portfolio optimization which is the problem of selecting assets based on the historical returns of the assets and the level of risk to be taken. The second algorithm that we will apply is the **Quantum Amplitude Estimation** (QAE) that we will also discuss here. Using this algorithm we will price call or put European options.

These two algorithms will be implemented on IBMQ platform belonging to IBM. Several quantum platforms are available such as Quantum Azure, #Q (Microsoft) and QuTip. The platform of IBM (which uses qiskit as its Software Develeopment Kit (SDK)) was selected because of its simplicity and because it has many quantum coding examples available. The programming language is python and Jupyter notebooks have been used to run the code.

In the end, it will be discussed how these two algorithms compare to their classical counterparts taking into account the limited capability quantum computers have now (limited number of qubits) and how this comparison will turn out to be in the future.

Chapter 2

Theory

2.1 Introduction to quantum computing

2.1.1 Bit: the basic unit of information

Quantum computing takes advantage of the nature of quantum mechanics. In classical computing each piece of information is stored and processed in binary form (0 or 1).

Combinations of these bits (binary strings) produce numbers or letters. The binary number system is a positional notation with a radix of 2. This means using the two characters 0 and 1 to express numbers. For example the number 9 is represented by a 4 bit binary string and can be written as:

$$9 = (1 \times 2^3) + (0 \times 2^2) + (0 \times 2^1) + (1 \times 2^0) \quad or \quad |9\rangle = |1001\rangle$$

This is how numbers are represented in classical computing. And not only numbers.. Letters, images, or sound, they can be represented in the form of binary strings.

Quantum computers use the same principle. The main difference is that they use qubits that can be manipulated using quantum operations.

In computer science a basic way to represent calculation with bits is a circuit diagram. A circuit consist of basic gates. These gates are operators that manipulate the bits in order to have a desirable output.

Basic logic gates that apply for computing are the following:

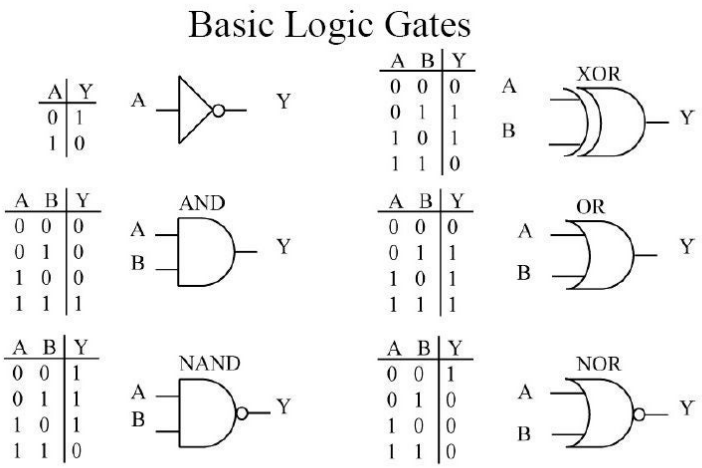

Figure 2.1: Logic gates [1]

The OR gate acts on two bits and outputs 0 if both bits are 0, else 1. The AND gate acts on two bits and outputs 1 if both bits are 1, else 0. The NOT gate is a very simple gate that flips a target bit from 0 to 1 or from 1 to 0.
The CNOT gate (exclusive OR) performs a NOT operation on the target bit if the controlled bit is 1 (basically an if statement).

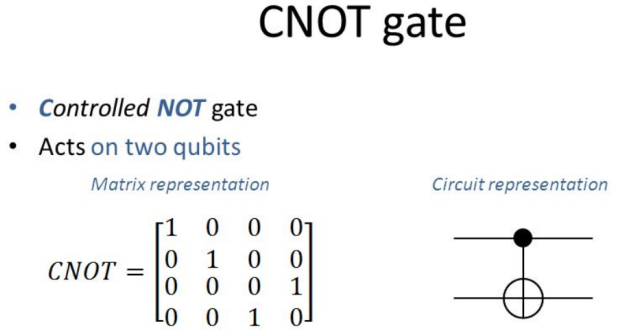

Figure 2.2: A CNOT or XOR gate [2]

The Toffoli gate is basically a CCNOT gate, [see Fig.(2.3)] performing a NOT operation in a target qubit if two control qubits are 1.

Inputs			Ouputs		
a	b	c	a'	b'	c'
0	0	0	0	0	0
0	0	1	0	0	1
0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

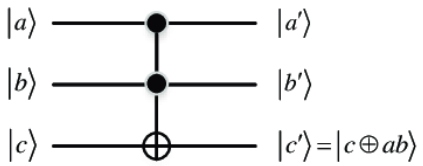

Figure 2.3: A CCNOT or Toffoli gate [3]

All these gates are combined into a circuit in order to construct basic algorithms. A very simple circuit that performs the sum of two $|1\rangle$ bits is the following:

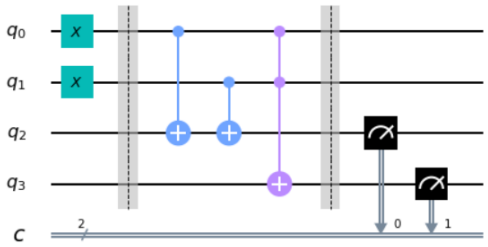

Figure 2.4: Example of a simple circuit (see [9])

Let's look deeper into the algorithm. We want to add two $|1\rangle$ bits. The output should be $|10\rangle = |2\rangle$. The basic idea is that we have two input bits and 2 output bits. The first step is to prepare our state. We perform the NOT gates to our two input bits in order to prepare our $|1\rangle + |1\rangle$ state. We want to manipulate our circuit in order to produce the $|0\rangle$ bit in our q2 bit register and the $|1\rangle$ bit in our q3 register. The first two CNOT gates target bit q2 flipping the target bit q2 to 1 and then back to 0 cancelling each other out. Finally the Toffoli gate performs a flip to 1 for the q3 register resulting in our $|10\rangle$ state. The two controlled gates that target q2 are used in order to connect our inputs to our outputs. In case we want to add a different number let's say $|1\rangle + |0\rangle$ then we just remove the second NOT gate without changing the algorithm between the input and the output.

Note that although this is a classical algorithm we can run it on a quantum computer just to showcase the basic operations that are performed. Unlike classical computers, in quantum computers we want to measure the qubit registers that correspond to our result.

Running this algorithm in IBM's Quantum Experience platform will have the following results:

We should expect to get $|01\rangle$ as our result with 100% certainty as in a classical computer and indeed we do. But quantum computers behave differently. Due to their quantum nature qubits that store our results need to be measured at the end

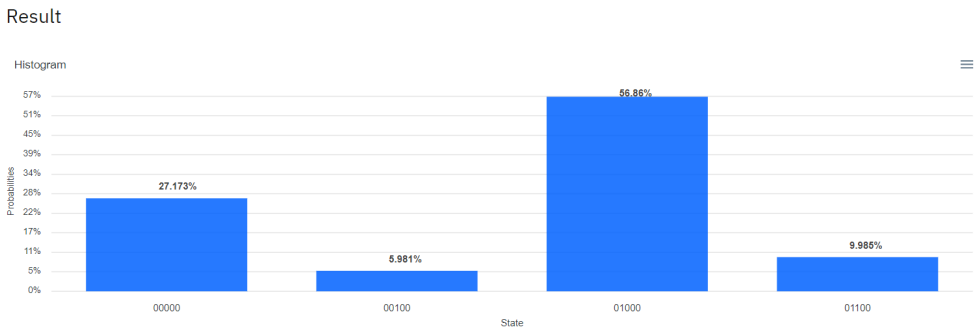

Figure 2.5: Results of the algorithm executed at ibmqx2 in IBMQ (IBM's quantum computer)

of the algorithm. Furthermore qubits have a probability to be in a specific state unlike classical bits that will always be in either 0 or 1 state. The measurement outputs a classical bit string (uncertainty collapses).

This measurement is executed a couple of times in a real quantum computer (usually 4000) in order to get a clear view of our result statistically. Quantum computers though are very sensitive to noise (interaction with the enviroment). Because we need to measure the particles encoded in a circuit, even if a small disturbance happens then the result is altered. That's what happens here. The algorithm is executed with 4096 shots in the quantum computer and 2294 times (56%) we had the correct result. That's one of the biggest problems quantum computers face right now and that's why most algorithms in my thesis will be run on quantum simulators.

2.1.2 Notation of quantum states

In quantum mechanics we use statevectors to describe the state of our system. For example, in order to define the spin of a particle we need to describe if it is up or down. To do so we use a vector. The $|0\rangle$ state corresponds to the up spin and the $|1\rangle$ to the down spin. This is the bra-ket notation and as you may have noticed it is used to describe the state of a qubit.

The corresponding statevector of the $|0\rangle$ state is :

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

and of the $|1\rangle$ state:

$$|1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

vspace0.3cm

In other words this notation describes the probability of measuring the particle with spin up or down. The probability is 1 if we are in the $|0\rangle$ state and 0 otherwise and vise versa. The same accounts for qubits. Each qubit has an amplitude (complex number) representing it's current state. The amplitude squared is the probability of finding the qubit in either state $|0\rangle$ *or* $|1\rangle$. For example, the following vector $|x\rangle$ stores the amplitudes of states $|0\rangle$ *and* $|1\rangle$.

$$|x\rangle = \begin{pmatrix} \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} \end{pmatrix}$$

The probability of measuring the qubit in the $|0\rangle$ state is 50 % and in the $|1\rangle$ also 50%.

$$|x\rangle = \tfrac{1}{\sqrt{2}}|0\rangle + \tfrac{1}{\sqrt{2}}|1\rangle$$

$$p(|x\rangle) = |\langle\psi|x\rangle|^2$$

In this example the qubit is in a state of equal superposition because we have the same probability of measuring it in the $|0\rangle$ *or* $|1\rangle$ state.

2.1.3 The Bloch Sphere

The representation of our qubit state can be visualized using the Bloch Sphere. The Bloch Sphere is basically a 3-D space that represents any possible combination of qubit states. The general state of a qubit $|q\rangle$ can be written as:

$$|q\rangle = a|0\rangle + b|1\rangle$$

with a and b being complex numbers. In order to write the above state using real numbers we can add a relative phase between them as:

$$|q\rangle = a|0\rangle + e^{i\phi}b|1\rangle$$

$$a, b, \phi \in \mathbb{R}$$

The qubit state can also be written using two variables ϕ *and* θ (a and b has to be normalized) as it is easier to represent it in the 3-D space, Bloch Sphere (surface with r=1) .

$$|q\rangle = \cos\tfrac{\theta}{2}|0\rangle + e^{i\phi}\sin\tfrac{\theta}{2}|1\rangle \tag{2.1}$$

A visual representation of the Bloch Sphere can be seen below:

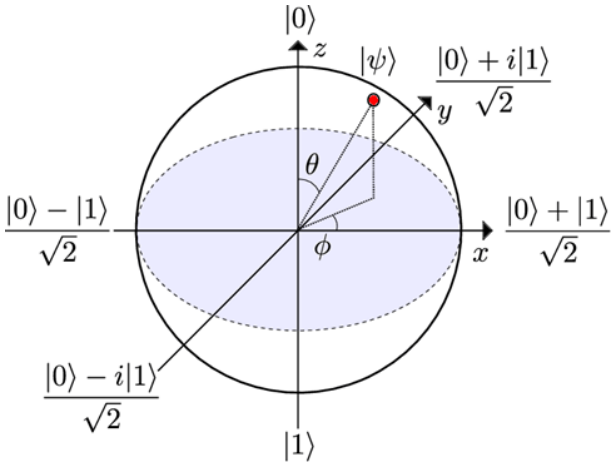

Figure 2.6: Visual representation of various qubit states in 3-D space [4].

It's now time to introduce more quantum gates that are essential in understanding the structure of more complex circuits and algorithms.

2.1.4 Basic quantum gates

In section 2.2.1 some basic logic gates were covered that are most frequently used in classical algorithms. These gates are the NOT, XOR (exclusive OR) and the XXOR gate and theirthat equivalent quantum gates are named Pauli X , CNOT and Toffoli respectively.These classical logic gates though are not enough to construct quantum algorithms. The need for more complex gates that manipulate quantum states efficiently are needed.

One of the most important gate that operates qubits states is the Hadamard gate **H**. Its role is to create a superposition. If the gate acts on a $|0\rangle$ state then the state transforms to :

$$|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle$$

else for the $|1\rangle$ state it transforms to :

$$|-\rangle = \frac{1}{\sqrt{2}}|0\rangle - \frac{1}{\sqrt{2}}|1\rangle$$

As we will see later its role is to expose the quantum nature of qubits and allow a quantum speed up for certain algorithms. The matrix representation of the gate can be seen below:

$$H = \frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

The next two gates that are used frequently in quantum algorithms are the Pauli-Y and Pauli-Z gates. Like the Pauli-X gate they perform a $\phi = \pi$ rotation in the y and z axis of the Bloch sphere likewise. The Pauli Z gate has the property of flipping the $|+\rangle$ *to the* $|-\rangle$ *state*, and vice versa.

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix} \qquad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$$

Next is the R_ϕ gate. The R_ϕ gate performs a rotation of ϕ around the Z-axis.

$$R_\phi = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\phi} \end{bmatrix}$$

Finally we have U_3 the most general quantum gate with parameters θ , λ and ϕ. All the quantum gates can be built from this gate given the right parameters.

$$U_3 = \begin{bmatrix} \cos(\theta/2) & -e^{i\lambda}\sin(\theta/2) \\ e^{i\phi}\sin(\theta/2) & e^{i\lambda+i\phi}\cos(\theta/2) \end{bmatrix}$$

2.1.5 Grover's search algorithm

On this subsection an elegant quantum algorithm will be explained, Grover's search algorithm[17]. It is one of the first quantum algorithms that introduces a quantum speed to its classical counterpart and is very effective for searching databases.

Suppose there is a function f(x) which is 0 for all x except for x=u. The goal is to find u.

$$f(x) = \begin{cases} 0 & \text{if } x \neq u \\ 1 & \text{if } x = u \end{cases} \tag{2.2}$$

For example, let's say we have a phonebook and we want to find a certain number u. A classical algorithm would need at most O(n) steps, equal to the number of cellphone numbers we are searching in order to find u. Grover's algorithm can find the desired phone number in at most $O(\sqrt{n})$ steps.

1) First of all, we start by creating a uniform superposition over all qubits. To do that we apply a Hadamard gate to each qubit register of our circuit.

$$|\psi\rangle = H^{\otimes n}|0\rangle^n = \frac{1}{\sqrt{2^n}} \sum_{x=0}^{2^n-1} |x\rangle \tag{2.3}$$

The reason behind this, is that we want to create all possible combinations of numbers in our phonebook/list with equal probability when a measurement happens. For simplicity, let's say we have 7 numbers in our phonebook. The numbers are : [0,1,2,3,4,5,6,7] or [000,001,010,011,100,101,110,111] in bit format. The probability of finding any number in the phonebook is p= $\frac{1}{8}$. Here each qubit register will have an amplitude of a = $\frac{1}{\sqrt{8}}$.

Now, in order to find a specific number let's say $|3\rangle = |011\rangle$ we have to apply Grover's search algorithm second step .

2) Apply an oracle function U_f in our state $|\psi\rangle$ that maps the number we are searching for with a negative amplitude. The oracle function is a black box which distinguishes the number we are searching from other numbers.

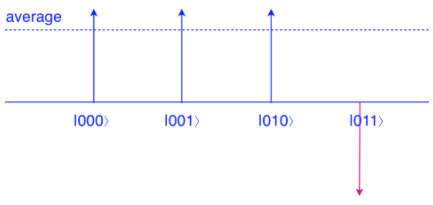

Figure 2.7: Visualization of the oracle function [18]

Below you can see the oracle function that is applied in our state $|\psi\rangle$ after the Hadamard transformation. Each number from 0 to 7 has a different oracle function. Here the simplest version is implemented which is for the number$|7\rangle = |111\rangle$. As you can see a Toffoli gate is acted on qubit register q[2] with two controlled qubits (register q[0] and q[1]) resulting in a flip of qubit q[2] in case the 2 first two qubits happen to be in the $|1\rangle$ state. Basically after the Hadamard transformation the Toffoli gate entangles the first two qubits with the third. The result is a change of amplitude on the third qubit. When a Hadamard gate is applied on a qubit and then it is acted again the superposition collapses and the qubit becomes a normal bit again but if a CNOT or Toffoli gate acts in-between then the states get entangled (control-target qubit) and it's impossible to get the initial state again thus changing the amplitude.

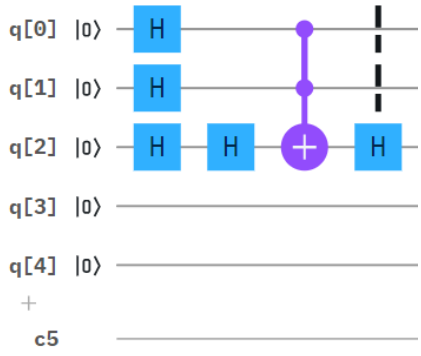

Figure 2.8: Implementation of Hadamard and Oracle transformation [IBMQ]

3) Apply the Grover operator :

$$G = 2|\psi\rangle(\langle\psi| - I)O$$

The Grover operator is responsible for boosting the amplitude of our desired quantum state and decreasing the amplitude of the rest. As you can see in figure 2.7 after the Grover operator is applied the average amplitude $\frac{1}{2\sqrt{2}} = 0.35$ drops to 0.265. Then the Grover operator is applied and it basically flips the amplitudes around the average. So the previously positive quantum states that were above the average in figure 2.7 will decrease and the only negative amplitude will increase.

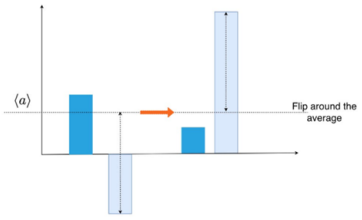

Figure 2.9: Visualization of the Grover operator [18]

The part of the circuit that implements the Grover operator can be seen below :

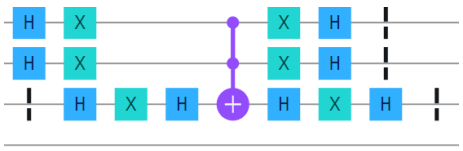

Figure 2.10: Implementation of the Grover operator on a quantum circuit [IBMQ]

As you can see the oracle function is again implemented but now there are Pauli-X and Hadamard gates that act on the qubits too.

Finally the complete 1-iteration algorithm can be seen below. The dashed lines are called barriers and they play the role of separating the different parts of the algorithm.

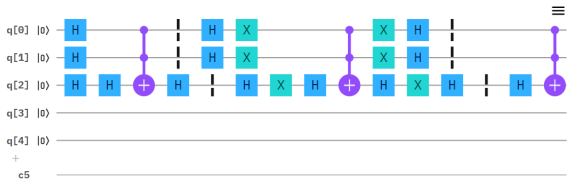

Figure 2.11: Complete circuit for Grover's algorithm[IBMQ]

4) Repeat $\sqrt{n}$ times where n are the number of qubits and $N = 2^n$ are the total number of combinations. In order to finally find the item or number we are searching in a list we have to repeat Grover's algorithm $\sqrt{n}$ times. For our example, with 3 qubits and 8 possible combinations of numbers the main body of the algorithm has to be repeated 2 times.
To sum up, the most important things about the algorithm are the manipulation of superpositions, the Oracle function and the Grover operator. With only 3 qubits registers we can create 8 different numbers while we would normally need more bits for a classical algorithm. But that is not enough because the Oracle function and the Grover operator manipulate our quantum state efficiently and manage to differentiate the desired quantum state from the rest.

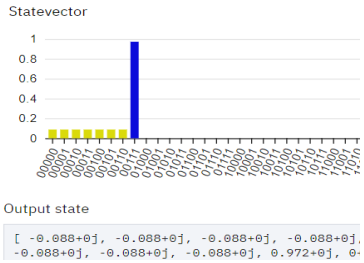

Figure 2.12: Amplitude visualization on statevector simulator[IBMQ].

2.1.6 Quantum Fourier Transform

The quantum Fourier transform (QFT) is the quantum implementation of the discrete Fourier transform over the amplitudes of a wavefunction.

The Discrete Fourier Transform takes a basis $x_n = [x_1, x_2, ..., x_n]$ and transforms it into a different base $y_n = [y_1, y_2, ..., y_n]$ using the formula below :

$$y_n = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \omega_N^{jn} \tag{2.4}$$

where $\omega_N^{jn} = e^{2\pi i \frac{jn}{N}}$.

It is very useful when dealing with periodic data because we are able to extract the most important features from our data e.g. time-frequency.
Likewise the QFT is the quantum version of the DFT. The only thing that changes is that the bases x_n, y_n are quantum with :

$$|x\rangle \mapsto \frac{1}{\sqrt{N}} \sum_{y=0}^{N-1} \omega_N^{xy} |y\rangle$$

or:

$$\sum_{i=0}^{N-1} x_i |i\rangle \mapsto \sum_{i=0}^{N-1} y_i |i\rangle$$

Like all quantum algorithms the above formula is implemented on a circuit by applying controlled rotations on multiple qubits.
The transforming basis y_n is stored in the qubit registers as following: The leftmost qubit in our circuit has the lowest frequency (least rotations) and the rightmost the highest frequency (most rotations around the Z-axis). Furthermore as you can see below each qubit is targeted by the above qubits $(n-j, ..., n-2, n-1)$ with rotations $(\frac{\pi}{2^{n-1}}, \frac{\pi}{2^{n-2}}, .., \frac{\pi}{2^{n-j}})$ accordingly .

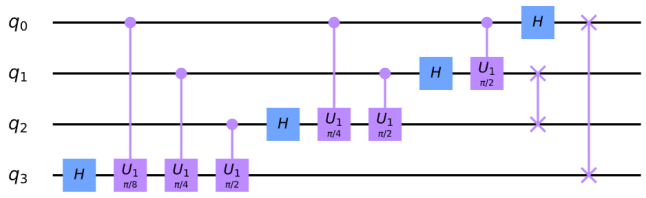

Figure 2.13: Circuit for the QFT for 4 qubits[IBMQ]

2.1.7 Quantum phase estimation

Quantum phase estimation is a quantum method used in many quantum algorithms to calculate the phase θ of a unitary matrix U or it's eigenvalue $e^{2\pi i\theta}$. The corresponding transformation is :

$$U|\psi\rangle = e^{\mathbf{2\pi i\theta}}|\psi\rangle \tag{2.5}$$

The unitary matrix U is the phase shift gate R_ϕ we saw in chapter (2.1.4) with an angle of $\phi = 2\pi\theta$. Note that U has a norm of 1 due to its unitary nature. The quantum subroutine can be seen below. Each step of it will be discussed analytically.

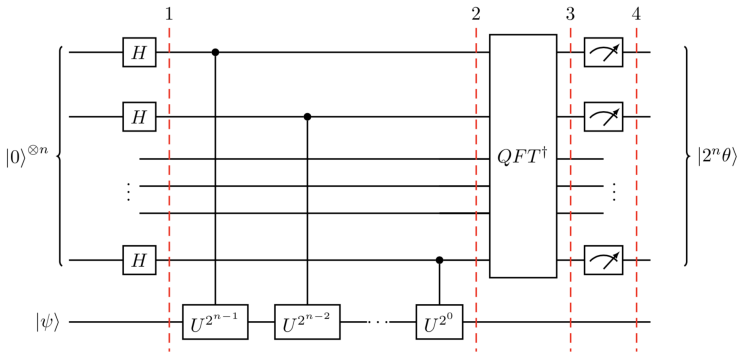

Figure 2.14: Visualization of the quantum phase estimation circuit [9]

First of all, we have to prepare a state $|\psi\rangle = \sum_{k=1}^{n} |\psi_k\rangle$ where n is the number of register counting qubits. Then we have to apply n Hadamard gates to these qubits transforming the state into :

$$|\psi\rangle = \frac{1}{\sqrt{2^n}} \left(|0\rangle + |1\rangle\right)^{\otimes n} |\psi\rangle \tag{2.6}$$

Then for $0 \leq i \leq n$ apply $C_{n-i-1}U$ gates(controlled U gates), U gates are applied only if its corresponding control bit is $|1\rangle$ to an ancilla qubit q (register qubits m) where our initial state $|\psi_\theta\rangle$ is stored . The state $|\psi_\theta\rangle$ corresponds to the transformation :

$$|\psi\rangle = \frac{1}{\sqrt{2^n}} \left(|0\rangle + e^{\mathbf{2\pi i\theta}2^{n-1}}|1\rangle\right) \otimes \cdots \otimes \left(|0\rangle + e^{\mathbf{2\pi i\theta}2^1}|1\rangle\right) \otimes \left(|0\rangle + e^{\mathbf{2\pi i\theta}2^0}|1\rangle\right) \otimes |\psi\rangle$$

$$= \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} e^{\mathbf{2\pi i}\theta k} |k\rangle \otimes |\psi\rangle \tag{2.7}$$

Now we are half-way there. All we need to do is apply an Inverse Quantum Fourier Transform(IQFT), inverse of QFT we saw in the previous section [2.1.6]. The correspondig state is :

$$|\psi\rangle = \frac{1}{\sqrt{2^n}} \sum_{k=0}^{2^n-1} e^{\mathbf{2\pi i}\theta k} |k\rangle \otimes |\psi\rangle \xrightarrow{\mathcal{QFT}_n^{-1}} \frac{1}{2^n} \sum_{x=0}^{2^n-1} \sum_{k=0}^{2^n-1} e^{-\frac{2\pi i k}{2^n}(x-2^n\theta)} |x\rangle \otimes |\psi\rangle \tag{2.8}$$

Now the final step is to measure our state $|\psi\rangle$ that corresponds to the register counting qubits (not the ancilla ones). Generally more counting register qubits correspond to better accuracy. The measurement measures state:

$$|\psi\rangle = |2^n\theta\rangle \otimes |\psi\rangle \tag{2.9}$$

The final angle θ that we wanted to find is $\theta = \frac{measurement}{2^n}$ with n the number of register qubits. The measurement can be any integer or float e.g. for $|001\rangle = 1$, $\theta = 1/8$ for n = 3 .

2.2 Variational Quantum Eigensolver

As we will see later in section 2.3 the main goal of optimization algorithms is to find the minimum of a function L in order to determine it's optimal state.
In physics, for example, the goal is to find the lowest eigenvalue of a Hermitian matrix H describing an atom, also known as the Hamiltonian, in order to determine the ground state of the atom. The lowest eigenvalue corresponds to the minimum energy that atom can have. For simple atoms this problem is solvable in a classical computer but for more complex ones or even molecules the optimization problem scales exponentially due to a huge number of parameters. Quantum computers solve that problem by utilizing a hybrid classical-quantum model by the name of Variational Quantum Eigensolver (VQE)[10], [12]. The basic idea is to take the advantages of both quantum and classical computers by allocating classically easy tasks to classical computers and other more complex tasks to quantum computers.

2.2.1 Variational Theorem

Before exploring the algorithm we have to understand it's mathematical background. The Variational Theorm [VT] is used in quantum mechanics to determine the ground state of an atom. Given a Hermitian matrix H that represents the Hamiltonian and a wavefunction $|\psi\rangle$ we know that :

$$H\,|\psi\rangle = \lambda\,|\psi\rangle \tag{2.10}$$

where λ are the eigenvalues of the Hermitian matrix H. Furthermore, the expectation value of the observable H (i.e. energy) on a quantum state $|\psi\rangle$ is given by :

$$\langle H\rangle_\psi \equiv \langle\psi|H|\psi\rangle \tag{2.11}$$

The variational theorm states that :

$$\langle H\rangle_\psi \geq \lambda_{min}$$

Eigenvalue λ_{min} corresponds to the lowest energy of the system E_o.
Equation (2.11) can also be written as a linear combination of states $|\psi_i\rangle$ with eigenvalues λ_i as:

$$E_o \leq \langle H\rangle_\psi = \langle\psi|H|\psi\rangle = \sum_{i=1}^{N} \lambda_i |\langle\psi_i|\psi\rangle|^2 \tag{2.12}$$

2.2.2 Preview of the algorithm

The basics steps that VQE will use in order to find the global minimum of an optimization problem are the following:

1) Prepare a variational state $|\psi(\theta_i)\rangle$ with parameters θ_i by generating trial wave functions (ansatz) using a parametrized quantum circuit $U(\theta_i)$ such as $|\psi(\theta_i)\rangle = U(\theta_i)\,|0\rangle$.

2) Calculate the expectation value of the Hamiltonian $E = \frac{\langle\psi|H|\psi\rangle}{\langle\psi|\psi\rangle}$.

3) Use a classical optimizer that suits the problem to find new optimal θ_i in order to reach the global minimum.

4) Iterate the above until convergence.

Now let's look each step more closely. Firstly, the simplified (without parameters θ_i, λ_i) parametrized quantum circuit $U(\theta_i)$ is described as :

$$U(\theta_i) = \begin{pmatrix} cos(\frac{\theta_i}{2}) & -sin(\frac{\theta_i}{2}) \\ sin(\frac{\theta_i}{2}) & cos(\frac{\theta_i}{2}) \end{pmatrix} \tag{2.13}$$

Also known as the rotation matrix. This matrix will act on the qubits of our problem and prepare a trial wave function with random initial angles θ_i. Then controlled Pauli-Z gates will act on the qubits in order to entangle them and then again the rotation matrix will act on the qubits and will create an initial "guess" (ansatz) concerning the lowest eigenvalue of our problem by calculating the expectation value $\langle\psi(\theta_i)|H|\psi(\theta_i)\rangle$.
In order to implement the above theorem in a quantum computer one has to prepare the Hamiltonian in a way that quantum operations can be acted upon it. That is done by splitting our Hamiltonian as a sum of Pauli-Z matrices that are acted as gates in our quantum circuit.
The circuit that implements this procedure is shown below:

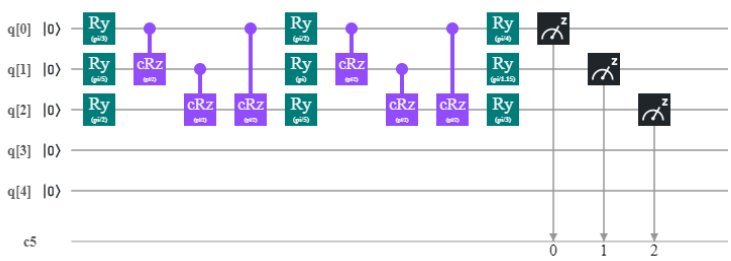

Figure 2.15: VQE variatonal form for n=3 qubits and p=2 depth with random selected θ_i(circuit designed in IBMQ)

The above circuit is an example of implementing the algorithm in a 3 qubit system. The circuit's depth p describes the number of times the basic circuit (3 controlled-Z and 3 Ry gates here) will be repeated. For a complex optimization problem p needs to be adjusted accordingly in order for the algorithm to converge.

The result of the algorithm (without classical optimization yet) is shown below using a statevector simulator in IBMQ.

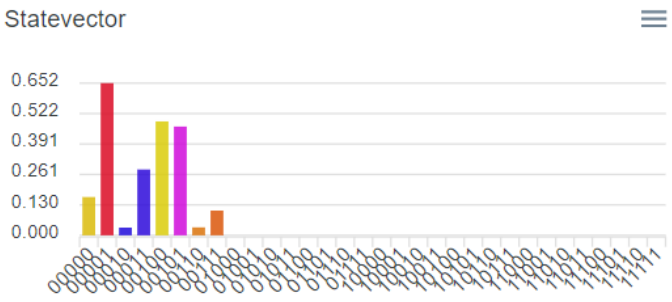

Figure 2.16: Results of VQE algorithm for a random problem

Now the classical optimizer will find new values of θ_i that will be used again for step **1)** in order to further decrease our expectation value and find the global minimum. The number of times this process needs to be repeated is unclear.Each problem requires different number of iterations in order to converge. The classical optimizer that is used for most problems is the Constrained Optimization by Linear Approximation optimizer (COBYLA).

A complete description of the algorithm is described by the figure below:

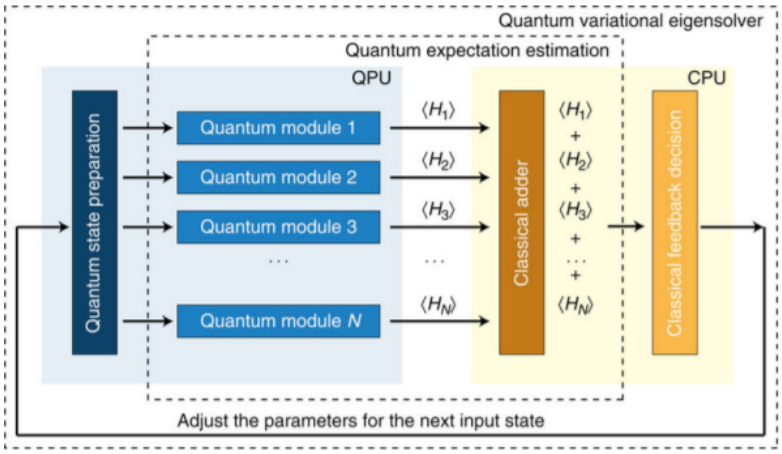

Figure 2.17: Summary of VQE Ref.[15] .

2.3 Modern portfolio theory

Modern portfolio theory[8] is a mathematical framework that is used for the optimization of a given portfolio with **n** financial assets. A portfolio is a list of stocks an investor holds. The goal is to find the best possible portfolio out of all possible combinations of the **n** assets by maximizing the expected return of the portfolio given a risk level **q** that depends on the investor. The idea behind modern portfolio theory is "the spreading of risk", the idea that holding different kinds of financial assets is less risky, often called diversification.

2.3.1 The Markowitz model

The mathematical model that solves this optimization problem is called the Markowitz model[20], named after the economist that proposed it who also got a Nobel for it.

There are two important parameters for this model, the mean and the variance of the of the portfolio return. The model uses the mean expected returns of the assets:

$$\vec{R} = E[\vec{r}] \tag{2.14}$$

where $\vec{r} = [r_1, r_2, .., r_i]$ is the vector of daily returns for the n stocks. The expected return of the portfolio is then

$$\mu = R^T w \tag{2.15}$$

where w is the weight vector which denotes how much money will be invested in each selected stock e.g. [0.3,0.2,0.5].
The co-variance matrix is defined as :

$$\vec{\Sigma} = V[(\vec{r} - \vec{R})(\vec{r} - \vec{R})^T] = \frac{\sum_{i=1}^{N}(\vec{r} - \vec{R})(\vec{r} - \vec{R})^T}{N - 1} \tag{2.16}$$

with N being the total number of trading days that are used to calculate the mean daily returns for the portfolio. Each value of the covariance matrix has the covariance between stock i and j. In other words, the model minimizes the risk given the correlations between the n stocks (if they are "moving together" or not). .

The variance is calculated by calculating the covariance of the portfolio through the covariance matrix and multiplying it by the weights of each stock selected.
The variance of portfolio return is defined as:

$$v = w^T \Sigma w \tag{2.17}$$

The following term must be minimized to acquire the optimal portfolio :

$$min(w^T \Sigma w - qR^T w) \tag{2.18}$$

where $q \geq 0$ is the "risk level" factor, where 0 results in a risk-neutral investor and the optimal solution would only maximize the expected return or minimize the the portfolio variance, independent of the risk and as q increases the solution gets more and more risk-averse resulting in the portfolio variance to be penalized more.

In other words the variance of the portfolio is the risk that has to be minimized (different from the risk level factor q) and the expected return is the value that will be maximized if we choose to denote the problem as a maximization one instead of a minimization one. Different risk level factors q give us different portfolio selections. All these portfolios lie in a 2-D space with the standard deviation as the x-axis and the expected return of the assets as the y-axis. In this 2-D space there is a parabolic line which is called the efficient frontier. A portfolio lying on the efficient frontier represents the combination offering the best possible expected return for given risk level. There is also one more constraint subject to the problem and that is :

$$1^T w = b \qquad or \qquad \sum_{i=1}^{n} (w_i) = b \tag{2.19}$$

If the investor wishes to be able to choose any possible combination instead of a defined budget **b**, even those portfolios with 1 or n stocks, then the above term may equal 1.
To sum up the optimization problem that needs to be solved is :

$$min(w^T \Sigma w)$$

subject to

$$qR^T w = \mu \quad \&$$
$$1^T w = b$$

It will be solved using Lagrangian mechanics.
The linear system to be solved[21] is defined as:

$$\begin{pmatrix} 0 & 0 & R^T w \\ 0 & 0 & 1^T \\ R & 1 & \Sigma \end{pmatrix} \begin{pmatrix} \lambda \\ z \\ w \end{pmatrix} = \begin{pmatrix} \mu \\ b \\ 0 \end{pmatrix} \tag{2.20}$$

or

$$Ax = B$$

A more analytical interpretation where the Lagrangian becomes more obvious is the following:
Equation (2.17) can be also written as : $\sigma^2 = \sum_{i,j=1}^{n} w_i w_j \sigma_{ij}$ with the Lagrangian defined as:

$$\Lambda = \sum_{i,j=1}^{n} w_i w_j \sigma_{ij} - \lambda(\sum_{i=1}^{n} w_i - b) - z(\sum_{i=1}^{n} w_i r_i - \mu) \tag{2.21}$$

and the minimization of the following terms will give us the optimal portfolio $w = [w1, w2, ..., w_n]$:

$$\frac{\partial \Lambda}{\partial w_i} = 0 \qquad i = 1, 2, ..n \quad \& \quad \frac{\partial \Lambda}{\partial \lambda} = 0 \quad \& \quad \frac{\partial \Lambda}{\partial z} = 0 \tag{2.22}$$

Linear system (2.20) can be solved by inversing matrix A(by any method available) :

$$x = A^{-1}B$$

with

$$x = (\lambda, z, w)$$

The optimal portfolio vector w is the final solution. As we will see next the hybrid quantum-classical model [VQE] that runs on a quantum computer will follow the same process as here to compute w.

2.4 Options

2.4.1 Definition

Options are financial contracts that are based on the value of underlying securities such as stocks, bonds [7]. These type of contracts like future contracts fall in the category of derivatives. There are two basic kinds of options contracts : the call option and the put option.

A call option gives the buyer the right, but not the obligation, to buy an underlying asset at an agreed price (the strike price) at some time in the future (the maturity).

A put option gives the buyer the right, but not the obligation, to sell an underlying asset at an agreed price (the strike price) at some time in the future (the maturity).

For example, let's say Danai has 100$ as savings and she wants to invest them in a global airline company. She heard that the airline company made a new deal with a large number of hotels around the world for this summer season. She believes that the company's underlying stock will thrive in the summer and she is most probably right. Let's say the company's stock trades at 10$ at April. Danai would be able to purchase 10 stocks and most probably wait until the end of summer to sell them for a profit.

But Danai has a strange feeling. She senses that this summer an unexpected disease may occur that will make the price of the stock fall, a risk Danai doesn't want to take. Her friend Mike tells her that there is a special contract on the stock market that allows her to buy these 10 stocks at a specific date (maturity date) in the future at the same price (strike price) she would buy them back at April, just by paying a premium or, in other words, a payment in advance. The key thing here is that Danai is not obligated to buy these stocks at all if the price falls in the future because then she would make no profit, she would just pay a small fee or premium. But if she bought the stocks themselves she would lose a lot of money if the price fell. In the positive scenario where the price increased then Danai would exercise the contract at the specific date and make a profit with the cost of paying just a small fee (call option).

That's the basic idea behind a call option. The put option is similar. If Danai wanted to sell her stocks at the end of summer but wasn't sure if the price could go any higher before falling in the next months then she would purchase a put option contract that would give her the right, but not the obligation, to sell the stock back at summer. If the stock price did indeed drop the next months then Danai would exercise the contract and make a profit by selling the stocks at summer's price just by paying a small fee. But if the price actually increased in the next months, then Danai would not exercise the contract.She would still have the stocks on her portfolio and she would have the chance to make even more profit by selling them now and not back in the summer.

2.4.2 Options pricing

Pricing these option contracts [13] is a matter that concerned exchanges for a very long time. In order to do so various methods have been used specifically for options. Here we will see the simplest kind of option, a European call or put option, which is priced very simply via the following methods. The first method we will talk about is the Monte Carlo pricing method and the second one is the Black-Scholes-Merton method.

The first method is a simulation process (Wiener process) where we generate random paths for the price of the desired asset in a specific time frame in the future. By keeping the last value from every random price path we calculate the pay-off (profit) relevant to the initial Strike price. Finally we price the option by calculating the mean pay-off. If the previous sound confusing, think of how you and your friends try to predict the weather 10 days in advance. Each one of you makes a prediction and every day you are allowed to update your prediction. In a sense you are creating a random path for the temperature value because weather prediction depends on randomness (for the non-scientists). After 10 days each one of you has predicted a temperature which you compare with the actual temperature. One of you almost got it right! You suddenly realise that the best approach would be to take the average over all these possible predictions and compare it with the actual temperature. That is the basic idea behind the Monte Carlo method.

The second method relies on calculating the cumulative standard normal probability distribution function of a stock and using it along with other parameters that we will see in a while to price the option.

2.4.2.1 Monte Carlo model

Stocks are often assumed to follow a random walk, like particles that collide with each other. This type of motion is called Brownian motion[19] and is derived from the random motion of particles suspended in a fluid resulting from their collision with the fast-moving molecules in the fluid. It is thus possible to model this random walk using this technique, by simulating the stochastic process of the stock price.

The price of the underlying stock S(t) is assumed to follow the following stochastic differential equation :

$$dS(t) = \mu S(t)dt + \sigma S(t)dW \tag{2.23}$$

where $\mu \, and \, \sigma$ are the drift and the volatility of the specific stock and and dW is a Wiener process [11]. Volatility is the standard deviation of returns and the drift μ is the mean of returns.

We want to keep only prices greater or equal to zero and thus take the logarithmic term with :

$$dlogS(t) = (\mu - \frac{\sigma^2}{2})dt + \sigma dW \Longrightarrow logS(t) = logS(0) + (\mu - \frac{\sigma^2}{2})t + \sigma \int_0^t dW$$

The final stock price at maturity time T (the time we exercise the option contract) is :

$$S(T) = S(0) \times \exp\left[\left(r - \frac{1}{2}\sigma^2\right)T + \sigma\sqrt{T}N(0,1)\right]. \tag{2.24}$$

where r is the risk-free interested rate that replaces the drift μ for simplicity (risk neutral investor). The interval of the Wiener process dW from time 0 to t is s a normally distributed random variable with mean zero and a variance of 1 multiplied by the square root of time.

In a Monte Carlo simulation the goal is to create a large number of stock price estimations using the above expression. These generated stock prices are then used to estimate the option price. We calculate the option price by taking the mean of a pay-off function f and then discounting by the increase in value due to the risk-free interest rate r.

The pay-off function f is a very simple mathematical expression, depending only on the stock price at maturity and the strike price K that was set by the investor when buying the contract.

Call option:

$$f_c = max(S(t) - K, 0) \tag{2.25}$$

Put option:

$$f_p = max(K - S(t), 0) \tag{2.26}$$

The final option price p is :

$$p = e^{-rT}\mathbb{E}\left[f(S(t), K)\right] \tag{2.27}$$

The above agree with the definition of options in section 2.2.1. An option is only exercised if the underlying price S(t) is greater than the strike price K for a call option or less than the strike price K for a put option.

As we will see later the price of the option for a stock depends mainly on the underlying volatility of the stock and the day to maturity for the option. The greater the uncertainty both in time and in price the greater the option costs and vice versa.

2.4.2.2 Black-Scholes-Merton Model

The Black-Scholes model is used in finance to price European call or put options. It's based on a partial differential equation which models the price evolution of the asset. Just like the Monte Carlo method it takes into account various parameters such as the volatility, risk-free interest, strike price and time to maturity of the asset. It is notable that professors Merton and Scholes won the Nobel prize (Bank of Sweden Prize in Economic Sciences in Memory of Alfred Nobel) back in 1997 for "a new method to determine the value of derivatives".

The partial differential equation derived from a geometric Brownian motion is:

$$\frac{\partial V}{\partial t} + \frac{1}{2}\sigma^2 S^2 \frac{\partial^2 V}{\partial S^2} + rS\frac{\partial V}{\partial S} - rV = 0 \tag{2.28}$$

The Black-Scholes model as said before makes certain assumptions (that simplify the problem to be solved) :

First of all the option is European and can only be exercised at expiration. American options can be exercised at any time and are thus harder to model. Secondly, no dividends are paid out during the life of the option. Thirdly, there are no transaction costs in buying the option (no platform fees). Furthermore, the risk-free return and volatility of the underlying asset are known and are constant (something not entirely true in the stock market). Finally, the returns of the underlying asset are normally distributed.

The final solution that gives us the price of the option (premium) can be seen below:

$$C = S_t N(d_1) - Ke^{-rt}N(d_2) \qquad with \tag{2.29}$$

$$d_1 = \frac{ln(\frac{S_t}{K}) + (r + \frac{\sigma^2}{2})t}{\sigma\sqrt{t}} \tag{2.30}$$

$$d_2 = d_1 - \sigma\sqrt{t} \tag{2.31}$$

where C is the call option price, S is the current stock price, K is the strike price, σ is the volatility, t the time to maturity, r the risk-free interest rate and N the standard normal cumulative probability distribution function.

As we will see later using this method is much faster than method 1 and less computationally expensive which is something we should always consider.

2.5 Quantum Amplitude Estimation for options pricing

In this section we will briefly see how to use Quantum Amplitude Estimation (QAE) [14], a quantum algorithm that is implemented on a quantum computer to price European call or put options[22],[23]. The algorithm is a generalization of Grover's search algorithm and is very similar to Quantum Phase Estimation (QPE) we saw in section [2.1.7].

Assume a unitary operator U acting on (m+1) qubits that applies the transformation:

$$U|0\rangle_{n+1} = \sqrt{1-a}|\psi_o\rangle_n|0\rangle + \sqrt{a}|\psi_1\rangle_n|1\rangle \tag{2.32}$$

The quantum states $|\psi\rangle_n$ are normalized and **a**, the amplitude, is unknown with a value of $a \in [0, 1]$. The goal is to estimate the value of a that is the probability of measuring qubit $|1\rangle$ in the last qubit. Furthermore, we need n additional sampling/counting qubits to store the result, a total of $M = 2^m$ applications of U in the Quantum Phase Estimation subroutine to compute the eigenvalues of U. Like in section (2.1.7) we first initialize the register counting qubits m on an equal superposition by using Hadamard gates. Then we use these sampling qubits to control different powers of U on our ancilla register qubits n. Then we apply an Inverse Quantum Fourier Transform(IQFT) and measure the counting register qubits m as integers e.g. $x \in [000, 001, ...111]$. These numbers have different probabilities as expected and the value x with the highest probability is our final amplitude $a = sin^2(\frac{x\pi}{M})$.

In order to use this algorithm for options pricing we need a linear function f that will be our payoff function (subject to a random variable X) that stores various payoffs/profits from different time paths of the asset just like in Monte Carlo method(2.4.2.1). The amplitude a is then calculated by computing the mean of these various payoffs (expectation value) such as $a = \mathbb{E}[f(X)]$ by using the Quantum Amplitude Estimation algorithm.

The payoff function f can be written as : $f(i) = f_1 i + f_0$ where i is an index referring to each n qubit i.e. $[0, ..., 2^{n-1}]$. We perform linearly controlled Y-rotations on the target ancilla qubits using different powers of the operator U. The U operator can be seen below:

$$U = \begin{pmatrix} \cos(2\theta) & \sin(2\theta) \\ -\sin(2\theta) & \cos(2\theta) \end{pmatrix} \tag{2.33}$$

The quantum state $|\psi\rangle$ now becomes:

$$|\psi\rangle = U_i|0\rangle = (cos(f(i))|0\rangle + sin(f(i))|1\rangle)|i\rangle \tag{2.34}$$

We now need a way to obtain the expectation value $\mathbb{E}[f(X)]$. This will be done by transforming our register qubits n into (2.34) and also by transforming our function $f(i)$. Our initial state is of the form :

$$|\psi\rangle = \sum_{i=0}^{2^n-1} \sqrt{p_i}|i\rangle_n|0\rangle \tag{2.35}$$

and becomes the following after the transformation:

$$\sum_{i=0}^{2^n-1} \sqrt{p_i}|i\rangle_n[cos(c\tilde{f}(i) + \frac{pi}{4})|0\rangle + sin(c\tilde{f}(i) + \frac{pi}{4})|1\rangle] \tag{2.36}$$

with :

$$\tilde{f}(i) = 2\frac{f(i) - f_{min}}{f_{max} - f_{min}} - 1 \tag{2.37}$$

The function $\tilde{f}(i)$ just rescales the function $f(i)$ into the interval [-1,1] with $c \in [0, 1]$ just being a scaling factor. The f_{min} and f_{max} are just the lowest and largest numbers that store the corresponding payoffs in our distribution e.g. [$|000\rangle$ *and* $|111\rangle$] for n=3 qubits.

Finally, as we said earlier we need to measure only the $|1\rangle$ qubit that stores our options payoffs. The measurement has probability:

$$P_1 = \sum_{i=0}^{2^n-1} p_i sin^2(c\tilde{f}(i) + \frac{pi}{4})$$

that is approximated for small c into :

$$P_1 = \sum_{i=0}^{2^n-1} p_i(c\tilde{f}(i) + \frac{1}{2}) = c\frac{2\mathbb{E}[f(X)] - f_{min}}{f_{max} - f_{min}} - c - \frac{1}{2} \tag{2.38}$$

All we need to do is solve for $\mathbb{E}[f(X)]$ to find the option's premium. Something worth noting here is that the operator-gates U which are control gates distinguish the price paths that make profit (strike price $K \geq$ spot price S for a call option) by flipping the target qubit to $|1\rangle$ from $|0\rangle$ and thus encoding the price paths into the payoff function f . The payoff function f is simply equation (2.25) for a call option where S(t) is denoted by i in the circuit.

Chapter 3

VQE for portfolio optimization

3.1 Introduction

In this chapter the basic programming steps for optimizing a portfolio on a classical computer will be introduced (using python and the IBMQ [5] platform).
The same procedure will be followed using the quantum optimization model VQE [16], [12].
The VQE model takes the classical parameters μ (2.14), b (2.19) , Σ (2.16) and uses equation (2.21) to map them in a Hamiltonian whose ground state corresponds to the optimal solution.

The procedure is slightly different from the classical Markowitz model. The VQE model encompasses simplifications of the Markowitz model in order to work efficiently. In the Markowitz model the vector $\vec{w}$ contains the weights of each stock after optimization in the following manner:

$$\vec{w} = [w_1, w_2, .., w_i]$$

with each w_i having a different value in the [0,1] interval and the sum of all weights must equal 1. The full budget **b** (2.19) has to be spent which means the model has to select exactly **b** assets out of **n**.

Furthermore all stocks selected will have the same weights w_i which makes the problem much easier to solve. For example if the model selects $w_x = [0, 1, 0, 1, 1, 1]$ as the optimal portfolio with binary selection of 1=' selected' and 0='not selected' then all 4 stocks selected will have a weight of $w_i = 0.25$ which means that if we invest 100 $ in this portfolio, 25$ will be invested in each stock.

3.2 Classical approach

3.2.1 Preparing the data

In this section it will be shown how to calculate the mean return μ and covariance matrix Σ of a given portfolio $\mathbf{P}$ with 6 assets by means of a simple program (programmed in python). These two parameters (along with the budget $\mathbf{b}$) are used to calculate the optimal selection w_i (3.1,2.1) using a **classical optimization model** [**2.20**]. Coding samples are available in Appendix C. Here I will introduce the most important parts of the algorithm.

Firstly, it's essential to find stock market data in order to create a portfolio. The dataset includes closing stock prices from 2010 to 2017 for 12 stocks. The dataset was downloaded (as csv) and imported in Jupyter Notebook using a Pandas DataFrame. Six assets were selected, Apple, Microsoft, Intel, Amazon, S&P500 , Gold.
The reason for the selection of a 6-stock portfolio and not of a 12-one is simple. If we increase the number of assets we also increase the computational complexity both for the classical and the quantum algorithm, the main concern being the time it takes to run the portfolio optimization problem and the time it needs to converge (e.g. 12×12 covariance matrix).

1258 trading days were used to calculate daily returns for each asset and their mean daily returns. The daily return is just the percentage difference (or the log-difference) between the closing price of day n to day n-1. By calculating the mean return of each asset for these 1258 trading days we get the mean daily return vector $\vec{\mu}$.

For the covariance matrix we generate a matrix of (1258,6) that contains the daily returns of each asset and then subtract from each asset its mean daily return, creating the excess return matrix R_e. Then by calculating the dot product $R_e \cdot R_e^T$ we get the covariance matrix Σ (6,6).

Now that we have our parameters estimated it's time to use them in the classical optimization model. Relevant code is available in Appendix B.

3.2.2 Applying the algorithm

While the previous code was executed on a regular notebook in my local Jupyter Notebok host, the following will be executed on IBMQ. The reason is the availability of packages that are necessary to run the models.

First of all, we have to encode the 3 values μ, Σ, **b** along with the risk level factor q and a penalty term n (parameter to scale the budget penalty term, equals to the number of assets) in a portfolio function that will be later used to create the quantum circuit (it can also be used for the classical part). That is done by calling the method **get_operator** (mu, sigma, q, budget, penalty) on the **portfolio** module (all necessary packages and modules are imported at the top of the notebook) .

Relevant code (snippet):

```
mu = [ μ1, μ2, ..., μi ]
sigma = Σ
q = 0.5
budget = 3
penalty = 6
qubitOp = portfolio.get_operator(mu, sigma, q, budget, penalty)
```

The risk level factor q is a hyperparameter of our problem.Like in section (2.1) different values of q give different value of optimal portfolio selections. It will be set as q = 0.5 for the most part of this chapter.

Next we need to call the ExactEigensolver classical model in order to optimize our portfolio classically by inversing matrix (2.20) of section [2.1]. Full code is presented in Appendix C.

```
exact_eigensolver = ExactEigensolver(qubitOp, k=1)
result = exact_eigensolver.run()
print_result(result)
```

3.2.3 Results

The classical optimization algorithm returns the vector : [0. ,1., 0., 1,. 1., 0.] as the optimal selection [Microsft, Amazon, S&P 500]. The computation takes (0.1secs), something to be expected due to a low number of parameters.

If we run the classical algorithm for a different budget e.g. b=2 or 4, we get the following results :

b=4 $\quad w_i = [\ 1\ ,\ 1\ ,\ 0\ ,\ 1\ ,\ 1\ ,\ 0\ ]$

b=2 $\quad w_i = [\ 0\ ,\ 1\ ,\ 0\ ,\ 1\ ,\ 0\ ,\ 0\ ]$.

3.3 Quantum approach

3.3.1 Applying the algorithm

It is now time to apply the quantum model to our data. First of all we have to select a backend. A backend is the device the quantum circuit will run. We select a backend from available providers (quantum computers or simulators available at that time).

```
4
5 from qiskit import IBMQ
6 provider = IBMQ.load_account()
7 provider = IBMQ.get_provider(group='open')
8 provider.backends() #shows available providers
9 provider = provider.get_backend('ibmq_16_melbourne')
10 #selects a 16-qubit quantum computer to run the circuit
```

We will select the ' **statevector simulator** ' as a backend because it is the simplest backend to run on. It is a simulator that runs the quantum algorithm with no noise and without decoherence.

Furthermore, we have to set up the classical optimizer that will update the θ_i parameters (weights) at each iteration. For the statevector simulator, IBMQ suggests the Constrained optimization by linear approximation (COBYLA) optimizer. Each portfolio optimization problem (different q, number of assets) needs different number of iterations in order to converge (find the global minimum-lowest eigenvalue). For example, a 4-stock portfolio will need less iterations to converge than a 6-stock portfolio due to the different number of qubits that are used in the quantum circuit.

```
1 """
2 backend = BasicAer.get_backend('statevector_simulator')
3 from qiskit.aqua import aqua_globals
4 seed = 50 #random_seed
5 aqua_globals.random_seed = seed
6 cobyla = COBYLA()#optimizer
7 cobyla.set_options(maxiter=3500,disp=True) #we set the classical
     iterations of the optimizer
8 """
```

After the classical optimizer setup we construct the quantum circuit and set the **depth** parameter which will determine how many times the basic quantum algorithm will be iterated in the circuit. Like COBYLA iterations, the depth parameter needs to be adjusted accordingly. We know the threshhold of these two parameters when the solution of the optimization problem doesn't change each time we run it (optimal portfolio selection stays the same even if we re-run the code).

```
"""
ry = RY(qubitOp.num_qubits, depth=11, entanglement='full')#circui#
    depth = how many times it will loop the basic circuit
vqe = VQE(qubitOp, ry, cobyla) #completed model
vqe.random_seed = seed
quantum_instance = QuantumInstance(backend=backend,seed_simulator=
    seed)
result = vqe.run(quantum_instance)#runs the algorithm
"""
```

Finally we concatenate the classical optimizer **COBYLA**, the quantum circuit **RY** and the encoded portfolio's parameters **qubitOp** together by using function **VQE** . We set the **quantum_instance**, which is the device the algorithm will be executed (backend) along with its random seed and run the final model.

3.3.2 Results

The quantum optimization algorithm returns vector [1., 0., 0., 1., 1., 0.] as the optimal selection [Apple, Amazon, S&P 500] with a probability of 22.75%.It seems to be pretty unsure about the selection but we will keep that for now and compare with the classical algorithm. The computation takes around 5 minutes to run on the simulator.

If we run the quantum algorithm for a different budget(2,4) we get the following results:

b=4 w_i = [1, 1, 0, 0, 1, 1] p = 39.39%
b=2 w_i = [1, 0, 0, 0, 0, 1] p = 34.32%

In order to see the impact of these portfolio selections it is necessary to see the daily portfolio value some months in the future for both methods and compare the results for three different budgets.

As mentioned in section [3.2.1] 1258 trading days were used to calculate the optimal portfolio selection. We will thus see how the portfolio performs 210 days after the selection.

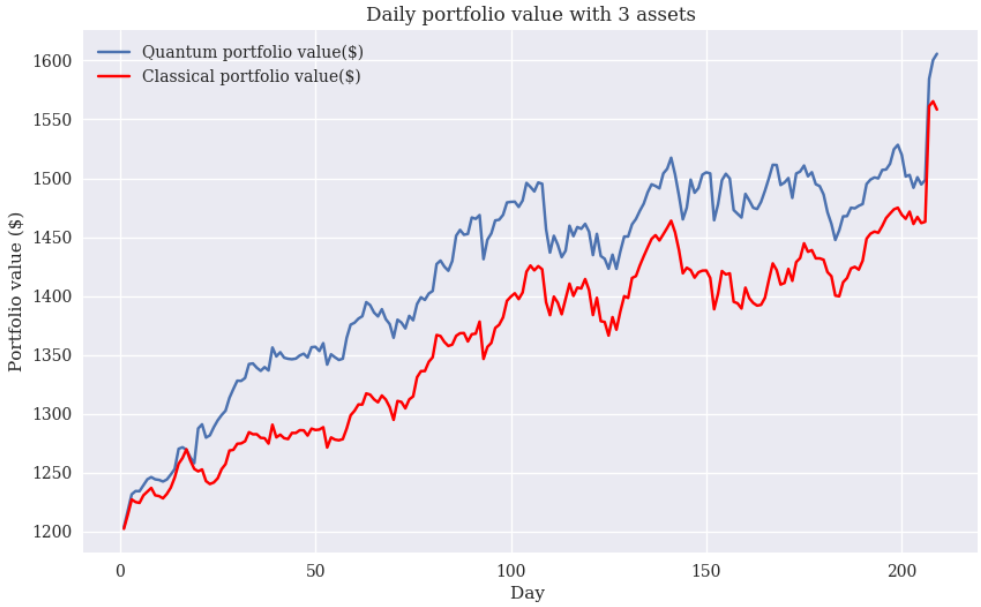

Figure 3.1: Daily value of quantum and classical portfolios with 3 assets

As we can see the quantum algorithm seems to have done a better selection of our portfolio here. Assuming an investor invests 1200 $ in 3 stocks (400$ to each one) at day 1, in 210 days their portfolio value will have increased to 1605.69$ with a total profit of 405.69$.

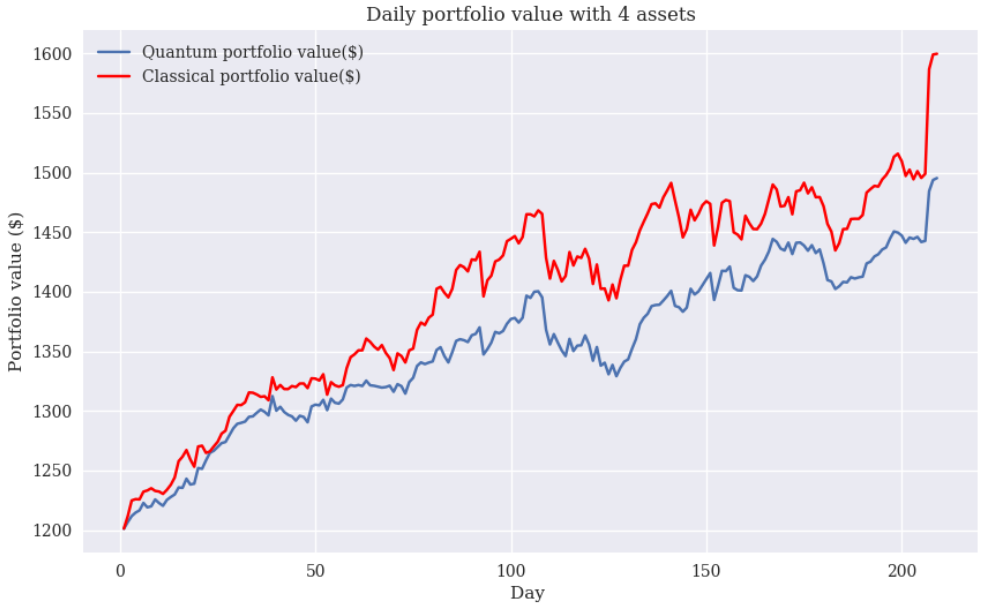

Figure 3.2: Daily value of quantum and classical portfolios with 4 assets

For the second scenario were 4 assets are selected the classical algorithm prevails. Here if the investor invests in 4 stocks with the same capital they will make 399.6$ in 210 days, slightly less than the first scenario.

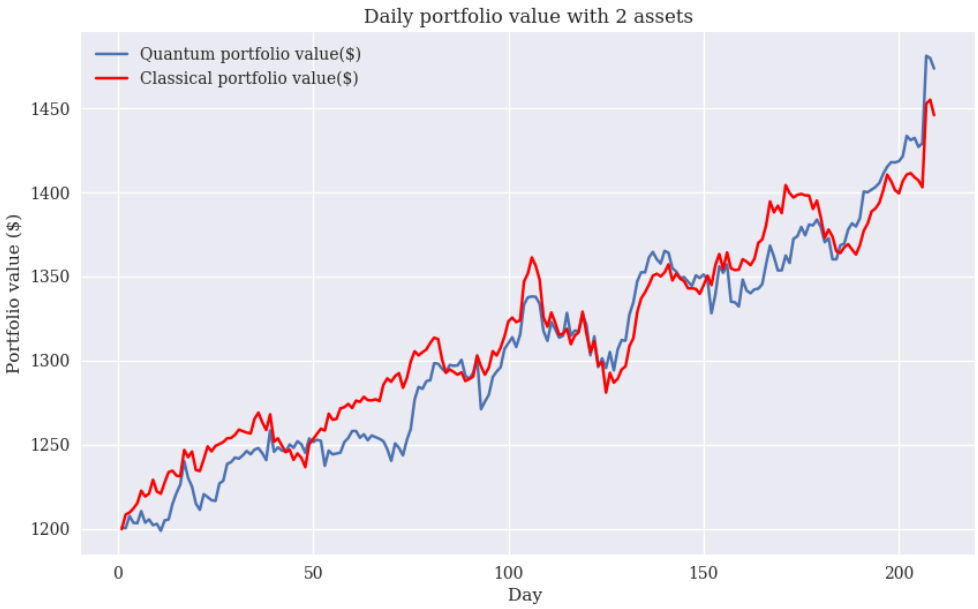

Figure 3.3: Daily value of quantum and classical portfolios with 2 assets

For the third scenario were 2 assets are selected out of 6 possible ones the quantum and classical algorithm seem to be very closely in terms of price movement. Both algorithms selected two different portfolios that follow a close relationship.

Finally the question arises, what if we were risky investors and wanted to pick only one asset out of these 6? Well in that case the profit for each stock after 210 days would be :

Asset	Profit($)	Return(%)
Apple	546.43	45.53
Microsoft	395.01	32.91
Intel	291.47	24.28
Amazon	559.83	46.65
S&P 500	168.71	14.05
Gold	116.37	9.69

Table 3.1: Profit and return per asset for a 1-stock portfolio

As we can see we would have made more profit than scenario 1 and 2 if we have invested all of our capital in Apple and Amazon, a total of 2 out of 6 cases. We should keep in mind that this portfolio contains top companies/assets that perform well in the stock market. For a random portfolio things would be different. That's why it is important to spread the risk and invest in multiple assets based on the VQE and the classical model that were analysed above.

Chapter 4

QAE for options pricing

4.1 Introduction

In this chapter we will see how a quantum model comes up against a classical Monte Carlo method for pricing a call option. The asset that we will investigate is the stock of Apple. The dataset will be the same as the one in chapter 3 [24]. The parameters we need for both models are the initial spot price S_0, the volatility of the asset σ, the time to maturity t, the risk-free return and finally the strike price K.

The initial spot price S_0 of the asset as well as the strike price K were $S_0 = 115.82\,\$$ and $K = 113\,\$$. Apple was trading at that price in 30-12-2016. The risk-free return r, also called risk-free interest rate was r = 0.025 at that time. It was derived from the interest rates of US Treasure Bonds at the end of 2016 (who are subject to minimal risk). The volatility of Apple was initially calculated by computing the standard deviation of its returns on a 4-year period from 2012, something that proved to be very conservative because Apple was much more volatile during the 2015-2016 period. That's why an implied volatility was chosen . Implied volatility is the volatility based on recent prices (adding a stronger weight on recent prices) and was set to be $\sigma = 0.1474$ which was much more representative for the short term period we were looking for . The time to maturity was 2 months and it is used in the model by dividing this value with 12 months.

First of all, we will use a classical Monte Carlo method to price the option and then a classical Black-Scholes method. Then we will use a quantum model that uses Quantum Amplitude Estimation (QAE) to price the option. The quantum model can be considered as a sparse censor Black-Scholes model with few ground truths. Finally, we will compare the classical and quantum models based on their convergence rate (number of bits or qubits needed to reach a solution). Just like in Chapter 3 all code will be executed in Jupyter Notebook and the quantum models will run on IBMQ using qiskit.

4.2 Option pricing using classical models

4.2.1 Monte carlo simulation

In section (2.4.2) we saw how we could use the concept of Brownian motion to model stock prices. The basic idea was to run a number of simulations on the asset, each one of them following a different random path (random walk) that depends on the asset's underlying volatility and time to maturity. The code for this and the next section is available at Appendix D. Here a short number of simulations will be run in order to be able to compare with the quantum algorithm (that has limitations on qubits).

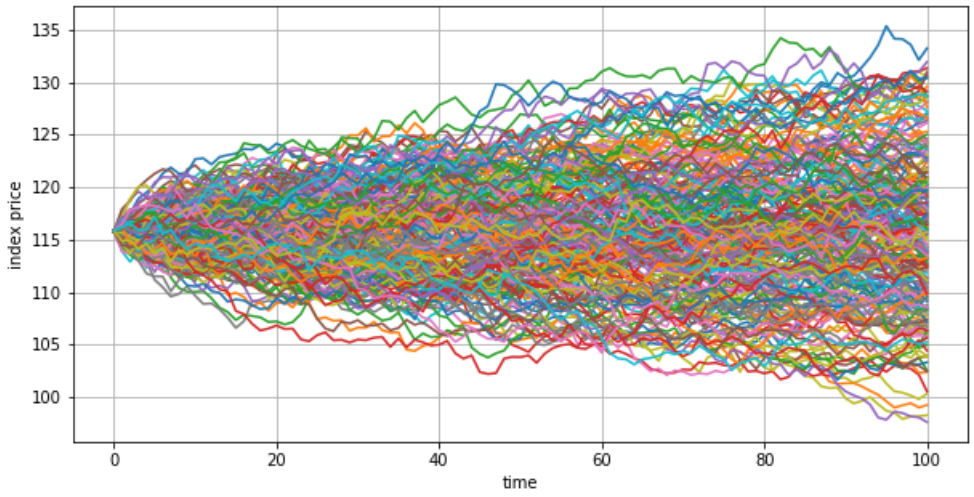

Figure 4.1: Monte carlo simulation for 100 time steps and 220 simulations

We want to keep the last price of each simulation and then use equation (2.25) or (2.26) depending on the type of option. After computing the payoff (if there is any) for the last price of each simulation we compute the mean payoff which is the option's price.

The price of the option after running 220 simulations is :

$$call = 4.545\,\$$$

$$put = 1.493\,\$$$

The total time spent to run this simulation was 0.35 secs and the amount of register bits needed for the simulation were at least 220 due to the 220 different numbers needed to be stored in a classical computer.

4.2.2 Black-Scholes model

Now we will use the Black-Scholes model to calculate the option's value. Here the computation is much more simple. There is no need for multiple simulations in order to price the option. We use equations (2.30) and (2.31) along with equation (2.29) to price the contract. The option's price according to the Black-Scholes model is :

$$call = 4.669\,\$$$

$$put = 1.387\,\$$$

The model needs only 0.0039 secs to run. If we increase the number of simulations in the Monte Carlo method e.g. 10^6 then the prices of the options will converge to the Black-Scholes prices. It will be though more computationally expensive i.e. at least 2 minutes to run the simulations.

Apple Option	Monte Carlo method	Black-Scholes	Percentage Π(%)
Call option price($)	4.545	4.669	2.66
Put option price($)	1.493	1.387	7.64

Table 4.1: Option's price per method

4.3 Option pricing using a quantum model

In this section we will show how to use a quantum method to price options. We will be using IBMQ as always. The model will run on a quantum simulator just like in Chapter 3 . Here though we will be using a different simulator which does have some noise, called qasm simulator. All relevant code can be found at Appendix E.

First of all, we have to create a log-normal random distribution based on the Black-Scholes model and load the different numbers of this distribution in our circuit. In order for our model to run smoothly we will be using a small number of register qubits, only 3. These 3 qubits encode $2^3 = 8$ different numbers. A visual representation of the distribution can be seen bellow:

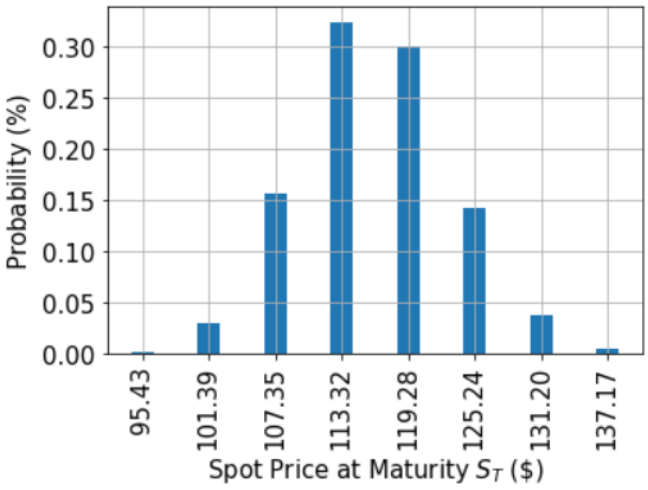

Figure 4.2: Price distribution of Apple at maturity based on its volatility and other parameters loaded in a three-qubit register. The price 95.4 $ is loaded in the $|000\rangle$ qubit and 137.16 $ in the last qubit $|111\rangle$.

As you can see the prices follows a log-normal random distribution with limits $[-3\sigma, 3\sigma]$ where σ is the standard deviation of returns or volatility. The analytical equation that models this distribution is :

$$P(S_T) = \frac{1}{S_T \sigma \sqrt{2\pi T}} e^{-\frac{(ln(S_T)-\mu)^2}{2\sigma^2 T}} \tag{4.1}$$

After we have loaded the distribution to the circuit it's now time to build our payoff function f using equations (2.25), (2.26) depending on the type of option contract.

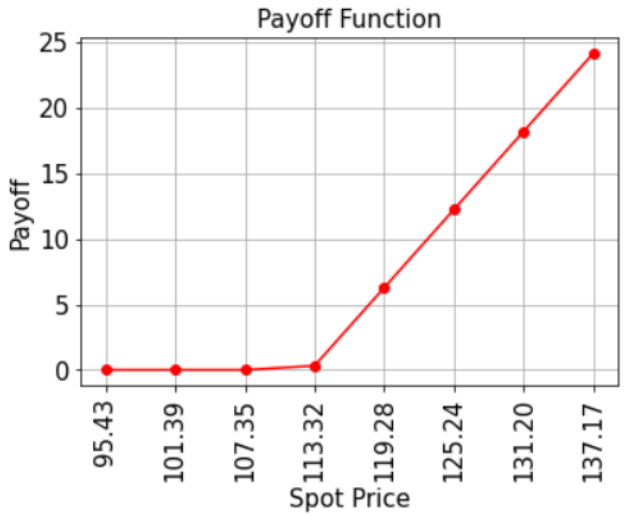

Figure 4.3

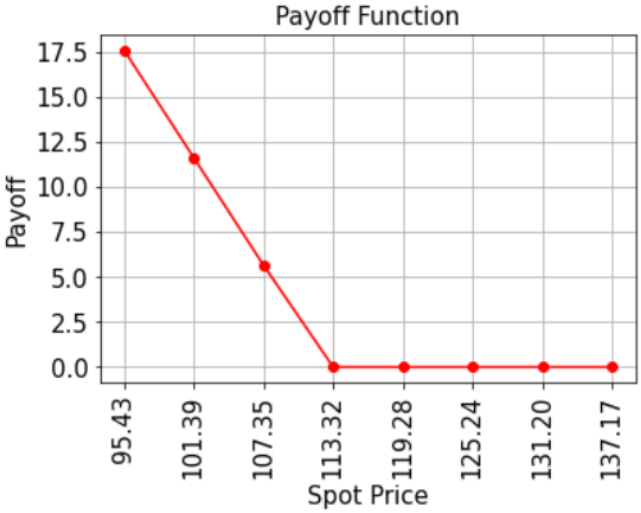

Figure 4.4

Figure 4.5: Payoff function of our call and put option

Next we are going to compute the classical option price based on these 8 different prices by computing a dot product of the corresponding state probability (Figure 4.2) and its payoff (for positive payoffs only). We will be calling this method "Black-Scholes weighted average".

The Black-Scholes weighted average method is a sparse censor Black-Scholes with few ground truths. By simulating only a few prices (samples) at maturity we can approximate the value of the option. It is the classical equivalent of the quantum model and is used to compare the two methods for convergence. We will call the quantum model "Quantum Black-Scholes weighted average (w.a.)".

The corresponding classical values with this method are:

$$call = 4.5353\,\$$$

$$put = 1.2684\,\$$$

Now it's time to run the quantum algorithm. We will be using m = 7 evaluating qubits for the call option and m= 9 for the put option. In total, 10 qubits for the call and 12 for the put option model.

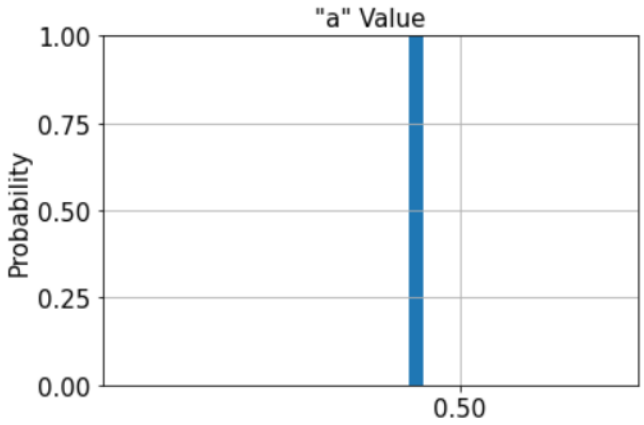

Figure 4.6: Amplitude value "a" for a call option(see equation(2.32))

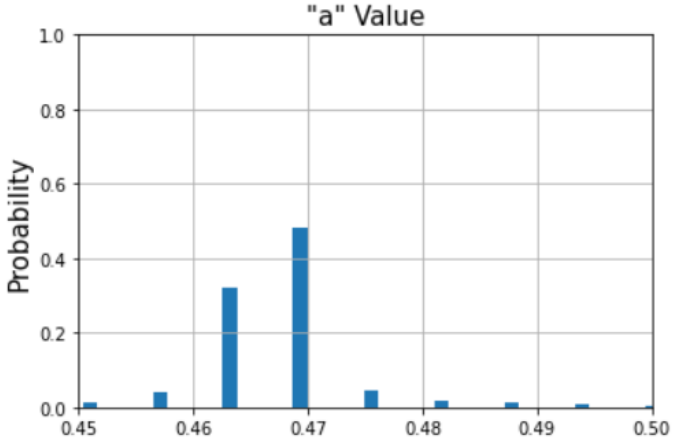

Figure 4.7: Amplitude value "a" for a put option

As you can see the call option agrees with the classical result with 100 % certainty while the put option doesn't, even with 9 evaluation qubits. The more evaluation qubits we use the better the accuracy of the model. For example, if we used less than 7 evaluation qubits on the first model, the call option one, then the output value would have many probabilistic values just like the put option one. The red dashed line is the classical value we talked about earlier (weighted average) while the blue ones are the values the quantum algorithm computes with their corresponding probability.

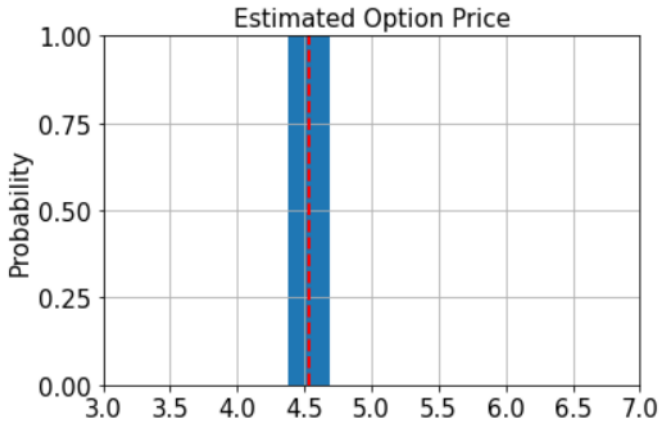

Figure 4.8: Call option price

Figure 4.9: Put option price

The results for both options are:

$$call = 4.5342\,\$$$

$$put = 1.9263\,\$\,(48\%\,probability)$$

Finally, comparisons of results obtained by the different methods are presented in the following table

Option's method	Call option price ($)	Put option price ($)	Runtime (secs)
Monte Carlo	4.545	1.493	0.34
Black-Scholes	4.669	1.387	0.0039
Black-Scholes weighted avg.	4.5353	1.2684	0.00255
Quantum Black-Scholes w.a.	4.5342	1.9263	2.5

Table 4.2: Option's pricing results using different methods

Chapter 5

Conclusions

The purpose of this thesis is to apply certain quantum algorithms on computational applications in finance. In chapter 2, we presented how quantum algorithms work and how it is theoretically possible to achieve speed-up over classical algorithms. We also described the two quantum algorithms (VQE , QAE) that we applied to financial applications.

In chapter 3, we applied the VQE algorithm to the portfolio optimization problem and in chapter 4 we applied the QAE algorithm in the options pricing problem. Due to limited resources offered by the IMBQ platform (up to 16 qubits) we restricted the size of our problems to a portfolio consisting of maximum 6 assets.It should be noted that the quantum algorithm is in no way faster than the classical one. Quantum computers usually solve problems faster than classical computers when the problem's underlying nature is quantum e.g. for molecule energy spectrum. Here the underlying nature of the problem is solving a complex linear system with a Hamiltonian that is computed based on stock market indicators (mean return, covariance matrix) .

Even though the quantum algorithm doesn't manage to surpass the classical in speed it manages to surpass it in the selection of the optimal portfolio.

We investigated how it's possible to use VQE for portfolio optimization. We concluded that it's possible to use a hybrid quantum-classical algorithm for optimization problems. Even though the computation is longer and the algorithm needs more time to find the optimal selection, the number of qubits used are far less than those classical computers use.

In chapter 4, we found that the quantum algorithm for options pricing is capable of converging with the classical one with only a few number of qubits. Specifically 10 qubits were needed for the call option and 12+ for the put option. So the main conclusion we can draw here is that quantum algorithms do indeed need a lot less number of bits in order to converge. In section [4.2] we found that at least 220 bits were needed in order to register the different values of the Monte Carlo simulation and the result is almost identical to the quantum algorithm. The speed-up in qubits/bits is large.

But what about the runtime? For the problems we investigated (which were of limited size), the classical algorithm takes less time to run. The difference is not a

large one. The main obstacle is the large number of gates the quantum algorithm utilizes.

If we increase the number of register qubits n for the quantum algorithm (QAE) it is possible to further converge with the classical Black-Scholes (2nd method) but it would take more time. A case with 6 register qubits was tested instead of 3 with 9 evaluation qubits. The call and put option price was the same as the one with 3 register qubits and the runtime was 3 minutes and 56 secs. Still, that's possible to happen even in a classical algorithm. Just like in Monte Carlo if we slightly increase the number of simulations then the result may not differ at all. Furthermore, the increase in the number of qubits was not possible due to hardware limitations.

The quantum algorithm converged more efficiently in terms of the number of qubits used than the classical ones. Even though the computation time is a bit longer, the quantum algorithm converges with the Monte Carlo algorithm for a few number of simulations and theoretically even for many simulations if appropriate quantum hardware is available in the future. Comparisons of results obtained by the different methods are presented in the following table :

Option's method	Call option price ($)	Put option price ($)	Runtime (secs)
Monte Carlo	4.545	1.493	0.34
Black-Scholes	4.669	1.387	0.0039
Black-Scholes weighted avg.	4.5353	1.2684	0.00255
Quantum Black-Scholes w.a.	4.5342	1.9263	2.5

Table 5.1: Option's pricing results using different methods

Bibliography

[1] Figure (2.1). URL: https://images.app.goo.gl/KDsHTfQYe7xfSzBy7.

[2] Figure (2.2). URL: https://images.app.goo.gl/fTrYa1uCJgwUdh6t5.

[3] Figure (2.3). URL: https://cs.stackexchange.com/questions/69725/understanding-implemetation-of-the-toffoli-gate-using-other-gates/69775.

[4] Figure (2.6). URL: https://www.researchgate.net/publication/284259345_Quantum_optics_with_artificial_atoms/figures?lo=1.

[5] Ibmq. URL: https://quantum-computing.ibm.com/.

[6] Ibmq definition. URL: https://en.wikipedia.org/wiki/IBM_Q_Experience.

[7] . Investopedia(finance), . URL: https://www.investopedia.com.

[8] Modern portfolio theory. URL: https://en.wikipedia.org/wiki/Modern_portfolio_theory.

[9] Yael Ben-Haim Sergey Bravyi Lauren Capelluto Almudena Carrera Vazquez Jack Ceroni Richard Chen Abraham Asfaw, Luciano Bello. Quantumtextbook, 2018. URL: https://qiskit.org/textbook.

[10] Yael Ben-Haim Sergey Bravyi Lauren Capelluto Almudena Carrera Vazquez Jack Ceroni Richard Chen Abraham Asfaw, Luciano Bello. Simulating molecules using vqe, 2018. URL: https://qiskit.org/textbook/ch-applications/vqe-molecules.html#groundstate.

[11] Dipesh Amin. Monte carlo options pricing, 2016. URL: https://pawsdevelopment.wordpress.com/2016/11/22/monte-carlo-european-vanilla-option-pricing-with-python/.

[12] Panagiotis Kl Barkoutsos, Giacomo Nannicini, Anton Robert, Ivano Tavernelli, and Stefan Woerner. Improving variational quantum optimization using cvar. *arXiv preprint arXiv:1907.04769*, 2019.

[13] Fischer Black and Myron Scholes. The pricing of options and corporate liabilities. *The Journal of Political Economy, Vol. 81, No. 3 (May - Jun.), pp. 637-654*, 1973.

[14] Gilles Brassard, Peter Hoyer, Michele Mosca, and Alain Tapp. Quantum amplitude amplification and estimation. *Contemporary Mathematics*, 305:53–74, 2002.

[15] Patrick J Coles, Stephan Eidenbenz, Scott Pakin, Adetokunbo Adedoyin, John Ambrosiano, Petr Anisimov, William Casper, Gopinath Chennupati, Carleton Coffrin, Hristo Djidjev, et al. Quantum algorithm implementations for beginners. *arXiv preprint arXiv:1804.03719*, 2018.

[16] Harper R Grimsley, Sophia E. Economou, Edwin Barnes, and Nicholas J Mayhall. An adaptive variational algorithm for exact molecular simulations on a quantum computer. *Nature communications*, 10(1):1–9, 2019.

[17] Lov K Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the twenty-eighth annual ACM symposium on Theory of computing*, pages 212–219, 1996.

[18] Jonathan Hui. Grover's algorithm, 2018. URL: `https://medium.com/@jonathan_hui/qc-grovers-algorithm-cd81e61cf248`.

[19] John C. Hull. *Options, Futures, and Other Derivatives (9th Edition)*. Pearson, 2017.

[20] Harry Markowitz. Portfolio selection. *The Journal of Finance, Vol. 7, No. 1. , pp. 77-91.*, 1952.

[21] Patrick Rebentrost and Seth Lloyd. Quantum computational finance: quantum algorithm for portfolio optimization. *arXiv preprint arXiv:1811.03975*, 2018.

[22] Nikitas Stamatopoulos, Daniel J Egger, Yue Sun, Christa Zoufal, Raban Iten, Ning Shen, and Stefan Woerner. Option pricing using quantum computers. *arXiv preprint arXiv:1905.02666*, 2019.

[23] Stefan Woerner and Daniel J Egger. Quantum risk analysis. *npj Quantum Information*, 5(1):1–8, 2019.

[24] Yuxing Yan. *Python for Finance Second Edition*. Packt Publishing Ltd., 2017.

Appendices

Appendix A

Accessing IBMQ

The IBM Q Experience [6] is an online platform that gives users in the general public access to a set of IBM's prototype quantum processors via the Cloud, an online internet forum for discussing quantum computing relevant topics, a set of tutorials on how to program the IBM Q devices, and other educational material about quantum computing. It is an The IBM Q Experience is an online platform that gives users in the general public access to a set of IBM's prototype quantum processors via the Cloud, an online internet forum for discussing quantum computing relevant topics, a set of tutorials on how to program the IBM Q devices, and other educational material about quantum computing. It is an example of cloud based quantum computing.

The steps to access IBMQ are the following :

Step 1: Press the following link and make an account : https://quantum-computing.ibm.com/login.

Step 2 :

This is the central page of IBMQ. By clicking the left bar where the Jupyter Notebook symbol lies you can find all available code IBMQ offers such as quantum turorials, projects etc. By clicking the circuit symbol in the same bar (above JN) you can find the circuit designer where quantum or classical circuits can be drawn. Above the circuit symbol you can find the "results" symbols where all the results of your algorithms will be stored. In the right you can see different public quantum computers available that you can use.

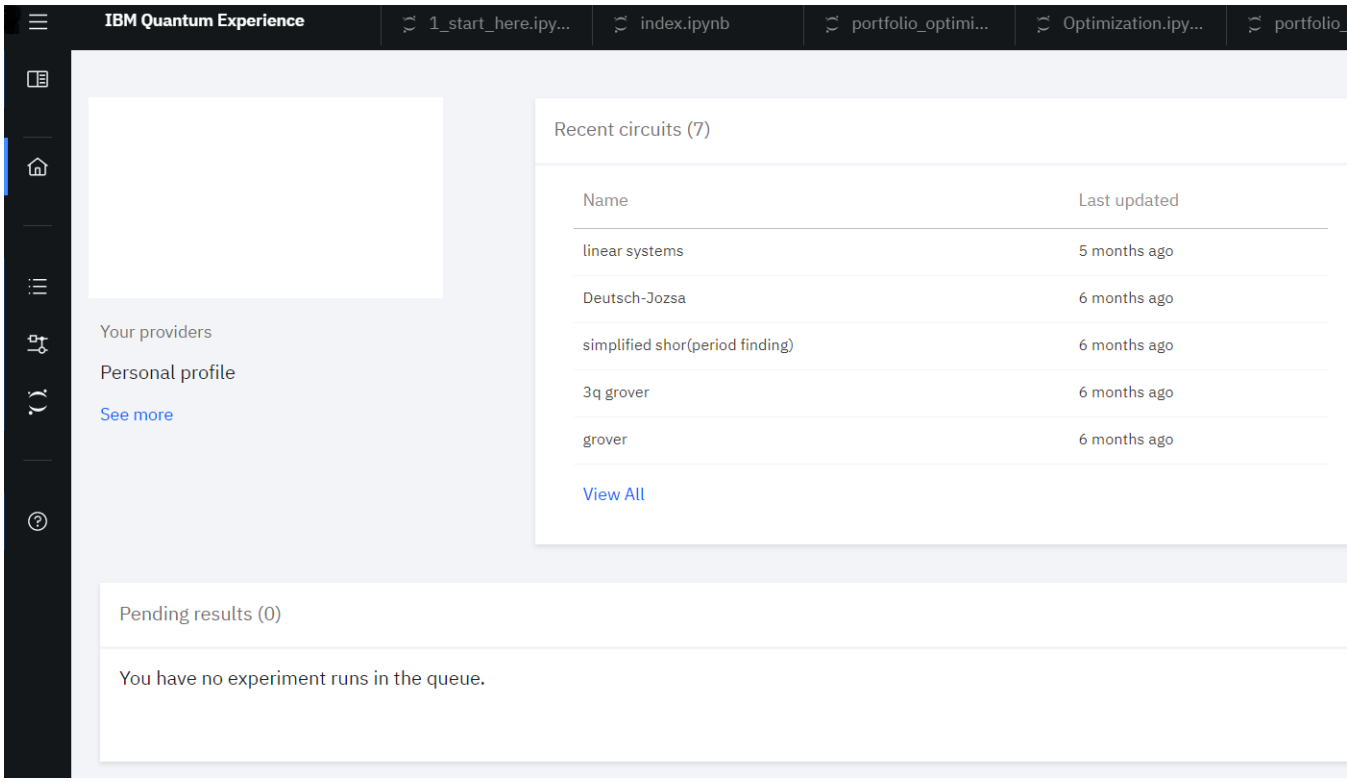

Figure A.1: Central page[IBMQ]

Step 3: Here you can import an existing Notebook to run it on IBMQ or create a new one. You can also open IBMQ's notebooks available in the upper right part of the page.

Step 4: Here you can create your own circuit just like we saw in chapter 2 were some circuits were shown depending on the algorithm we were using.

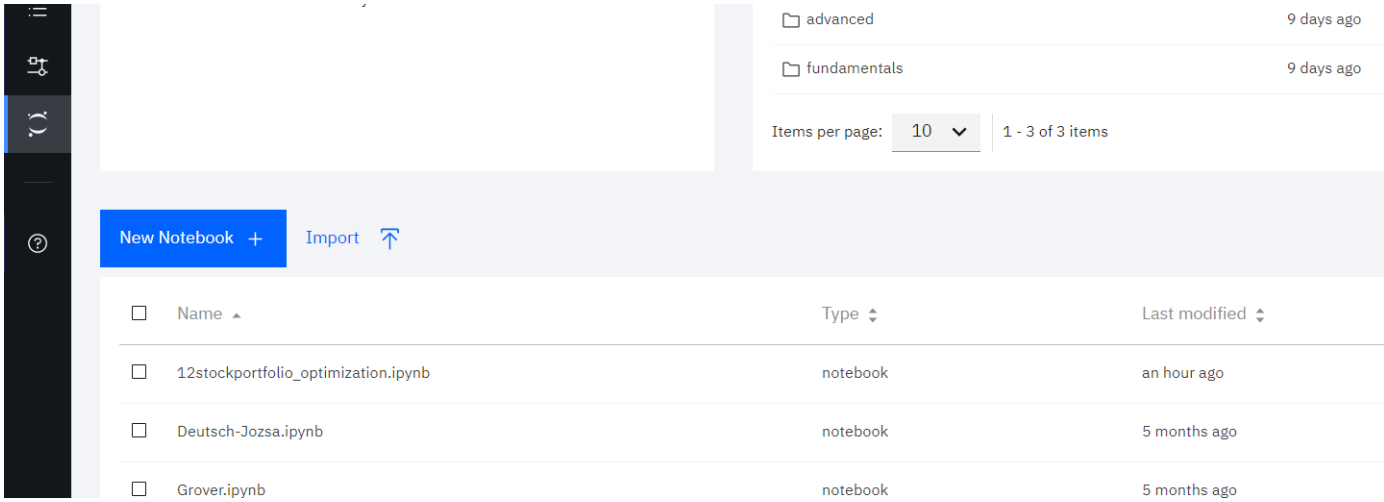

Figure A.2: Code page [IBMQ]

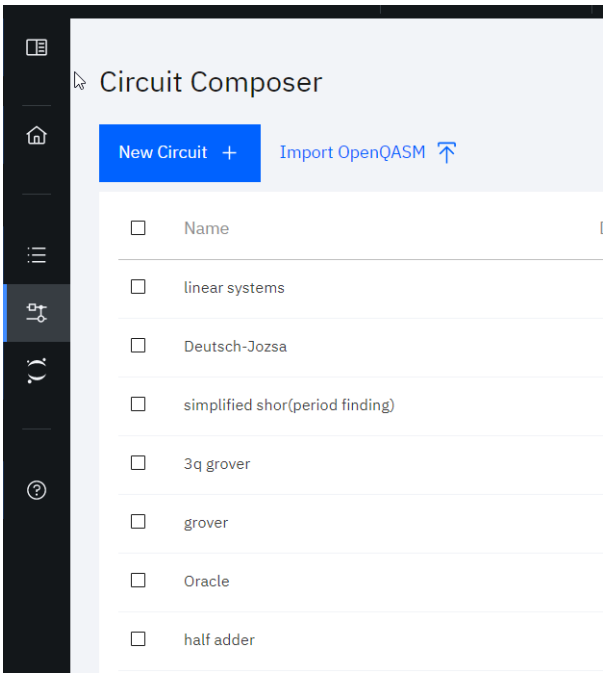

Figure A.3: Circuit page[IBMQ]

Appendix B

Code for preprocessing the data

Code for chapter 3.2.1:

```
#import necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as pl
from pylab import plt
plt.style.use('seaborn')
import matplotlib as mpl
mpl.rcParams['font.family'] = 'serif'
#import data
data = pd.read_csv('tr_eikon_eod_data-Copy1.csv',index_col=0,
    parse_dates=True)
#choose data for the model and data to check the results
datatrain = data.loc["2012-01-01":"2016-12-31"]
datatest =  data.loc["2017-01-01":]
stocksnames = ['Apple Stock', 'Microsoft Stock', 'Intel Stock', '
    Amazon Stock', 'S&P 500 Index','Gold Price']
retcol=['AAPL.Oret','MSFT.Oret','INTC.Oret','AMZN.Oret','SPXret','
    Gold.Nret']
#calculate_returns_function
def calculate_returns(stocks,columns,Returns):
    for i in range(0,len(columns)):
        stocks[Returns[i]] = np.log(stocks[columns[i]]/stocks[
    columns[i]].shift(1))
calculate_returns(datatrain6,datatrain6.columns,retcol)
#select columns that have the returns of each asset
ret6 = datatrain6.iloc[1:,6:]
returns6 = np.array(ret6)
meanreturns6 = np.array(ret6.mean()).reshape(-1,1)
print(meanreturns6) #mu (3.2.1)

def make_covariance_matrix(numberofstocks,stocksreturns,meanreturn)
    :
    meanreturns2 = np.zeros((len(stocksreturns),numberofstocks))
    for i in range(0,numberofstocks-1):
        for j in range(0,len(stocksreturns)-1):
            meanreturns2[j,i] = meanreturn[i]
    npexcessreturns = stocksreturns - meanreturns2
    covarm=np.dot(npexcessreturns.T,npexcessreturns)/(len(
    stocksreturns)-1)
    return covarm
covariancematrix6 = make_covariance_matrix(6,returns6,meanreturns6)
```

Appendix C

Code for VQE portfolio optimization

Code for chapter 3.2.2:

```
1
2  from qiskit import BasicAer
3  from qiskit.aqua import QuantumInstance
4  from qiskit.finance.ising import portfolio
5  from qiskit.optimization.ising.common import sample_most_likely
6  from qiskit.finance.data_providers import RandomDataProvider
7  from qiskit.aqua.algorithms import VQE, QAOA, ExactEigensolver
8  from qiskit.aqua.components.optimizers import COBYLA
9  from qiskit.aqua.components.optimizers import SPSA
10 from qiskit.aqua.components.variational_forms import RY
11 import numpy as np
12 import datetime
13
14 from qiskit import IBMQ
15 provider = IBMQ.load_account()
16 IBMQ.providers()
17 provider = IBMQ.get_provider(group='open')
18 provider.backends()
19
20 num_assets = 6 #how many assets will be optimized
21 All =  ['Apple Stock', 'Microsoft Stock', 'Intel Stock', 'Amazon
       Stock', 'S&P 500 Index','Gold Price']
22 mu = np.array([ 0.00054001, 0.00067008, 0.00031081, 0.0011395 ,
       0.00044661,-0.00026355])
23 #mu= mean returns
24 print(All)
25 print(mu)
26 #sigma = covariance matrix
27 sigma = np.array([ [ 2.71695895e-04,  7.73333928e-05 , 7.34944318e
       -05 , 7.54258143e-05,
28    6.63245938e-05 , 2.32045549e-06],
29  [ 7.73333928e-05 , 2.16089133e-04,  1.06985085e-04 , 9.88414797e
       -05,
30    7.27489419e-05, -2.95452437e-06],
31  [ 7.34944318e-05 , 1.06985085e-04 , 1.98037274e-04 , 7.05554838e
       -05,
32    6.96549409e-05, -2.44650650e-06],
```

```
 [ 7.54258143e-05 , 9.88414797e-05 , 7.05554838e-05,  3.74851607e-04,
   7.78604746e-05, -7.42632050e-06],
 [ 6.63245938e-05 , 7.27489419e-05 , 6.96549409e-05,  7.78604746e-05,
   6.54419127e-05, -1.96820530e-06],
 [ 2.32045549e-06 ,-2.95452437e-06, -2.44650650e-06, -7.42632050e-06,
  -1.96820530e-06 , 1.08235308e-04]])
print(sigma)


def index_to_selection(i, num_assets):
    s = "{0:b}".format(i).rjust(num_assets)
    x = np.array([1 if s[i]=='1' else 0 for i in reversed(range(num_assets))])
    return x
def print_result(result):
    selection = sample_most_likely(result['eigvecs'][0])
    value = portfolio.portfolio_value(selection, mu, sigma, q, budget, penalty)
    print('Optimal: selection {}, value {:.4f}'.format(selection, value))

    probabilities = np.abs(result['eigvecs'][0])**2
    i_sorted = reversed(np.argsort(probabilities))
    print('\n----------------- Full result ---------------------')
    print('selection\tvalue\t\tprobability')
    print('---------------------------------------------------')
    for i in i_sorted:
        x = index_to_selection(i, num_assets)
        value = portfolio.portfolio_value(x, mu, sigma, q, budget, penalty)
        probability = probabilities[i]
        print('%10s\t%.4f\t\t%.4f' %(x, value, probability))


budget = 3
penalty = num_assets

q = 0.5
qubitOp, offset = portfolio.get_operator(mu, sigma, q, budget, penalty)
print("Number of qubits are :",qubitOp.num_qubits)

exact_eigensolver = ExactEigensolver(qubitOp, k=1)
result = exact_eigensolver.run()
print_result(result)

#quantum
backend = BasicAer.get_backend('statevector_simulator')#simulator
seed = 50 #random_seed
cobyla = COBYLA()#optimizer
cobyla.set_options(maxiter=4500,disp=True) #we need as much iterations as possible to reach the minimum,3k optimall
from qiskit.aqua import aqua_globals
aqua_globals.random_seed = seed

```

```
ry = RY(qubitOp.num_qubits, depth=11, entanglement='full')#circuit# optimal depth = 7 # depth = how many times it will loop the basic circuit
vqe = VQE(qubitOp, ry, cobyla)
vqe.random_seed = seed

quantum_instance = QuantumInstance(backend=backend, seed_simulator=seed, seed_transpiler=seed)

result = vqe.run(quantum_instance)

print_result(result)

import qiskit.tools.jupyter
get_ipython().run_line_magic('qiskit_version_table', '')
get_ipython().run_line_magic('qiskit_copyright', '')


budget = 4#try2,3
q = 0.5
qubitOp, offset = portfolio.get_operator(mu, sigma, q, budget, penalty)
print("Number of qubits are :",qubitOp.num_qubits)

exact_eigensolver = ExactEigensolver(qubitOp, k=1)
result = exact_eigensolver.run()
print_result(result)

backend = BasicAer.get_backend('statevector_simulator')# running on a q simulator
#if we change backends we can set the device to a quantum computer
seed = 50 #random_seed
cobyla = COBYLA()#optimizer
cobyla.set_options(maxiter=5000,disp=True)
from qiskit.aqua import aqua_globals

aqua_globals.random_seed = seed
ry = RY(qubitOp.num_qubits, depth=11, entanglement='full')#circuit# optimal depth = 7 # depth = how many times it will loop the basic circuit
vqe = VQE(qubitOp, ry, cobyla) #oloklirwmeno montelo
vqe.random_seed = seed
quantum_instance = QuantumInstance(backend=backend, seed_simulator=seed, seed_transpiler=seed)
result = vqe.run(quantum_instance)
print_result(result

budget = 2
q = 0.5
qubitOp, offset = portfolio.get_operator(mu, sigma, q, budget, penalty)
print("Number of qubits are :",qubitOp.num_qubits)


# In[16]:


exact_eigensolver = ExactEigensolver(qubitOp, k=1)
```

```
result = exact_eigensolver.run()
print_result(result)


backend = BasicAer.get_backend('statevector_simulator')
seed = 50 #random_seed
cobyla = COBYLA()#optimizer
cobyla.set_options(maxiter=3500,disp=True)
from qiskit.aqua import aqua_globals

aqua_globals.random_seed = seed

ry = RY(qubitOp.num_qubits, depth=13, entanglement='full')#circuit# optimal depth = 7 # depth = how many times
#it will loop the basic circuit
vqe = VQE(qubitOp, ry, cobyla) #oloklirwmeno montelo
vqe.random_seed = seed

quantum_instance = QuantumInstance(backend=backend, seed_simulator=seed, seed_transpiler=seed)
result = vqe.run(quantum_instance)#[0,1,0,1,0,0] = classical
#quantum [1,1,0,0,0,0]
print_result(result)

from qiskit import IBMQ
provider = IBMQ.load_account()
IBMQ.providers()

#run on real quantum hardware
qprovider = provider.get_backend('ibmq_16_melbourne')#here we use a 16-qbit quantum computer

budget = 3#try2,4
q = 0.5
qubitOp, offset = portfolio.get_operator(mu, sigma, q, budget, penalty)
print("Number of qubits are :",qubitOp.num_qubits)

from qiskit.aqua.components.optimizers import SPSA, SLSQP
from qiskit.aqua.operators import Z2Symmetries
from qiskit import  Aer
from qiskit.aqua.components.variational_forms import RYRZ
from qiskit.aqua import aqua_globals
seed = 50
aqua_globals.random_seed = seed
backend = qprovider
optimizer = SPSA(max_trials=500)
quantum_instance2 = QuantumInstance(backend=backend, seed_simulator=seed, seed_transpiler=seed,skip_qobj_validation=False,shots=2000)

var_form = RYRZ(qubitOp.num_qubits, depth=7, entanglement="full") # or linear
vqe2 = VQE(qubitOp, var_form, optimizer=optimizer)
result2 = vqe2.run(quantum_instance2)
print_result(result2)
```

Appendix D

Code for classical options pricing models

Code for chapter 4.2.1,4.2.2:

```
#Monte Carlo method
import scipy as sp
import matplotlib.pyplot as plt
import numpy as np
# input area
stock_price_today = 115.82 # stock price at time zero
T =0.164 # maturity date (in years)
n_steps=100 # number of steps
#mu =0.15 # expected annual return
sigma = 0.1474 # annualized volatility
sp.random.seed(12345) # fixed our seed
n_simulation = 220# 2^10 so 10 bits needed # number of simulations
X = 113 # strike price
dt =T/n_steps
r = 0.025

import time
sp.random.seed(12345)
call = sp.zeros([n_simulation], dtype=float)
put = sp.zeros([n_simulation], dtype=float)
#call = []
t0 = time.time()
x = range(0, int(n_steps), 1)
for j in range(0, n_simulation):
    sT=S0
    for i in x[:-1]:
        e=sp.random.normal()
        sT*=sp.exp((r-0.5*sigma*sigma)*dt+sigma*e*np.sqrt(dt))
        call[j]=max(sT-X,0)
        put[j] = max(X-sT,0)
        #print(max(sT-X,0))
        #call.append(max(sT-x,0))

#

call_price=sp.mean(call)*sp.exp(-r*T)
put_price = sp.mean(put)*sp.exp(-r*T)
```

```
print('call price = ', round(call_price,3))
print('put price = ', round(put_price,3))
t1 = time.time()
print('Time Taken: ' + str(t1-t0) + 's')
#visual representation
I = 220
M = 100
dt = T / M
S = np.zeros((M + 1, I))
S[0] = S0
for t in range(1, M + 1):
    S[t] = S[t - 1] * np.exp((r - 0.5 * sigma ** 2) * dt
            + sigma * np.sqrt(dt) * np.random.standard_normal(I))
plt.figure(figsize=(10,5))
plt.plot(S[:,:], lw=1.5)
#plt.title("Monte carlo simulation for 100 time steps and 220
   simulations")
plt.xlabel('time')
plt.ylabel('index price')
plt.grid(True)
plt.savefig('local path')
plt.show()

#Black-Scholes method

import numpy as np
import scipy.stats as si
import sympy as sy
from sympy.stats import Normal, cdf
def euro_vanilla_call(S, K, T, r, sigma):

    #S: spot price
    #K: strike price
    #T: time to maturity
    #r: interest rate
    #sigma: volatility of underlying asset

    d1 = (np.log(S / K) + (r + 0.5 * sigma ** 2) * T) / (sigma * np
   .sqrt(T))
    d2 = (np.log(S / K) + (r - 0.5 * sigma ** 2) * T) / (sigma * np
   .sqrt(T))

    call = (S * si.norm.cdf(d1, 0.0, 1.0) - K * np.exp(-r * T) * si
   .norm.cdf(d2, 0.0, 1.0))
    put = (-S * si.norm.cdf(-d1, 0.0, 1.0) + K * np.exp(-r * T) *
   si.norm.cdf(-d2, 0.0, 1.0))
    return call,put
import time
t0 = time.time()
call = euro_vanilla_call(115.82,113,0.164,0.025,0.1474)[0]
put = euro_vanilla_call(115.82,113,0.164,0.025,0.1474)[1]
t1 = time.time()
print("Time taken:",t1-t0,"secs")
print("Option price:",call,"$")
print("Option price:",put,"$")
```

Appendix E

Code for QAE options pricing

Code for chapter 4.2.3:

```
import matplotlib.pyplot as plt
get_ipython().run_line_magic('matplotlib', 'inline')
import numpy as np

from qiskit import BasicAer
from qiskit.aqua.algorithms import AmplitudeEstimation
from qiskit.aqua.components.uncertainty_models import
    LogNormalDistribution
from qiskit.aqua.components.uncertainty_problems import
    UnivariateProblem
from qiskit.aqua.components.uncertainty_problems import
    UnivariatePiecewiseLinearObjective as PwlObjective


num_uncertainty_qubits = 3#was 3

# parameters for considered random distribution
Sapple = 115.82 # initial spot price (30-12-16)
volapple = 0.1474 # volatility (daily of 1-year) #historical
    volatility is preservative(only 0.014)(so we use implied
    volatility)
rapple = 0.025 # annual interest rate of 2.5%
Tapple = 60 / 365 # 60 days to maturity #peripou 50 trading days
T = 90
# resulting parameters for log-normal distribution
muapple = ((rapple - 0.5 * volapple**2) * Tapple + np.log(Sapple))
    # black-scholes distribution model
sigmaapple = volapple * np.sqrt(Tapple) #volatility per 60 days
meanapple = np.exp(muapple + sigmaapple**2/2)
varianceapple = (np.exp(sigmaapple**2) - 1) * np.exp(2*muapple +
    sigmaapple**2)
stddevapple = np.sqrt(varianceapple)

# lowest and highest value considered for the spot price; in
    between, an equidistant discretization is considered.
lowA  = np.maximum(0, meanapple - 3*stddevapple)#normalize se
    kanoniki katanomi
highA = meanapple + 3*stddevapple

```

```
# construct circuit factory for uncertainty model
uncertainty_modelA = LogNormalDistribution(num_uncertainty_qubits,
    mu=muapple, sigma=sigmaapple, low=lowA, high=highA)


x1 = uncertainty_modelA.values
y1 = uncertainty_modelA.probabilities
plt.bar(x1, y1, width=2)
plt.xticks(x1, size=15, rotation=90)
plt.yticks(size=15)
plt.grid()
plt.xlabel('Spot Price at Maturity $S_T$ (\$)', size=15)
plt.ylabel('Probability ($\%$)', size=15)
plt.savefig('local path')
plt.show()


# set the strike price (should be within the low and the high value
    of the uncertainty)
strike_price = 113

# set the approximation scaling for the payoff function
c_approx = 0.05#0.25
breakpoints = [uncertainty_modelA.low, strike_price]
slopes = [0, 1]
offsets = [0, 0]
f_min = 0
f_max = uncertainty_modelA.high - strike_price
european_call_objective = PwlObjective(
    uncertainty_modelA.num_target_qubits,
    uncertainty_modelA.low,
    uncertainty_modelA.high,
    breakpoints,
    slopes,
    offsets,
    f_min,
    f_max,
    c_approx
)
# construct circuit factory for payoff function
european_call = UnivariateProblem(
    uncertainty_modelA,
    european_call_objective
)


x2 = uncertainty_modelA.values
y2 = np.maximum(0, x2 - strike_price)
plt.plot(x2, y2, 'ro-')
plt.grid()
plt.title('Payoff Function', size=15)
plt.xlabel('Spot Price', size=15)
plt.ylabel('Payoff', size=15)
plt.xticks(x2, size=15, rotation=90)
plt.yticks(size=15)
plt.savefig('local path')
plt.show()

```

```
87
88
89  exact_value_montecarlo = np.exp(-rapple*Tapple)*np.dot(
        uncertainty_modelA.probabilities, y2)
90  exact_delta_montecarlo = sum(uncertainty_modelA.probabilities[x2 >=
        strike_price])
91  print('exact expected value by monte carlo for premium :\t%.4f' %
        exact_value_montecarlo,'$')
92  print('exact delta value by monte carlo:   \t%.4f' %
        exact_delta_montecarlo)#delta is the percent change of the
        option price
93  #if the stock price increases
94  #4.57 is value with 1024 values
95  #4.6242 with 6 register qubits
96
97
98
99  # set number of evaluation qubits (=log(samples))#mallon sqrt(
        samples)
100 m = 7 #7 optimal alla pairnei poli xrono
101 # construct amplitude estimation
102 ae = AmplitudeEstimation(m, european_call)
103
104
105
106
107 from qiskit import IBMQ
108 provider = IBMQ.load_account()
109 IBMQ.providers()
110 provider = IBMQ.get_provider(group='open')
111 provider.backends()
112 qprovider = provider.get_backend('ibmq_qasm_simulator')#qprovider =
        provider.get_backend('ibmq_16_melbourne'), if we used a 16qbit
        qcomputer.
113
114
115
116
117 import time
118 t0 = time.time()
119 result = ae.run(quantum_instance=BasicAer.get_backend('
        statevector_simulator'))
120 t1 = time.time()
121 print('Time Taken: ' + str(t1-t0) + 's')
122 #ae.set_backend(qprovider)
123
124
125
126 from qiskit.aqua import QuantumInstance
127 seed = 42
128 quantum_instance = QuantumInstance(backend=qprovider,
        seed_simulator=seed, seed_transpiler=seed,skip_qobj_validation=
        False)
129 t0 = time.time()
130 result = ae.run(quantum_instance=quantum_instance)
131 t1 = time.time()
132 print('Time Taken: ' + str(t1-t0) + 's')
133 #2.6 seconds to run on qasm simulator,the rest time is for
```

```
    validation in IBMQ for 3 register qubits.
#3 minutes 56 secs with 6 register qubits and 7 evaluation,4.5342
    same value! for 6 register qubits.



print('Exact value:    \t%.4f' % exact_value_montecarlo)
print('Estimated value:\t%.4f' % result['estimation'])
print('Probability:    \t%.4f' % result['max_probability'])
#4.6242 for 6 register qubits exact value
#4.5353 for 3 register qubits



#quantum beats classical for few samples(in number of bits/qubits)
    !!



plt.bar(result['values'], result['probabilities'], width=0.05/len(
    result['probabilities']))
plt.xticks([0, 0.25, 0.5, 0.75, 1], size=15)
plt.yticks([0, 0.25, 0.5, 0.75, 1], size=15)
plt.title('"a" Value', size=15)
plt.ylabel('Probability', size=15)
plt.ylim((0,1))
#plt.xlim((0.3,0.6))
#plt.xlim((2,3))
plt.grid()
plt.savefig('local path')
plt.show()
print(result['values'])
# plot estimated values for option price (after re-scaling and
    reversing the c_approx-transformation)
plt.bar(result['mapped_values'], result['probabilities'], width
    =0.3)
plt.plot([exact_value_montecarlo, exact_value_montecarlo], [0,1], '
    r--', linewidth=2)
plt.xticks(size=15)
plt.yticks([0, 0.25, 0.5, 0.75, 1], size=15)
plt.title('Estimated Option Price', size=15)
plt.ylabel('Probability', size=15)
plt.ylim((0,1))
plt.xlim((3,7))
plt.grid()
plt.savefig('local path')
plt.show()



#estimation of put option:
strike_price = 113

# set the approximation scaling for the payoff function
```

```
c_approx = 0.05

# setup piecewise linear objective fcuntion
breakpoints = [uncertainty_modelA.low, strike_price]
slopes = [-1, 0]
offsets = [strike_price - uncertainty_modelA.low, 0]
f_min = 0
f_max = strike_price - uncertainty_modelA.low
european_put_objective = PwlObjective(
    uncertainty_modelA.num_target_qubits,
    uncertainty_modelA.low,
    uncertainty_modelA.high,
    breakpoints,
    slopes,
    offsets,
    f_min,
    f_max,
    c_approx
)

# construct circuit factory for payoff function
european_put = UnivariateProblem(
    uncertainty_modelA,
    european_put_objective)




x = uncertainty_modelA.values
y = np.maximum(0, strike_price - x)
plt.plot(x, y, 'ro-')
plt.grid()
plt.title('Payoff Function', size=15)
plt.xlabel('Spot Price', size=15)
plt.ylabel('Payoff', size=15)
plt.xticks(x, size=15, rotation=90)
plt.yticks(size=15)
plt.savefig('local path')
plt.show()




exact_value_montecarlo = np.exp(-rapple*Tapple)*np.dot(
    uncertainty_modelA.probabilities, y)#trekse kai monte carlo
    allou na sigrineis to "exact" value
exact_delta_montecarlo = sum(uncertainty_modelA.probabilities[x <=
    strike_price])
print('exact expected value by monte carlo for premium :\t%.4f' %
    exact_value_montecarlo,'$')
print('exact delta value by monte carlo:   \t%.4f' %
    exact_delta_montecarlo)




m = 9

```

```
# construct amplitude estimation
ae = AmplitudeEstimation(m, european_put)




import time
t0 = time.time()
result = ae.run(quantum_instance=BasicAer.get_backend('
    statevector_simulator'))
t1 = time.time()
print('Time Taken: ' + str(t1-t0) + 's')




#qasm simulator
t0 = time.time()
result = ae.run(quantum_instance=quantum_instance)
t1 = time.time()
print('Time Taken: ' + str(t1-t0) + 's')




print('Exact value:    \t%.4f' % exact_value_montecarlo)
print('Estimated value:\t%.4f' % result['estimation'])
print('Probability:    \t%.4f' % result['max_probability'])
#3.2970 with statevector
#1.2684 classical
# 3.2970  with qasm,2.9 secs to run




plt.bar(result['values'], result['probabilities'], width=0.05/len(
    result['probabilities']))
#plt.xticks([0, 0.25, 0.5, 0.75, 1], size=15)
#plt.yticks([0, 0.25, 0.5, 0.75, 1], size=15)
plt.title('"a" Value', size=15)
plt.ylabel('Probability', size=15)
plt.ylim((0,1))
plt.xlim((0.2,0.6))
plt.xlim((0.45,0.5))
plt.savefig('local path')
plt.grid()
plt.show()

# plot estimated values for option price (after re-scaling and
    reversing the c_approx-transformation)
plt.bar(result['mapped_values'], result['probabilities'], width
    =0.2)
plt.plot([exact_value_montecarlo, exact_value_montecarlo], [0,1], '
    r--', linewidth=2)
plt.xticks(size=15)
plt.yticks([0, 0.25, 0.5, 0.75, 1], size=15)
plt.title('Estimated Option Price', size=15)
plt.ylabel('Probability', size=15)
```

```
plt.ylim((0,1))
plt.xlim((0,7))
plt.grid()
plt.savefig('local path')
plt.show()

#run on actual quantum device


from qiskit import IBMQ
provider = IBMQ.load_account()
IBMQ.providers()
provider = IBMQ.get_provider(group='open')
provider.backends()



#result2 = ae.run(quantum_instance=provider.get_backend('
    ibmq_16_melbourne'))

m = 3

# construct amplitude estimation
ae = AmplitudeEstimation(m, european_call)
ae.construct_circuit(measurement=True)
#run on real quantum hardware.Takes too long for even 3 evaluation
    qubits..
#backend2 = provider.get_backend('ibmq_16_melbourne')
#quantuminstance = quantum_instance(backend2, shots=4096,
    skip_qobj_validation=False)
#result2 = ae.run(backend2)
#or
#result2 = ae.run(quantum_instance=backend2)
```