Technical University Of Crete

Quantum Algorithms for solving Linear Systems of Equations and Implementation in Prototype Quantum Computers

by

Alexandros Myron Politis

Supervisor: **Prof.D. Ellinas**
Commitee 1st : **Assoc.Prof.D G. Angelakis**
Commitee 2nd : **Assoc.Prof G.Chalkiadakis**

October 2021

Abstract

In this thesis we study quantum algorithms for solving linear systems of equations which is a problem that arises both on each own and as a part of larger problems in academic and industry. The first quantum linear solver that we study is the HHL algorithm[1]. HHL was invented by A. W. Harrow, A. Hassidim and S. Lloyd in 2008 and is an exact quantum algorithm that uses the quantum phase estimation as a subroutine for eigenvalue finding. HHL scales tremendously well with respect to number of variables and size of equations, but is qubit demanding and noise sensitive. The proper implementation of this algorithm requires many and fully functional qubits. Neither of these prerequisites is satisfied in today's prototype quantum computers and for this reason, the attention has been recently shifted towards variational hybrid-quantum classical algorithms that combine prototype quantum hardware with classical optimizers. In the second part of this thesis we analyze the variational quantum linear solver. After studying in detail the inner workings of both algorithms, we experiment with various test equation sets of different size gap, condition numbers and sparsity, running both perfect and noisy simulations as well as real hardware. The result of these experiments, that were ran in the IBMQ devices on the cloud, is that VQLS and HHL can produce the same result in regards to fidelity on perfect simulators, but VQLS outperformed HHL while working on a realistic noisy setting, as it is the case for current day prototype quantum processors. We conclude by discussing briefly possible applications of the algorithms presented in this work in other areas including machine learning.

Acknowledgments

First of all, I would like to express my warm thanks to my supervising professor, Prof Dimitrios Angelakis, for the trust he showed me during the assignment of this dissertation, as well as for the excellent cooperation. I would also like to thank the PhD candidate, Kalogerakis Michalis, for his valuable and selfless help. Still, I would like to thank my friends for my support and encouragement all these months, contributing in their own special way in completing my dissertation. In closing, I sincerely thank my parents for their moral and unwavering support and love.

Contents

List of Figures

List of Tables

Introduction

Quantum computing is a recent field of research in the computing science which utilizes certain properties of quantum mechanics such as quantum superposition, quantum entanglement and quantum interference. Quantum computing is the merging of quantum mechanics and computer science.

In classical computer science the bit is the fundamental unit of information. In a similar fashion qubit (quantum bit) is the fundamental unit of quantum computing science. Classical computations are performed on bits by logical gates such as 'AND' and 'OR' and quantum computations are performed similarly on quantum gates. However, there are some core differences between bits and qubits and logical and quantum gates which we will make clear through the course my thesis.

Quantum computing was born from the inefficiency of contemporary classical computers to tackle a number of hard computational problems. The last few years we have reached to a stalemate which prohibits the further miniaturization of transistors which are printed on silicon substrates.

The transistor size nowadays is so small and as such it cannot function properly due to the appearance of quantum phenomena. So, it is apparent that in order to achieve the sufficient computational force we need a new perspective namely, quantum computing.

Quantum computers are more prominent than their classical counterparts in certain tasks such as linear system solving and integer factorization because they harness quantum properties such as quantum superposition that allows almost boundless (limited by hardware inefficiencies) parallel computations and exponential speedups.

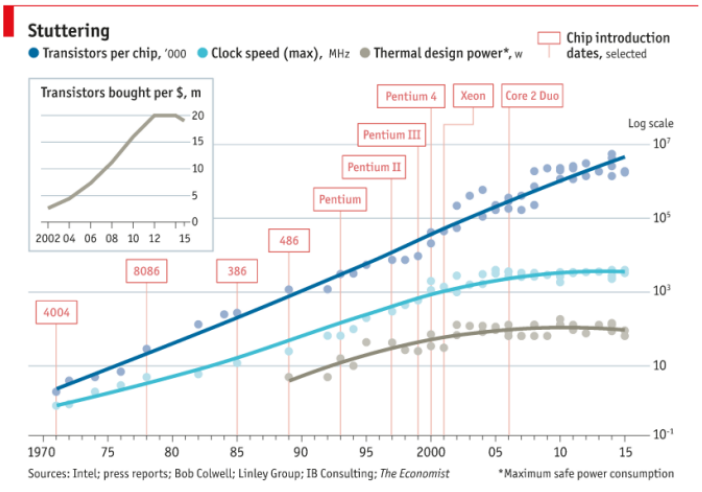


Figure 1: The end of Moore's law

In this thesis we will initially try to outline the fundamentals of quantum computing and subsequently dive into the main subject of my thesis i.e exact and approximate quantum algorithms for linear systems of equations and implementation in quantum hardware. We will introduce two quantum algorithms that solve linear systems of equations. The first quantum algorithm is HHL[1]which is completely quantum and the second is the variational quantum linear solver[2] which is a heuristic quantum-classical hybrid approach. These algorithms offer two fundamentally different perspectives to the quantum linear system problem. We will compare the way they work and also the pros and cons of each approach. Finally we will run a diverse set of simulations on both perfect simulators and noise imbued simulators in order to have a better understanding of how these algorithms work.

Chapter 1

Background and basics of quantum computing

Quantum computing is a relatively recent and fast evolving field of science. Although quantum mechanics was first developed around 1900, quantum computers were initially proposed nearly 80 years later by Paul Benioff who proposed a quantum mechanical model of the Turing machine. Later Richard Feynman and Yuri Manin proposed the universal quantum simulator[3]. Those early quantum simulators promised to solve problems unsolvable by classical computers, mainly chemistry problems at that time. The first quantum algorithm was proposed by David Deutsch and Richard Jozsa [4] in 1992 and in 1994 Peter W. Shor[5] proposed the homonym algorithm, which is a factoring algorithm that could theoretically break modern encryption protocols such as the RSA public-key cryptosystem. Later in 1996 Lov K. Grover proposed a quantum mechanical algorithm for database search[6]. Since 2000 the field has attracted the attention of the scientific community and more and more people from diverse fields of technology and science get involved in quantum computing leading to big quantum breakthroughs. The highlight of the quantum hype was undoubtedly in September of 2019 when Google has reached the so called "quantum supremacy" with an array of 54 qubits (out of which 53 were functional), which were used to perform a series of operations in 200 seconds that would

take a supercomputer about 10,000 years to complete[7].

1.1 Fundamentals of quantum computing

1.1.1 Qubit and quantum states

A bit(binary digit) is the basic unit of classical information and can be in the state of either 0 or 1. It represents a logical state that can take one of two possible values. The bit is most commonly physically represented as electrical voltage. Accordingly, the qubit is the basic unit of quantum information and can be in the state of either $|0\rangle$ or $|1\rangle$. Physically it can be represented as the spin of an electron with possible states spin up, spin down or by the polarization of a photon in which the two possible states can be the vertical and the horizontal polarization of the photon. The quantum states are written according to Dirac's bra-ket notation. Bra is a row vector and ket is a column vector. The main difference between bits and qubits is that bits can be 1 or 0, in contrast qubits can be in one of the two basis states $|0\rangle, |1\rangle$ but also in a superposition of these states. This can be written as a linear combination of the two basis states as follows[8]:

$$|\psi\rangle = a\,|0\rangle + b\,|1\rangle \quad , \quad a\,, b \in \mathbb{C} \ , \quad |a|^2 + |b|^2 = 1$$

a,b complex coefficients are probability amplitudes. When we measure in the standard basis we can find the qubit in the state $|0\rangle$ with probability $|a|^2$ and in the state $|1\rangle$ with probability $|b|^2$. As we all know probabilities sum up to 1.

Another way to represent a quantum state is the following:

$$|0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \text{ so}, |\psi\rangle = a|0\rangle + b|1\rangle = a\begin{bmatrix} 1 \\ 0 \end{bmatrix} + b\begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} a \\ 0 \end{bmatrix} + \begin{bmatrix} 0 \\ b \end{bmatrix} = \begin{bmatrix} a \\ b \end{bmatrix}$$

The complex conjugate of state $|\psi\rangle$ is the following:

$$\langle\psi| = a^* \langle 0| + b^* \langle 1| = a^* \begin{bmatrix} 1 & 0 \end{bmatrix} + b^* \begin{bmatrix} 0 & 1 \end{bmatrix} = \begin{bmatrix} a^* & b^* \end{bmatrix}$$

The inner product of a quantum state is :

$$\langle\psi \mid \psi\rangle = \begin{bmatrix} a^* & b^* \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = a^*a + b^*b = |a|^2 + |b|^2 = 1$$

The outer product of a quantum state is:

$$|\psi\rangle\langle\psi| = \begin{bmatrix} a \\ b \end{bmatrix} \begin{bmatrix} a^* & b^* \end{bmatrix} = \begin{bmatrix} aa^* & ab^* \\ ba^* & bb^* \end{bmatrix}$$

Linear algebra is the mathematical language used to describe quantum systems. A base set of vectors $B = \{b_0, b_1 ... b_{n-1}\}$ in a vector space V is any set of vectors such that any vector U in the span of the set can be written as a linear combination of the base $v = a_0b_0 + a_0b_1 + ...a_{n-1}b_{n-1}$, $a_0, a_1, ..a_{n-1} \in \mathbb{C}$. Now, let us check if we can use $\{|0\rangle, |1\rangle\}$ as a base set.$|0\rangle, |1\rangle$ need to be orthonormal, meaning orthogonal and normalized.

$|0\rangle, |1\rangle$ are orthogonal: $\langle 1 \mid 0\rangle = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 0$, $\langle 0 \mid 1\rangle = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 0$

$$|0\rangle, |1\rangle \text{ are normalized: } \langle 1 \mid 1\rangle = \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 1, \ \langle 0 \mid 0\rangle = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 1$$

The general form of a quantum state is the following:

$$|\psi\rangle = e^{i\delta}\left(\cos\left(\frac{\theta}{2}\right)|0\rangle + e^{i\varphi}\sin\left(\frac{\theta}{2}\right)|1\rangle\right), \delta \in [0, 2\pi), \theta \in [0, \pi], \varphi \in [0, 2\pi)$$

$e^{i\delta}$ is a global phase that can be ignored due to the fact that it has no physically observable consequences.

Another way to think of the abstract until now concept of qubit is provided by the geometrical representation of qubit, that of the Bloch Sphere.

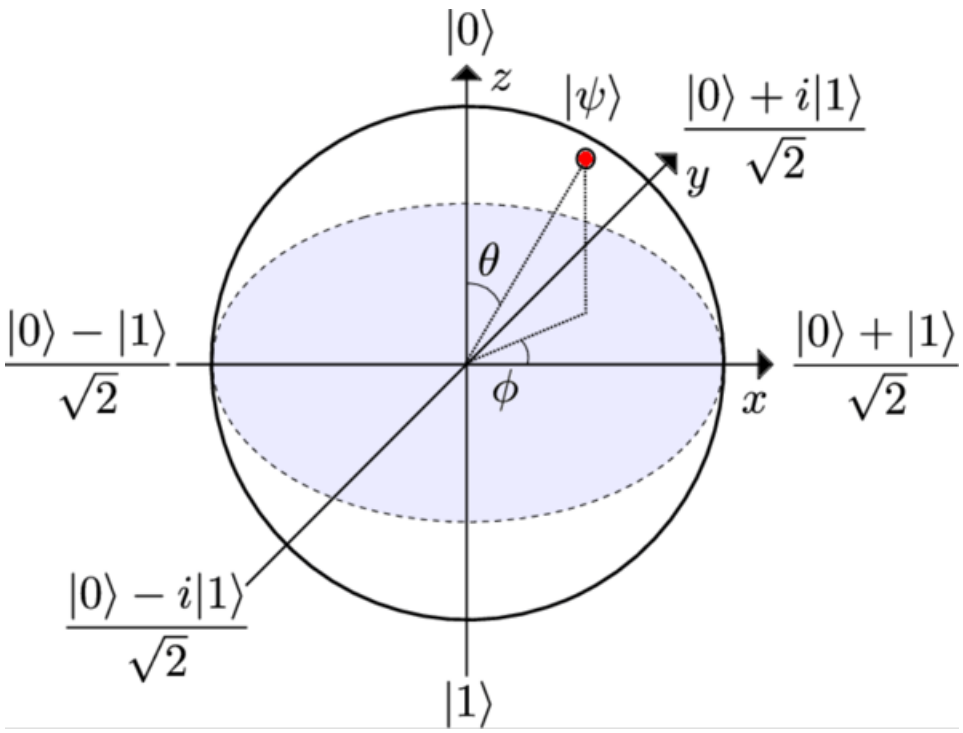


Figure 1.1: Bloch Sphere

1.1.2 Single qubit gates

Now that I have defined the quantum state ,we are able to perform computations using qubits. We can go from one quantum state to another by applying certain operators called quantum gates. Since we have only discussed one qubit state until this point, we will deal only with single qubit gates. Since a single qubit state is a 2x1 column vector, a quantum gate must be 2x2 matrix. Let A be a 2x2 quantum gate then:

$$A = \begin{bmatrix} c & d \\ e & f \end{bmatrix}, |\psi\rangle = a|0\rangle + b|1\rangle \rightarrow A|\psi\rangle = \begin{bmatrix} c & d \\ e & f \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} ac + db \\ ea + fb \end{bmatrix} = \left|\psi'\right\rangle$$

$\left|\psi'\right\rangle$ should remain a quantum state :

$$\left\langle\psi'\middle|\psi'\right\rangle = \begin{bmatrix} (a \cdot c + d \cdot b)^* & (e \cdot a + f \cdot b)^* \end{bmatrix} \begin{bmatrix} a \cdot c + d \cdot b \\ e \cdot a + f \cdot b \end{bmatrix} =$$

$$= (a \cdot c + d \cdot b)^* (a \cdot c + d \cdot b) + (e \cdot a + f \cdot b)^* (e \cdot a + f \cdot b) = |a|^2|c|^2 + |b|^2|d|^2 + |a|^2|e|^2 + |b|^2|f|^2$$

$$= |a|^2 \left(|c|^2 + |e|^2\right) + |b|^2 \left(|d|^2 + |f|^2\right) = 1 = |a|^2 + |b|^2 \Rightarrow |c|^2 + |e|^2 = 1,\ |d|^2 + |f|^2 = 1$$

Operators that obey this condition are the unitary operators. An operator is called unitary if and only if it satisfies the condition: $UU^\dagger = U^\dagger U = I$ where I is the identity matrix (do-nothing matrix) and $U^\dagger$ is the conjugate transpose of U. Another applicable group of gates are the Hermitian operators. Hermitian operators can be turned into unitary operators ($U = e^{iA}$). A operator equals its conjugate transpose. The most commonly known Hermitian operators are the Pauli operators.

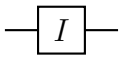
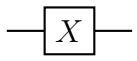
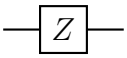
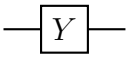
Pauli operators		
Operator	matrix representation	circuit symbol
I = σ_I=ID	$\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$	
X = σ_x=NOT	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	
Z = σ_z	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	
Y = σ_y	$\begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}$	

Table 1.1: Pauli operators

- The Pauli Matrices are involutory, which means that each Pauli matrix squared equals the identity matrix: $I^2 = X^2 = Y^2 = Z^2 = I$
- The Pauli matrices are anti-commute: {X,Z}=0, XZ=-ZX,{X,Y}=0, XY=-YX,{Y,Z}=0, YZ=-ZY

Other notable single qubit gates are the Hadamard gate, the rotation gates and the phase shift gates.

The Hadamard gate is the gate that creates superposition:

$$H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \; H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) = |+\rangle_x \; , \; H|1\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |-\rangle_x$$

Phase shift gates
$P(\pi) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} = \mathrm{Z}$
$P\left(\frac{\pi}{2}\right) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{2}} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix} = \sqrt{Z} = S$
$P\left(\frac{\pi}{4}\right) = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & \frac{1}{\sqrt{2}}(1+i) \end{bmatrix} = \sqrt{S} = T$

Table 1.2: Phase shift gates

Rotation gates
$R_x(\theta) = e^{-iX\frac{\theta}{2}} = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -i\sin\left(\frac{\theta}{2}\right) \\ -i\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix}$
$R_y(\theta) = e^{-iY\frac{\theta}{2}} = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ -\sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{bmatrix}$
$R_z(\theta) = e^{-iZ\frac{\theta}{2}} = \begin{bmatrix} e^{-i\frac{\theta}{2}} & 0 \\ 0 & e^{i\frac{\theta}{2}} \end{bmatrix}$

Table 1.3: Rotation gates

Another set of single qubit quantum gates, are the general U gates. These gates are parameterized gates of the form:

$$U(\theta, \varphi, \lambda) = \begin{bmatrix} \cos\left(\frac{\theta}{2}\right) & -e^{i\lambda}\sin\left(\frac{\theta}{2}\right) \\ e^{i\varphi}\sin\left(\frac{\theta}{2}\right) & e^{i(\varphi+\lambda)}\cos\left(\frac{\theta}{2}\right) \end{bmatrix}$$

By choosing the right parameters we can get any single qubit gate:

$$U\left(\frac{\pi}{2},0,\pi\right)=\begin{bmatrix}\cos\left(\frac{\pi}{4}\right) & -e^{i\pi}\sin\left(\frac{\pi}{4}\right)\\ \sin\left(\frac{\pi}{4}\right) & e^{i\pi}\cos\left(\frac{\pi}{4}\right)\end{bmatrix}=\frac{1}{\sqrt{2}}\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}=H$$

$$U\left(0,0,\pi\right)=\begin{bmatrix}\cos\left(0\right) & -e^{i\pi}\sin\left(0\right)\\ e^{i0}\sin\left(0\right) & e^{i\pi}\cos\left(0\right)\end{bmatrix}=\begin{bmatrix}1 & 0\\ 0 & -1\end{bmatrix}=Z$$

$$U\left(\pi,0,0\right)=\begin{bmatrix}\cos\left(\frac{\pi}{2}\right) & -e^{i0}\sin\left(\frac{\pi}{2}\right)\\ e^{i0}\sin\left(\frac{\pi}{2}\right) & e^{i0}\cos\left(\frac{\pi}{2}\right)\end{bmatrix}=\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}=X$$

$$U\left(0,0,0\right)=\begin{bmatrix}\cos\left(0\right) & -e^{i0}\sin\left(0\right)\\ e^{i0}\sin\left(0\right) & e^{i0}\cos\left(0\right)\end{bmatrix}=\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}=I$$

1.1.3 Multiple qubit state

Now that we have outlined how a single qubit works, it is time to present how multiple qubit states and multiple qubit operators work. In order to design useful quantum algorithms many qubits are needed to encode information and interact with each other for the algorithm to produce the desired results. We will first show what a 2-qubit state looks like and later we will introduce several qubit systems. Earlier we have shown that a qubit can be in one of the two basis states, or in a superposition of these two basis states $|\psi\rangle = a\,|0\rangle + b\,|1\rangle \quad , \quad a\,,\,b \in \mathbb{C} \quad , \quad |a|^2 + |b|^2 = 1$. Now it is essential to introduce the tensor product. The tensor product of each basis state with itself is the following:

$$|0\rangle \otimes |0\rangle = \begin{bmatrix}1\\0\end{bmatrix} \otimes \begin{bmatrix}1\\0\end{bmatrix} = \begin{bmatrix}1\cdot\begin{bmatrix}1\\0\end{bmatrix}\\ 0\cdot\begin{bmatrix}1\\0\end{bmatrix}\end{bmatrix} = \begin{bmatrix}1\\0\\0\\0\end{bmatrix} = |0\rangle\,|0\rangle$$

$$|1\rangle \otimes |1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix} \otimes \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \\ 1 \cdot \begin{bmatrix} 0 \\ 1 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = |1\rangle\,|1\rangle$$

So, let $|\psi_1\rangle, |\psi_1\rangle$ two separate qubits can be described more concisely as a composite system as follows:

$$|\psi_1\rangle = a\,|0\rangle + b\,|1\rangle \quad a, b \in \mathbb{C} \;,\; |a|^2 + |b|^2 = 1$$

$$|\psi_2\rangle = c\,|0\rangle + d\,|1\rangle \quad c, d \in \mathbb{C} \;,\; |c|^2 + |d|^2 = 1$$

$$|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle = a\,|0\rangle + b\,|1\rangle \otimes c\,|0\rangle + d\,|1\rangle = ac\,|00\rangle + ad\,|01\rangle + bc\,|10\rangle + bd\,|11\rangle$$

$$ac, ad, bc, bd \in \mathbb{C} \;,\; |ac|^2 + |ad|^2 + |bc|^2 + |bd|^2 = 1$$

We can alternatively write the 2-qubit system as follows :

$$|\psi\rangle = a\,|00\rangle + b\,|01\rangle + c\,|10\rangle + d\,|11\rangle =$$

$$= a \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} + b \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} + c \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} + d \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix}$$

$$a, b, c, d \in \mathbb{C} \;,\; |a|^2 + |b|^2 + |c|^2 + |d|^2 = 1$$

We added one qubit to our system and the coefficient vector has doubled in size. Generally, if we have an n-qubit state, which denotes a system that consists of n qubits, the coefficient vector will be $N = 2^n$.

1.1.4 Multiple qubit gates

Earlier we acted on the single qubit state using single qubit operators. In the same way we can act on two qubit quantum states using 2-qubit operators. We have constructed the qubit state by applying the tensor product and we will do the same in order to construct a 2-qubit operator. First, we will examine the 2-qubit X gate[9].

$$X \otimes X = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \otimes \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 0 \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & 1 \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \\ 1 \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} & 0 \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

We can construct any 2-qubit version of the Pauli matrices accordingly. However, it is evident that there is no difference between applying the X gate to each qubit separately and applying the 2-qubit X gate to a 2-qubit state. Until this point the qubits do not really interact with each other, they just coexist. Another perspective on quantum operators can be given by the spectral decomposition theorem:

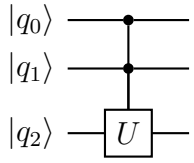
Let operator A with eigenvalues $\{\lambda_1, \lambda_2, ...\lambda_n\}$ and eigenvectors $\{|v_1\rangle, |v_2\rangle, ... |v_n\rangle\}$ can be written as :

$$\hat{A} = \sum_{j=1}^{n} \lambda_i |v_i\rangle \langle v_i|$$

There is a commonly used set of multiple qubit gates, the control-U gates that use one or more qubits as control and the other qubit as target. The control gate changes the target qubit quantum state if the Control qubit is $|1\rangle$ and does nothing to the target qubit if the control qubit is $|0\rangle$.

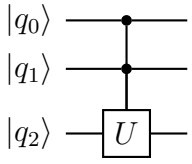
Generally, C-unitary gates have the following form :

$$\text{Let unitary operator U}=\begin{bmatrix} a & b \\ c & d \end{bmatrix}, C-U=\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & c & d \end{bmatrix}$$



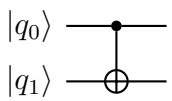
Accordingly, CC-unitary gates have two control qubits and one target qubit, and they have the following form:

$$\text{CC - U} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & a & b \\ 0 & 0 & 0 & 0 & 0 & 0 & c & d \end{bmatrix}$$



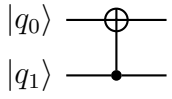
The matrix representation of the control unitary changes according to which qubit is the control and which is the target. For example if we use the first qubit as control and the second as target:

$$\text{C - NOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$



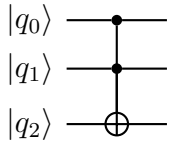
And if we use the first qubit as target and the second as control:

$$\text{C - NOT} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

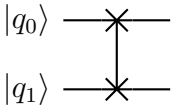


Other notable multiqubit gates are the following:

$$\text{Toffoli=CC-Z=}\begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$



$$\text{SWAP=}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



1.2 Quantum circuit

Quantum circuit is a sequence of quantum gates (reversible transformations) acting on qubits with the intention to perform a desired computation. A quantum circuit consists of qubits (later we will refer to them as quantum registers) which appear like wires that move horizontally in the schematic representation. Qubits are aligned vertically, are enumerated and do not cross each other's route. Quantum gates look like nodes in which the wires go through. Each node has a name that describes its action, and the vertical lines show which qubit acts as the target and which as the control.

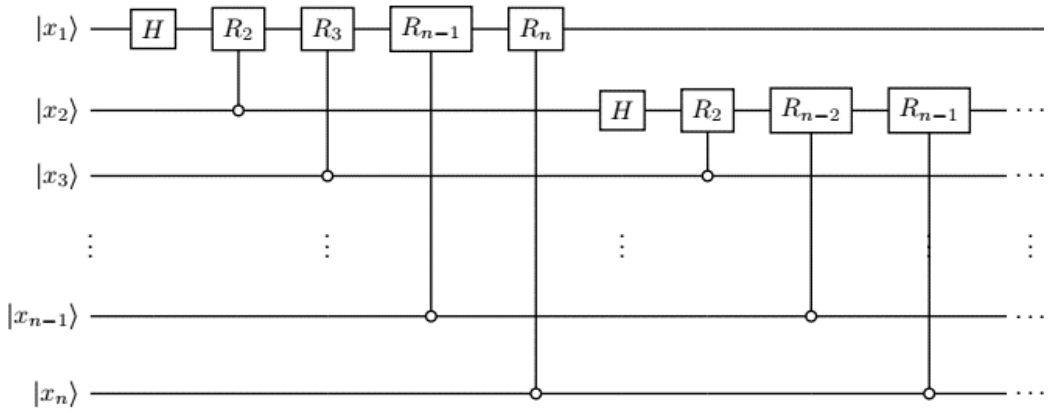


Figure 1.2: Quantum circuit example

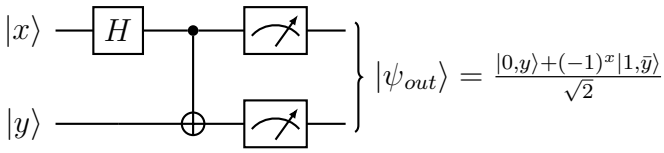
1.2.1 Quantum register

A classical register is the storage used by a classical processor. Accordingly, a quantum register "stores" the value of one or many qubits in order to be used for computations in a quantum processor. A classical register of n size can store just one of the 2^n possible bitstrings on the other hand, the quantum register can store all 2^n possible bitstrings at once. However, there are some things that a classical register can do that the quantum

register can not. For example, if we want to fetch a value from a classical register, we can do it without losing the rest of the stored data. On the other hand, if we make a single measurement on a quantum register, we get one value, and all the other information is gone. We will analyze this uniquely quantum phenomenon in chapter 1.4. Another thing that we can not do with a quantum register is to make a copy of a stored value. This is known as the "No-cloning theorem"[10].

1.3 Quantum entanglement

Now, this is a feature only found in quantum mechanics. There is no classical analogue. Quantum entanglement can be described as the invisible link between two or more particles. These particles remain entangled no matter how far apart is one from the other and any measurement (observation) of any of these particles affects the other particles as well. We can not describe the quantum state of one of these particles without referring to the others. They share a common unified state. Now let us see how we can create an entangled pair (a Bell state or an EPR pair).



$$x = |0\rangle, y = |0\rangle, |\psi_{out}\rangle = \frac{|00\rangle + |11\rangle}{\sqrt{2}} = B_{00} \quad , \quad x = |0\rangle, y = |1\rangle, |\psi_{out}\rangle = \frac{|01\rangle + |10\rangle}{\sqrt{2}} = B_{01}$$
$$x = |1\rangle, y = |0\rangle, |\psi_{out}\rangle = \frac{|00\rangle - |11\rangle}{\sqrt{2}} = B_{10} \quad , \quad x = |1\rangle, y = |1\rangle, |\psi_{out}\rangle = \frac{|01\rangle - |10\rangle}{\sqrt{2}} = B_{11}$$

1.4 Quantum Measurement

Quantum measurement plays a fundamental role in quantum computation because, at some point, we have to be able to get information out of the computational system. Let

us consider a general qubit $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$, $a\,,\, b \in \mathbb{C}$. When a measurement is made, the qubit will be forced into the state $|\psi\rangle \mapsto |0\rangle$ *or* $|\psi\rangle \mapsto |1\rangle$. After measurement, the original state is lost, and the quantum superposition collapses. With a single measurement it is not possible to determine a, b. So, if we want to extract information about the complex coefficients a,b we have to produce the same state and measure it over and over. So, given a quantum state $|\psi\rangle = a\,|0\rangle + b\,|1\rangle$, $a\,,\, b \in \mathbb{C}$ what is the probability of measuring $|0\rangle$ *or* $|1\rangle$? The most common measurement model is the projective or Von Neumann measurement. At this point we must present the projection operator.

Projection operators are Hermitian $\Rightarrow P = P^{\dagger}$

Projection operators are involutory $\Rightarrow P^2 = P$

Projection operators are orthogonal $\Rightarrow \mathrm{P}_0 * \mathrm{P}_1 = \mathrm{P}_1 * \mathrm{P}_0 = 0$

$$P_0 = |0\rangle\langle 0| = \begin{bmatrix} 1 & 0 \\ 0 & 0 \end{bmatrix} \quad , P_1 = |1\rangle\langle 1| = \begin{bmatrix} 0 & 0 \\ 0 & 1 \end{bmatrix} \quad , \sum_{i=1}^{2} P_i = P_0 + P_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = I$$

The probability of measuring $|\psi\rangle$ in the state $|0\rangle$ is $\Pr(0) = \langle\psi|\,P_0\,|\psi\rangle = Tr\left(P_0\,|\psi\rangle\langle\psi|\right)$

The probability of measuring $|\psi\rangle$ in the state $|1\rangle$ is $\Pr(1) = \langle\psi|\,P_1\,|\psi\rangle = Tr\left(P_1\,|\psi\rangle\langle\psi|\right)$

Chapter 2

Basic quantum algorithms and subroutines

The main quantum linear solving algorithm that we are going to analyse in my thesis is the HHL algorithm invented by A. W. Harrow, A. Hassidim and S. Lloyd in 2008[1]. The quantum algorithms that are presented in this chapter are vital parts of the HHL. The quantum phase estimation algorithm lies in the heart of HHL and quantum Fourier transform is a key subroutine for quantum phase estimation.

Figure 2.1: HHL QPE QFT

2.1 Quantum Fourier transform

The quantum Fourier transform (QFT)[11] is the quantum analogue of the classical inverse discrete Fourier transform (IDFT).The Fourier transform (FT) is one of the most useful mathematical tools in modern science and engineering. FT transforms data from the time domain to the frequency domain. It is used, amongst many other things, to:

- remove noise from data
- produce holograms
- compress data
- process digital signals

The FT is especially useful when we process data with underlying periodicity.

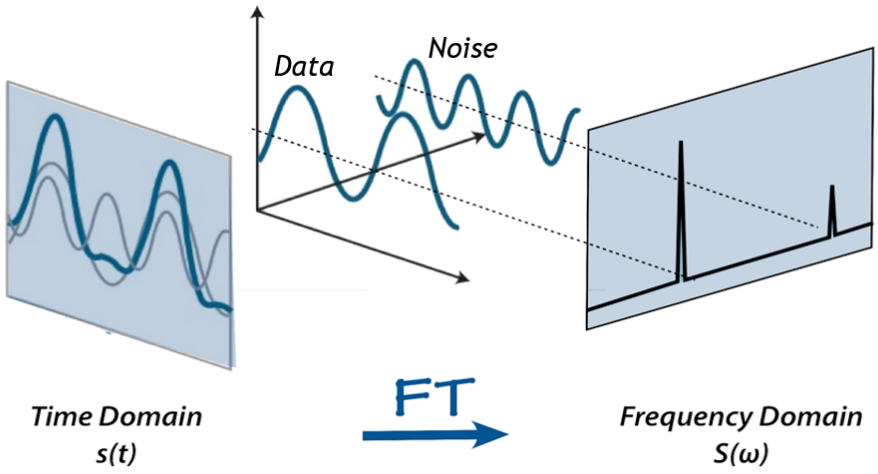


Figure 2.2: Fourier transform

For example, imagine a sinusoid wave with a high-frequency noise:

1. We apply classical Fourier transform to the data in order to obtain the frequency spectrum
2. We discard the high frequency peak (and its mirrored peak!)

3. We apply inverse Fourier transform

4. Finally, we retrieve the original data without the high-pitched noise

First, we must define the discrete version of the Fourier transform, which will form the basis for the quantum algorithm. DFT acts on a sequence of N complex numbers, for instance a column vector $x = (x_1, x_{2,}...x_{N-1})^T$ maps it to a new column vector $y = (y_1, y_{2,}...y_{N-1})^T$ using the following mapping formula:

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{-2\pi i \frac{kj}{N}} = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \left[\cos\left(2\pi i \frac{kj}{N}\right) - i \sin\left(2\pi i \frac{kj}{N}\right) \right]$$

The discrete Fourier transform is an invertible linear transformation. The inverse discrete Fourier transform given the same sequence of complex N numbers has the following form:

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j e^{2\pi i \frac{kj}{N}} = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} x_j \left[\cos\left(2\pi i \frac{kj}{N}\right) + i \sin\left(2\pi i \frac{kj}{N}\right) \right]$$

Fast Fourier transform

As the name implies, the fast Fourier transform (FFT) is an algorithm that determines the discrete Fourier transform of an input significantly faster than computing it directly. The main idea behind the FFT is that of "divide and conquer". In other words, if we split the problem size until we are left with groups of two and then directly compute the discrete Fourier transform for each of these pairs and then merge the results of these discrete DFTs we can get a significant speedup over applying DFT on the original N-size sequence.

However, the speedup depends on the size of the sequence. If N=2^n the FFT provides maximum speedup O($N\log_2 N$). If N is a prime number, then FFT provides no speedup O(N^2) and if N fits neither case FFT provides a significant speedup. There are many different FFT variations such as the Cooley-Tukey algorithm and Bruun's FFT algorithm.

Some FFT algorithms manage to produce results with complexity $O(N^2)$ no matter the size of N.

Quantum Fourier transform

Since quantum states are vectors of complex numbers, we can also apply the DFT on them, only this time we call it quantum Fourier transform(QFT). QFT is the quantum analogue of the inverse discrete Fourier transform.

QFT acts on a quantum state : $|\psi\rangle = \sum_{j=0}^{N-1} a_j |j\rangle$ and transforms it to the quantum state: $|\psi'\rangle = \sum_{k=0}^{N-1} b_k |k\rangle$. Now, the complex coefficients are written on the Fourier basis: $b_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} a_j e^{2\pi i \frac{kj}{N}}$ and $|\psi'\rangle$ has the following form:

$$|\psi'\rangle = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} \sum_{j=0}^{N-1} a_j e^{2\pi i \frac{kj}{N}} |k\rangle$$

QFT operator

We have mentioned in the first chapter that all applicable quantum operators are unitary. Therefore, if we want to apply the QFT on a quantum state we must find a unitary operator $\hat{F}$ that acts on a quantum state and transforms it to its IDFT. By definition a unitary operator satisfies the condition: $UU^\dagger = U^\dagger U = I$. The QFT operator and the inverse QFT operator can be written as follows:

$$\hat{F} = \sum_{j,k=0}^{N-1} \frac{1}{\sqrt{N}} e^{2\pi i \frac{kj}{N}} |k\rangle \langle j| \quad , \hat{F}^\dagger = \sum_{j,k=0}^{N-1} \frac{1}{\sqrt{N}} e^{-2\pi i \frac{jk}{N}} |j\rangle \langle k|$$

And the quantum Fourier transformation as follows:

$$QFT : \hat{F} |\psi\rangle = \sum_{j,k=0}^{N-1} \frac{1}{\sqrt{N}} e^{2\pi i \frac{kj}{N}} |k\rangle \langle j| \sum_{j=0}^{N-1} a_j \ |j\rangle \ = \frac{1}{\sqrt{N}} \sum_{j,k=0}^{N-1} e^{\frac{2\pi i kj}{N}} a_j |k\rangle$$

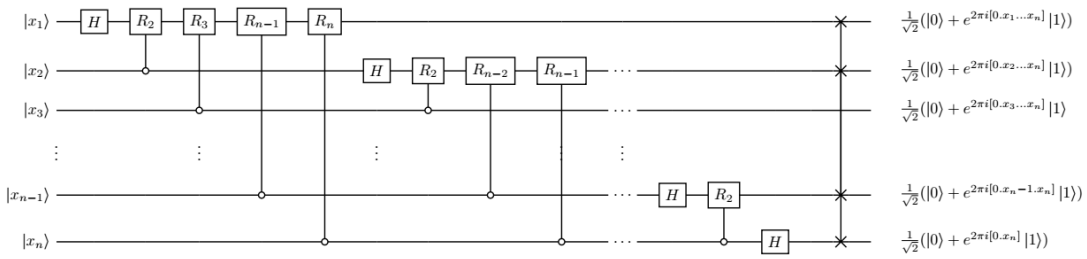


Figure 2.3: General circuit QFT

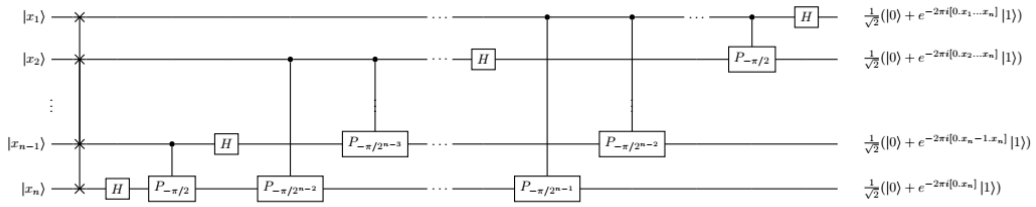


Figure 2.4: General circuit IQFT

Now that we have presented the preliminaries of generalized QFT let us see the QFT in more detail. The 2-qubit QFT is implemented by the following circuit:

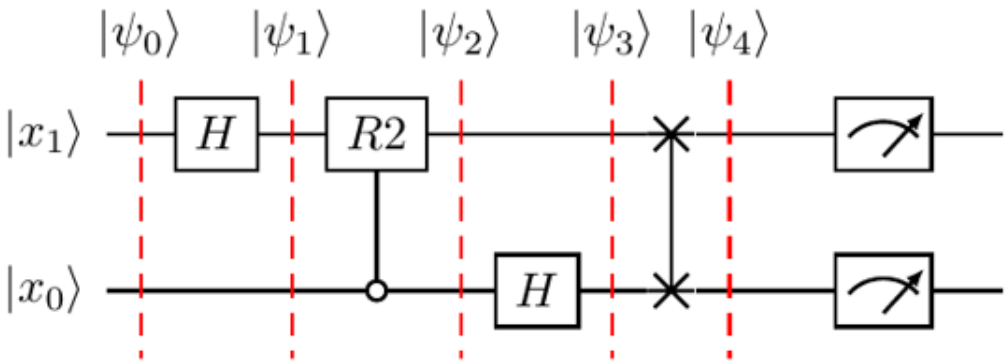


Figure 2.5: 2-qubit QFT

Steps of the QFT

1. $|\psi_0\rangle = |x_1\rangle \otimes |x_0\rangle = |x_1 x_0\rangle$

2. $|\psi_1\rangle = H \otimes I\,|\psi_0\rangle$

3. $|\psi_2\rangle = C - R_2\,|\psi_1\rangle$

4. $|\psi_3\rangle = I \otimes H\,|\psi_2\rangle$

5. $|\psi_4\rangle = SWAP\,|\psi_3\rangle$

QFT consists almost entirely of Hadamard gates and controlled rotation gates. The SWAP gates at the end can be avoided simply by classically rearranging the qubits after measurement. The sequence of gates is fully decomposed below:

- $H \otimes I = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & 1 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \\ 1 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} & -1 \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix}$

- $C - R_n = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{\frac{2\pi i}{2^n}} \end{bmatrix} \Rightarrow C - R_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{\frac{2\pi i}{2^2}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & i \end{bmatrix}$

- $I \otimes H = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \otimes \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} & 0 \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \\ 0 \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} & 1 \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \end{bmatrix} = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix}$

- $SWAP = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

$$hence\ \ \hat{F}_2 = SWAP(I \otimes H)(C - R_2)(H \otimes I) = \frac{1}{2} \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & -i & -1 & i \end{bmatrix}$$

IBM Qiskit

At this stage we are ready to confirm the math behind this algorithm by simulating some proof of principle examples using the IBM platform Qiskit. IBM has built working quantum computers and has also developed a software platform based on the open quantum assembly language (OpenQASM). IBM has made an open source framework for developing and testing quantum algorithms, called the quantum information science kit (Qiskit) and the cloud computing platform IBM Q Experience. Qiskit is a Python package that combines highlevel interface with OpenQASM language. Qiskit lets the user run quantum simulations locally as well as on IBM's backends, quantum simulators and quantum computers[12].

Figure 2.6: IBM quantum computer at New York US

Qiskit simulation 4-qubit QFT

We will show 3 distinct runs to show what the algorithm really does.

Case 1 :$|\psi_{IN}\rangle = |0000\rangle$

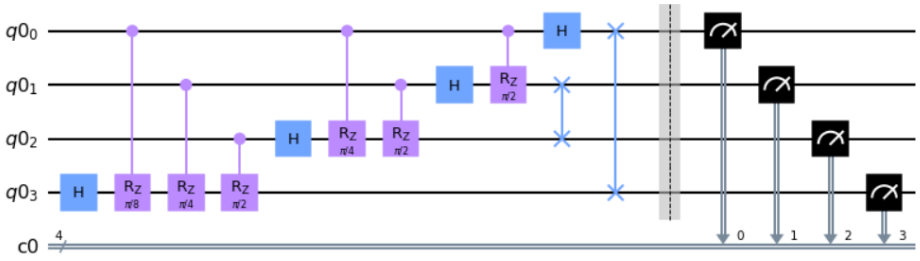


Figure 2.7: QFT circuit $|\psi_{IN}\rangle = |0000\rangle$

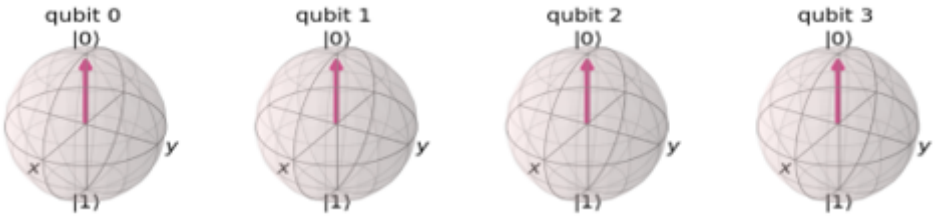


Figure 2.8: $|\psi_{IN}\rangle = |0000\rangle$

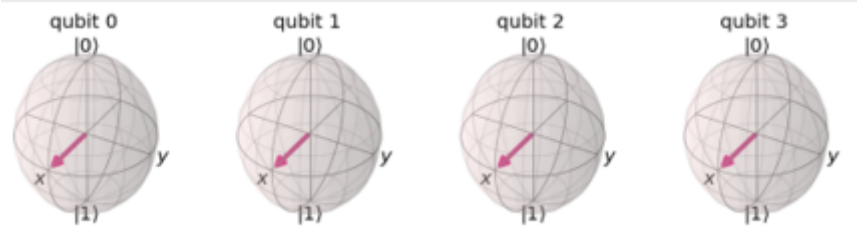


Figure 2.9: $|\psi_{OUT}\rangle = |++++\rangle$

Case 2 :$|\psi_{IN}\rangle = |0100\rangle$

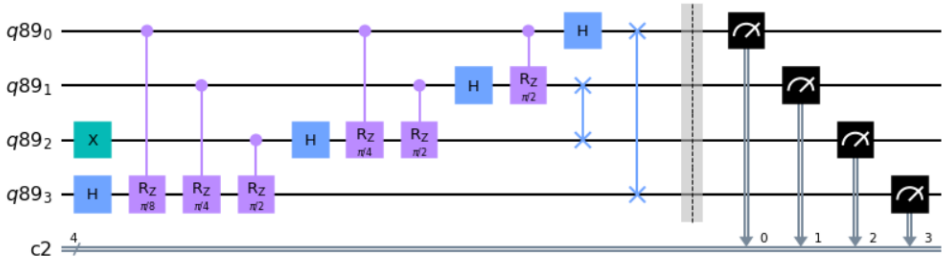


Figure 2.10: QFT circuit $|\psi_{IN}\rangle = |0100\rangle$

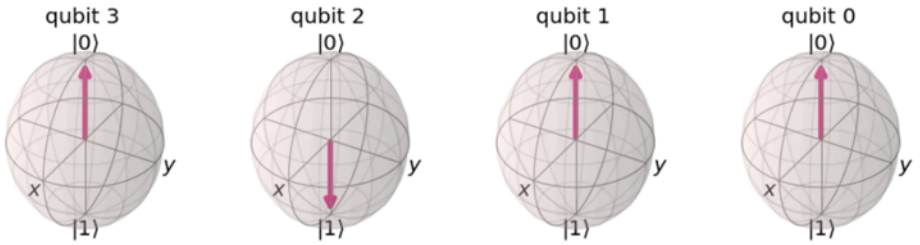


Figure 2.11: $|\psi_{IN}\rangle = |0100\rangle$

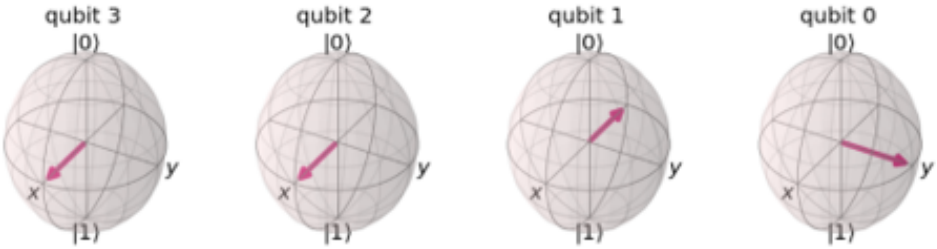


Figure 2.12: $|\psi_{OYT}\rangle = |+\rangle\,|+\rangle\,|-\rangle\,\frac{1}{\sqrt{2}}(|0\rangle + i\,|1\rangle)$

Case 3 :$|\psi_{IN}\rangle = |1000\rangle$

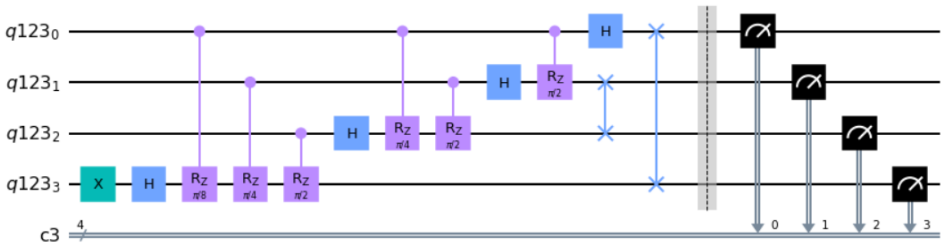


Figure 2.13: QFT circuit $|\psi_{IN}\rangle = |1000\rangle$

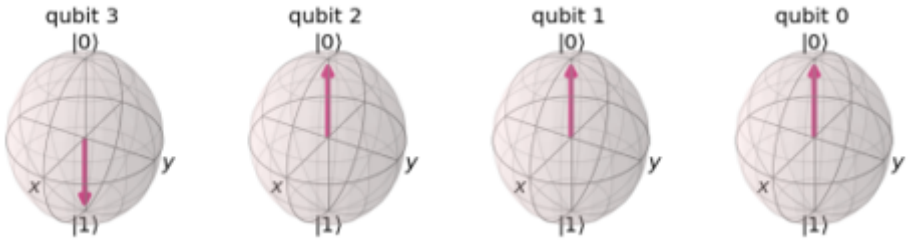


Figure 2.14: $|\psi_{IN}\rangle = |1000\rangle$

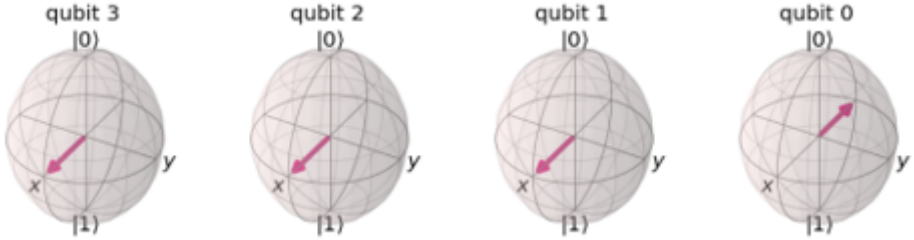


Figure 2.15: $|\psi_{OYT}\rangle = |+\rangle\,|+\rangle\,|+\rangle\,|-\rangle$

As we can see the QFT takes as input a number in binary form, which acts as the frequency for the rotation around the z axis. These rotations on the XY hyper-plane are controlled by the frequency. For example :

$$|\psi_{\text{IN}}\rangle = |x_3x_2x_1x_0\rangle = |1000\rangle = 8_{10}$$

$$|\psi_{out}\rangle = \frac{1}{\sqrt{2}}(|0\rangle + e^{2\cdot(2\cdot(2\cdot(2\pi i\frac{8}{16})))}\,|1\rangle)\frac{1}{\sqrt{2}}(|0\rangle + e^{2\cdot(2\cdot(2\pi i\frac{8}{16}))}\,|1\rangle)\frac{1}{\sqrt{2}}(|0\rangle + e^{2\cdot(2\pi i\frac{8}{16})}\,|1\rangle)\frac{1}{\sqrt{2}}(|0\rangle + e^{2\pi i\frac{8}{16}}\,|1\rangle)$$

$$|\psi_{out}\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) = |+\rangle\,|+\rangle\,|+\rangle\,|-\rangle$$

Now it is obvious that the LSB(x_0) rotates with frequency $\frac{8{:}input}{16{=}2^4{:}4\,qubit\,QFT} = \frac{1}{2}$. x_1 rotates two times faster than x_0. x_2 rotates two times faster than x_1 and four times faster than x_0 and so on.

Classical vs quantum Fourier transform
Time complexities and quantum speedup

As we have already seen the QFT circuit consists of Hadamard and controlled rotation gates and possibly SWAP gates . We can avoid the Swap gates if we classically rearrange the qubits. We can estimate the space complexity of QFT by counting the number of gates in the circuit :

- Hadamard and C-Rotation gates :$1 + 2 + 3 + ... + n = \sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- SWAP gates:$\lfloor \frac{n}{2} \rfloor$

$$QFT : O(n^2)\ Gates\ \text{, n number of qubits}$$

$$DFT : O(2^{2n})\ Gates\text{, n numbers of bits}$$

$$FFT : O(n2^n)\ Gates\text{, n numbers of bits}$$

The best QFT algorithms manage to achieve efficient approximations with just $O(n \log n)$ gates, n number of qubits.

2.2 Quantum phase estimation

Quantum phase estimation (QPE) is a quantum algorithm[13] which estimates the eigenvalue(phase) of an eigenvector of a unitary U operator. As we have shown in chapter 1, we can write an operator according to the spectral decomposition theorem as follows:

Let operator A with eigenvalues $\{\lambda_1, \lambda_2, ...\lambda_n\}$ and eigenvectors $\{|v_1\rangle, |v_2\rangle, ... |v_n\rangle\}$ can be written as :

$$\hat{A} = \sum_{j=1}^{n} \lambda_i |v_i\rangle \langle v_i|$$

The eigenvalues of unitary operators are roots of unity and can be written as follows:

$$\lambda_i = e^{2\pi i\varphi} \ , \ \varphi = 0.\varphi_1.\varphi_2...\varphi_N \ , \ 0 \leqslant \varphi < 1$$

It is useful for the rest of this QPE overview to present ϕ as a binary fraction. Let v_i an eigenvector of a unitary matrix U and λ_i its corresponding eigenvalue then:

$$U |v_i\rangle = \lambda_i |v_i\rangle = e^{2\pi i\varphi} |v_i\rangle$$

The algorithm takes as input arguments the unitary matrix U accessed by an oracle and a vector b and outputs an eigenvalue ϕ of U. Actually, the algorithm outputs a quantum state $|2^n\phi\rangle$ where, n is the number of qubits in the vector register and $\varphi = 0.\varphi_1.\varphi_2...\varphi_N$. The quantum circuit consists of 2 quantum registers R_a, R_v. R_a is an m-qubit ancillary quantum eegister that is initialized on $|0\rangle^{\otimes m}$. It is important to choose the right size(accuracy) for this register because it is the register that will eventually store the eigenvalue. R_v is an n-qubit register that stores the inputted eigenvector of U. The initial and the ultimate state of the algorithm are the following:

$$|\psi_{IN}\rangle = |R_a\rangle |R_v\rangle = |0\rangle^{\otimes m} |v\rangle_n \ : \text{ Initial state}$$

$$|\psi_{OUT}\rangle = \frac{1}{2^m} \sum_{k,j=0}^{2^m-1} e^{-2\pi\iota j \frac{(k-2^m\varphi)}{2^m}} |k\rangle |v\rangle_n \ : \text{ Output state}$$

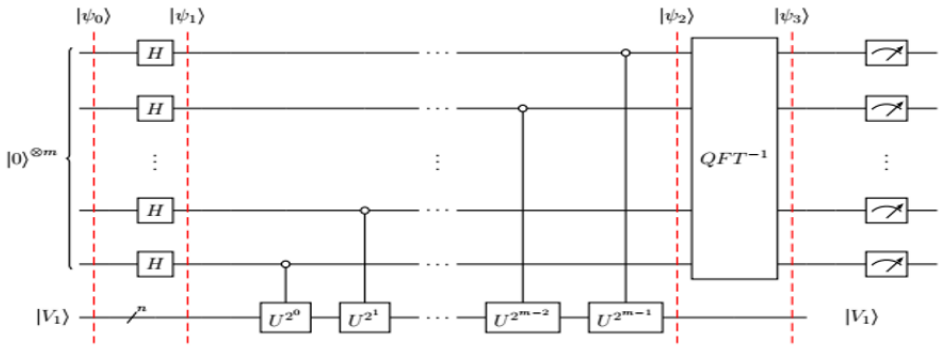


Figure 2.16: Quantum phase estimation generalized circuit

Steps of QPE

- $|\psi_0\rangle = |R_a\rangle\,|R_v\rangle = |0\rangle^{\otimes m}|v\rangle_n$: Initial state
- $|\psi_1\rangle = \frac{1}{\sqrt{2^m}}\sum_{j=0}^{2^m-1}|j\rangle|v\rangle_n$: Superposition in anc register
- $|\psi_2\rangle = C-U^{2^j}\,|\psi_1\rangle = \frac{1}{\sqrt{2^m}}\sum_{j=0}^{2^m-1} e^{2\pi i\varphi j}\,|j\rangle|v\rangle_n$: Control Rotations
- $|\psi_3\rangle = \frac{1}{2^m}\sum_{k,j=0}^{2^m-1} e^{-2\pi\imath j\frac{(k-2^m\varphi)}{2^m}}\,|k\rangle\,|v\rangle_n$: $\mathrm{QFT}^\dagger$ on anc register

QPE measurement

The end state of the algorithm is the following:

$$|\psi_{out}\rangle = \frac{1}{2^m}\sum_{k,j=0}^{2^m-1} e^{-2\pi\imath j\frac{(k-2^m\varphi)}{2^m}}\,|k\rangle\,|v\rangle_n\,,\varphi\in\mathbb{R}\,,\,0\leqslant\varphi<1\,,k\in\mathbb{N}$$

From this expression arise two possible cases:

Case 1: $2^m\varphi\in\mathbb{N}$

In this case if we make a measurement we get the following:

$$\Pr\left(|2^m\varphi\rangle\right) = \left|\langle 2^m\varphi|\,\frac{1}{2^m}\sum_{k,j=0}^{2^m-1} e^{-2\pi ij\frac{(k-2^m\varphi)}{2^m}}\,|k\rangle\right|^2 = \left|\langle 2^m\varphi|\,\frac{1}{2^m}\sum_{j=0}^{2^m-1} e^{-2\pi ij\frac{(2^m\varphi-2^m\varphi)}{2^m}}\,|2^m\varphi\rangle\right|^2$$

$$= \frac{1}{2^{2m}}\left|\sum_{j=0}^{2^m-1} e^{-2\pi ij\frac{(2^m\varphi-2^m\varphi)}{2^m}}\,\langle 2^m\varphi\mid 2^m\varphi\rangle\right|^2 = \frac{1}{2^{2m}}\left|\sum_{j=0}^{2^m-1} e^0\right|^2 = \frac{2^{2m}}{2^{2m}} = 1$$

Case 2: $2^m\varphi \notin \mathbb{N}$

Now we can only approximate the value of ϕ by rounding $2^m\varphi$ to the nearest integer. So, we write $2^m\varphi = 2^m\delta + \alpha$ where α is the nearest integer to $2^m\varphi$ and $0 \leqslant |2^m\delta| \leqslant \frac{1}{2}$.We reform the final state as follows:

$$|\psi_{out}\rangle = \frac{1}{2^m}\sum_{k,j=0}^{2^m-1} e^{-2\pi\iota j\frac{(k-(2^m\delta+\alpha))}{2^m}}|k\rangle = \frac{1}{2^m}\sum_{k,j=0}^{2^m-1} e^{-2\pi\iota j\frac{(k-a)}{2^m}}e^{2\pi ij\delta}|k\rangle$$

Now, this is the probability of measuring the best possible approximation α:

$$\begin{aligned}\Pr(|\alpha\rangle) &= \left|\langle\alpha|\frac{1}{2^m}\sum_{k,j=0}^{2^m-1} e^{-2\pi\iota j\frac{(k-a)}{2^m}}e^{2\pi ij\delta}|k\rangle\right|^2 = \left|\langle\alpha|\frac{1}{2^m}\sum_{j=0}^{2^m-1} e^{-2\pi\iota j\frac{(a-a)}{2^m}}e^{2\pi ij\delta}|\alpha\rangle\right|^2 \\ &= \frac{1}{2^{2m}}\left|\sum_{j=0}^{2^m-1} e^0 e^{2\pi ij\delta}\langle\alpha \mid \alpha\rangle\right|^2 = \frac{1}{2^{2m}}\left|\sum_{j=0}^{2^m-1} e^{2\pi ij\delta}\right|^2\end{aligned}$$

$$\begin{aligned}\Pr(|\alpha\rangle) &= \frac{1}{2^{2m}}\left|\frac{1-e^{2\pi i\delta 2^m}}{1-e^{2\pi i\delta}}\right|^2, \delta \neq 0 \\ &= \frac{1}{2^{2m}}\left|\frac{2\sin(\pi 2^m\delta)}{2\sin(\pi\delta)}\right|^2 = \frac{1}{2^{2m}}\left|\frac{\sin(\pi 2^m\delta)}{\sin(\pi\delta)}\right|^2 \quad , \left|1-e^{2ix}\right|^2 = 4|\sin(x)|^2 \\ &\geqslant \frac{1}{2^{2m}}\left|\frac{\sin(\pi 2^m\delta)}{\pi\delta}\right|^2 , |\sin(\pi\delta)| \leqslant |\pi\delta| \;\; for\ \delta \leqslant \frac{1}{2^m}+1 \\ &\geqslant \frac{1}{2^{2m}}\frac{|2\cdot 2^m\delta|^2}{|\pi\delta|^2} , |2\cdot 2^m\delta| \leqslant |\sin(\pi 2^m\delta)| \;\; for\ \delta \leqslant \frac{1}{2^m}+1 \\ &= \frac{1}{2^{2m}}\frac{2^2 2^{2m}\delta^2}{\pi^2\delta^2} = \frac{4}{\pi^2} \approx 0.405\end{aligned}$$

QPE Qiskit simulation

The first gate we ran was the phase gate S which is a diagonal matrix so, we can easily find its eigenvectors and corresponding eigenvalues. S gate has the following matrix representation[14]:

$$Phase\,gate\,S: \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i\frac{1}{4}} \end{pmatrix} = 1 \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} + e^{2\pi i\frac{1}{4}} \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$|v_1\rangle = |0\rangle \ , \lambda_1 = 1$$

$$|v_2\rangle = |1\rangle \ , \lambda_2 = i = e^{2\pi i\frac{1}{4}}$$

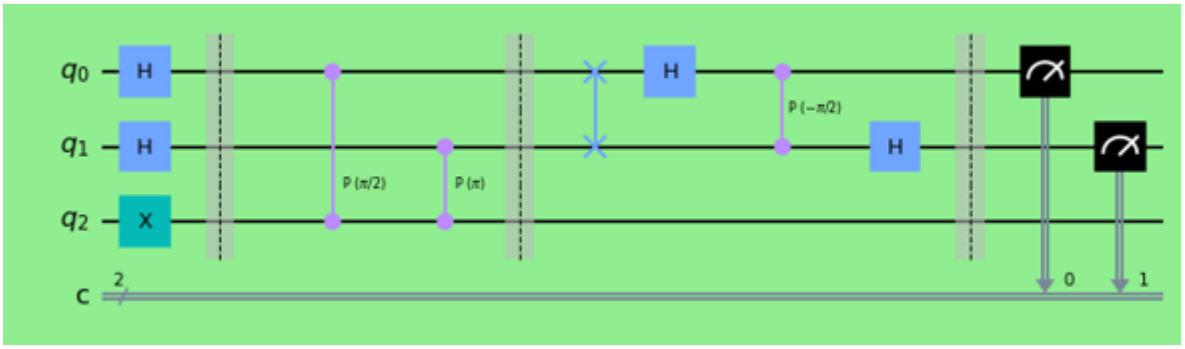


Figure 2.17: QPE S gate Qiskit circuit

$$|2^m\varphi\rangle = |01\rangle \ \ , \ \ 2^2\varphi = 1 \Rightarrow \varphi = \frac{1}{4} \Rightarrow e^{2\pi i\varphi} = e^{\frac{\pi i}{2}} = i$$

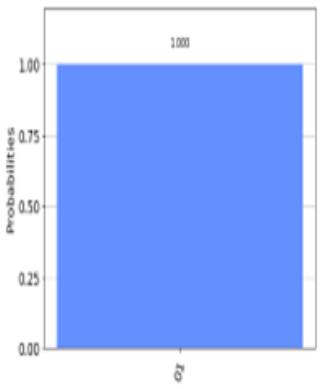


Figure 2.18: QPE S gate QASM Sim

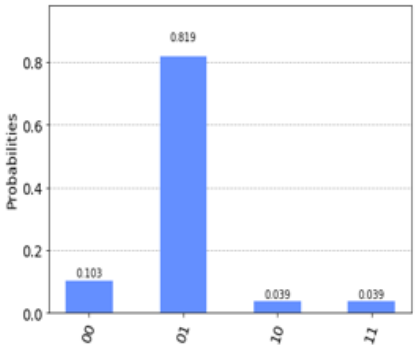


Figure 2.19: QPE S gate Ibmq Lima

Next, we ran the QPE algorithm for the phase gate T which has the following matrix representation:

$$Phase\,Gate\,T: \begin{pmatrix} 1 & 0 \\ 0 & e^{2\pi i\frac{1}{8}} \end{pmatrix} = 1 \cdot \begin{pmatrix} 1 \\ 0 \end{pmatrix} + e^{2\pi i\frac{1}{8}} \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$|v_1\rangle = |0\rangle \ , \lambda_1 = 1$$

$$|v_2\rangle = |1\rangle \ , \lambda_2 = e^{\frac{\pi i}{4}} = \cos\left(\frac{\pi i}{4}\right) + i\sin\left(\frac{\pi i}{4}\right) = \frac{1}{\sqrt{2}} + \frac{i}{\sqrt{2}}$$

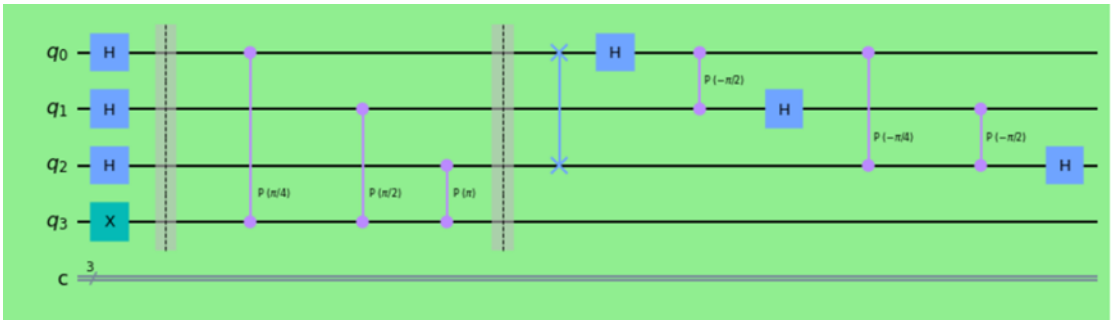


Figure 2.20: QPE T gate Qiskit circuit

$$|2^m\varphi\rangle = |001\rangle \ , \ 2^3\varphi = 1 \Rightarrow \varphi = \frac{1}{8} \Rightarrow e^{2\pi i\varphi} = e^{\frac{\pi i}{4}} = \frac{1}{\sqrt{2}} + \frac{i}{\sqrt{2}}$$

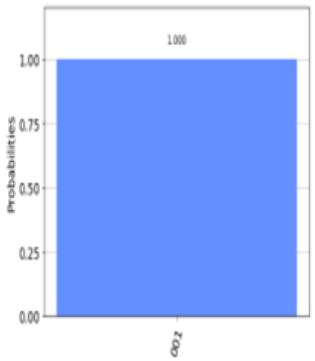


Figure 2.21: QPE T gate QASM sim

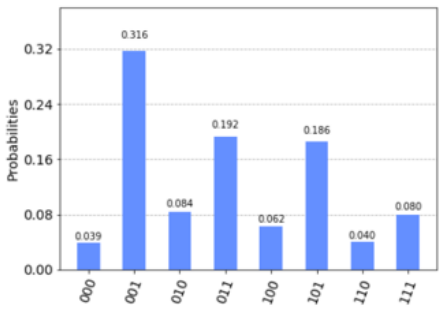


Figure 2.22: QPE T gate Ibmq Manila

Finally, we ran QPE for a random gate that has $\varphi = \frac{31}{32}$.

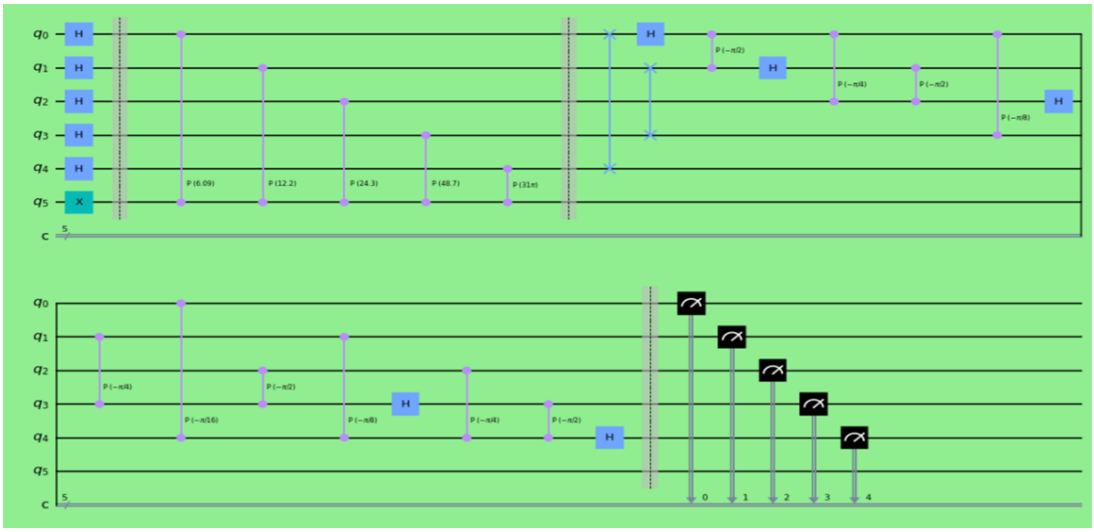


Figure 2.23: QPE random gate Qiskit circuit

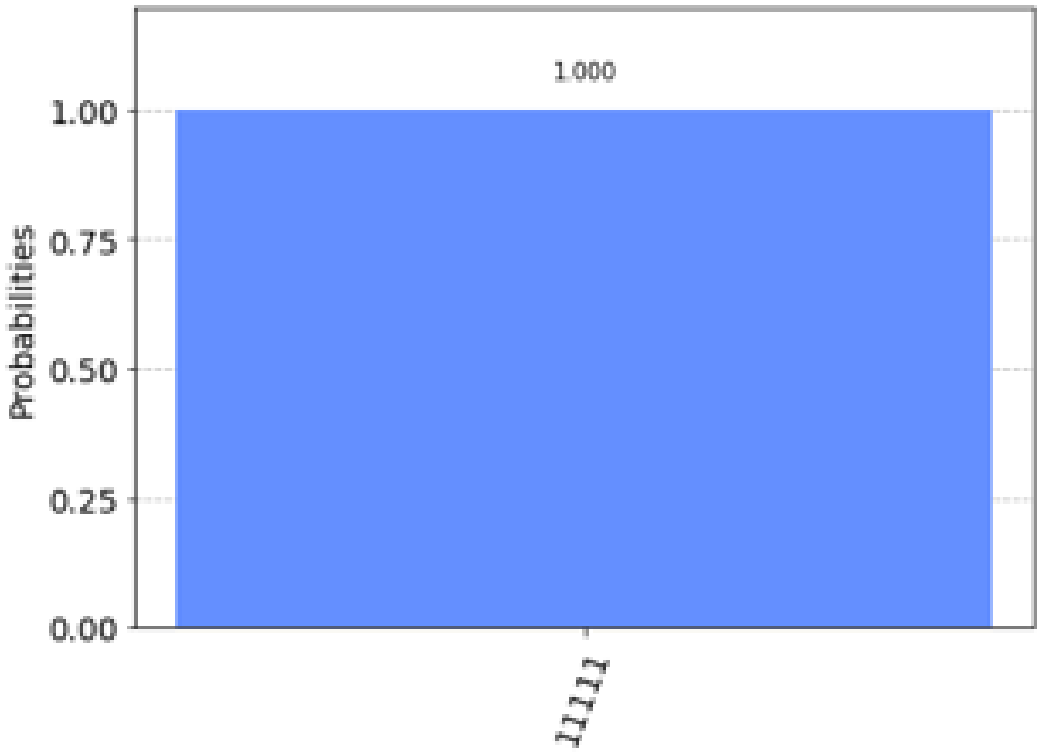


Figure 2.24: QPE random gate Qasm Simulator

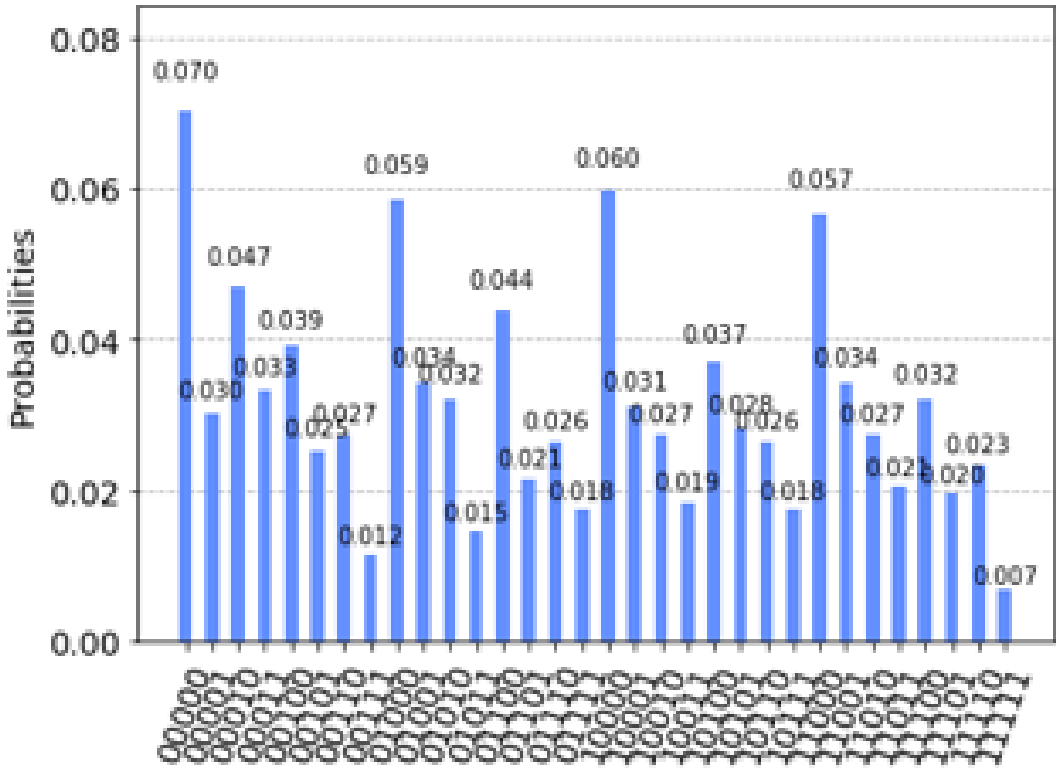


Figure 2.25: QPE random gate Ibmq Melbourne comprising of 14 qubits

$$|2^m\varphi\rangle = |11111\rangle \ , \ 2^5\varphi = 31 \Rightarrow \varphi = \frac{31}{32}$$

Chapter 3

Quantum linear system solver

3.1 Introduction

The linear system problem (LSP) is a very common mathematical problem that arises both on each own and as a subroutine in more complex problems. A general system of m linear equations and n unknowns can be written as follows:

$$a_{11}x_1 + a_{12}x_2 + ... + a_{1n}x_n = b_n$$

$$a_{21}x_1 + a_{22}x_2 + ... + a_{2n}x_n = b_2$$

$$\vdots$$

$$a_{m1}x_1 + a_{m2}x_2 + ... + a_{mn}x_n = b_m$$

$$x_1 \begin{bmatrix} a_{11} \\ a_{21} \\ \vdots \\ a_{m1} \end{bmatrix} + x_2 \begin{bmatrix} a_{12} \\ a_{22} \\ \vdots \\ a_{m2} \end{bmatrix} + \cdots + x_n \begin{bmatrix} a_{1n} \\ a_{2n} \\ \vdots \\ a_{mn} \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix} \rightarrow \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_m \end{bmatrix}$$

The general definition of the LSP is the following:

Given an invertible matrix A $\in \mathrm{C}^{N\times N}$, and a vector b $\in \mathrm{C}^{N}$ find a vector x $\in \mathrm{C}^{N}$ such that:

$$A\vec{x} = \vec{b}$$

3.2 Classical approaches for solving linear equations

3.2.1 Gaussian elimination

The most common way to solve a system of linear equations is Gaussian elimination. Gaussian elimination is an algorithm that makes use of elementary row operations in order to produce a convenient matrix decomposition of the original matrix (row echelon form and reduced row echelon form), assign values to the independent variables and use back substitution to determine the values of the dependent variables.
Elementary row operations:

1. Swap the positions of two rows
2. Multiply by a non-zero scalar
3. Add to one row a scalar multiple of another

A matrix is in row echelon form if and only if all the following conditions hold:

1. The first nonzero entry in each row is 1
2. Each successive row has its first nonzero entry in a later column
3. All entries below the first nonzero entry of each row are zero
4. All full rows of zeroes are the final rows of the matrix

$$\mathrm{A} = \begin{bmatrix} 1 & a & b & c \\ 0 & 1 & d & e \\ 0 & 0 & 1 & f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

matrix in row echelon form

The Gaussian elimination algorithm has arithmetic complexity O(n^3), n: number of variables.

3.2.2 Conjugate gradient descent

Another classical algorithm that solves the LSP is the conjugate gradient descent[15]. In this case we must add some limitations to the given matrix A. For this algorithm to work A must be positive semi-definite and sparse.

- A matrix M is said to be positive semi - definite if $\mathrm{x}^* Mx \geqslant 0$ for all $\mathrm{x} \in \mathbb{C}^n$
- A matrix M is s-sparse if it has at most s nonzero entries per row

If we meet those expectations, then the conjugate gradient descent can iteratively find a solution for the LSP. To simply outline this method :

- The algorithm makes a guess for the solution x
- And then gradually minimizes the quadratic function $\|Ax - b\|^2$ to finds its global minimum.

Conjugate gradient descent has complexity: O $\left(nks \log\left(\frac{1}{\varepsilon}\right)\right)$ where s defines the sparsness of A, k is the condition number of A, $cond\,(A) = \|A\|_\infty \|A^{-1}\|_\infty$, n: number of variables, ϵ: the accepted error.

Classical linear system solvers for large matrices

Classical computers are evolving at a tremendous rate both in hardware and algorithmic efficiency allowing us to solve bigger and bigger linear systems of equations. With the increased speed comes an increase in the size of problems that can be solved in reasonable time, provided that sufficient memory is available.

More precisely we present a list below that shows the biggest solvable matrices from 1993 to 2021. The data is from the TOP500[?]

Machine	Date	n
Fugaku	June 2021	2.1×10^7
Jaguar	June 2010	6.3×10^6
ASCI RED	June 2000	3.6×10^5
CM-5/1024	June 1993	5.2×10^4

Figure 3.1: Largest solvable matrices

3.3 Quantum approach

As we have previously seen the classical approach on LSP has reached a certain stalemate. Quantum computers promise to solve efficiently that kind of problems. Due to the nature of quantum computation, the size of the linear system grows exponentially with the increasing number of functional qubits.

There are two main ways to solve LSP using quantum computers:

- Exact algorithms: Exact algorithms such as the HHL [1], which is completely quantum and is the first algorithm that we will go through.

- Approximate algorithms: Approximate algorithms that are hybrid quantum-classical algorithms. In the next chapter we will work with the variational quantum linear

solver (VQLS) [2].

We will cross-examine these two algorithms in respect to circuit depth, number of qubits needed and result fidelity.

3.3.1 HHL algorithm

The HHL algorithm was introduced by A. W. Harrow, A. Hassidim and S. Lloyd in 2009 in order to solve the quantum variation of the linear system problem. The definition of the quantum linear system problem is the following:

Let A N × N Hermitian matrix and $|\mathrm{b}\rangle$ unit vector, find $|x\rangle$ such that: $\mathrm{A}\,|x\rangle = |b\rangle$

There are some restrictions on matrix A. A must be sparse, square and well conditioned. If matrix A is not Hermitian, we can reformulate the problem as follows:

$$if\ A \neq A^{\dagger}\ then \begin{pmatrix} 0 & A \\ A^{\dagger} & 0 \end{pmatrix} \begin{pmatrix} 0 \\ \overrightarrow{x} \end{pmatrix} = \begin{pmatrix} \overrightarrow{b} \\ 0 \end{pmatrix}$$

Matrix A is accessed by an oracle and the input vector b should be normalized to be initialized.

$$|b\rangle := \frac{\sum_i b_i\,|i\rangle}{\left\|\sum_i b_i\,|i\rangle\right\|_2}$$

The goal of the algorithm is to find a $|x\rangle$ solution vector such that $A\,|x\rangle = |b\rangle$. So, all we need to do is to construct $|x\rangle$ as follows:

$$A\,|x\rangle = |b\rangle \Rightarrow |x\rangle = A^{-1}\,|b\rangle$$

We need to apply the reciprocal of matrix A to vector $|b\rangle$ to get the solution vector $|x\rangle$. This is the reason that many refer to this problem as quantum matrix inversion problem.

The HHL is an exact algorithm, that finds a $|\tilde{x}\rangle$ vector, which is proportional to $|x\rangle$. If we use HHL to obtain the whole $|x\rangle$, we lose the computational speedup over the classical algorithms. Instead, we can extract information about the global properties of the solution vector $|x\rangle$ by computing the expectation value $\langle x| M |x\rangle$ where M is a Pauli operator.

Now, we must introduce some notations. We need three quantum registers for this problem.

- Memory register: is initialized to $|b\rangle$ and will eventually store the solution vector $|x\rangle$. This register will store n qubits where $n = \log_2 N$ where N: number of variables
- Eigenvalue register: is initialized to 0 and it has the necessary size to store the eigenvalues of A. So, its size depends on the accuracy, which we need for the first step of the algorithm(QPE) to work efficiently
- Ancillary register: is initialized to 0 and lets us perform the controlled-rotations in the second step of the algorithm. At the end of the algorithm, we postprocess (value=1) this register that will also serve as a flag indicating the success or failure of the algorithm. The ancillary register stores just one qubit.

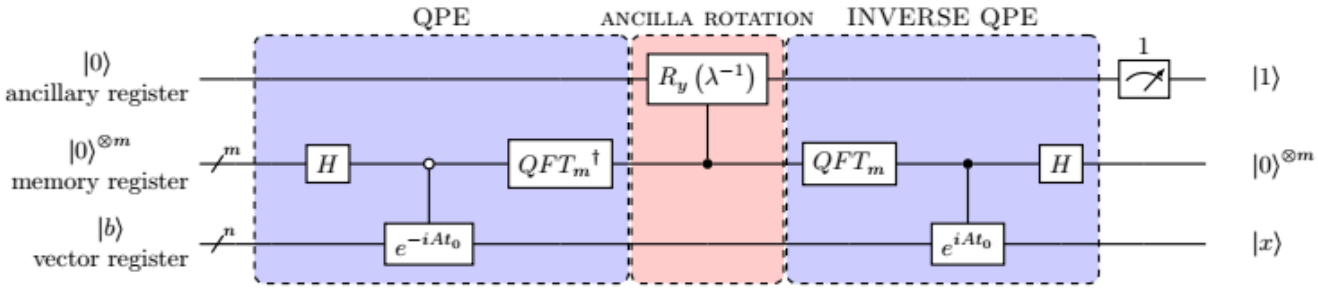


Figure 3.2: HHL general circuit

State progression

$$|\psi_1\rangle = |0\rangle_a |0\rangle_m^{\otimes m} |b\rangle_v \xrightarrow{clock} |0\rangle_\alpha \sqrt{\frac{2}{\mathrm{T}}} \sum_{\tau=0}^{\mathrm{T}-1} \sin\left(\frac{\pi\,(\tau+1/2)}{\mathrm{T}}\right) |\tau\rangle_m |b\rangle_v = |\psi_1\rangle$$

$$|\psi_1\rangle = |0\rangle_\alpha \sqrt{\frac{2}{\mathrm{T}}} \sum_{\tau=0}^{\mathrm{T}-1} \sin\left(\frac{\pi\,(\tau+1/2)}{\mathrm{T}}\right) |\tau\rangle_m |b\rangle_v \xrightarrow{QPE} \sum_{j=0}^{2^n-1} |0\rangle_a |\lambda_j\rangle_m \beta_j |u_j\rangle_v = |\psi_2\rangle$$

$$|\psi_2\rangle = \sum_{j=0}^{2^n-1} |0\rangle_a |\lambda_j\rangle_m \beta_j |u_j\rangle_v \xrightarrow{anc\,Rotation} \sum_{j=0}^{2^n-1} \left(\sqrt{1-\frac{C^2}{\lambda_j^2}}|0\rangle_a + \frac{C}{\lambda_j}|1\rangle_a\right) |\lambda_j\rangle_m \beta_j |u_j\rangle_v = |\psi_3\rangle$$

$$|\psi_3\rangle = \sum_{j=0}^{2^n-1} \left(\sqrt{1-\frac{C^2}{\lambda_j^2}}|0\rangle_a + \frac{C}{\lambda_j}|1\rangle_a\right) |\lambda_j\rangle_m \beta_j |u_j\rangle_v \xrightarrow{QPE^{-1}} \sum_{j=0}^{2^n-1} \left(\sqrt{1-\frac{C^2}{\lambda_j^2}}|0\rangle_a + \frac{C}{\lambda_j}|1\rangle_a\right) |0\rangle_m^{\otimes m} \beta_j |u_j\rangle_v$$

$$|\psi_4\rangle = \sum_{j=0}^{2^n-1} \left(\sqrt{1-\frac{C^2}{\lambda_j^2}}|0\rangle_a + \frac{C}{\lambda_j}|1\rangle_a\right) |0\rangle_m^{\otimes m} \beta_j |u_j\rangle_v \xrightarrow{post\ selection} \sum_{j=0}^{2^n-1} \left(\frac{C}{\lambda_j}\beta_j\right) |1\rangle_a |0\rangle_m^{\otimes m} \beta_j |u_j\rangle_v = |\psi_5\rangle$$

HHL consists of 4 main steps:

1. Quantum phase estimation in order to extract the eigenvalues λ_j of matrix A
2. Controlled-rotation on the ancilla qubit to create $\frac{1}{\lambda_j}$
3. Inverse quantum phase estimation to uncompute the eigenvalue register
4. Classical post-selection on the ancillary register and then the computation of the expectation value $\langle x| M |x\rangle$

If we decompose matrix A according to the eigendecomposition, we get the following:

$$A|x\rangle = |b\rangle \overset{initial}{\Rightarrow} R^\dagger \Lambda R |x\rangle = |b\rangle \overset{QPE}{\Rightarrow} \Lambda R |x\rangle = R|b\rangle \overset{anc\,Rt}{\Rightarrow} R|x\rangle = \Lambda^{-1} R|b\rangle \overset{QPE^{-1}}{\Rightarrow} |x\rangle = R^\dagger \Lambda^{-1} R|b\rangle$$

Step1: Quantum phase estimation

In the HHL algorithm the given matrix A is not unitary but Hermitian. A is Hermitian and we know that e^{iAt} is unitary, so we need to apply conditional Hamiltonian evolution to create e^{iAt}. The conditional Hamiltonian evolution is conditioned over t (time). The eigenvalue register will be initialized as a clock register in order to apply properly the conditional Hamiltonian evolution.

$$|\psi_{\mathrm{I}}\rangle = |0\rangle_a |0\rangle_m^{\otimes m} |b\rangle_v \xrightarrow{clock} |0\rangle_\alpha \sqrt{\frac{2}{\mathrm{T}}} \sum_{\tau=0}^{\mathrm{T}-1} \sin\left(\frac{\pi\left(\tau+1/2\right)}{\mathrm{T}}\right) |\tau\rangle_m |b\rangle_v = |\psi_1\rangle$$

The conditional Hamiltonian operator is the following:

$$\sum_{\tau=0}^{\mathrm{T}-1} |\tau\rangle \langle\tau| \otimes e^{\frac{\iota A \tau t_0}{T}}$$

So, if We apply conditional Hamiltonian evolution on the clock register we get :

$$\left(\sum_{\tau=0}^{\mathrm{T}-1} |\tau\rangle \langle\tau| \otimes e^{\frac{\iota A \tau t_0}{T}}\right) \sqrt{\frac{2}{\mathrm{T}}} \sum_{\tau=0}^{\mathrm{T}-1} \sin\left(\frac{\pi\left(\tau+1/2\right)}{\mathrm{T}}\right) |\tau\rangle_m \otimes |b\rangle_v =$$

$$= \sqrt{\frac{2}{\mathrm{T}}} \sum_{\tau=0}^{\mathrm{T}-1} |\tau\rangle \langle\tau| \sin\left(\frac{\pi\left(\tau+1/2\right)}{\mathrm{T}}\right) |\tau\rangle_m \otimes e^{\frac{\iota A \tau t_0}{T}} |b\rangle_v$$

Where $\mathrm{T} = 2^t$ and $t_0 = \mathrm{O}\left(\frac{\kappa}{\varepsilon}\right)$. It is possible to run a proof of principle example with a simpler clock (Haddamard superposition) but this uniform clock minimizes error.

After the conditional Hamiltonian simulation, we apply inverse quantum Fourier transform to the memory-clock register and we get the following quantum state:

$$|\psi_1\rangle = |0\rangle_\alpha \sqrt{\frac{2}{\mathrm{T}}} \sum_{\tau=0}^{\mathrm{T}-1} \sin\left(\frac{\pi\left(\tau+1/2\right)}{\mathrm{T}}\right) |\tau\rangle_m |b\rangle_v \xrightarrow{QPE} \sum_{j=0}^{2^n-1} |0\rangle_a |\lambda_j\rangle_m \beta_j |u_j\rangle_v = |\psi_2\rangle$$

Step2: Controlled rotation on the ancillary register

In this step we implement a non-unitary map : $|\lambda_j\rangle \xrightarrow{map} {\lambda_j}^{-1} |\lambda_j\rangle$. We perform a controlled unitary transformation $R_y\left(\theta_j\right)$ on the ancilla register, controlled by the eigenvalue register.

$$R_y\left(\theta_j\right) = U\left(\theta, \varphi = 0, \lambda = 0\right) = \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -e^{i\lambda}\sin\left(\frac{\theta}{2}\right) \\ e^{i\varphi}\sin\left(\frac{\theta}{2}\right) & e^{i(\lambda+\varphi)}\cos\left(\frac{\theta}{2}\right) \end{pmatrix} = \begin{pmatrix} \cos\left(\frac{\theta}{2}\right) & -\sin\left(\frac{\theta}{2}\right) \\ \sin\left(\frac{\theta}{2}\right) & \cos\left(\frac{\theta}{2}\right) \end{pmatrix}$$

Where $\theta_j = \cos^{-1}\left(\frac{C}{\tilde{\lambda}_j}\right)$, $\tilde{\lambda}_j$ is a m-qubit approximation of λ_j and $C \leqslant \min|\lambda_j|$

If the eigenvalues of A are too small, then the matrix is ill-conditioned, and we can not perform

the matrix inversion. After this step our system is in the following state:

$$|\psi_2\rangle = \sum_{j=0}^{2^n-1} |0\rangle_a |\lambda_j\rangle_m \beta_j |u_j\rangle_v \xrightarrow{anc\,Rotation} \sum_{j=0}^{2^n-1} \left(\sqrt{1 - \frac{C^2}{\lambda_j^2}} |0\rangle_a + \frac{C}{\lambda_j} |1\rangle_a \right) |\lambda_j\rangle_m \beta_j |u_j\rangle_v = |\psi_3\rangle$$

Step3: Inverse quantum phase estimation

Now that we have successfully created the reciprocal of the eigenvalues $\sqrt{1 - \frac{C^2}{\lambda_j^2}} |0\rangle_a + \frac{C}{\lambda_j} |1\rangle_a$ we no longer need the memory register. We perform inverse quantum phase estimation (IQPE) to disentangle (uncompute) the memory register. The system is now:

$$|\psi_3\rangle = \sum_{j=0}^{2^n-1} \left(\sqrt{1 - \frac{C^2}{\lambda_j^2}} |0\rangle_a + \frac{C}{\lambda_j} |1\rangle_a \right) |\lambda_j\rangle_m \beta_j |u_j\rangle_v \xrightarrow{QPE^{-1}} \sum_{j=0}^{2^n-1} \left(\sqrt{1 - \frac{C^2}{\lambda_j^2}} |0\rangle_a + \frac{C}{\lambda_j} |1\rangle_a \right) |0\rangle_m^{\otimes m} \beta_j |u_j\rangle_v$$

Step4: Post selection and expectation values

At this point if we make a measurement and get 1 on the ancilla register, then we know we have successfully "constructed" a state that is proportional to the desired solution.

$$|\psi_4\rangle = \sum_{j=0}^{2^n-1} \left(\sqrt{1 - \frac{C^2}{\lambda_j^2}} |0\rangle_a + \frac{C}{\lambda_j} |1\rangle_a \right) |0\rangle_m^{\otimes m} \beta_j |u_j\rangle_v \xrightarrow{post\,selection} \sum_{j=0}^{2^n-1} \left(\frac{C}{\lambda_j} \beta_j \right) |1\rangle_a |0\rangle_m^{\otimes m} \beta_j |u_j\rangle_v = |\psi_5\rangle$$

We know that a measurement on the vector register will yield just one of the eigenvectors. In order to make full use of the quantum superposition we have created, we measure an expectation value instead. It is possible to get the full solution but in order to do so we will need many successive runs and all the quantum speedup will be lost.

Computational complexity: quantum vs classical

As we have mentioned before the best-known classical algorithm that solves LSP is the conjugate gradient descent with complexity: $\mathrm{O}\left(Nks \log\left(\frac{1}{\varepsilon}\right)\right)$. HHL has complexity: $\mathrm{O}\left(\log(N)\, k^2 s^2 \frac{1}{\varepsilon}\right)$ where, N:number of variables, k:condition number$=k(A) = \left\|A^{-1}\right\|_\infty \|A\|_\infty$, s:sparsity = maximum non-zero entries per row, ε= accepted error

HHL provides an exponential speedup over the number of variables, but it is sensitive to the changes of sparseness, condition number and error. We can see that eventually the algorithm can be useful for inverting big matrices with the condition that the future quantum hardware is able to provide a plethora of functional qubits.

3.3.2 HHL implementation

We ran the HHL algorithm in Qiskit with 3 ancillary qubits and the conditional Hamiltonian evolution has expansion_mode= 'Suzuki' and number of slices =30 We ran it first without noise using the state vector Simulator as backend and then we added noise using the noise model from ibmq_quito.

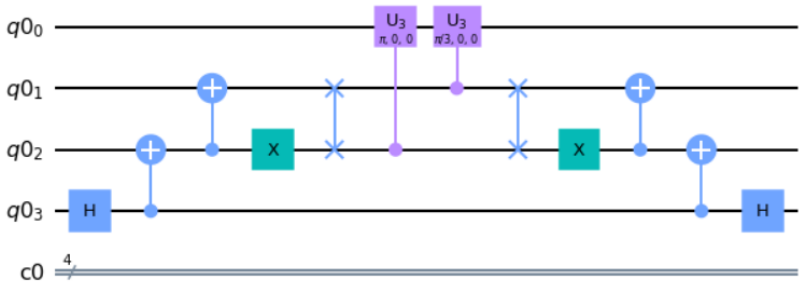


Figure 3.3: HHL circuit for a 2x2 optimized problem

The above Gate sequence implements the following Linear System:

$$\begin{pmatrix} \frac{3}{2} & \frac{1}{2} \\ \frac{1}{2} & \frac{3}{2} \end{pmatrix} \quad for \quad |b\rangle = 0, \quad \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ -1 \end{pmatrix}$$

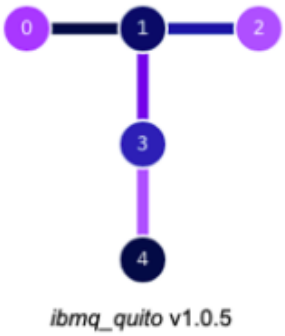


Figure 3.4: ibmq_quito circuit comprising of 5 qubits connected as in the figure

```
In [27]: noise_model.basis_gates
Out[27]: ['cx', 'id', 'reset', 'rz', 'sx', 'x']

In [29]: noise_model.noise_instructions
Out[29]: ['cx', 'id', 'measure', 'reset', 'sx', 'x']

noise_model.noise_qubits
[0, 1, 2, 3, 4]
```

Figure 3.5: ibmq_quito noise model

The measured variables for the subsequent simulations are the following:

- fidelity: $fid\,(x,y) = Tr\left(\sqrt{\sqrt{x}y\sqrt{x}}\right)^2, x = \frac{HHL}{\|HHL\|_2}, y = \frac{classical}{\|classical\|_2}$
- Circuit depth
- Circuit width (minimum qubits needed)

The test matrices are on the appendix A.2

2x2 matrix implementation in Qiskit

We ran the HHL algorithm from IBM Qiskit Textbook [16] with 3 ancillary qubits and apply the conditional Hamiltonian evolution with expansion_mode= 'Suzuki' and number of slices =30 for 7 different diagonal 2x2 matrices with condition numbers (1, 1.5, 2, 5, 10, 100, 1000) to see what is the correlation between condition number and fidelity. As backend we have used the 'state_vector_simulator'

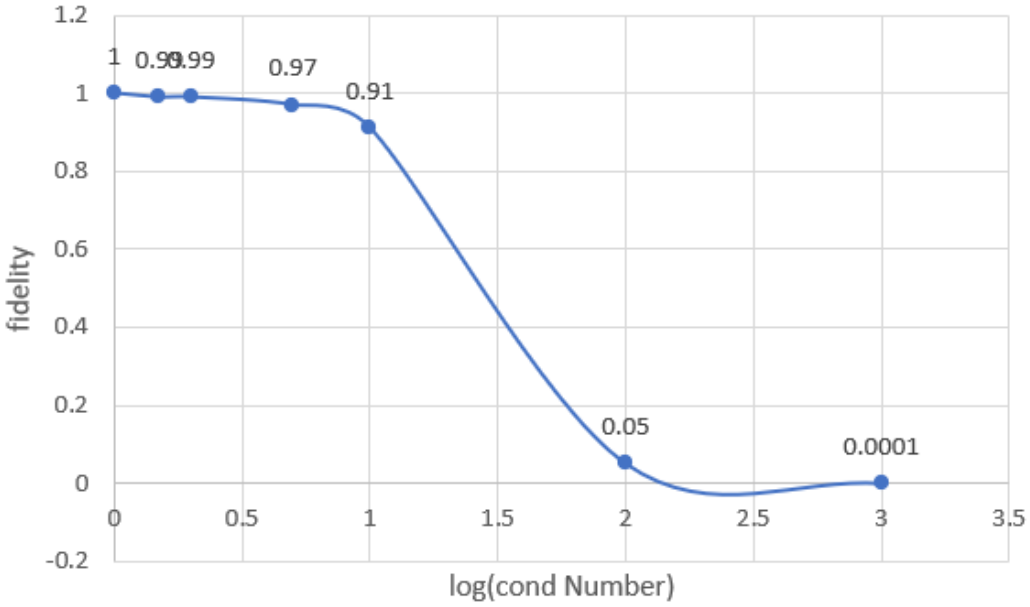


Figure 3.6: 2x2 diagonal matrices 'state_vector_simulator'

Afterwards we ran one more simulation for the 2x2 case but this time we ran the HHL for 5 matrices with condition numbers(1.5, 2, 5, 10, 100) with s=1 and s=2 respectively. By doing so we opt to show the correlation between sparsity and fidelity.

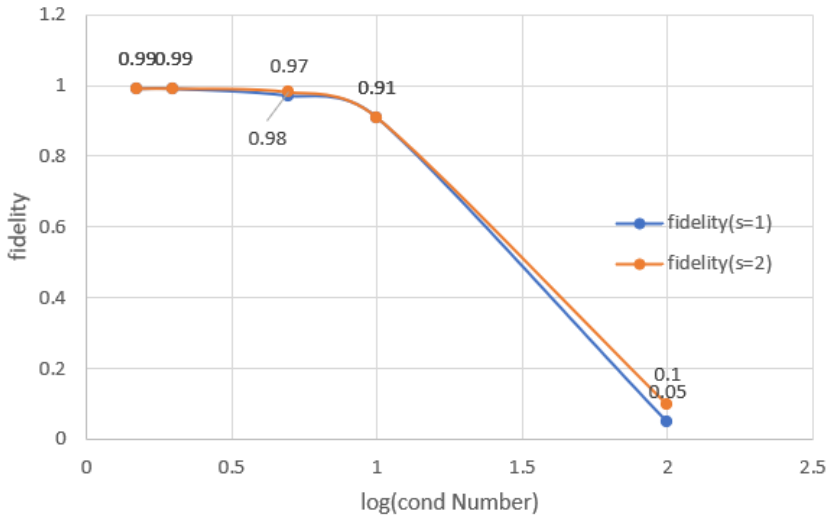


Figure 3.7: 2x2 matrices with s=1,2 'state_vector_simulator'

In the last simulation for the 2x2 case we added noise using the ibmq_quito noise model and ran the previous simulation for 5 matrices with condition numbers(1.5, 2, 5, 10, 100) with s=1

and s=2 in order to show how noise sensitive is the algorithm.

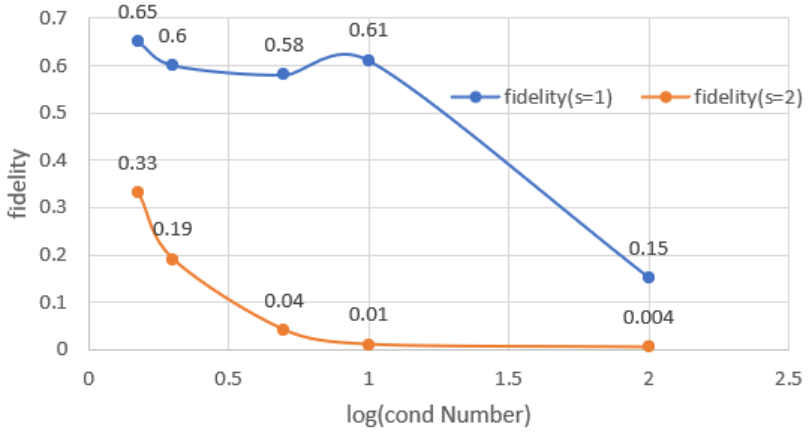


Figure 3.8: 2x2 matrices with s=1,2 noisy simulation

4x4 Matrix implementation in Qiskit

We ran the HHL algorithm from IBM Qiskit Textbook [16] with 3 ancillary qubits and apply the conditional Hamiltonian evolution with expansion_mode= 'Suzuki' and number of slices =30 for 7 different 4x4 diagonal matrices with condition numbers (1, 1.5, 2, 5, 10, 100, 1000) to see what is the correlation between condition number and fidelity. As backend we have used the 'state_vector_simulator'

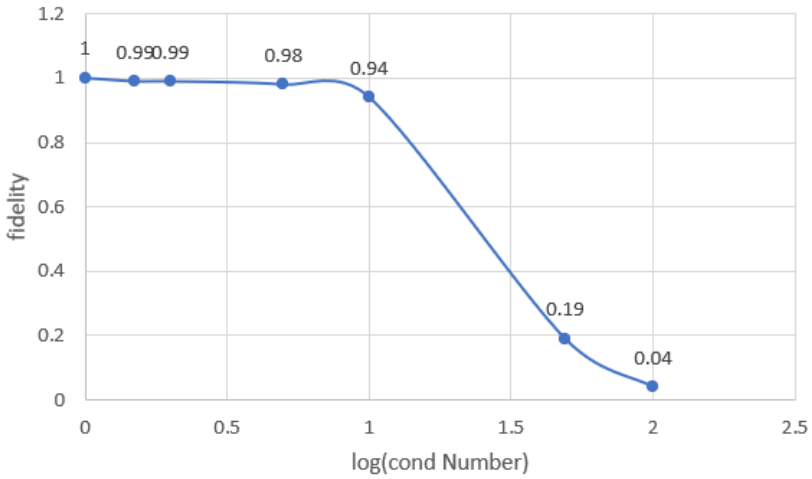


Figure 3.9: 4x4 diagonal matrices 'state_Vector_simulator'

Afterwards, we ran the HHL for 4 matrices with condition number (10) with s=1-4. By doing so we opt to show the correlation between sparsity and fidelity.

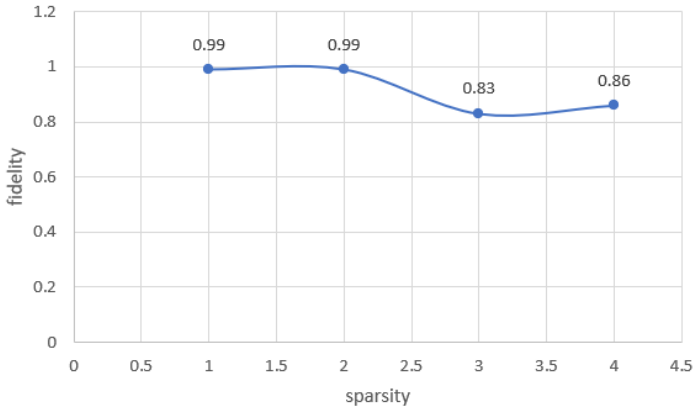


Figure 3.10: 4x4 matrices with s=1-4, c=10 'state_Vector_simulator'

In the last simulation for the 4x4 case we added noise using the ibmq_quito noise model and ran a noisy simulation for five 4x4 matrices with condition numbers(1. 5, 2, 5, 10, 100) with s=1,2 respectively in order to show how noise sensitive is the algorithm.

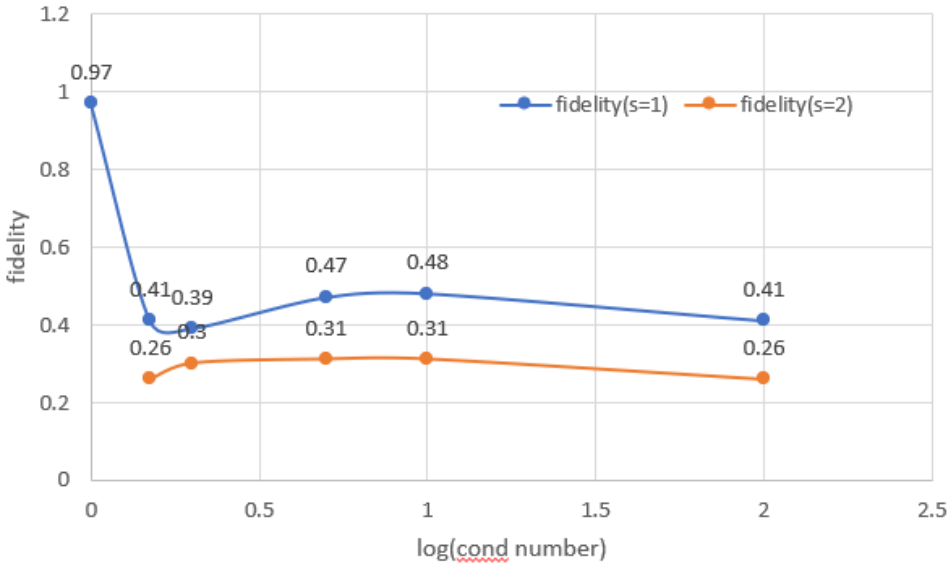


Figure 3.11: 4x4 matrices with s=1,2 noisy simulation

8x8 Matrix implementation in Qiskit

We ran the HHL algorithm from IBM Qiskit Textbook[16] while keeping the default arguments (3 ancillary qubits, expansion_mode= 'Suzuki') for 7 different diagonal 8x8 matrices with condition numbers (1, 1.5, 2, 5, 10, 50, 100) to see what is the correlation between condition number and fidelity. We run it first without noise (backend : 'state_Vector_simulator') and then with noise (noise model=ibmq_quito). The results are presented below:

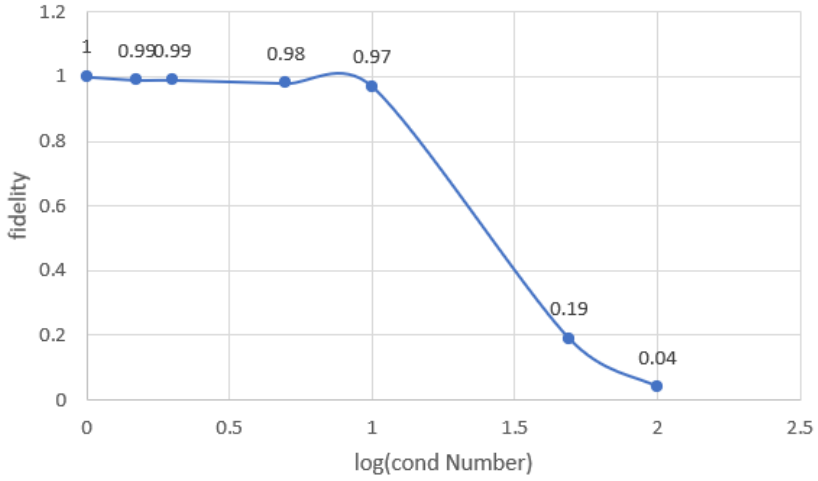


Figure 3.12: 8x8 diagonal matrices 'state_vector_simulator'

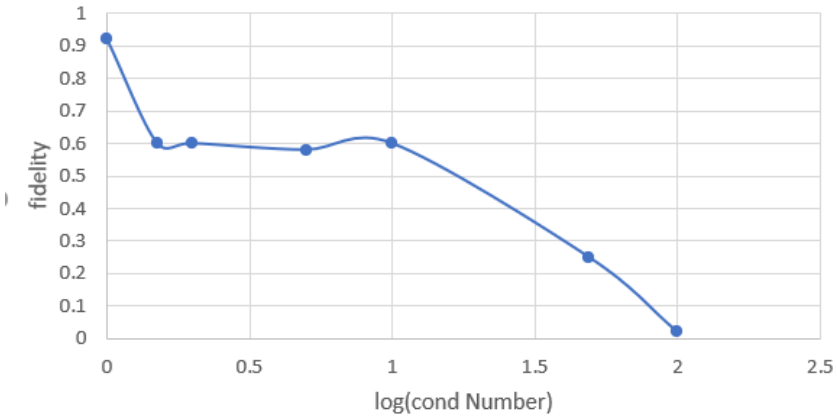


Figure 3.13: 8x8 diagonal matrices noisy simulation

3.3.3 Running HHL on a real quantum device: optimised example

In the previous section we ran the standard algorithm provided in Qiskit and saw that it uses 7 qubits, has a depth of ~ 100 gates and requires a total of 54 CNOT gates[17]. These numbers are not feasible for the current available hardware, therefore we need to decrease these quantities. In particular, the goal will be to reduce the number of CNOTs by a factor of 5 since they have worse fidelity than single-qubit gates. Furthermore, we can reduce the number of qubits to 4 as was the original statement of the problem: the Qiskit method was written for a general problem and that is why it requires 3 additional auxiliary qubits.

However, solely decreasing the number of gates and qubits will not give a good approximation to the solution on real hardware. This is because there are two sources of errors: those that occur during the run of the circuit and readout errors.

Qiskit provides a module to mitigate the readout errors by individually preparing and measuring all basis states, a detailed treatment on the topic can be found in the paper by Dewes et al.[3] To deal with the errors occurring during the run of the circuit, Richardson extrapolation can be used to calculate the error to the zero limit by running the circuit three times, each replacing each CNOT gate by 1, 3 and 5 CNOTs respectively[4]. The idea is that theoretically the three circuits should produce the same result, but in real hardware adding CNOTs means amplifying the error. Since we know that we have obtained results with an amplified error, and we can estimate by how much the error was amplified in each case, we can recombine the quantities to obtain a new result that is a closer approximation to the analytic solution than any of the previous obtained values.

Below we give the optimised circuit that can be used for any problem of the form

$$A = \begin{pmatrix} a & b \\ b & a \end{pmatrix} \quad and \quad |b\rangle = \begin{pmatrix} \cos(\theta) \\ \sin(\theta) \end{pmatrix} \quad , \quad a, b, \theta \in \mathbb{R}$$

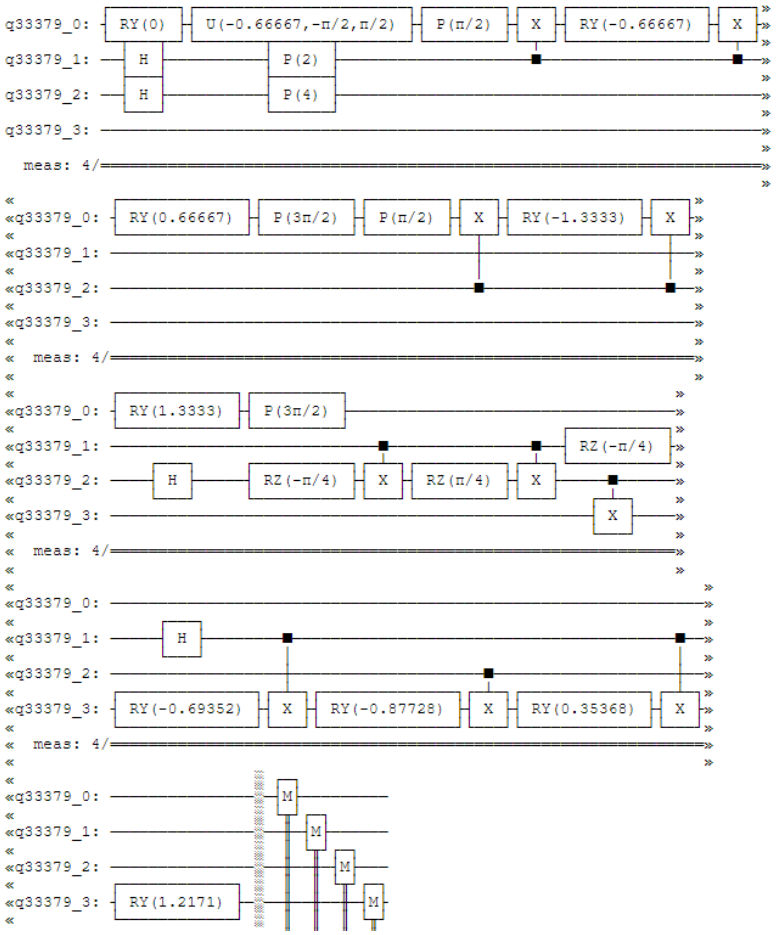


Figure 3.14: HHL optimized circuit

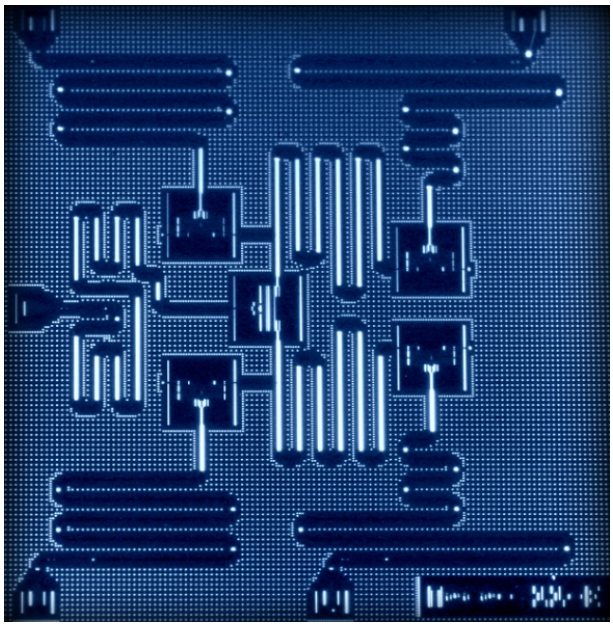
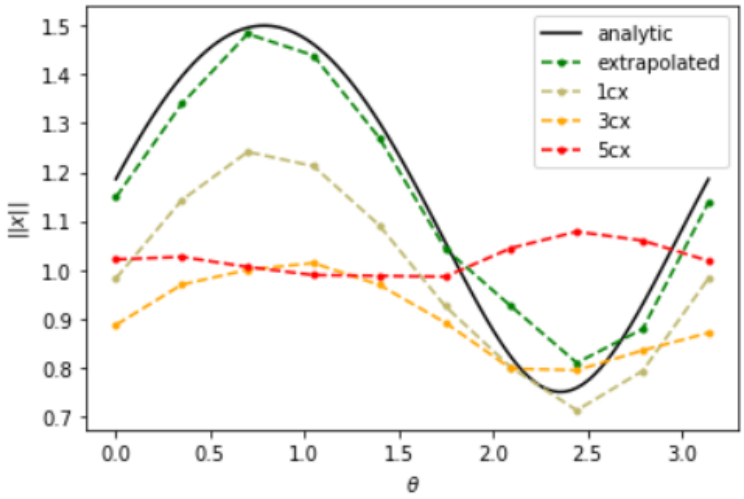
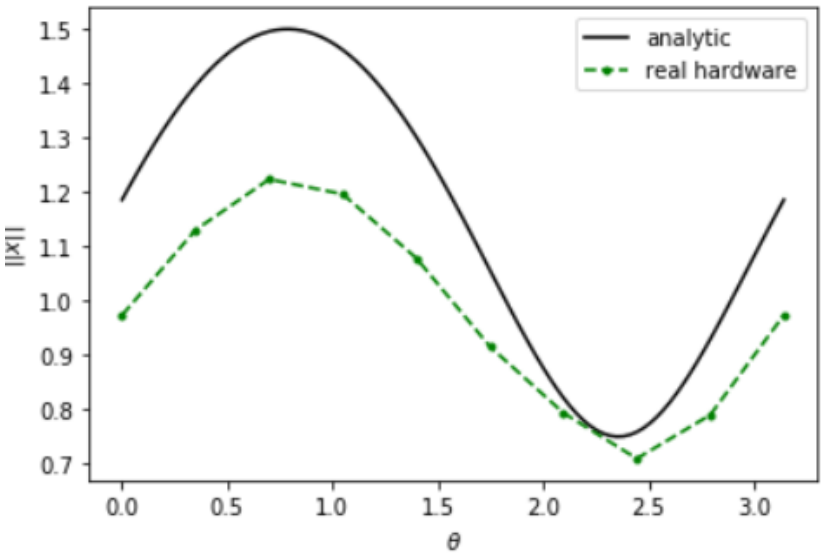


Figure 3.15: IBM 5-qubit quantum processor

The following plot, shows the results from running the circuit above on real hardware for 10 different initial states[18]. The x-axis represents the angle θ defining the initial state in each case. The results where obtained after mitigating the readout error and then extrapolating the errors arising during the run of the circuit from the results with the circuits with 1, 3 and 5 CNOTs.



The next plot shows results without error mitigation nor extrapolation from the CNOTs[18].



HHL implementation-results analysis

After experimenting with different matrix sizes, condition numbers and sparsities we have reached the following conclusions:

- As expected, the condition number relates strongly to fidelity. The bigger the condition number ,the lower the fidelity. Especially when k=1-10 (identity matrix) fidelity is almost 1 and when we run the extremely ill-conditioned (k=100) matrices, fidelity is close to 0
- Sparsity does not affect the fidelity in this proof of principle example
- Noise affects fidelity strongly in each case
- The max depth of the circuit does not vary with different condition numbers and sparsity

Matrix size	2x2	4x4	8x8
Qubits	7	8	9
Max depth	101	104	111

Table 3.1: HHL circuit depth, circuit width

Chapter 4

Variational quantum linear solver

4.1 Variational algorithms

Variational algorithms are hybrid quantum-classical algorithms that can be utilized in order to solve general optimization problems as long as we can represent the problem as quantum observable. Variational algorithms are heuristic. In quantum mechanics, variational algorithms are commonly used to approximate the lowest energy state (ground state) of a quantum system for chemistry problems[19]. Variational algorithms can also be utilized for linear system solving[2]. The procedure of a variational algorithm is the following:

- Construct an Ansatz, which is trainable gate sequence with some parameters. One can vary both, the structure of the whole circuit and the parameters
- Make an initial guess by feeding the Ansatz with a quantum state
- The algorithm defines an efficient quantum circuit as the cost function
- The result of the cost function gets classically minimized. The classical optimizer returns the set of parameters which are then fed to the quantum Ansatz
- This loop continues until the cost function reaches an accepted minimum value

VQLS is a hybrid quantum-classical approach to the quantum linear system problem using both classical and quantum computing methods to solve linear systems of equations. VQLS is not an iteration on the HHL, but a proposed intermediate solution to HHL's high demand of qubits. The variational nature of the algorithm allows for it to be performed on NISQ(noisy intermediate-scale quantum) devices.

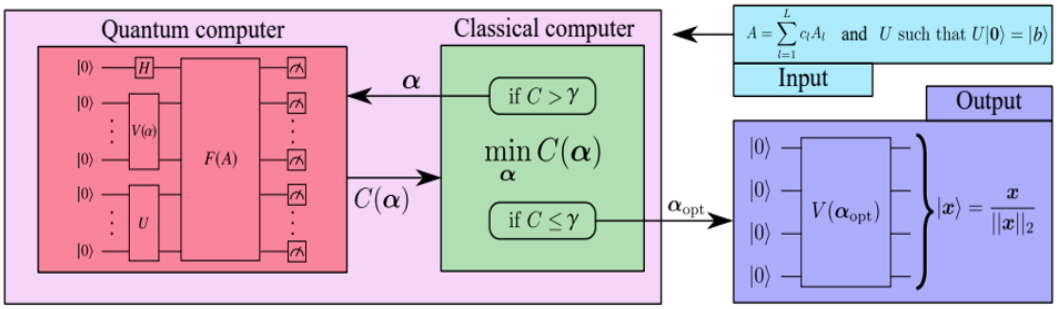


Figure 4.1: Variational quantum linear solver general scheme[2]

Inputs and outputs

Inputs:

- Matrix A written as a linear combination of unitaries $A = \sum_{l=1}^{L} c_l A_l$
- $|b\rangle$ which is prepared by a short-depth unitary transformation U such that: $U|0\rangle = |b\rangle$

Output:

- The output of VQLS is a quantum state $|x\rangle$ which is proportional to the solution x of the linear system: $Ax = b$

Hybrid optimization loop

The algorithm starts with the creation of an Ansatz V(a), which is a circuit that prepares an arbitrary state $|\psi(k)\rangle = V(a)|0\rangle$ as a potential solution $|x(a)\rangle = |\psi(k)\rangle = V(a)|0\rangle \quad a, k\, parameters.$ The VQLS algorithm defines a cost function in terms of overlap between the quantum states $|b\rangle\, and \frac{A|x\rangle}{\sqrt{\langle x|A^\dagger A|x\rangle}}$.

The result of the cost function C(a) is inputted to the classical device in order to be optimized via a classical optimization algorithm which adjusts a, k in attempt to reduce the value of the cost function. The process is iterated a number of times until the cost function's output reaches

a predetermined lower bound γ such that: $C(a) \leqslant \gamma$

Quantum Ansatz

The quantum Ansatz prepares a potential solution $|x(a)\rangle = \mathrm{V(a)}\,|0\rangle$ where V(a) is a trainable gate sequence. V(a) can be expressed as a sequence of L gates $V(a) = G_{K_L}(\theta_L)...G_{K_1}(\theta_1)$, where $G_K(\theta)$ is the k^{th} gate with input θ. Therefore a encompasses both k and θ(a=(k,θ)). k is a discrete parameter determining the circuit layout, k $\in \mathbb{N}$. θ are continuous parameters such as rotational angles of gates. If we choose to vary both of the parameters k,θ we get a variable structure ansatz and if we choose to vary one of the parameters k,θ we get a fixed structure Ansatz[20].

For this example we will use a fixed structure Ansatz so, the configuration of the gates remains the same for each run of the quantum circuit and all that changes are the θ parameters.

Cost function

The output of the quantum circuit is $|\Phi\rangle = \mathrm{A}\,|\psi(k)\rangle = \sum_{l=1}^{L} c_l A_l(V(a)\,|0\rangle)$. At this point we need to introduce an efficient cost function that comprises the overlap between a projector $|\psi\rangle\langle\psi|$ and the subspace orthogonal to $|b\rangle$ where $|\psi\rangle = \mathrm{A}\,|x\rangle$.

We want the cost function's output to be:

- very small when $|\Phi\rangle = \mathrm{A}\,|\psi(k)\rangle$ is almost parallel to $|b\rangle$
- very large when $|\Phi\rangle, |b\rangle$ are close to orthogonal

Here we need to introduce the projection Hamiltonian :$H_P = I - |b\rangle\langle b|$

$$C_P = \langle\Phi| H_P |\Phi\rangle = \langle\Phi| (I - |b\rangle\langle b|) |\Phi\rangle = \langle\Phi \mid \Phi\rangle - \langle\Phi \mid b\rangle\langle b \mid \Phi\rangle$$

From the latter equation arise 2 terms :

The first term is : $\langle\Phi \mid \Phi\rangle = \||\Phi\rangle\|^2$

The second term is: $\langle\Phi \mid b\rangle\langle b \mid \Phi\rangle = \|\langle b \mid \Phi\rangle\|^2$

Even if $\|\langle b \mid \Phi\rangle\|^2$ is small, which means $|\Phi\rangle, |b\rangle$ are close to orthogonal, if $\||\Phi\rangle\|^2$ is also small ($|\Phi\rangle$ has small norm) the cost function will still be low, which is not the desired result. So, if we want to make our cost function even better we can replace $|\Phi\rangle$ with $\frac{|\Phi\rangle}{\sqrt{\langle\Phi|\Phi\rangle}}$.

The revised cost function will be the following:

$$\hat{C}_P = \frac{\langle\Phi|}{\sqrt{\langle\Phi \mid \Phi\rangle}} H_P \frac{|\Phi\rangle}{\sqrt{\langle\Phi \mid \Phi\rangle}} = \frac{\langle\Phi|}{\sqrt{\langle\Phi \mid \Phi\rangle}} (I - |b\rangle\langle b|) \frac{|\Phi\rangle}{\sqrt{\langle\Phi \mid \Phi\rangle}} = \frac{\langle\Phi \mid \Phi\rangle}{\langle\Phi \mid \Phi\rangle} - \frac{\langle\Phi \mid b\rangle\langle b \mid \Phi\rangle}{\langle\Phi \mid \Phi\rangle} = 1 - \frac{\|\langle b \mid \Phi\rangle\|^2}{\||\Phi\rangle\|^2}$$

So, all we need to do is to calculate these two terms.

Hadamard test

The Hadamard test is a quantum subroutine that allows us to find the expectation value $\langle\psi| U |\psi\rangle$ of a unitary U with respect to a state $|\psi\rangle$.

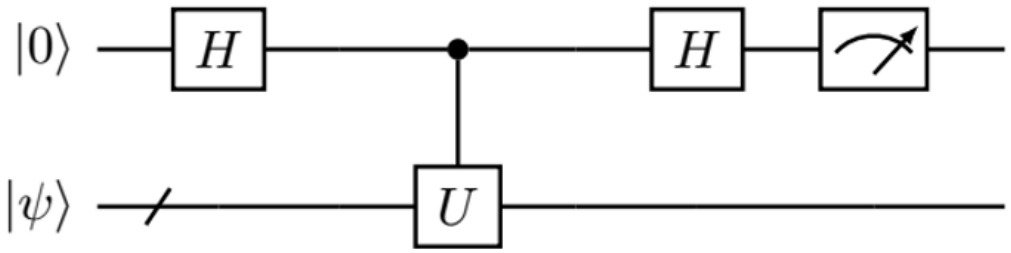


Figure 4.2: Hadamard test

$$|\psi_1\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \otimes |\psi\rangle = \frac{|0\rangle \otimes |\psi\rangle + |1\rangle \otimes |\psi\rangle}{\sqrt{2}}$$
$$|\psi_2\rangle = \frac{|0\rangle \otimes |\psi\rangle + |1\rangle \otimes U|\psi\rangle}{\sqrt{2}}$$
$$|\psi_3\rangle = \frac{1}{2}((|0\rangle + |1\rangle) \otimes |\psi\rangle + (|0\rangle - |1\rangle) \otimes U|\psi\rangle)$$
$$|\psi_3\rangle = \frac{|0\rangle \otimes |\psi\rangle + |1\rangle \otimes U|\psi\rangle}{\sqrt{2}} \to \frac{1}{2}[(|0\rangle + |1\rangle) \otimes |\psi\rangle + (|0\rangle - |1\rangle) \otimes U|\psi\rangle]$$

$$\Rightarrow \frac{1}{2}[|0\rangle \otimes |\psi\rangle + |1\rangle \otimes |\psi\rangle + |0\rangle \otimes U|\psi\rangle - |1\rangle \otimes U|\psi\rangle] = \frac{1}{2}[|0\rangle \otimes (I+U)|\psi\rangle + |1\rangle \otimes (I-U)|\psi\rangle]$$

$$P(0_1) = \langle\psi_3| P_0 \otimes I |\psi_3\rangle$$

$$P_0 \otimes I |\psi_3\rangle = |0\rangle\langle 0| \otimes I |\psi_3\rangle = |0\rangle\langle 0| \otimes I\frac{1}{2}[|0\rangle \otimes (I+U)|\psi\rangle + |1\rangle \otimes (I-U)|\psi\rangle]$$
$$= \frac{1}{2}[|0\rangle\langle 0||0\rangle \otimes (I+U)|\psi\rangle + |0\rangle\langle 0||1\rangle \otimes (I-U)|\psi\rangle] = \frac{1}{2}[|0\rangle \otimes (I+U)|\psi\rangle]$$

$$P(0_1) = \langle\psi_3| P_0 \otimes I |\psi_3\rangle = \frac{1}{2}[\langle 0| \otimes \langle\psi|(I+U^\dagger) + \langle 1| \otimes \langle\psi|(I-U^\dagger)]\ \frac{1}{2}[|0\rangle \otimes (I+U)|\psi\rangle]$$
$$= \frac{1}{4}[\langle 0||0\rangle \otimes \langle\psi|(I+U^\dagger)(I+U)|\psi\rangle] = \frac{1}{4}[\langle\psi|(I+U^\dagger)(I+U)|\psi\rangle] = \frac{1}{4}[\langle\psi| I^2 + U + U^\dagger + UU^\dagger|\psi\rangle]$$
$$= \frac{1}{4}[\langle\psi| 2I + U + U^\dagger|\psi\rangle] = \frac{1}{4}[2 + \langle\psi| U|\psi\rangle^* + \langle\psi| U|\psi\rangle] = \frac{1}{2}(1 + \mathrm{Re}\langle\psi| U|\psi\rangle)$$

$$\bullet P(1_1) = \frac{1}{2}(1 - \mathrm{Re}\langle\psi| U|\psi\rangle)$$

$$P(0_1) - P(1_1) = \frac{1}{2}(1 + \mathrm{Re}\langle\psi| U|\psi\rangle) - \frac{1}{2}(1 - \mathrm{Re}\langle\psi| U|\psi\rangle) = \mathrm{Re}\langle\psi| U|\psi\rangle$$

We want to calculate the term: $\langle\Phi \mid \Phi\rangle = \||\Phi\rangle\|^2$

$$\bullet\ |\Phi\rangle = \mathrm{A}|\psi(k)\rangle\ ,\ |\psi(k)\rangle = V(k)|0\rangle\ ,\ A = \sum_{l=1}^{L} c_l A_l$$

$$\langle\Phi \mid \Phi\rangle = \langle\psi(k)| A^\dagger A|\psi(k)\rangle = \langle 0| V(k)^\dagger A^\dagger A V(k)|0\rangle = \langle 0| V(k)^\dagger (\sum_n c_n A_n)^\dagger \sum_n c_n A_n V(k)|0\rangle$$

$$\langle\Phi \mid \Phi\rangle = \sum_m \sum_n {c_m}^* c_n \langle 0| V(k)^\dagger {A_m}^\dagger A_n V(k)|0\rangle$$

We want to calculate the term: $\|\langle b \mid \Phi\rangle\|^2$

$$\|\langle b \mid \Phi\rangle\|^2 = |\langle b| AV(k) |0\rangle|^2 = \left|\langle 0| U^\dagger AV(k) |0\rangle\right|^2$$
$$= \langle 0| U^\dagger AV(k) |0\rangle \left(\langle 0| U^\dagger AV(k) |0\rangle\right)^\dagger$$
$$= \langle 0| U^\dagger AV(k) |0\rangle \langle 0| V(k)^\dagger A^\dagger U |0\rangle$$
$$= \sum_m \sum_n {c_m}^* c_n \langle 0| U^\dagger A_n V(k) |0\rangle \langle 0| V(k)^\dagger {A_m}^\dagger U |0\rangle$$

For this Qiskit demonstration: $\langle 0| U^\dagger AV(k) |0\rangle = \left(\langle 0| U^\dagger AV(k) |0\rangle\right)^\dagger = \langle 0| V(k)^\dagger A^\dagger U |0\rangle$

So, $\|\langle b \mid \Phi\rangle\|^2 = \sum_m \sum_n c_m c_n \langle 0| U^\dagger A_n V(k) |0\rangle \langle 0| U^\dagger A_m V(k) |0\rangle$

4.2 VQLS implementation

We ran the VQLS algorithm in Qiskit utilizing the "COBYLA" minimizer with maximum iterations=200[21]. We ran it first without noise using the state_vector_simulator as backend and then we added noise using the noise model from ibmq_quito.

We did not run any simulation on quantum hardware because it would take too much time due to backend's queue and the required iterations.

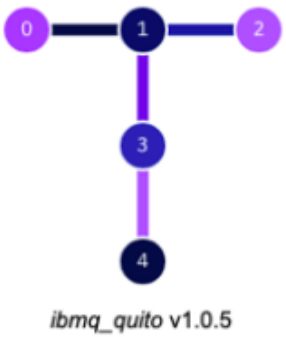


Figure 4.3: ibmq_quito circuit comprising of 5 qubits connected as in the figure

```
In [27]: noise_model.basis_gates
Out[27]: ['cx', 'id', 'reset', 'rz', 'sx', 'x']

In [29]: noise_model.noise_instructions
Out[29]: ['cx', 'id', 'measure', 'reset', 'sx', 'x']

noise_model.noise_qubits
[0, 1, 2, 3, 4]
```

Figure 4.4: ibmq_quito noise model

The measured variables for the subsequent simulations are the following:

- fidelity: $fid\,(x,y) = Tr\left(\sqrt{\sqrt{x}y\sqrt{x}}\right)^2$, $x = \frac{VQLS}{\|VQLS\|_2}$, $y = \frac{classical}{\|classical\|_2}$
- Circuit depth
- Circuit width(minimum qubits needed)

The test matrices are on the appendix A.2

2x2 matrix implementation in Qiskit

We ran the VQLS algorithm from IBM Qiskit textbook [21] with the above arguments for 6 different 2x2 matrices with condition numbers (1, 1.5, 2, 5, 10, 100) and s=1, 2 to see what is the correlation between condition number, sparsity and fidelity. As backend we have used the 'state_vector_simulator'.

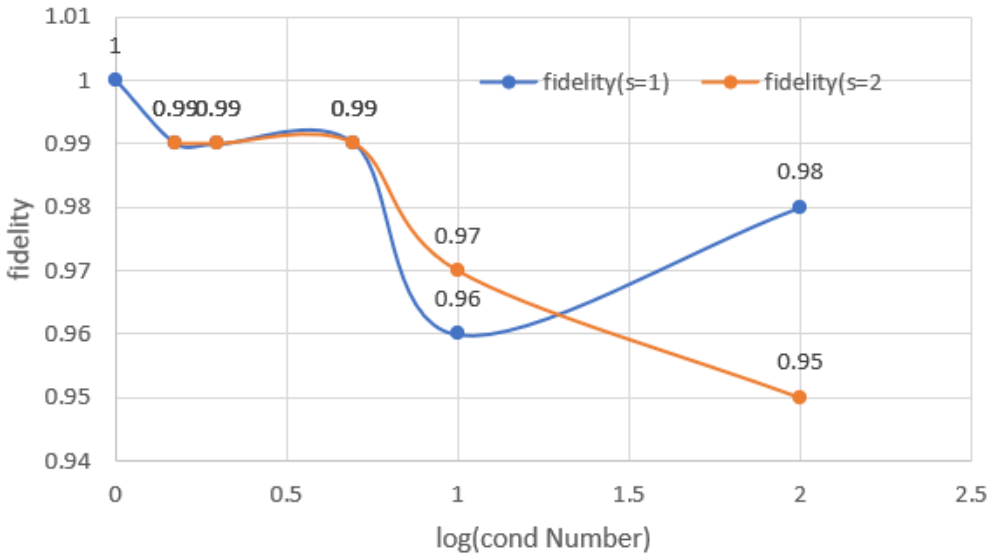


Figure 4.5: VQLS 2x2 simulation, s=1, 2

Afterwards we ran the VQLS algorithm from IBM Qiskit textbook [21] for 6 different 2x2 matrices with condition numbers (1, 1.5, 2, 5, 10, 100) and s=1, 2 adding noise, to see how the

additive noise affects fidelity. We have used the noise model from ibmq_quito and the results are presented below:

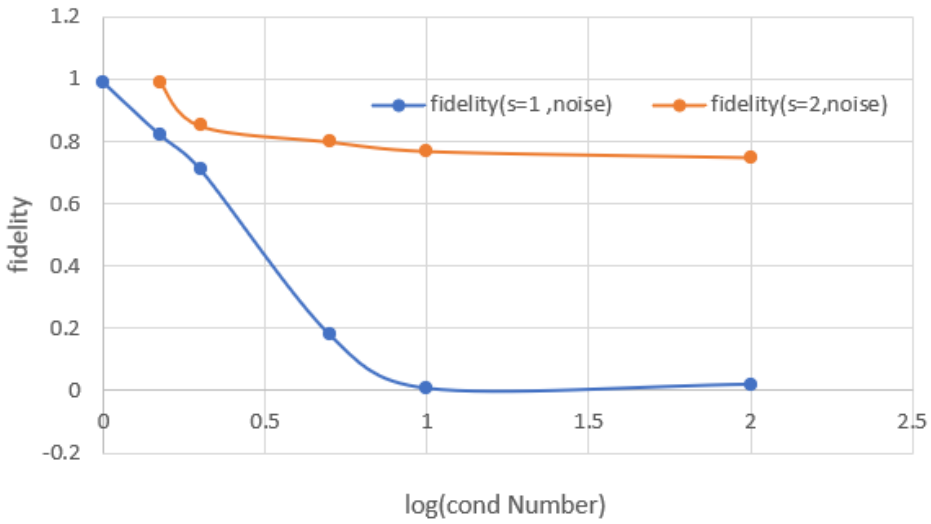


Figure 4.6: VQLS 2x2 noisy simulation s=1, 2

4x4 matrix implementation in Qiskit We ran the VQLS algorithm from IBM Qiskit textbook [21] with the above arguments for 6 different 4x4 matrices with condition numbers (1, 1.5, 2, 5, 10, 100) and s=1, 2 to see what is the correlation between condition number, sparsity and fidelity. As backend we have used the 'state_vector_simulator' and the results are presented below:

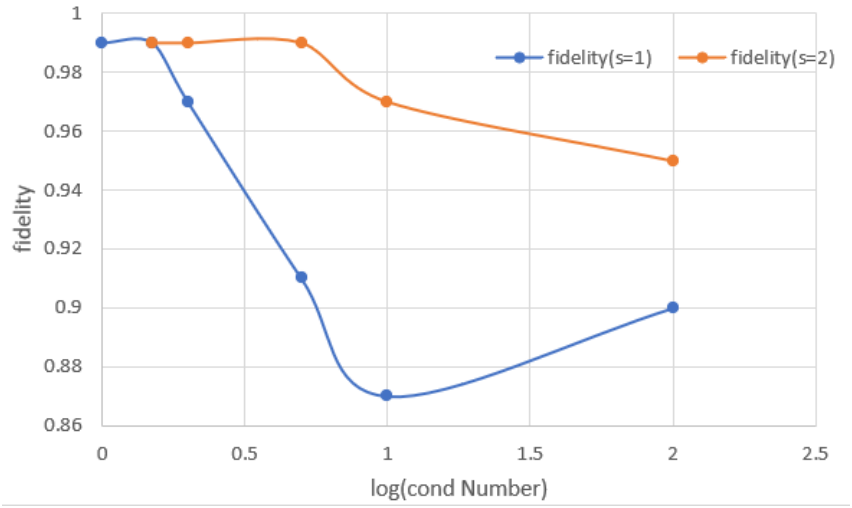


Figure 4.7: VQLS 4x4 simulation s=1, 2

Afterwards, we ran the VQLS algorithm from IBM Qiskit textbook [21] for 6 different 4x4 matrices with condition numbers (1, 1.5, 2, 5, 10, 100) and s=1, 2 adding noise, to see how the additive noise affects fidelity. We have used the noise model from ibmq_quito and the results are presented below:

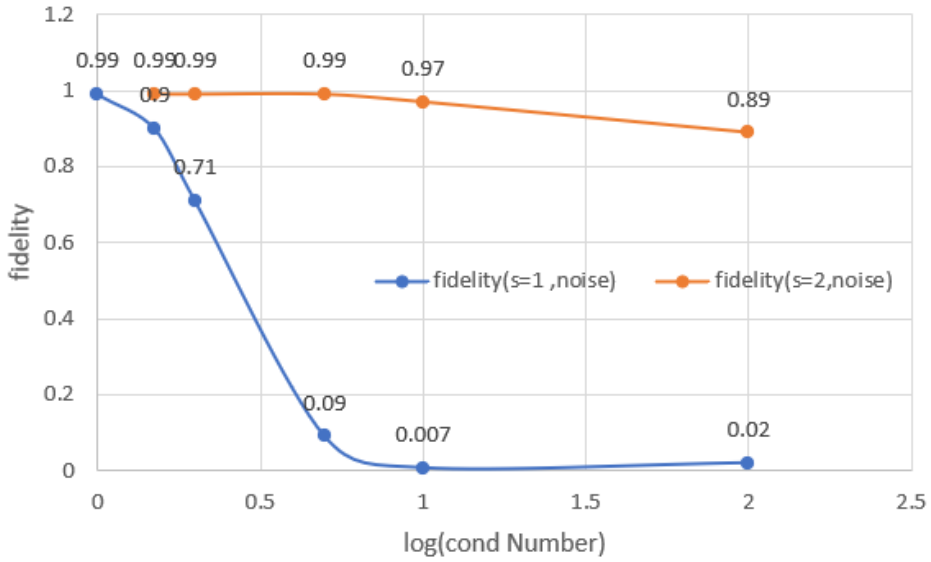


Figure 4.8: VQLS 4x4 noisy simulation s=1, 2

8x8 matrix implementation in Qiskit

We ran the VQLS algorithm from IBM Qiskit textbook [21] with the above arguments for 6 diagonal 8x8 matrices with condition numbers (1, 1.5, 2, 5, 10, 100) initially without and then with noise to see what is the correlation between condition number, noise and fidelity.

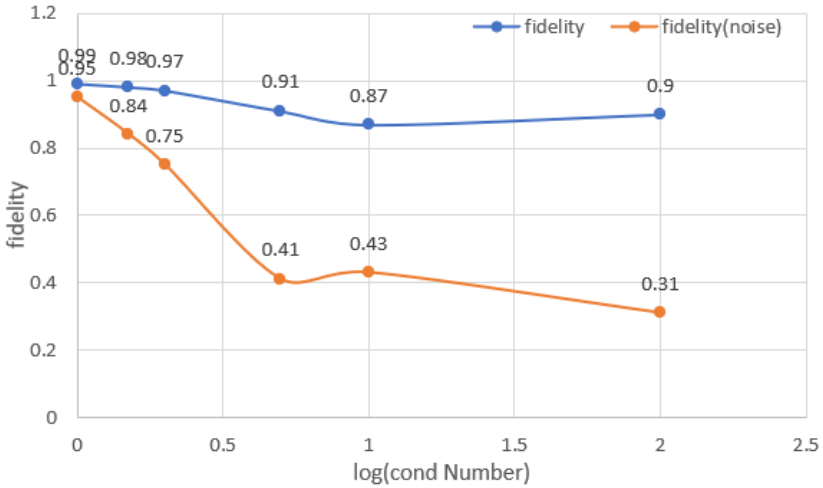


Figure 4.9: VQLS 8x8 diagonal matrices with and without noise simulation

VQLS implementation-results analysis

After experimenting with different matrix sizes, condition numbers and sparsities we have reached the following conclusions:

- As expected, similarly to HHL, for the VQLS the condition number relates strongly to fidelity. The bigger the condition number, the lower the fidelity
- The max depth of the circuit does not vary with different condition numbers and sparsity
- Noise does affect the results fidelity but not as much as in the HHL simulations. This is expected since we already know that variational algorithms can work efficiently on NISQ devices
- Circuit depth may seem small but we iterate on it multiple times

Matrix size	2x2	4x4	8x8
Qubits	3	4	5
Max depth	8	10	26

Table 4.1: VQLS circuit depth, width

Conclusion

In the course of this thesis we have experimented on many different simulations on 2 different approaches to the quantum linear system problem. HHL is a fully quantum algorithm while VQLS is a hybrid approach that utilizes both quantum properties and classical optimizers. At this moment neither of those algorithms can provide any significant advantage over their classical counterparts. HHL has a complexity of $\mathrm{O}\left(\log\left(N\right)k^2s^2\frac{1}{\varepsilon}\right)$ which scales extremely promisingly on number of variables but is too sensitive to error and condition number. On the other hand, VQLS has heuristic scaling and is less sensitive to error and condition number. Another vital difference between these two algorithms is that of the qubit dependency. HHL requires 2n+1 qubits and many steps(big circuit depth) to produce results. The quantum phase estimation part of HHL (conditional Hamiltonian evolution) is the most demanding on computational steps. VQLS requires less qubits and has a significantly shallower circuit which makes it much more feasible to run on near term quantum hardware. Furthermore both these algorithms insert some restrictions when it comes to the input data such as square Hermitian matrices. Another problem that arises when it comes to input data is that of the quantum representation of the original data and also that of the storing of these data.

Quantum linear solvers and variational approaches on optimization problems are in the midst of the ongoing research and may soon with the improvement of quantum hardware and the further development of efficient algorithms be able to solve real life problems and change the way we solve linear systems of equations.

The linear system problem (LSP) is a very common mathematical problem that arises both on each own and as a subroutine in more complex problems. Linear systems of equations can be found in problems of various different fields, such as engineering, physics, chemistry and

economics. Quantum linear system solving algorithms like HHL can estimate exponentially faster useful features of the solution of a linear system. For this reason it has many applications. HHL managed to solve the radar cross-section of a complex shape which is a problem that appears in electromagnetic scattering [22] while providing exponential speedups over its classical counterpart. Other notable applications of HHL are the finite element method and the linear differential equation solving. The quantum algorithm for linear systems of equations is applicable for machine learning tasks. Tasks in machine learning frequently involve manipulating and classifying a large volume of data in high-dimensional vector spaces. Quantum computers can process these high-dimensional vectors using tensor product spaces. Quantum linear system solver has already been applied to a support vector machine. Rebentrost et al.[23] has shown that quantum support vector machine can be used for big data classification and achieve an exponential speedup over classical methods.

Bibliography

[1] A. W. Harrow, A. Hassidim, and S. Lloyd, *Quantum algorithm for linear systems of equations*, Physical review letters **103**, 150502 (2009).

[2] C. Bravo-Prieto, R. LaRose, M. Cerezo, Y. Subasi, L. Cincio, and P. J. Coles, *Variational quantum linear solver*, arXiv preprint arXiv:1909.05820 (2019).

[3] R. P. Feynman, *Simulating Physics with Computers*, International Journal of Theoretical Physics **21**, 467–488 (1982).

[4] D. Deutsch and R. Jozsa, *Rapid solution of problems by quantum computation*, Proceedings of the Royal Society of London. Series A: Mathematical and Physical Sciences **439**, 553–558 (1992).

[5] P. W. Shor. *Algorithms for quantum computation: discrete logarithms and factoring.* In *Proceedings 35th annual symposium on foundations of computer science*, pages 124–134. Ieee, (1994).

[6] L. 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).

[7] E. Pednault, J. A. Gunnels, G. Nannicini, L. Horesh, and R. Wisnieff, *Leveraging secondary storage to simulate deep 54-qubit sycamore circuits*, arXiv preprint arXiv:1910.09534 (2019).

[8] *Qubits and Quantum States*, chapter 2, pages 11–37. John Wiley and Sons, Ltd, (2007).

[9] *Matrices and Operators*, chapter 3, pages 39–72. John Wiley & Sons, Ltd, (2007).

[10] V. Bužek and M. Hillery, *Quantum copying: Beyond the no-cloning theorem*, Physical Review A **54**, 1844 (1996).

[11] Y. S. Weinstein, M. Pravia, E. Fortunato, S. Lloyd, and D. G. Cory, *Implementation of the quantum Fourier transform*, Physical review letters **86**, 1889 (2001).

[12] A. Asfaw, A. Corcoles, L. Bello, Y. Ben-Haim, M. Bozzo-Rey, S. Bravyi, N. Bronn, L. Capelluto, A. C. Vazquez, J. Ceroni, R. Chen, A. Frisch, J. Gambetta, S. Garion, L. Gil, S. D. L. P. Gonzalez, F. Harkins, T. Imamichi, H. Kang, A. h. Karamlou, R. Loredo, D. McKay, A. Mezzacapo, Z. Minev, R. Movassagh, G. Nannicini, P. Nation, A. Phan, M. Pistoia, A. Rattew, J. Schaefer, J. Shabani, J. Smolin, J. Stenger, K. Temme, M. Tod, S. Wood, and J. Wootton. *Learn Quantum Computation Using Qiskit*, (2020).

[13] A. Y. Kitaev, *Quantum measurements and the Abelian stabilizer problem*, arXiv preprint quant-ph/9511026 (1995).

[14] H. Lee and M. Scully, *The Physics of EIT and LWI in V-Type Configurations*, Found. Phys. **28**, 585–600 (1998).

[15] M. R. Hestenes and E. Stiefel, *Methods of conjugate gradients for solving linear systems*, Journal of Research of the National Bureau of Standards **49**, 409–436 (1952).

[16] T. Q. Team. *Solving Linear Systems of Equations using HHL*, (2021).

[17] T. Q. Team. *Solving linear systems of equations using HHL*, (2021).

[18] A. Carrera Vazquez, A. Frisch, D. Steenken, H. Barowski, R. Hiptmair, and S. Woerner, *Enhancing Quantum Linear System Algorithm by Richardson Extrapolation*, Bulletin of the American Physical Society **65** (2020).

[19] A. Peruzzo, J. McClean, P. Shadbolt, M.-H. Yung, X.-Q. Zhou, P. J. Love, A. Aspuru-Guzik, and J. L. O'brien, *A variational eigenvalue solver on a photonic quantum processor*, Nature communications **5**, 1–7 (2014).

[20] S. Hadfield, Z. Wang, B. O'Gorman, E. G. Rieffel, D. Venturelli, and R. Biswas, *From the quantum approximate optimization algorithm to a quantum alternating operator ansatz*, Algorithms **12**, 34 (2019).

[21] T. Q. Team. *Variational Quantum Linear Solver*, (2021).

[22] B. D. Clader, B. C. Jacobs, and C. R. Sprouse, *Preconditioned quantum linear system algorithm*, Physical review letters **110**, 250504 (2013).

[23] P. Rebentrost, M. Mohseni, and S. Lloyd, *Quantum support vector machine for big data classification*, Physical review letters **113**, 130503 (2014).

Appendix A

Appendices

A.1 Test Matrices

2x2 Test Matrices

$$md21 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, s = 1, c = 1$$

$$md22 = \begin{bmatrix} 1 & 0 \\ 0 & 0,66 \end{bmatrix}, s = 1, c = 1.5$$

$$md23 = \begin{bmatrix} 1 & 0 \\ 0 & 0,5 \end{bmatrix}, s = 1, c = 2$$

$$md24 = \begin{bmatrix} 1 & 0 \\ 0 & 0,2 \end{bmatrix}, s = 1, c = 5$$

$$md25 = \begin{bmatrix} 1 & 0 \\ 0 & 0,1 \end{bmatrix}, s = 1, c = 10$$

$$md26 = \begin{bmatrix} 1 & 0 \\ 0 & 0,01 \end{bmatrix}, s = 1, c = 100$$

$$m22 = \begin{bmatrix} 1 & 0.2 \\ 0.2 & 1 \end{bmatrix}, s = 2, c = 1.5$$

$$m23 = \begin{bmatrix} 0.75 & 0.25 \\ 0.25 & 0.75 \end{bmatrix}, s = 2, c = 2$$

$$m24 = \begin{bmatrix} 1 & 0.666 \\ 0.666 & 1 \end{bmatrix}, s = 2, c = 5$$

$$m25 = \begin{bmatrix} 0.409 & 0.5 \\ 0.5 & 0.409 \end{bmatrix}, s = 2, c = 10.2$$

$$m26 = \begin{bmatrix} 0.5 & 0.49 \\ 0.49 & 0.5 \end{bmatrix}, s = 2, c = 100.2$$

4x4 Test Matrices

$$md41 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, s = 1, c = 1 \qquad md44 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0.2 \end{bmatrix}, s = 1, c = 5$$

$$md42 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0.666 \end{bmatrix}, s = 1, c = 1.5 \qquad md45 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0.1 \end{bmatrix}, s = 1, c = 10$$

$$md43 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0.5 \end{bmatrix}, s = 1, c = 2 \qquad md46 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0.01 \end{bmatrix}, s = 1, c = 100$$

Figure A.1: 4x4 diagonal Test Matrices

$$m42 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0.2 & 0 \\ 0 & 0.2 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, s = 2, c = 1.5$$

$$m43 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0.333 & 0 \\ 0 & 0.333 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, s = 2, c = 2$$

$$m44 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0.666 & 0 \\ 0 & 0.666 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, s = 2, c = 5$$

$$m45 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0.818 & 0 \\ 0 & 0.818 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, s = 2, c = 10$$

$$m46 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0.98 & 0 \\ 0 & 0.98 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, s = 2, c = 100$$

$$m43s3 = \begin{bmatrix} 1 & 0.75 & 1 & 0 \\ 0.75 & 1 & 0.325 & 0 \\ 0 & 0.325 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, s = 2, c = 5$$

$$m43s4 = \begin{bmatrix} 1 & 0.76 & 0.05 & 0.15 \\ 0.76 & 1 & 1.5 & 0.15 \\ 0.05 & 1.5 & 1 & 0.1 \\ 0.15 & 0.15 & 0.1 & 1 \end{bmatrix}, s = 2, c = 5$$

Figure A.2: 4x4 non diagonal Test Matrices

8x8 Test Matrices

$$md81=\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&1&0\\0&0&0&0&0&0&0&1\end{bmatrix},s=1,c=1 \qquad md84=\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&0.2&0&0&0\\0&0&0&0&0&0.2&0&0\\0&0&0&0&0&0&0.2&0\\0&0&0&0&0&0&0&0.2\end{bmatrix},s=1,c=5$$

$$md82=\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&0.666&0&0&0\\0&0&0&0&0&0.666&0&0\\0&0&0&0&0&0&0.666&0\\0&0&0&0&0&0&0&0.666\end{bmatrix},s=1,c=1.5 \qquad md85=\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&0.1&0&0&0\\0&0&0&0&0&0.1&0&0\\0&0&0&0&0&0&0.1&0\\0&0&0&0&0&0&0&0.1\end{bmatrix},s=1,c=10$$

$$md83=\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&0.5&0&0&0\\0&0&0&0&0&0.5&0&0\\0&0&0&0&0&0&0.5&0\\0&0&0&0&0&0&0&0.5\end{bmatrix},s=1,c=2 \qquad md86=\begin{bmatrix}1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&0.01&0&0&0\\0&0&0&0&0&0.01&0&0\\0&0&0&0&0&0&0.01&0\\0&0&0&0&0&0&0&0.01\end{bmatrix},s=1,c=100$$

Figure A.3: 8x8 diagonal Test Matrices

A.2 Code

QFT

```
from qiskit import QuantumCircuit,QuantumRegister,ClassicalRegister, Aer, assemble ,execute ,transpile
from math import pi
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from qiskit.visualization import plot_histogram, plot_bloch_multivector
from qiskit.quantum_info import Statevector
from qiskit_textbook.tools import array_to_latex
from qiskit.providers.ibmq import least_busy
from qiskit.tools.monitor import job_monitor
style = {'backgroundcolor': 'lightgreen'}
# oi 3 simulators
svsim = Aer.get_backend('statevector_simulator')
usim = Aer.get_backend('unitary_simulator')
qasm_sim = Aer.get_backend('qasm_simulator')
shots=1000
from qiskit.providers import ibmq
```

```
def qft_rotations(circ, n):
    if n == 0:
        return circ
    n -= 1
    circ.h(n)
    for qubit in range(n):
        circ.crz(pi/2**(n-qubit), qubit, n)
    qft_rotations(circ, n)
```

```
def swap_registers(circ, n):
    """Swapping order of qubits to fix qiskit natural ordering upon measurement"""
    for qubit in range(n//2):
        circ.swap(qubit, n-qubit-1)
    return circ

def qft_full(circ, n):
    """QFT on the first n qubits in circ"""
    qft_rotations(circ, n)
    swap_registers(circ, n)
    return circ
```

```
q = QuantumRegister(4)
c = ClassicalRegister(4)
circ = QuantumCircuit(q,c)
circ.x(q[3])

test = transpile(circ,svsim)
qobj = assemble(test)
initial_state = svsim.run(qobj).result().get_statevector()
plot_bloch_multivector(initial_state, reverse_bits=True)
```

QPE

```
from qiskit import QuantumCircuit,QuantumRegister,ClassicalRegister, Aer, assemble ,execute ,transpile, IBMQ
from math import e
from math import pi
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from qiskit.visualization import plot_histogram, plot_bloch_multivector
from qiskit.quantum_info import Statevector
from qiskit_textbook.tools import array_to_latex
from qiskit.providers.ibmq import least_busy
from qiskit.tools.monitor import job_monitor

style = {'backgroundcolor': 'lightgreen'}

# oi 3 simulators
svsim = Aer.get_backend('statevector_simulator')
usim = Aer.get_backend('unitary_simulator')
qasm_sim = Aer.get_backend('qasm_simulator')

shots=1000 #gia to qasm_simulator , poses fores na trexei

from qiskit.providers import ibmq
n = 6

IBMQ.save_account('2f24d8fe65f0ee08e556cf3cbad237bae71a413385f38482025ae7358723491c34b2134d05caa9bc5f5a1fae561cdf688
```

```
#Initialization
qpe = QuantumCircuit(n,n-1)
for qubit in range(n-1):
    qpe.h(qubit)
qpe.x(n-1)
qpe.barrier()

qpe.draw(output='mpl',style=style)

c_u_rotations(qpe,n,float(31/32))
qpe.barrier()
qpe.draw(output='mpl',style=style)


qft_dagger(qpe,n-1)


qpe.barrier()
for qubit in range(n-1):
    qpe.measure(qubit,qubit)

qasm_sim = Aer.get_backend('qasm_simulator')
shots = 2048
t_qpe = transpile(qpe, qasm_sim)
qobj = assemble(t_qpe, shots=shots)
results = qasm_sim.run(qobj).result()
answer = results.get_counts()
```

```
plot_histogram(answer)

Provider = IBMQ.get_provider(hub='ibm-q')
Pprovider.backends()

provider = IBMQ.get_provider(hub='ibm-q')
backend = least_busy(provider.backends(filters=lambda x: x.configuration().n_qubits >= (n+1) and
                                    not x.configuration().simulator and x.status().operational==True))
print("least busy backend: ", backend)


shots = 1024
transpiled_qc = transpile(qpe, backend)
qobj = assemble(transpiled_qc, backend)
job = backend.run(qobj)
job_monitor(job, interval=2)

results = job.result()
answer = results.get_counts()

plot_histogram(answer)
```

HHL

```
from qiskit import Aer
from qiskit.circuit.library import QFT
from qiskit.aqua import QuantumInstance, aqua_globals
from qiskit.quantum_info import state_fidelity
from qiskit.aqua.algorithms import HHL, NumPyLSsolver
from qiskit.aqua.components.eigs import EigsQPE
from qiskit.aqua.components.reciprocals import LookupRotation
from qiskit.aqua.operators import MatrixOperator
from qiskit.aqua.components.initial_states import Custom
import numpy as np
from numpy import linalg as LA
import time
from pinak import *
```

```
def HHLalg(len(Amatrix),matrixname, Amatrix, bvector, FileName,
    noise = False,
    Nmodel = [],
    Bgates = [],
    num_ancillae = 3,
    negative_evals = False,
    expansion_mode = 'suzuki',
    num_time_slices = 50,
    order = 1):
    Amatrix, vector, truncate_powerdim, truncate_hermitian = HHL.matrix_resize(Amatrix, bvector)

    ne_qfts = [None, None]
    orig_size=len( Amatrix)
    if negative_evals:
        num_ancillae += 1
        ne_qfts = [QFT(num_ancillae-1), QFT(num_ancillae-1).inverse()]
```

```
    orig_size=len( Amatrix)
    if negative_evals:
        num_ancillae += 1
        ne_qfts = [QFT(num_ancillae-1), QFT(num_ancillae-1).inverse()]

    eigs = EigsQPE(MatrixOperator(matrix=Amatrix),
                   QFT(num_ancillae).inverse(),
                   num_time_slices=num_time_slices,
                   num_ancillae=num_ancillae,
                   expansion_mode='suzuki',
                   expansion_order = order,
                   evo_time=None,
                   negative_evals=negative_evals,
                   ne_qfts=ne_qfts)
    # Get total number of qubtis and ancillae
    num_q, num_a = eigs.get_register_sizes()
    # Initialize initial state module
    init_state = Custom(num_q, state_vector=bvector)
    # Initialize reciprocal rotation module
    reciprocal = LookupRotation(negative_evals=eigs._negative_evals,evo_time=eigs._evo_time)


    HHL_Setup = HHL(Amatrix, bvector, truncate_powerdim, truncate_hermitian,
        eigs, init_state, reciprocal, num_q, num_a, orig_size)
    if (noise):
        HHL_Setup.set_backend(Aer.get_backend('qasm_simulator'),
            basis_gates=Bgates, noise_model=Nmodel)
    else:
        HHL_Setup.set_backend(Aer.get_backend('statevector_simulator'))

    HHL_output = HHL_Setup.run()

    # Calculate classical results and fidelity with quantum solution
    result_classical = NumPyLSsolver(Amatrix, vector).run()
    Fi = fidelity(HHL_output['solution'], result_classical['solution'])
```

VQLS

```
import qiskit
from qiskit import QuantumCircuit, QuantumRegister, ClassicalRegister
from qiskit import Aer, transpile, assemble,execute
import math
import random
import numpy as np
from scipy.optimize import minimize
import time
from qiskit.quantum_info import state_fidelity
from qiskit.aqua.algorithms import HHL, NumPyLSsolver
from pinak import *
```

```
def control_b(auxiliary, qubits):

    for ia in qubits:
        circ.ch(auxiliary, ia)
```

```
def apply_fixed_ansatz(qubits, parameters):
    if len(qubits)==3 :
        for iz in range(0, len(qubits)):
            circ.ry(parameters[0][iz], qubits[iz])
        circ.cz(qubits[0], qubits[1])
        circ.cz(qubits[2], qubits[0])
        for iz in range(0, len(qubits)):
            circ.ry(parameters[1][iz], qubits[iz])
        circ.cz(qubits[1], qubits[2])
        circ.cz(qubits[2], qubits[0])
        for iz in range(0, len(qubits)):
            circ.ry(parameters[2][iz], qubits[iz])
    elif len(qubits)==2 :
        for iz in range(len(qubits)):
```

```
            circ.ry(parameters[0][iz], qubits[iz])
        circ.cz(qubits[0], qubits[1])
        for iz in range(len(qubits)):
            circ.ry(parameters[1][iz], qubits[iz])
    elif len(qubits)==1 :
        circ.ry(parameters[0][0], qubits[0])
```

```
def control_fixed_ansatz(qubits, parameters, auxiliary, reg):
    if len(qubits)== 3:
        for i in range (0, len(qubits)):
            circ.cry(parameters[0][i], qiskit.circuit.Qubit(reg, auxiliary), qiskit.circuit.Qubit(reg, qubits[i]))

        circ.ccx(auxiliary, qubits[1], 4)
        circ.cz(qubits[0], 4)
        circ.ccx(auxiliary, qubits[1], 4)

        circ.ccx(auxiliary, qubits[0], 4)
        circ.cz(qubits[2], 4)
        circ.ccx(auxiliary, qubits[0], 4)

        for i in range (0, len(qubits)):
            circ.cry(parameters[1][i], qiskit.circuit.Qubit(reg, auxiliary), qiskit.circuit.Qubit(reg, qubits[i]))

        circ.ccx(auxiliary, qubits[2], 4)
        circ.cz(qubits[1], 4)
        circ.ccx(auxiliary, qubits[2], 4)

        circ.ccx(auxiliary, qubits[0], 4)
        circ.cz(qubits[2], 4)
        circ.ccx(auxiliary, qubits[0], 4)

        for i in range (0, len(qubits)):
            circ.cry(parameters[2][i], qiskit.circuit.Qubit(reg, auxiliary), qiskit.circuit.Qubit(reg, qubits[i]))
```

```
        for i in range(nrqubits):
            circ.cry(parameters[0][i], qiskit.circuit.Qubit(reg, auxiliary), qiskit.circuit.Qubit(reg, qubits[i]))
        circ.ccx(auxiliary, qubits[1], nrqubits+1)
        circ.cz(qubits[0], nrqubits+1)
        circ.ccx(auxiliary, qubits[1], nrqubits+1)
        for i in range(nrqubits):
            circ.cry(parameters[1][i], qiskit.circuit.Qubit(reg, auxiliary), qiskit.circuit.Qubit(reg, qubits[i]))
    elif len(qubits)== 1:
        nrqubits = len(qubits)
        circ.cry(parameters[0][0], qiskit.circuit.Qubit(reg, auxiliary),qiskit.circuit.Qubit(reg, qubits[0]))
        circ.ccx(auxiliary, qubits[0], nrqubits+1)
        circ.cz(qubits[0], nrqubits+1)
        circ.ccx(auxiliary, qubits[0], nrqubits+1)
```

```
def special_had_test(gate_type, qubits, auxiliary_index, parameters, reg):

    circ.h(auxiliary_index)

    control_fixed_ansatz(qubits, parameters, auxiliary_index, reg)

    for ty in range (0, len(gate_type)):
        if (gate_type[ty] == 1):
            circ.cz(auxiliary_index, qubits[ty])


    control_b(auxiliary_index, qubits)

    circ.h(auxiliary_index)
```

```
def had_test(gate_type, qubits, auxiliary_index, parameters):

    circ.h(auxiliary_index)

    apply_fixed_ansatz(qubits, parameters)

    for ie in range (0, len(gate_type[0])):
        if (gate_type[0][ie] == 1):
            circ.cz(auxiliary_index, qubits[ie])

    for ie in range (0, len(gate_type[1])):
        if (gate_type[1][ie] == 1):
            circ.cz(auxiliary_index, qubits[ie])

    circ.h(auxiliary_index)
```

```
def calculate_cost_function(parameters):

    global opt

    overall_sum_1 = 0

    parameters = [parameters[0:3], parameters[3:6], parameters[6:9]]

    for i in range(0, len(gate_set)):
        for j in range(0, len(gate_set)):

            global circ

            qctl = QuantumRegister(5)
            qc = ClassicalRegister(5)
            circ = QuantumCircuit(qctl, qc)
```

```
        backend = Aer.get_backend('aer_simulator')

        multiply = coefficient_set[i]*coefficient_set[j]

        had_test([gate_set[i], gate_set[j]], [1, 2, 3], 0, parameters)

        circ.save_statevector()
        t_circ = transpile(circ, backend)
        qobj = assemble(t_circ)
        job = backend.run(qobj)

        result = job.result()
        outputstate = np.real(result.get_statevector(circ, decimals=100))
        o = outputstate

        m_sum = 0
        for l in range (0, len(o)):
            if (l%2 == 1):
                n = o[l]**2
                m_sum+=n

        overall_sum_1+=multiply*(1-(2*m_sum))

overall_sum_2 = 0

for i in range(0, len(gate_set)):
    for j in range(0, len(gate_set)):

        multiply = coefficient_set[i]*coefficient_set[j]
        mult = 1

        for extra in range(0, 2):

            qctl = QuantumRegister(5)
            qc = ClassicalRegister(5)
```

```
            if (extra == 0):
                special_had_test(gate_set[i], [1, 2, 3], 0, parameters, qctl)
            if (extra == 1):
                special_had_test(gate_set[j], [1, 2, 3], 0, parameters, qctl)

            circ.save_statevector()
            t_circ = transpile(circ, backend)
            qobj = assemble(t_circ)
            job = backend.run(qobj)

            result = job.result()
            outputstate = np.real(result.get_statevector(circ, decimals=100))
            o = outputstate

            m_sum = 0
            for l in range (0, len(o)):
                if (l%2 == 1):
                    n = o[l]**2
                    m_sum+=n
            mult = mult*(1-(2*m_sum))

        overall_sum_2+=multiply*mult

print(1-float(overall_sum_2/overall_sum_1))
```

Quantum library incomplete

```python
import numpy as np
import math
import random
import matplotlib.pyplot as plt

def is_unitary(M):
    M_star = numpy.transpose(M).conjugate()
    identity = numpy.eye(len(M))
    return numpy.allclose(identity, numpy.matmul(M_star, M))

class QuantumState:
    def __init__(self, vector):
        length = numpy.linalg.norm(vector)
        if not abs(1 - length) < 0.00001:
            raise ValueError('Quantum states must be unit length.')
        self.vector = numpy.array(vector)

    def measure(self):
        choices = range(len(self.vector))
        weights = [abs(a)**2 for a in self.vector]
        outcome = random.choices(choices, weights)[0]

        new_state = numpy.zeros(len(self.vector))
        new_state[outcome] = 1
        self.vector = new_state
        return outcome

    def compose(self, state):
        new_vector = numpy.kron(self.vector, state.vector)
        return QuantumState(new_vector)
```

```python
    def __repr__(self):
        return '<QuantumState: {}>'.format(', '.join(map(str, self.vector)))

class QuantumOperation:
    def __init__(self, matrix):
        if not is_unitary(matrix):
            raise ValueError('Quantum operations must be unitary')
        self.matrix = matrix

    def apply(self, state):
        new_vector = numpy.matmul(self.matrix, state.vector)
        return QuantumState(new_vector)

    def compose(self, operation):
        new_matrix = numpy.kron(self.matrix, operation.matrix)
        return QuantumOperation(new_matrix)

    def __repr__(self):
        return '<QuantumOperation: {}>'.format(str(self.matrix))


    def draw(c: Circuit, **kwargs) -> Any:
    """Draw the circuit. This function requires Qiskit."""
    return c.run(backend='ibmq', returns="draw", **kwargs)
```

```
def circuit_to_unitary(circ: Circuit, *runargs, **runkwargs) -> np.ndarray:
    """Make circuit to unitary. This function is experimental feature and
    may changed or deleted in the future."""
    runkwargs.setdefault('returns', 'statevector')
    runkwargs.setdefault('ignore_global', False)
    n_qubits = circ.n_qubits
    vecs = []
    if n_qubits == 0:
        return np.array([[1]])
    for i in range(1 << n_qubits):
        bitmask = tuple(k for k in range(n_qubits) if (1 << k) & i)
        c = Circuit()
        if bitmask:
            c.x[bitmask]
        c += circ
        vecs.append(c.run(*runargs, **runkwargs))
    return np.array(vecs).T

def binarytodecimal(numberOfQubits,Array,initialArray):
 newArray=zeros_like(initialArray)
 decimal=0

 for x in range(0,2**numberOfQubits):

  for y in range(numberOfQubits-1,-1,-
1):

     decimal+=Array[x][y]*(2**(numberOfQubits-1-y))
     newArray[x]=decimal
     decimal=0
 return newArray
```

```
    def measure_shots(self,shots=1000):
        choices = range(len(self.vector))
        weights = [abs(a)**2 for a in self.vector]

        oocount=0
        olcount=0
        locount=0
        llcount=0

        for i in range(1,shots):

            outcome = random.choices(choices, weights)[0]


            if outcome == 0:
                oocount+=1
            elif outcome == 1:
                olcount+=1
            elif outcome == 2:
                locount+=1
            else:
                llcount+=1

        return(oocount,olcount,locount,llcount)

    def compose(self, state):
        new_vector = numpy.kron(self.vector, state.vector)
        return QuantumState(new_vector)

def compose_all(*args):
  for op in args:
        result = np.kron(result, op)
         return QuantumState(result)
```