Quantum Meta-Programming for Dummies with a weak focus on Poisson Bullets, in the style of a book.

The Grandest of Cohort Grand Challenges

Jake Biele , Andrés Ducuara , Jonathan Frazer , Huili Hou , Friederike Jöhlinger , Ankur Khurana , Lana Mineh , David Payne , Ben Sayers , John Scott , Dominic Sulway , and last but not least, Oliver Thomas

Quantum Engineering Centre for Doctoral Training

H. H. Wills Physics Laboratory and Department of Electrical & Electronic Engineering, University of Bristol, BS8 1FD, UK

July 18, 2018

Preface

Hello there! Thanks for stopping by our quantum programming guide. Quantum programming, as you might expect, requires understanding of both quantum theory and programming, but do not despair, whether you are an experienced programmer with little or no experience with quantum, a quantum theoretician with little or no experience with programming, or an enthusiast willing to get into the field of quantum computing, we have specially designed this guide to take you from the absolute basics of both fields, to the implementation of quantum programs in already existing quantum computers.

We are living in the so-called second quantum revolution, and quantum computing is leading the way. The construction of quantum computers in the last few years has led to an explosion in the development of quantum programming languages, and so is the need for quantum programming guides. In this guide we provide a self contained introduction to both quantum theory and programming, an overview of the current quantum programming languages, so you can decide which language better suits your needs, followed by example programs and exercises implemented in different quantum languages, so that you acquire the fundamentals to start writing your own quantum programs.

We are the fourth cohort of the Quantum Engineering Centre for Doctoral Training (QE-CDT) at the University of Bristol, and this guide is the outcome of a whole year of learning the arts of quantum programming. We are addressing this whole subject in the way that we would have liked to learn about it, with tips and tricks that we have found along this journey, that the future quantum software developers might find useful.

Quantum regards!

QE-CDT Cohort 4

September, 2018

Some Quotes that we can use throughout the guide

This one from [**?**]

Third, scientists need to establish a quantum programming community to nurture an ecosystem of software. This community must be interdisciplinary, inclusive and focused on applications.

This one from [**?**]

One of the challenges the quantum computing field currently faces is a shortage of people that have been trained to write and develop quantum software.

This one from [**?**]

Just as classical computers are meaningless pieces of hardware without appropriate software, quantum computers need quantum software to function.

"So do not take the lecture too seriously, feeling that you really have to understand in terms of some model what I am going to describe, but just relax and enjoy it." (Feynman [Fey65])

How to use this guide (loading...)

Depending on your background, you might want to approach our guide differently. Here we describe three recommended paths depending on your background, but feel free to explore our guide as you wish!

Profile 1: The programmer

You can start from Chapter 1, if things are getting difficult you might want to refresh your mind on linear algebra and vector spaces in Appendix A, and then come back to Chapter 1. From here you can easily follow chapter two which uses mostly Python.

Profile 2: The quantum theoretician

Depending on you programming experience, you might want to check our mini tutorial in python Appendix B. Then you can address Chapter 2, simple programs implemented in different languages.

Profile 3: The Enthusiast

We recommend first a fast course on quantum mechanics. Appendix A then Chapter 1. Then our basics for programming, Appendix B. After this, you can comfortably go to Chapter 2.

If you wanna explore more on Q theory references. Describing these references [**?**], [**?**], [**?**]
If you wanna explore more on programming references. Python coursera [**?**] what else here?

Contents

Introduction 9

1 Background 13
1.1 Weird Vector things . . . 13
1.2 The gate model and quantum circuits . . . 16

2 Quantum Programming Languages 19
2.1 Rigetti - pyQuil . . . 20
2.2 IBM - QISKit . . . 23
2.3 ProjectQ . . . 28
2.4 Q# . . . 30
2.5 Further languages . . . 33
2.6 Comparison Discussion? . . . 33
2.7 Exercises . . . 34

3 Short term quantum computing 35
3.1 Adiabatic quantum computing & quantum annealers . . . 35
3.2 Implementing an Eigensolver's algorithm . . . 39
3.2.1 Eigensolver's Algorithm with pyQuil . . . 39
3.2.2 Eigensolver Algorithm in QISKit . . . 40
3.2.3 Eigensolver's Algorithm with Project Q? . . . 41

4 Quantum Algorithms and Applications 43
4.1 Oracular Algorithms . . . 43
4.1.1 Deutsch-Jozsa algorithm . . . 44
4.1.2 Grover's algorithm . . . 44
4.2 Quantum fourier transform . . . 49
4.3 Shor's algorithm . . . 50

5 Programming a future universal quantum computer 53
5.1 Implementing Deutsch's Algorithm . . . 54
5.1.1 Deutsch's Algorithm with pyQuil . . . 54
5.1.2 Deutsch's Algorithm in Qasm (IBM Q-experience) . . . 56
5.1.3 Deutsch's Algorithm with Project Q . . . 56
5.1.4 Deutsch's Algorithm with Q# . . . 56
5.1.5 Deutsch's Algorithm Exercises . . . 57

5.2 Implementing Shor's algorithm . . . 58
5.2.1 Shor's algorithm with pyQuil . . . 58
5.2.2 Shor's algorithm with QISKit . . . 60
5.2.3 Shor's algorithm with ProjectQ . . . 60
5.2.4 Shor's algorithm with Q# . . . 61
5.2.5 Shor's Algorithm Exercise(s) . . . 67
5.3 The universal quantum computer . . . 67

6 Implementations **69**
6.1 Not another introduction! . . . 69
6.1.1 The development of classical computers . . . 70
6.2 Quantum computer hardware architecture . . . 70
6.2.1 What are the qubits? . . . 71
6.2.2 What are the operations? . . . 71
6.2.3 Trapped Ions . . . 71
6.2.4 Superconducting qubits . . . 71
6.2.5 Linear optical quantum computing . . . 71
6.2.6 Error Correction . . . 72
6.3 Quantum software architecture . . . 72
6.3.1 Quantum instruction sets . . . 74
6.3.2 Higher level languages . . . 82
6.3.3 High- and low- level quantum languages . . . 82
6.3.4 Examples of different types of languages . . . 82
6.3.5 Compilers . . . 83
6.3.6 Linkers . . . 83
6.4 The future . . . 83
6.4.1 QLANG +− ™ the Quantum programming language . . . 83

A A gentle introduction to quantum theory **85**
A.1 Quantum mechanics . . . 85
A.1.1 Quantum States . . . 85
A.1.2 Superposition . . . 87
A.1.3 Operators . . . 90
A.1.4 Beyond the Single Qubit . . . 92
A.1.5 An Intro to Quantum Information Theory . . . 93
A.2 Exercises? . . . 94

B A gentle tutorial to Python **95**
B.1 Unix install . . . 95
B.2 MacOS install . . . 95
B.3 Windows install . . . 95

C Answers to Exercises **97**

D Complete code examples **99**
D.0.1 Shor in Q# . . . 100

Introduction

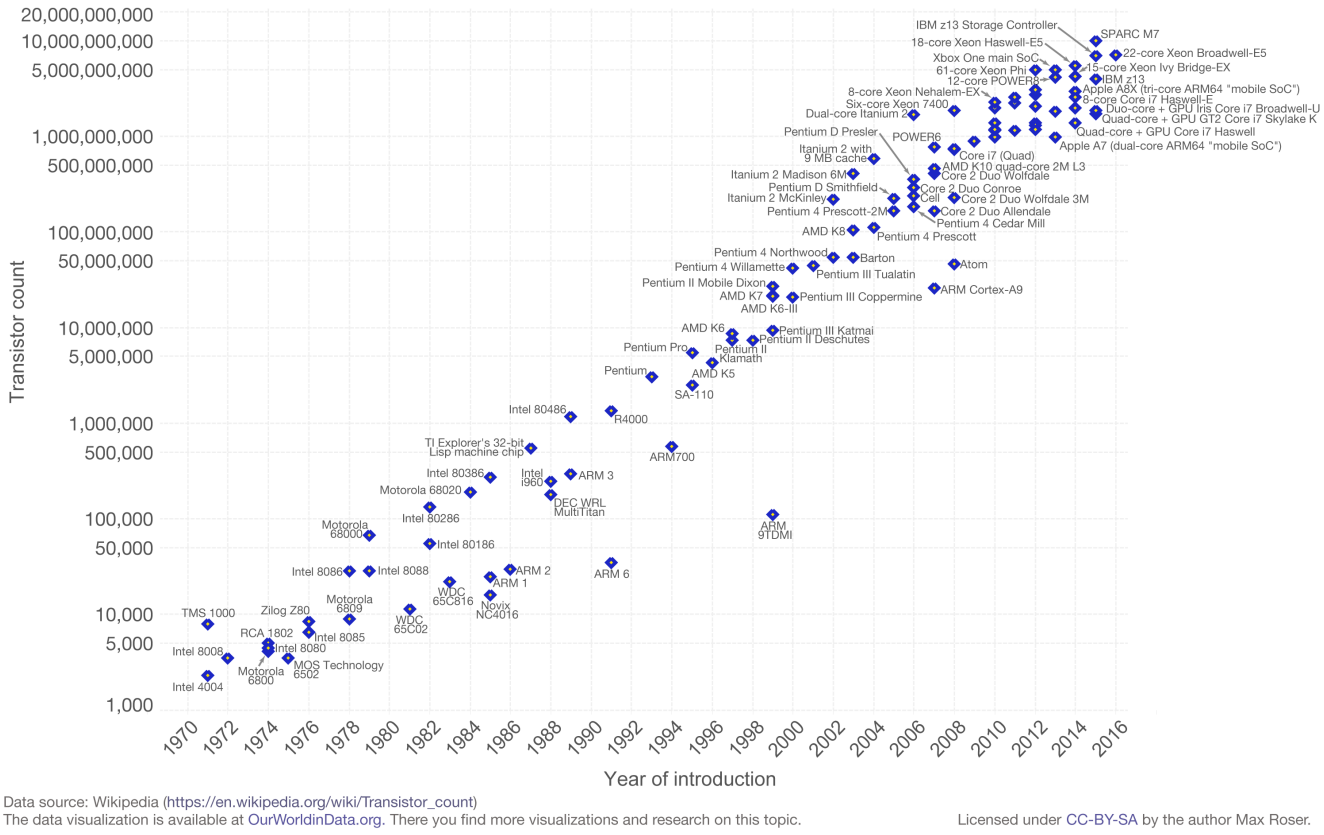
Quantum mechanics is one of the most well-tested theories in existence. It is also one of the most unintuitive, revealing aspects of nature at nanoscopic scales which are entirely incompatible of our a human's experience of the world.

There are two main facets which differentiate quantum from classical theory. The first, from which the field derives its name, is the quantisation of properties of a particle such as charge, angular momentum, and energy. This feature has already changed the world substantially over the course of the last 70 years. In addition to technologies such as the laser and magnetic resonant imaging (MRI) [**?** **?**], perhaps the greatest impact has been made through the manipulation of semiconductors. Since the invention of the first transistor in 1947 [**?**], semiconductor technology has laid the groundwork for scalable computing, bringing us into the the current information age.

The development of this technology comes at a critical time in the conventional silicon industry. The famous 'Moore's law', hypothesised in its current form in 1975, stated that the number of transistors per square inch would double every two years. This rule of thumb models the exponential scaling which computing power has followed since its conception incredibly well, as seen in Fig. 1. This was driven predominantly by the cost and power consumption per transistor going down as feature sizes decreased [**?**]. However, now increasingly small feature sizes have resulted in energy efficiencies and profitability are starting to plateau, while technical issues continue to increase. Some of these issues are in part due to quantum effects such as 'tunnelling', where an electron is able to access regions in space where, classically, it would be have insufficient energy to reach.

In fact, this feature is already used in so-called quantum annealers such as those built by D-Wave [**?**]. These machines use tunneling to find optimal solutions to a range of optimisation problems, where This example of tunnelling is related to the so-called 'wave-particle duality' which is the second key feature of quantum mechanics. The distinction between waves and particles, whose behaviour is well established in classical physics, becomes blurred. Every object in the universe, depending on their energy and confinement, will display both of these aspects to some degree.

The tantalising promise of quantum computing is hidden in this property. Particles such as electrons, which have comprised the backbone of electricity and classical information for the past century, have the ability to behave in a wave-like manner: they could contain not just the binary bit choices of 0 or 1, but one of an infinite number of continuous values, called 'qubits'. The ability to



Data source: Wikipedia (https://en.wikipedia.org/wiki/Transistor_count)
The data visualization is available at OurWorldinData.org. There you find more visualizations and research on this topic.

Figure 1: Chart showing Moore's law, with a logarithmic increase in transistor count on each chip from 1970-2016.

investigate the effect of a process on both 0 and 1 bits simultaneously means that quantum computers have the ability to scale exponentially (instead of polynomially) with the resources that are put in. This scaling is crucial. Currently for computationally difficult tasks (problems in science, AI and more) supercomputers must be used, which take up massive amounts of space and are costly to build and run. The current state of the art is shown in Fig. 2. If instead our computational scaling was exponential instead of polynomial we could perform the same task with many fewer qubits than bits - as the problem becomes more exaggerated [1]. Combined with entanglement to allow our qubits to influence each other in these states, we can harness a form of parallelism that results from the wave-like nature of controlled particles.

While in general it is doubtful that a quantum computer will be universally 'faster' than a classical computer, and is much harder to engineer, there is significant potential to outperform conventional computers at certain tasks. While the amount of information processing in a conventional computer scales linearly with the number of bits, a quantum computer scales exponentially with the number of

[1] A classic example of the power of exponential scaling is given by the legend of a vizier who presented a gift to his King. The king asked what he wanted in return, and the vizier replied that he wanted rice. Precisely, he wanted one grain of rice on the first square of the chessboard, two grains of rice on the second, four grains on the third, and so on, doubling on each square. This bankrupts the king, who has to find 2^{63} grains of rice for the last square alone. This is ~ 100 times greater than the *current* global annual food production. (At ~ 10^{12}kg [**?**])

qubits for these tasks. Thus adding a single extra qubit could double the computing power. These are discussed, along with the quantum algorithms used to implement them, in Chapter 4. At the point when quantum computers are able to outperform classical supercomputers at a task, the so-called 'quantum supremacy' [**?**] will have been achieved.

These tasks range from aiding the fields of medicine, chemistry and materials with applications including creating more powerful simulations[2] [**?**]; providing potential speedups for AI and machine learning [**?**]; assisting with modelling complex logistics problems; and improving financial models [**?**].

However, achieving this potential does not come without significant engineering difficulties. Readout or detection of the information in the qubit destroys the information contained within, resulting in us reverting to the classical bit values with an some probabilities. These probabilities are a fundamental (and irremovable) part of quantum theory, providing the link between . . . Furthermore, the technology is still very young and undeveloped. Algorithms exist for many of the applications above, but there may be many more as yet undiscovered. Academic institutions, large corporations (including Google [**? ?**], IBM, [**?**] and Intel [**?**]) and small start-ups, sucha as Rigetti, [**?**], alike have invested heavily in hardware. There are a wide range of platforms and architectures, including but not limited to superconducting qubits [**?**], ion traps [**?**], quantum dots [**?**], spin qubits in silicon [**?**], and silicon poisson bullets [**?**].

One crucial area that remains comparatively underdeveloped is software. It will be crucial to provide the missing link between theoretical algorithm and their implementation on a quantum computer. Ideally 'quantum programming' should adopt many of the features as its classical counterpart: it should be usable by any person without understanding the details of the hardware being used, while allowing access to the fundamental workings of the computer. However, all programming languages will have to trade off these two features to some degree. Finally it is required to translate the input of the user into a set of instructions that the computer can follow efficiently. This step is called compilation and is crucial to the ability to use computers.

On the other hand, several attempts to [**? ? ? ? ? ?**]. However, these documents have often focused on one or two languages. In this guide, we intend to keep a broad overview of the current state of the languages out there. In particular, in the manner of conventional programming guides, we provide many worked examples and problems to aid newcomers to the field.

Over the next few years and decades quantum computing is likely to become a reality, eventually becoming accessible to people from a range of disciplines via cloud services. It will be crucial when this becomes the case that people are able to understand how to use these machines in order to harness their applicability to the areas of mathematics, computer science, chemistry and finance. This guide is designed to be an introduction to the science of quantum computers and the current state of the field. Initially we explain in more depth how quantum computers work and their differences to classical computers in section 1.1. Since in the short-term, quantum computers are likely to be noisy,

[2] This application - the simulation of large, complex many body systems - was in fact one of the very first motivators for the development of quantum computers, most famously by Richard Feynman [**?**].

Figure 2: The summit supercomputer, currently set to be the worlds fastest supercomputer, taking up the area of two tennis courts. [**?**]

error-prone and limited in scale, we discuss how they can be used in this regime in Chapter 3.

Once the engineering of quantum computers have been improved, then a host of more impressive applications can be demonstrated, which are considered in Chapter 4. We examine the programming languages that will be able to interface between the algorithms and the quantum computer in Chapter 5, and hardware-specific implementations and architectures in Chapter 6.

A more complete description of quantum mechanics is given in Appendix A for any interested party.

Chapter 1

Background

Let's put a nice quote here!

Andres

1.1 Weird Vector things

In this section we will attempt to introduce quantum mechanics and its basic unit of information, the qubit, for those with little background in physics. Some knowledge of linear algebra may prove useful but is not necessary. In classical computing and information theory the fundamental unit of information is the familiar bit. Every bit is a binary number 0 or 1 that we use to represent false or true, or combine together to encode any information we wish. We can (for reasons that will become clear) represent a bit as 2-dimensional vector where,

$$0 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, 1 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}. \tag{1.1}$$

We can combine single bits in vector form to represent any register of bits,

$$\begin{pmatrix} x_0 \\ x_1 \end{pmatrix} \otimes \begin{pmatrix} y_0 \\ y_1 \end{pmatrix} = \begin{pmatrix} x_0 y_0 \\ x_0 y_1 \\ x_1 y_0 \\ x_1 y_1 \end{pmatrix}. \tag{1.2}$$

This notation captures the relevant information but appears rather unwieldy compared to the equivalent binary or decimal representations. In this form we write the decimal value 6 as,

$$6_{10} = 110_2 = \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}0\\1\end{pmatrix} \otimes \begin{pmatrix}1\\0\end{pmatrix} = \begin{pmatrix}0\\0\\0\\0\\0\\0\\1\\0\end{pmatrix}. \tag{1.3}$$

Note that we have a zero in every entry apart from the one corresponding the the decimal 6. The fundamental operation we can perform on this register is flipping the value of the nth bit i.e. we perform a logical NOT operation $X\begin{pmatrix}1\\0\end{pmatrix}$ to find $\begin{pmatrix}0\\1\end{pmatrix}$. The matrix X has the form,

$$\begin{pmatrix}0 & 1\\1 & 0\end{pmatrix}. \tag{1.4}$$

This can be extended to find a matrix that allows us to change any basis state into any other, and therefore we can represent all quantum operations in matrix form. Note also that the not operation is reversible; no information is lost in applying it as many times as we like. This turns out to be a general feature of logical operations in quantum computing.

Adding together numbers in either their binary or decimal form is obvious, however this clearly does not correspond to simple addition in their vector representation.

$$6_{10} + 5_{10} = 110_2 + 101_2 \neq \begin{pmatrix}0\\0\\0\\0\\0\\0\\1\\0\end{pmatrix} + \begin{pmatrix}0\\0\\0\\0\\0\\1\\0\\0\end{pmatrix}. \tag{1.5}$$

A vector representation of our bits however *should* allow addition and we will now see how to interpret this. Rather than our register being in a definite single *state* corresponding to a single decimal number we can allow superpositions of vectors with each element corresponding to a bit register. For example,

$$\begin{pmatrix}0\\1\\0\\0\end{pmatrix} + \begin{pmatrix}0\\0\\0\\1\end{pmatrix}, \tag{1.6}$$

is a valid state (however we are now moving away from classical information). This no longer has a clear and unambiguous representation in binary. Furthermore, we can change the signs between

vectors and have,

$$\begin{pmatrix}0\\1\\0\\0\end{pmatrix}-\begin{pmatrix}0\\0\\0\\1\end{pmatrix}. \tag{1.7}$$

What this represents in information terms is no longer obvious and, more importantly, how can we retrieve information stored in this way? Classically the state of an entire computer is in principle represented by a single string of bits and we can determine each one with certainty. Moving beyond classical information the situation becomes more complicated. If we attempt to measure a superposition of our vectors, asking "*which state is our register in?*" we will find $\begin{pmatrix}0\\1\\0\\0\end{pmatrix}$ with probability 0.5 and $\begin{pmatrix}0\\0\\0\\1\end{pmatrix}$ with probability 0.5. In order to ensure our probabilities sum to 1 we should normalise our even superposition with a factor of $\frac{1}{\sqrt{2}}$ however this will be ignored for clarity. We can even allow superpositions with more elements i.e. three or more possible outcomes from measurement and factors in front of each vector to adjust the probabilities. To further complicate things we can even allow complex factors in front of our basis vectors, and as it turns out this necessary to fulfil the condition that we can continuously transform any vector into another [**?**]. That is to say that there exists a matrix like X introduced above that allows us to move between any states or superpositions thereof.

So far we have mainly dealt with vectors more complicated than the simple representations of 0 and 1 we introduced earlier. Returning to these we can now introduce the qubit,

$$\alpha\begin{pmatrix}1\\0\end{pmatrix}+\beta\begin{pmatrix}0\\1\end{pmatrix} \tag{1.8}$$

where α and β are real or complex number such that $|\alpha|^2+|\beta|^2=1$. This is the reasonable requirement that probabilities should always sum to 1, but encapsulates the principle of superpositions that we measure to find outcomes. In more standard notation we represent vectors in quantum mechanics with a 'ket' $|\psi\rangle$. Anything inside the ket is simply a label and can be changed for convenience depending on the situation. For example we traditionally use ψ to denote an arbitrary quantum state with basis vectors in binary i.e.,

$$|\psi\rangle=\alpha|0\rangle+\beta|1\rangle, \tag{1.9}$$

is the same state as above in dirac notation. The probability of obtaining the outcome $|1\rangle$ from this state is $|\langle 1|\psi\rangle|^2$ where $\langle 1|$ is known as a 'bra,' forming a dirac 'bracket' with the state $|\psi\rangle$. This is nothing more than the inner product of vectors taken to the absolute value squared to ensure we always obtain real and positive probabilities.

A full model of computation requires more than representations of states and a conceptual method for reading out data. In the next section we will see how to process quantum information in the so-called circuit model.

A complete mathematical description of quantum mechanics is given in Appendix A.

1.2 The gate model and quantum circuits

This section briefly reviews the gate model for circuit based quantum computing and discusses the similarities between between digital and quantum computers. The gate model is one of the most popular architectures for quantum computation at the moment. A number of companies such as, *Intel [?] IBM [?], Google [?] and Rigetti [?]* are all using the gate model approach for quantum computing. There are other architectures for quantum computing however we think that the gate model is the most similar to digital computers.

Both forms of computation follow the same structure, you start with bits (or qubits), operations are performed on the (qu)bits and then you measure the new values of the (qu)bits. We show an example in Fig. 1.1.

(a) Digital operation

(b) Quantum operation

Figure 1.1: Digital and quantum logic circuits for implementing an arbitrary operation on bits A, B returning value Q

One of the main differences between the two figures is that in the quantum circuit the inputs A&B exist after the operation and the output Q is present before the operation. This is a feature of quantum computing being reversible (unitary).

One of the requirements for quantum computing to be universal is that we are able to perform any single qubit and a single two qubit gate. Most quantum algorithms make use of a Hadamard gate at the beginning of the computation.

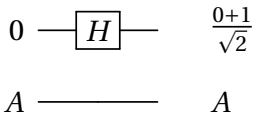


Figure 1.2: Hadamard gate acting on the top qubit and no gate performed on the bottom qubit

Multiple qubits gates are of the form, control-*Operation*, one of the most popular two-qubit gates used is the Controlled-NOT. The control means use the value of the first qubit to decide whether or not to perform the operation on the target qubit.

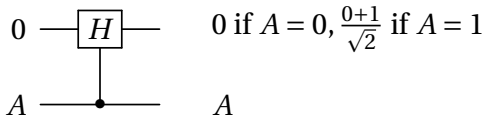
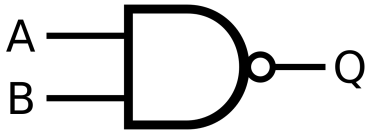


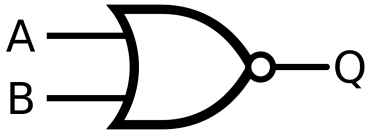
Figure 1.3: Hadamard gate acting on the top qubit and no gate performed on the bottom qubit

Digital logic

Every digital computing operation can be built up from NAND logic gates [**?**]. We can call a NAND logic gate a universal gate for computation. The NOR gate is also universal in the same way any computation can be constructed from NOR gates.



(a) The NAND logic gate [**?**].



(b) The NOR logic gate [**?**].

Table 1.1: Global caption

Input A	Input B	Output Q
0	0	1
0	1	0
1	0	0
1	1	0

(a) NOR gate truth table

Input A	Input B	Output Q
0	0	1
0	1	1
1	0	1
1	1	0

(b) NAND gate truth table

Chapter 2

Quantum Programming Languages

A new breed of quantum programmer is needed to study and implement quantum software — with a skillset between that of a quantum information theorist and a software engineer.

Will Zeng
Rigetti Computing [**?**]

I think here we should thoroughly explain the syntax of each language with a basic example implementation. with examples (not Deutsch cause we haven't explained it yet). I propose I super simple example, like generating a superposition state and performing some measurements.

For each language need to talk about the following:

- How is the program structured? (E.g. as a quantum processor/subprogram to regular programming language)
- Qubit, Ancilla bits, Classical bits
- How are gates applied?
- How are measurements handled?
- Available libraries
- Available for which operating systems
- Use case of language and future of language
- Support available
- Flexibility of language (hardware, simulation, cost estimator, etc.)

2.1 Rigetti - pyQuil

In this section we address the main features of the programming language by Rigetti and the description of its syntax with a simple example.

pyQuil is the Python based programming language created by Rigetti [**?**] as part of their quantum programming toolkit for their hardware. This consists of

- **Forest** - Rigetti's entire quantum programming toolkit.
- **Quil** – Quantum Instruction Language which is assembly-like as it lists the gates to apply. Acts as an intermediate step between pyQuil and the actual instructions that will go to the quantum computer.
- **pyQuil** - Open source Python library. The user writes their code in pyQuil and this gets translated to Quil.
- **QVM** - Rigetti's Quantum Virtual Machine which runs simulations of up to 26 qubits, an API key is required to access this.
- **QPU** – Quantum Processing Unit, Rigetti's actual chip with 19 qubits which requires special access.
- **Grove** - Open source Python library containing quantum algorithms developed with pyQuil.

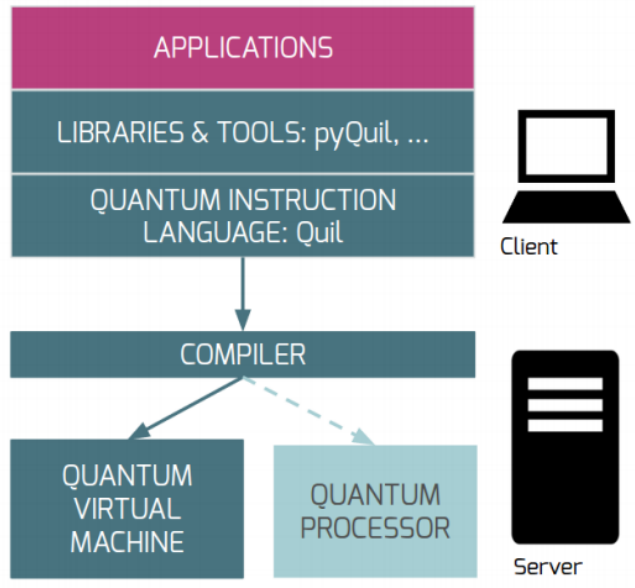


Figure 2.1: The structure of Forest. Figure taken from [**?**] - possibly need to create our own

A program would be written in pyQuil which gets translated to Quil. If the program is being run on the QVM, it is generally not necessary to decompose the Quil instructions further since the code

is being run on a simulator and everything comes down to mathematical operations. If the program is to be run on the QPU, a compiler will break down the Quil instructions into quantum gates which can be executed on the actual chip. This is necessary because QPUs currently have a more limited gate set due to the technology used to create the chip. For example if you would like to perform a controlled-NOT gate, this may have to be broken down into the elementary gates for the specific chip architecture.

Quil has been designed with the idea in mind that near term quantum computers will act as co-processors along with classical processors. It can execute quantum programs with classical control. pyQuil is more of a high-level language, hence in the guide we will be focusing on this rather than Quil.

Quantum Programs with pyQuil

Quantum algorithms require the implementation the three computational stages. We first need to initialise the state in a state of the form $|00...0\rangle$, we then need to apply gates in order to obtain a target state of interest and finally, we perform measurements in the computational basis. Let us see how to address these stages with the following specific example.

Example 1

Generating the one-qubit pure state

$$|\psi(\phi)\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + e^{i\phi}|1\rangle\right),$$

by applying gates to the initial state $|0\rangle$ and performing measurements.

Starting with the one-qubit state $|0\rangle$, we can generate the superposition with a Hadamard gate, and we then can add the desired phase with a gate of the form *here*. Finally, we consider projective measurements. We remark here however, that these steps can be implemented in any programming language that handles linear algebra. We can do it in python with the Python library Numpy in the following way Listing 2.1.

```
import numpy as np
# State initialisation
state0=np.array([[1],[0]])
# State manipulation
H=(1/np.sqrt(2))*np.array([[1,1],[1,-1]])
state1=np.dot(H,state0)
phi=np.pi/2
PHASE=np.array([[1,0],[0,np.exp(1j*phi)]])
state2=np.dot(PHASE,state1)
# Measurement: defining projectors P0,P1
ket0=np.array([[1],[0]])
ket1=np.array([[0],[1]])
P0=np.dot(ket0,ket0.T)
P1=np.dot(ket1,ket1.T)
```

```
# Probability of obtaining outcome 0
prob0=np.trace(np.dot(P0,np.dot(state2,np.conj(state2).T)))
# Probability of obtaining outcome 1
prob1=np.trace(np.dot(P1,np.dot(state2,np.conj(state2).T)))
```

Listing 2.1: Example with python only

However, this is local, and we are running this computation in our standard computers and therefore is a classical simulation of a quantum computation.

We would like to implement this in a real quantum computer, and this is where Rigetti comes in. We need further software to manipulate real QPUs, and therefore cannot be as straightforward as our previous code. but we are using the QVM.

0. Libraries: The connection and qubit initialisation of our qubits is given by:

```
from pyquil.quil  import Program
from pyquil.api   import QVMConnection
from pyquil.gates import I,X,Z,Y,H,PHASE
import numpy as np
```

1. Initialisation: The systems has been initialised into a state of n qubits in the $|00...0>$ state. Of course both the qvm and qpu are limited by this and that respectively.

the other point is that this is a lst of instructions and the actual computation has not taken place, and we need to invoke the command run to do it.

```
# Invoking and renaming
qvm=QVMConnection()
p=Program()
```

2. Gate implementation: So far we have only covered initialisation, now we need to consider state manipulation. The way that gates are being called is as follows:

```
# Gate implementation
p.inst(H(0))
theta=np.pi/2
```

by considering the object program which we have renamed as p with the methd isnt we apply fro left to right in order of application and inside parenthesis the qubit which is acting upon the qubits are listed from $0,1,...n$. For instance applying gates this and that. The complete set of gates can be found in HERE.

3. Measurement: Yadda describing

```
# Measurement
p.measure(0,0)
p.measure(1,1)
```

4. Run: So far this is only a list, now finally we run the program with the command.

```
# Running the program
```

```
cr=[]
results=qvm.run(p,cr,4)
print(results)
```

So in full our program looks like this Listing 2.2.

```
from pyquil.quil import Program
from pyquil.api import QVMConnection
from pyquil.gates import H,PHASE
import numpy as np
# Invoking and renaming
qvm=QVMConnection()
p=Program()
# Gate implementation
p.inst(H(0))
theta=np.pi/2
p.inst(PHASE(theta,0))
# Measurement
p.measure(0,0)
p.measure(1,1)
# Running the program
cr=[]
results=qvm.run(p,cr,4)
print(results)
```

Listing 2.2: Example algorithm implemented with pyQuil.

2.2 IBM - QISKit

IBM has launched the IBM Q experience that consists of a development environment called QISKit, a higher-level gate-building software development kit (SDK) that allows users to compose their own quantum algorithms, and OpenQASM, a low level quantum assembly language to realise the gates at the quantum prosessing unit (QPU). The devices used to perform the simulation and computation are based on a superconducting charge qubit implementation, which can be found described in more detail in Chapter 6.

There is ample space for development of algorithms in this environment as the visual arrays provide a clear structure to those familiar with quantum computation. The composer also displays the code on which it operates so that users interested in further development have the opportunity to learn how to code their own gates in OpenQASM as well as QISKit. The associated documentation which is linked on the IBM Q website provides more detail to the structure [**?**].

The architectures (physical or virtual) that the user can execute their quantum algorithms on are easily accessible via the IBM Q Experience web page, and can be seen in Fig. 2.2. IBM recently published a report detailing their state-of-the-art prototype 50 qubit chip in the 2017 IEEE ICRC conference [**?**].

The current state of IBM's stack, from top to bottom consists of the following elements:

- **QISKit** - IBM's open source quantum library for usage in the high level language Python.

- **QISKit ACQUA** - IBM's open source, modular Algorithms and Circuits for Quantum Applications library for usage in the high level language Python. The modules in this library as specifically targeted towards industry professionals in the research areas of Chemistry, artificial intelligence and optomization.
- **OpenQASM** – IBM's low-level Quantum Assembly Language which interprets commands and functions from QISKit and translates them into microwave pulses for use on the physical architecture (superconducting qubits). Acts as an intermediate step between QISKit and the actual instructions that will go to the quantum computer.
- **QPU** - IBM has multiple physical architectures for running quantum algorithms. These include 3, 4, 5 and 16 qubit devices accessible via an API provided when you create an account with the IBM Experience, and larger 20 qubit devices that are available via membership to the IMB Q Network (hubs, partner institutions etc).
- **Q QASM Simulator** - IBM's Quantum Virtual Machine which runs simulations of up to 32 qubits, an API key is required to access this.

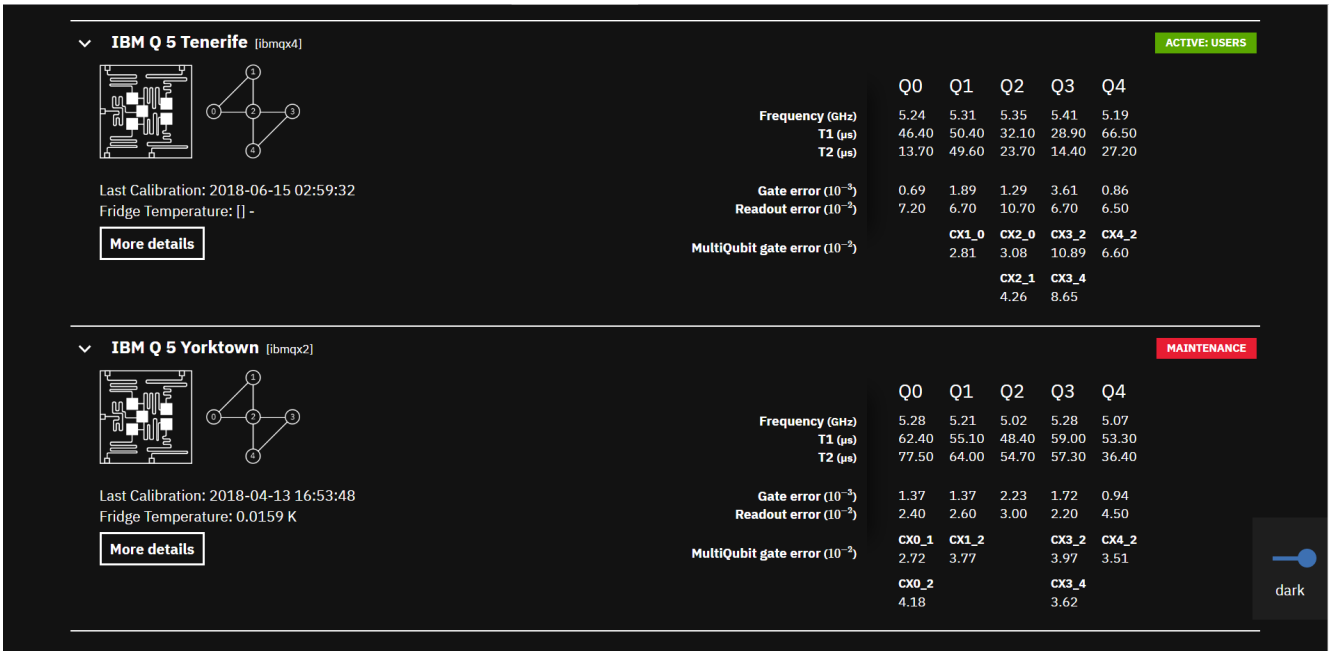


Figure 2.2: The description of the devices used for the IBM Q suite (to be updated closer to submission). Cooling refrigerator temperature along with the update structure is presented. A useful live display feed allows users to track both the physical and computational side of their own algorithms made on the composer.

Quantum Programs with QISKit

The first basic example of QISKit will be to generate a single qubit superposition, $|\psi\rangle = \frac{1}{\sqrt{2}}\left(|0\rangle + e^{i\phi}|1\rangle\right)$, beginning with the all qubits in the state $|0\rangle$. We will focus on the cases of $\phi = 0$ and $\phi = \pi$. These can

be broken down into the action of the Hadamard operator on $|0\rangle$ and the Hadamard operator followed by the Z Pauli gate. These actions are written in Eq. 2.1 and Eq. 2.2, respectively.

$$H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle) \tag{2.1}$$

$$ZH|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \tag{2.2}$$

We can simulate both of these actions on the composer in the IBM Q experience, or use experiment tokens to run the program on the processor IBM provides (referred to as ibmqx4) as seen in Fig. 2.3.

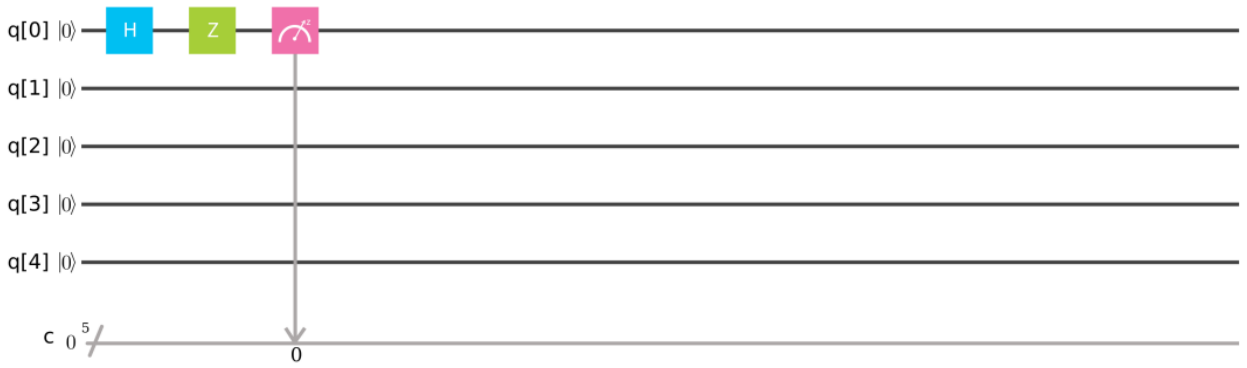


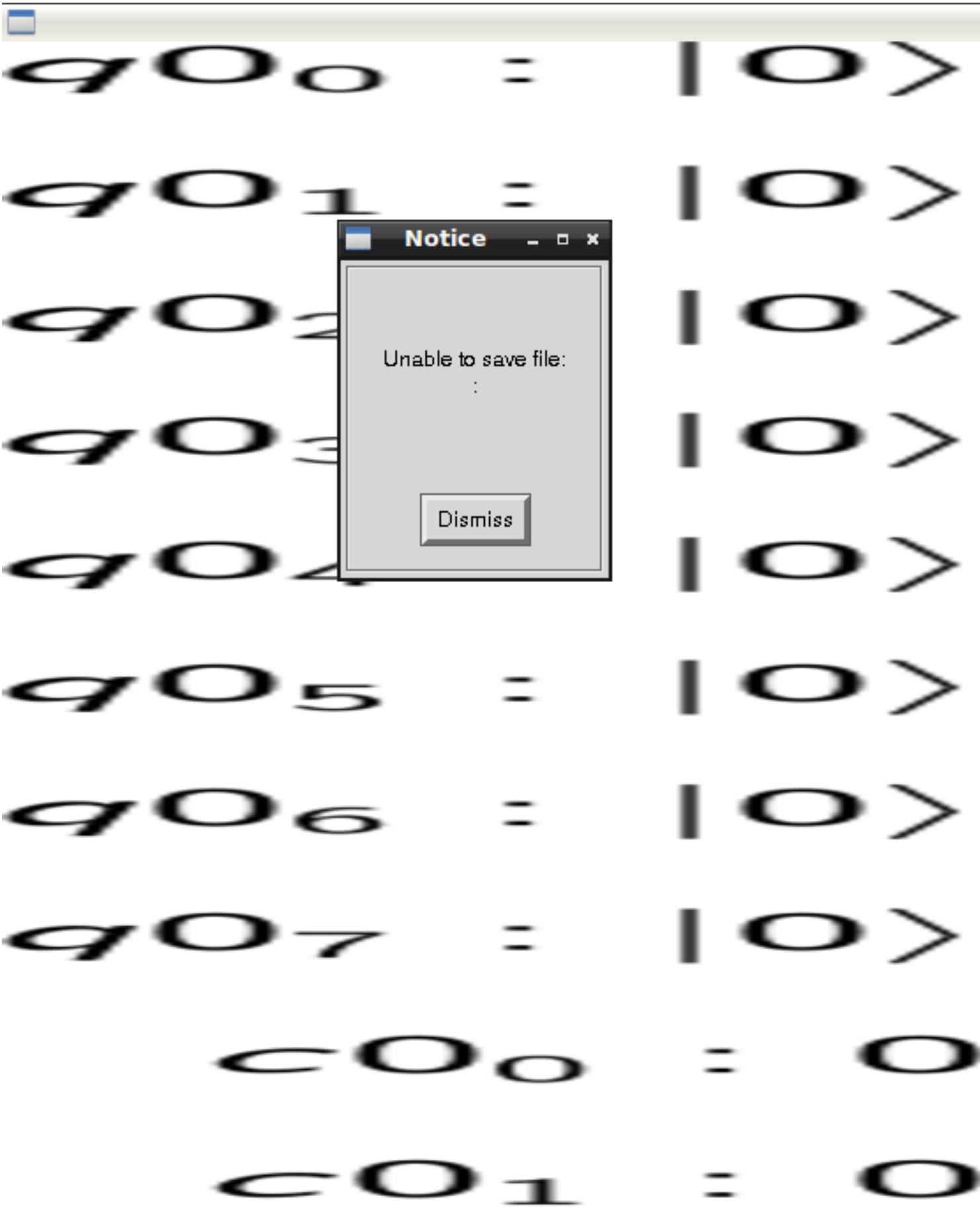
Figure 2.3: The composer view of the code that we can use to simulate the generation of $ZH|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$.

Though the composer is a useful place to get started with your own simulations of basic gate operations, it doesn't give us a an opportunity to truly experiment with the hardware at a deeper level. To do this, we need to start experimenting with the QISKit language itself.

Consider the previous example illustrated by the gate composer in 2.3. This relatively simple example can be expanded upon in QISKit via the generation and measurement of a Bell state, described by Eq. 2.3

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|01\rangle + |10\rangle) \tag{2.3}$$

To create this state, and measure it on IBM's Q QASM Simulator, the follwoing code can be utilised:


Notice
Unable to save file:
:
Dismiss

```
# Program written using QISKit to demonstrate the way in which to create an equal weighted
# superposition of states in the computational basis, and then simulate the measurement
# on IBM's Q QASM simulator

# Import relevant library functions from QISKit
from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit
from qiskit import available_backends, execute

# Initiate quantum registers for gate execution, and classical registers for measurements
q = QuantumRegister(2)
c = ClassicalRegister(2)
qc = QuantumCircuit(q, c)

# Preform a Hadamard on the qubit in the quantum register to create a superposition
qc.h(q[0])
# Preform a controlled-not operation between the first and second qubits in the register
qc.cx(q[0], q[1])
# Preform an X-pauli on the second qubit in the register
qc.x(q[1])
# Measure the superposition
qc.measure(q, c)

# Check simulation backends
print("Local backends: ", available_backends({'local': True}))

# Submit the job to the Q QASM Simulator (Up to 32 Qubits)
job_sim = execute(qc, "local_qasm_simulator")
# Fetch result
sim_result = job_sim.result()

#Print out the simulation measuement basis and corresponding counts
print("simulation: ", sim_result)
print(sim_result.get_counts(qc))
```

Listing 2.3: Example algorithm implemented with QISKit.

When ran, the output produced is as follows:

```
1 Local backends:  ['local_qasm_simulator', 'local_statevector_simulator', '
      local_unitary_simulator']
2 simulation:  COMPLETED
3 {'01': 513, '10': 511}
```

Listing 2.4: Bell State generation and measurement output.

Here, the back end in selected in line 1, completion confirmation is shown in line 2, and the measurement basis and counts are shown in line 3, illustrating that indeed the Bell State specified in 2.3 has been generated, and measured.

2.3 ProjectQ

ProjectQ is an open source Python based language. ProjectQ can run on a local simulator - your computer. But there is also an option to connect to the IBM back-end to run code on their simulator or chips, and this may be extended to other hardware in the future as well. This makes ProjectQ a general quantum programming language that is not platform specific unlike pyQuil, QISKit and Q#.

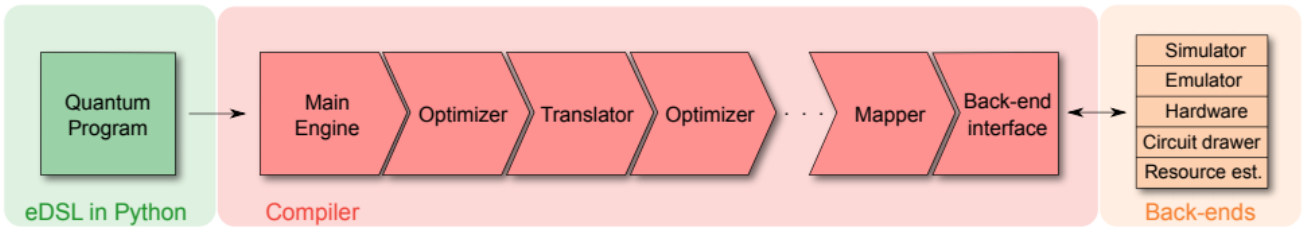


Figure 2.5: Structure of ProjectQ. Taken from [**?**], will make own diagram and possibly simplify.

ProjectQ is one of the more flexible quantum programming languages available in terms of the types of operations, compilers and back-ends that are available. Many of the typical gates that are used in quantum algorithms are already built in, and user defined gates that are either just matrices or based on mathematical operations are relatively simple to implement.

The compiler is designed to have a modular structure, so the user can pick and choose the level of optimisation required.

Quantum Programs with ProjectQ

We will now do a simple demonstration in ProjectQ to create the superposition $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$.

```
from projectq import MainEngine
from projectq.ops import H, Measure

# Set up the simulator
engine = MainEngine()
# Allocate a qubit
qubit = engine.allocate_qubit()
# Apply Hadamard gate to create superposition
H | qubit
# Measure the qubit to collapse to |0> or |1>
Measure | qubit
# Send the gates to the compiler/simulator
engine.flush()
# Display result
print('Output =', int(qubit))
```

The first thing to do is to set up the back-end that the program will be run on. This is done by calling `MainEngine` which has the local simulator as the default back-end, this can also be set up to be the IBM back-end or something else such as a resource estimator.

Qubits can now be allocated, either individually (as shown above) or as a register, from the `engine`. The syntax to do an operation is `Gate | Qubits`. Finally the gates are sent to the compiler by use of the `flush` command. And if you wish to display the result, the qubit can be converted to type integer after measurement.

2.4 Q#

One of the languages that is strongly quantum platform independent is Q#. It is being developed by Microsoft and very roughly based on their classical languages C# and F#. It is, however, not just a library, but designed to be its own language. Q# is developed to be a high-level quantum programming language. Though it still allows the basic single qubit operations, it also provides a more advanced library and allows user-defined “operations" which can be used to structure more complicated programs. The programming language should be platform independent to allow users to write programs now for quantum computers available in the future. Currently it runs on a simulator (either on your local computer or Azure, a Microsoft cloud computing service) or a cost estimator (trace simulator) which approximates qubit and gate requirements for the program.
The concept behind the language is that the quantum computer is treated like a coprocessor: like GPU is a graphical coprocessor, a QPU is the quantum one. This means that a program using the QPU requires a driver program, which controls the main processor (CPU) and which then calls on a subprogram controlling the QPU. The QPU program in this case will be written in Q#, whereas the driver program can be written in not just C# or F#, but any language supporting the .NET framework. In this programming guide the driver programs are written in C#, but you could easily change this if you are more comfortable with another language.
As Q# is dependent on the .NET framework, it is currently only available for Windows computers. The easiest way to get started is probably by programming with Visual Studio. Tutorials for installing this API and Q# can be found online.
Q# comes with a standard library which is divided into two sections: The basic building blocks of the language, such as the qubit and measurements representations and the operations given above, are part of the so-called prelude. The prelude is dependent on the target machine, e.g. a simulator. The second part, called the canon, is machine independent and built on the elements that form the prelude. It gives implementations of various quantum algorithms such as a quantum Fourier transform and a phase estimation.

Basic building blocks There are several primitive types specific to Q# and an overview of these is given in table 2.1. The first five are analogous to types in classical programming languages, e.g. C#. The others are more quantum programming specific.
It is also possible to create arrays, e.g. a `Qubit[]` or an `Int[][]`. Tuples, so e.g. (a, b, c), are also used. a, b and c could be of different types and tuples can be quite useful in returning multiple values from an operation. You can also define your own types, but they must be based on the primitive types.
Another thing a Q# programmer has to keep in mind, is that variables are normally immutable. Values are assigned with a `let` statement and they cannot be changed afterwards. To allow changing the value of a variable one can use the `mutable` keyword when assigning a value to a variable. Changing the variable is then done with the `set` statement.
Qubits are maybe the most important resource in a quantum program. You cannot assign a value to a `Qubit`, but you can measure it and then apply operations to change it to the value you want. The default value of a `Qubit` is $|0\rangle$, i.e. the computational `Zero` state, and it is generally good practice to reset a used `Qubit` to this before releasing it.
The easiest way to use actually use `Qubits` is to write a `using` statement, e.g. `using (qubits = Qubit[2]){<any code using these qubits>}`.

Type	Values
`Int`	64-bit signed integer
`Double`	double-precision floating-point number
`Bool`	Boolean value (`true` or `false`)
`String`	Unicode sequence, used to pass messages to classical driver program
`Range`	sequence of integers
`Qubit`	qubit
`Result`	outcome of measurement (`One` or `Zero`)
`Pauli`	elements of the Pauli group (`PauliI`, `PauliX`, `PauliY` or `PauliZ`)

Table 2.1: Primitive types in Q#

Quantum Programs with Q#

Q# programs require two main bits of code: a “classical" driver program and the quantum program. The driver program here is in C#, but as mentioned in the previous section this could be another language. It must use two namespaces: `Microsoft.Quantum.Simulation.Core` and `Microsoft.Quantum.Simulation.Simulators`. Next you want to define the namespace you want to work in: in this case Quantum.Superposition, which will be shared by your driver and your quantum program.
Then you start your driver program like you would any regular program. In C# you create a class and the entry point for the program is your `static void Main(string[] args)` method.
As we do not have an actual QPU, we are using a simulator. To create an instance of the simulator in your program you can use a `using` block. This creates the simulator instance and ensures it is disposed of when the block ends.
In the "using" block you can then call on the quantum operation defined in your quantum subprograms. In this case that will be the operation "Superposition". The operation is actually run with the method `Run(<QuantumSimulator>, arg1, arg2, ...)`, where the `arg` arguments are any potential arguments passed to the quantum operation. In this simple example the quantum operation does not take any arguments so we are only passing the Quantumsimulator `sim` to `Run`. Finally, we want to actually get the outcome of the quantum operation, which we get by adding `.Result` to the end.
Then the rest of the driver program can be completely classical again, with the exception that the objects returned can be specific quantum types.

```
1  using Microsoft.Quantum.Simulation.Core;
2  using Microsoft.Quantum.Simulation.Simulators;
3
4  namespace Quantum.Superposition{
5
6      class Driver{
7          static void Main(string[] args){
8
9              using (var sim = new QuantumSimulator()) {
10                 Result outcome = Superposition.Run(sim).Result;
11                 System.Console.WriteLine
12                         ($"Measuring equal superposition state gave outcome {outcome}.");
```

```
            }

            System.Console.WriteLine("Press any key to continue...");
            System.Console.ReadKey();
        }
    }
}
```

Listing 2.5: Starting a driver program for Q#

To specify the quantum operation you need to write Q# code. Again, you start by defining the namespace we are working in (`Quantum.Superposition`). Next there are two namespaces you will need to use, which are specified with the keyword `open` (similar to `using` in C#): `Microsoft.Quantum.Primitive` and `Microsoft.Quantum.Canon`.
New operations are defined with the keyword `operation` and the arguments taken and the return value is specified here in the form `operation OperationName (arg1: type1, arg2: type2, ...) : (returnType)`. Inside the operation you define the actual steps to take in the `body` block. Here we are defining a mutable variable res. It is assigned the value `Zero`, which is of type `Result`. This variable will be used to return the outcome of the quantum operation. Next, a `Qubit` array containing one `Qubit` is created and after applying a Hadamard (`H`) gate the `Qubit` will be in state $\frac{|0\rangle+|1\rangle}{2}$. Then the `Qubit` is measured (`M`) and afterwards returned to the $|0\rangle$ state. The measurement outcome is then returned to the driver program.
One thing to note here is that we needed to define the variable `res` outside the `using` block, as any variables defined in the block are local only to the `using` block. We cannot use the return statement inside the `using` block either, so we needed to use a variable local to the whole `body` block to return a value from the operation.

```
namespace Quantum.Superposition{

    open Microsoft.Quantum.Primitive;
    open Microsoft.Quantum.Canon;

    operation Superposition() : (Result){

      body{

        mutable res = Zero;

      // create a qubit reqister
      using (reg = Qubit[1]){
        // Apply the Hadamard gate to your qubit to create an equal superposition
          H(reg[0]);

        // Measure the qubit in superposition
        set res = M(reg[0]);

        // Reset the qubit to its clean |0> state
        if (res == One){
          X(reg[0]);
        }
      }
```

```
25
26          return res;
27
28          }
29      }
30  }
```

Listing 2.6: A simple program in Q# creating and measuring an equal superposition.

2.5 Further languages

TODO: short overview of other languages available.

Here is the list of some other languages also mentioned in [**?**]. Are there more?

- Strawberry Fields for CV by Xanadu [**?**]
- IonQ [**?**]
- Is Quantum Developer Kit the same QISKit?
- others?

2.6 Comparison Discussion?

I think we should talk about the pros and cons of each language, to give the readers an idea of which one better fits their needs?

- **Language-based?** Three of them are python-based, Q# is not python-based
- **Open source?** all of them are open source (which is good, I think)
- **QPU?** pyquil, qiskit, project Q have access to QPU, Q#? (could anyone confirm this?) although u need to request access to this.
- **QVM?** about QVM do they all offer QVM services?
- **how beginner-friendly is the syntax?**
- **particular conveniences** easy to implement oracles? qft? what other conveniences?
- what other general characteristics should we highlight?

2.7 Exercises

Implement the following examples in the language of your choice.

Exercise 1: Arbitrary one-qubit pure state

Generating an one-qubit state of the form:

$$\left|\psi\left(\phi,\theta\right)\right\rangle = \frac{1}{\sqrt{2}}\left(\cos\theta\left|0\right\rangle + e^{i\phi}\sin\theta\left|1\right\rangle\right),$$

by applying gates to the initial state $|0\rangle$ and performing measurements in the computational basis. What is the probability of obtaining outcome 0 for a given $\left|\psi\left(\phi,\theta\right)\right\rangle$?

List of further exercises?

- Creating a Bell state

Even though they are not algorithms perse, perhaps itd be nice to leave as an exercise one of the following?

Using the Bell state as a resource for one of these protocols?

- Teleportation?
- Super Dense coding?

Chapter 3

Short term quantum computing

Let's put a nice quote here! A Quantum supremacy quote?

Andres

In this section we aim to give a comprehensive overview of quantum computing platforms which are currently available and discuss the near-term advantages that these platforms can bring.

3.1 Adiabatic quantum computing & quantum annealers

As the universal computer is hard to achieve in the short term requiring high quality and scalability of the qubits, the technology making use of existing qubits were proposed as the intermediate steps. One of them is quantum annealing. Annealing is a process to search for the minimum energy of the state. Classically this is done by the thermal annealing changing the temperature adiabatically. Quantum annealing is expected to be implemented more efficiently with the assistance of entanglement and tunneling as showed in Fig. 3.1.

Released by D-Wave Systems Inc. in 2017, state of the art quantum annealing computer D-Wave 2000Q has 2048 superconducting qubits connected with entanglement. This is greatly improved from the first prototype with 128 qubits. it can solve certain type of problems more efficiently than the classical computer. i.e. NP hard optimisation problems such as coloring problem.
D-wave has its own software stack:

- **D-Wave Ocean**- D-Wave toolkits to help solve problems on D-Wave system.
- **qbsolv**- Open source hybrid optimization solver which can partition the Quadratic unconstrained binary optimization (QUBO) which is run either on tabu classical solver or D-Wave system. By default classical solver is used. To use D-Wave machine, separate arrangement should be made.

- **QPU**- Quantum processing unit. API key is needed to run the program on the quantum computer.

Alternatively, 1Qbit has developed the 1QBIT QUANTUM-READY™ software development kit targetting on hardware independent language. Currently it can use local classical solver or interface to D-Wave system with API key provided. I will introduce the 1QBIT QUANTUM-READY™ SDK in this section, as the example of the language used as the interface to D-Wave machine. Currently only online Jupyter Notebook version is available on `http://qdk.1qbit.com/`. By default classical solver is used. To use D-Wave machine, separate arrangement should be made.
Quantum annealing computer can be programmed to solve Quadratic unconstrained binary optimization (QUBO) problem. Given the coefficients h_i and J_{ij}, QUBO problems are such that to determine the minimum value of

$$C = \sum_{i=1}^{N} h_i q_i + \sum_{i<j}^{N} J_{ij} q_i q_j \; q_i \in \{-1, 1\} \tag{3.1}$$

To have a better idea of this, I will borrow the example of the light switching game from D-Wave website. Imagine there are several lights with certain bias marked for each of them and certain weight assigned between every two of them. For each switch, "ON" scores 1 point and "OFF" scores -1 point. The total score is the sum of the bias multiplied by the light configuration score. Additional score contribution exists from the weights multiplied by the two fold light configuration. The goal is to set the light configuration such that the total score is minimum. If we have positive bias to all of the lights, it is easy to know such configuration is setting all the light "OFF". However it can be notoriously complicating to take the weights between the lights into account.

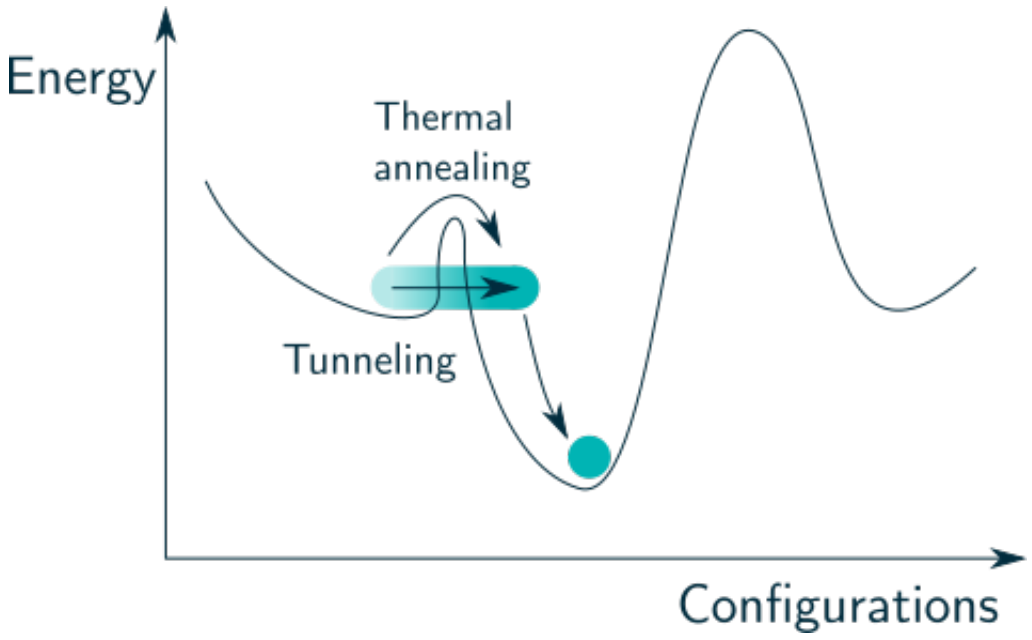


Figure 3.1: Caption

Example 1

Solve the light configuration problem with two lights involved. The bias are set to be -1 for both lights and the weight assigned between them is 2
i.e. Get the configuration to minimize the quadratic polynomial $-x_0 - x_1 + 2x_0x_1$

```
from qdk import*
# Create a quadratic polynmial  -x_0-x_1+2x_0x_1
builder = QuadraticBinaryPolynomialBuilder()
builder.add_term(-1.0, 0, 0)
builder.add_term(-1.0, 1, 1)
builder.add_term(2.0, 1, 0)
quad_poly = builder.build_polynomial()
# Create a local solver
dwave_local_solver = DWaveSolver()
# Get configuration of minimum energy solution
solution_list = dwave_local_solver.minimize(quad_poly)
print solution_list.solution_count
sol=solution_list.peek_minimum_energy_solution()
print sol
```

Vertex coloring problem is probably one of the most popular NP problems. Given the graph G(V, E) with the set of vertices V and the set of edges E, it is to determine the coloring configuration of the vertices for which no adjacent two nodes (the nodes connected with edge) have the same color. Mathematically it is the same as finding the minimum value for:

$$H = \sum_{n=0}^{N-1}(1 - \sum_{k=0}^{K-1} x_{n,k})^2 + \sum_{(u,v)\in E}\sum_{k=0}^{K-1} x_{u,k}x_{v,k} \tag{3.2}$$

$x_{n,k}$ with node n and color k is 1 only if the node n has color k, 0 otherwise. N is the total number of nodes and K is total number of the color used. The first term is the restriction that each node can only have one color. The second term is the restriction that the adjacent nodes have different color. As only one dimensional QUBO problem can be solved, the equation (2) should be reduced to one dimension by reordering the index $n, k \to nK + k$.

Example 2

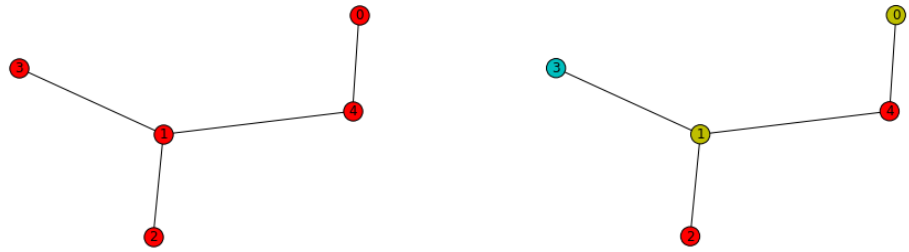
Vertex coloring problem for configuration showed in (a)

```
import networkx as nx
import matplotlib.pyplot as plt
# Set the graph configuration by setting nodes and neighbours
node=[0,1,2,3,4]
neighbour=[(0,4),(1,2),(1,3),(1,4)]
# Draw the graph configuration
g=nx.Graph()
g.add_nodes_from(node)
g.add_edges_from(neighbour)
pos = nx.spring_layout(g)
```

```
nx.draw(g, pos=pos, with_labels=True)
plt.show()
# Turn the two dimension index to one dimension
def ind2to1(i, j, r):
 return i * r + j
pallete = {0: 'r', 1: 'c', 2:'m', 3:'y'}
def get_color(i, sol, r):
 z = ind2to1(i, 0, r)
 ctr = 0
 for ctr in range(r):
  if sol[z + ctr]:
   break
 return pallete[ctr]
# Construct the quadratic equation (3.2) with one dimension index
from qdk.binary_polynomial import *
from qdk.common_solver_interface import *
builder = QuadraticBinaryPolynomialBuilder()
qubo = builder.build_polynomial()
N = 5
K = 4
for n in range(N):
 builder.add_constant_term(1)
for k in range(K):
 builder.add_term(-1, ind2to1(n,k,K))
l=builder.build_polynomial()
builder.power(2)
t = builder.build_polynomial()
qubo.sum(t)
builder.reset()
for (u,v) in g.edges():
 for k in range(K):
  builder.add_term(1, ind2to1(u,k,K), ind2to1(v,k,K))
qubo.sum(builder.build_polynomial())
print qubo
# Get the minimum energy solution by taking 300 samples
solver = DWaveSolver()
solver.solver.num_reads = 300
sol = solver.minimize(qubo).peek_minimum_energy_solution().configuration
print sol
# Draw the final configuration with color applied
nx.draw(g, pos=pos, with_labels=True, nodelist=g.nodes(),
node_color=[get_color(i, sol, K) for i in g.nodes()])
plt.show()
```

Listing 3.1: code adapted from KColoring.ipynb on http://qdk.1qbit.com/

[]



(a) A gull (b) A gull coloured

3.2 Implementing an Eigensolver's algorithm

3.2.1 Eigensolver's Algorithm with pyQuil

Loading...

3.2.2 Eigensolver Algorithm in QISKit

Applications that can use the five-qubit register include simulation of quantum state evolution and computing the properties of a ground state of the system. These applications are useful for quantum chemistry and solid state physics, where understanding strongly-correlated fermions is of interest. QISKit ACQUA provides tools that can be used to make an eigensolver for the ground state energy of simple molecules. The ground state problem may be understood as finding the solution to $H|\psi_g\rangle = E_g|\psi_g\rangle$, where H represents the Hamiltonian (energy operator) of the system and $|g\rangle$ and E_g represent the ground state and its associated energy, respectively. For complex systems such as molecules, the ground state energy can be challenging to compute in a simplified way. We need ACQUA to run these sorts of simulations as its library has a number of tools that simplify the process. QISKit ACQUA requires a number of external files to run its computation. It is recommended that these steps be taken in the route to simulating molecular ground state energies with ACQUA on QISKit. The other application is in simulating the ising model of spin chains, though we will not cover this in this section.

- Download the latest version of Anaconda Navigator in Python 3.6 at: https://www.anaconda.com/download/
- Download the relevant QISKit files from the website: https://nbviewer.jupyter.org/github/QISKit/qiskit-tutorial/blob/master/index.ipynb
- Sign up for QISKit via IBM Q experience and generate an API key for your Qconfig.py file found in the downloads from the QISKit website
- Use the command prompt to install with: pip install qiskit qiskit-acqua qiskit-acqua-chemistry in Anaconda

The molecule in this example will be Lithium hydride (LiH), as seen in Listing 3.2. The Hartree Fock method (used to determine the quantum state of a multi-particle system) is used to describe the initial state of the molecule. More detail on this method of state determination can be found in [**?**]. Constrained optimisation by linear approximation (COBYLA) is the numerical optimisation, among possible others, which is used in this example. The maximum number of iterations can be altered to suit the example along with the options to print convergence trends and termination tolerance. The backend can be changed to ibmqx4 for other examples by using an API token and experiment credits on IBM Q experience. You can compare results between the two backend options to see how well ibmqx4 compares to your classical simulation. (Another example could be added after this where there are additional elements to code by the user, perhaps an exercise on this would be helpful)

```
from qiskit_acqua_chemistry import ACQUAChemistry
# Define the problem using the parameters in the acqua dictionary
acqua_chemistry_dict = {
'driver': {'name': 'PYSCF'}, # Psycf driver for writing molecular configuration and species
# Writes the positions of the atoms in the molecule in 3 dimensions
'PYSCF': {'atom': 'Li .0 .0 -0.8; H .0 .0 0.8', 'basis': 'sto3g'},
# Maps Hamiltonian with parity qubit mapping and two-qubit reduction. Reduces the orbital
    size as well
'operator': {'name':'hamiltonian','qubit_mapping': 'parity', 'two_qubit_reduction': True, '
    freeze_core': True, 'orbital_reduction': [-3,-2]},
'algorithm': {'name':'VQE'}, # Variational quantum eigensolver
# Iterates the optimizer COBYLA 10000 times
'optimizer': {'name':'COBYLA','maxiter': 10000},
'variational_form': {'name':'UCCSD'},
# Hartree Fock initial state calculation
'initial_state': {'name': 'Hartree Fock'},
# Simulation performed on the local simulator
'backend':{'name': 'local_qasm_simulator'}
}
solver = ACQUAChemistry()
result = solver.run(acqua_chemistry_dict)
print(result['energy']) # Prints the result of the calculation for the ground state energy
```

Listing 3.2: The Lithium hydride eigensolver with Hartree Fock initial state adapted from https://qiskit.org/acqua/chemistry.

3.2.3 Eigensolver's Algorithm with Project Q?

Chapter 4

Quantum Algorithms and Applications

> You are not expected to understand this
>
> *John Lions, Lions' Commentary on UNIX 6th Edition, with Source Code*

In this section we discuss three of the most famous algorithms that provide some speed-up on a quantum computer for different problems. We start with two oracular algorithms, Deutsh's and Grover's. The last of these we go into in detail, as it is a useful example for developing some intuition of how superpositions can be used effectively. The other algorithms are described more briefly, as we outline some idea of how they work and provide a resource-focused outlook of the algorithm. Finally, several algorithms, their resource requirements and success probabilities are summed up in a table in section **??** for reference.

4.1 Oracular Algorithms

Oracular algorithms are those which are based on a unitary operation called oracle. Suppose there is a function $f(x)$ on one-bit domain range, then there exits an operation which transforms the state $|x, y\rangle \rightarrow |x, y \oplus f(x)\rangle$, where $\oplus$ indicates addition modulo 2. It is well known that this operation is unitary and can be simulated by appropriate sequence of gates. More importantly, if this operation is applied to the state $|x\rangle(|0\rangle - |1\rangle)/\sqrt{2}$, then it outputs the state $(-1)^{f(x)}|x\rangle(|0\rangle - |1\rangle)/\sqrt{2}$. This operation is generally known as oracle and represented by U_f. There are many algorithms which exploit this operation because it allows to do computation by phase manipulation. Some of the well known algorithms have been discussed below.

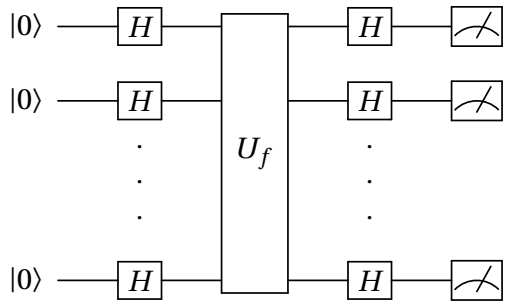
4.1.1 Deutsch-Jozsa algorithm

Deutsch-Jozsa algorithm is one of the first algorithms to demonstrate "quantum parallelism". In very simple terms, quantum parallelism is a feature of quantum computer which allows it to evaluate a function $f(x)$ simultaneously for many different values of x. Deutsch-Jozsa algorithm doesn't have many practical applications but it does provide insight into how quantum computing can trump classical computation. Suppose there is a function $f(x) : \{0,1\}^n \rightarrow \{0,1\}$ which is either constant or balanced for all values of x, the problem is to find with certainty the nature of $f(x)$. A classical solver would need at least $N = 2^n/2+1$ queries to determine with certainty whether the function is balanced or constant while the quantum algorithm can solve the problem in just one query. The box below describes how Deutsch-Jozsa algorithm features the property of quantum parallelism for a 2-qubit case ($N = 2^2 = 4$). It should be noted that there is an additional qubit required for oracle as well.

Deutsch-Jozsa Algorithm

1. Start with the state $|00\rangle$.
2. Apply Hadamard gate to all the qubits which leads to the state: $|\Psi\rangle = \frac{1}{2}\big(|0\rangle+|1\rangle\big)\big(|0\rangle+|1\rangle\big)$.
3. Apply the oracle 'U_f' which transforms the state $|\Psi\rangle \rightarrow \frac{1}{2}\big((-1)^{f(00)}|00\rangle + (-1)^{f(01)}|01\rangle + (-1)^{f(10)}|10\rangle + (-1)^{f(11)}|11\rangle\big)$.
4. Apply Hadamard to the first two qubits. Now, if $f(x)$ is constant, the first two qubits end up in the state $|00\rangle$ but if $f(x)$ is balanced, the first two qubits are in the state $|01\rangle$, $|10\rangle$ or $|11\rangle$.
5. Measuring the first two qubits reveals the nature of $f(x)$.

The circuit diagram for Deutsch-Jozsa algorithm for a general n-qubit case looks like:



4.1.2 Grover's algorithm

One of the earliest algorithms that were designed to use quantum resources was described in 1996 paper by Lov Grover [**?**]. The algorithm attempts to solve the following problem: imagine you have a

database of elements. We can represent them as bit strings, but we know that one of them is 'marked' by some function acting on that bit string. Examining the case where we have 4 numbers (2 bits), we have the following truth table.

	x	$f(x)$
0	00	0
1	01	0
2	10	1
3	11	0

(4.1)

This unstructured search is an important problem in computer science. If we used a classical computer to try to find the marked element 10 above we'd have to try at least 3 times, since we could always end up with it being the last element applied to f(x). This scales as expected, so we can write that at worst it takes N attempts to find the marked element, which can be written O(N).

However, using the principle of superposition, we can explore the whole space of elements simultaneously. To do this we need two matrices (or gates): one which is a diagonal matrix with $(-1)^{f(x)}$ as its elements. For the marked element being 10 as above, we have

$$U_f = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{4.2}$$

The second ingredient is the following matrix, (irrespective of which element is marked)

$$D = \frac{1}{2}\begin{pmatrix} -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \end{pmatrix} \tag{4.3}$$

Now we will look at the algorithm step-by-step for this simple four element (two qubit) case. Starting with the qubits in the 00 state, we generate a superposition using a so called Hadamard gate (represented by H) on each qubit, which takes $00 \rightarrow 00 + 01 + 10 + 11$ (we have ignored normalisation for simplicity). This can be represented by the matrix transformation

$$\frac{1}{2}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \tag{4.4}$$

In the next step of the algorithm, we apply U_f. This picks out the marked element, giving it a minus sign and adding a π phase shift to the other elements.

$$\frac{1}{2}\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&-1&0\\0&0&0&1\end{pmatrix}\begin{pmatrix}1\\1\\1\\1\end{pmatrix}=\frac{1}{2}\begin{pmatrix}1\\1\\-1\\1\end{pmatrix} \tag{4.5}$$

The final step is to apply D. The construction is D is such that each row, when multiplied by the vector, converts the π phase difference into unit value . This can be thought of as a constructive interference on the marked element instead of destructive interference on all of the other elements.

$$\frac{1}{2}.\frac{1}{2}\begin{pmatrix}-1&1&1&1\\1&-1&1&1\\1&1&-1&1\\1&-1&-1&1\end{pmatrix}\begin{pmatrix}1\\1\\-1\\1\end{pmatrix}=\frac{1}{4}\begin{pmatrix}0\\0\\4\\0\end{pmatrix}=\begin{pmatrix}0\\0\\1\\0\end{pmatrix} \tag{4.6}$$

From this we can see that the general recipe of Grover's algorithm is to create a superposition of all the possible states, add a π phase shift to the marked states with the special unitary U_f, and then use D to pick out this phase shift.

In this case the algorithm has successfully found the marked element with certainty (though this does not take into account any experimental imperfections observed in real life). However, using superpositions inevitably leads to success with a non-unity success rate, as demonstrated in the next section, where we take the three qubit case. Furthermore we now look at using Dirac notation instead of matrices, as now we would have to use 8×8 matrices - this demonstrates the exponential scaling that quantum computing demonstrates, with $2^n\times 2^n$ matrices being required for n qubits. Hence the transition to Dirac notation is quite a natural progression to larger system sizes.

Alternative representation: Dirac notation

We again consider a problem on $N = 8$ elements, where the fifth element is marked. The algorithm can be applied with a minimum of 3 ($2^3 = 8$) qubits in the following manner:

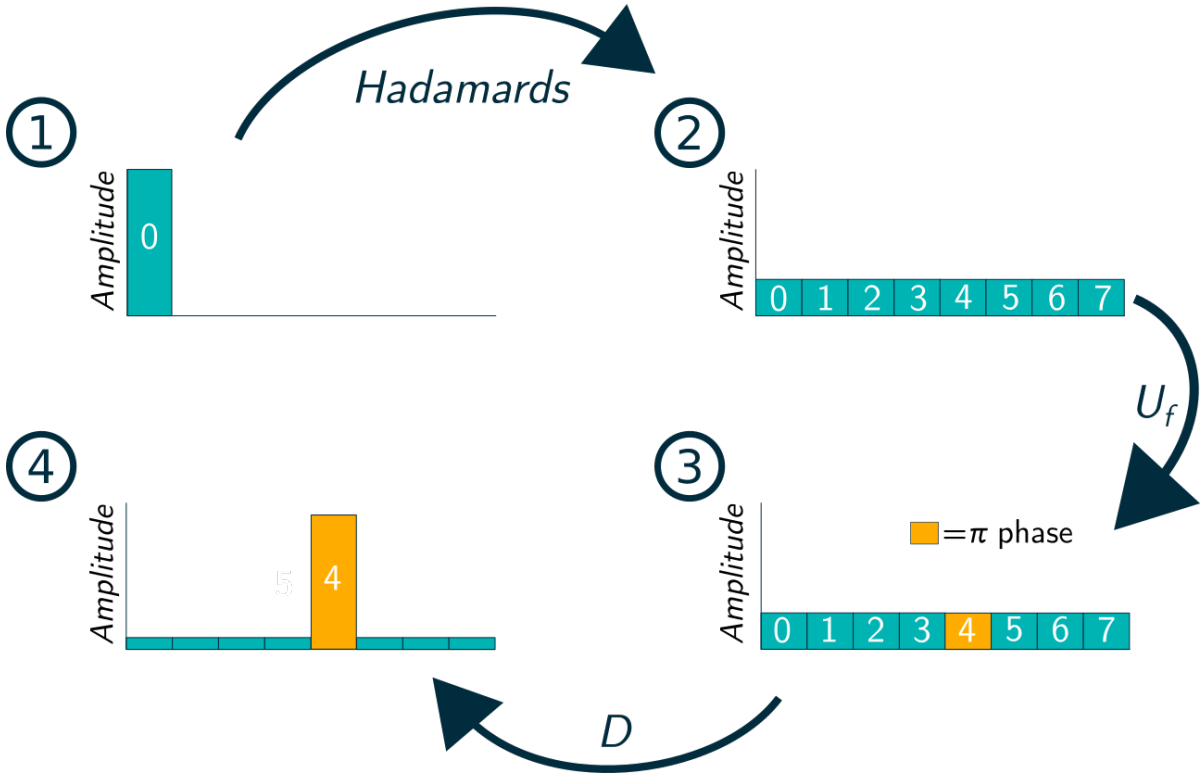


Figure 4.1: Pictorial demonstration of the steps of Grover's Algorithm

Grover's Algorithm

1. Start with the state $|000\rangle$.
2. Apply the Hadamard gates which result in the state: $|\Psi\rangle = \frac{1}{2\sqrt{2}}\big(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle\big)$
3. Apply U_f, which reverses the sign on the fifth element: $|\Psi\rangle = \frac{1}{2\sqrt{2}}\big(|000\rangle + |001\rangle + |010\rangle + |011\rangle - |100\rangle + |101\rangle + |110\rangle + |111\rangle\big)$
4. Apply D, which leads to state $|\Psi\rangle = \frac{1}{4\sqrt{2}}\big(|000\rangle + |001\rangle + |010\rangle + |011\rangle + 5|100\rangle + |101\rangle + |110\rangle + |111\rangle\big)$
5. Repeat steps 3 and 4 "T" times, where T is to be determined later.
6. Measure the state $|\Psi\rangle$.

In the step 2 above, applying Hadamard gate to all the qubits leads to a state which is in superposition of all possible states (elements). It is important to start with the state $|000\rangle$ so that all the states

in the superposition have the same initial phase. In step 3 we apply U_f gate which is dependent on $f(x)$ and is able to recognise the marked element. What it does is that it applies a π phase shift to the marked element but leaves all the other states unchanged. Next step is to apply gate D to the state obtained in step 3. The application of gate D can be understood as the operation which increases the amplitude of the phase shifted element in step 3. As mentioned in chapter 1, the measurement of a state leads to an output with a probability equal to the amplitude squared and thus, increasing the amplitude of the marked element state results in a higher probability of measuring that state. It is worth noting that repeating steps 3 and 4 a fixed number of times will increase the probability of detecting the marked element but after a certain point, the probability will start to decrease. For clarification, if we another iteration of step 3 and 4 in the above example, we get the state:

$$|\Psi\rangle = \frac{-1}{8\sqrt{2}}\big(|000\rangle + |001\rangle + |010\rangle + |011\rangle - 11\,|100\rangle + |101\rangle + |110\rangle + |111\rangle\big). \tag{4.7}$$

Now if a measurement is performed on this state, there is a 94.53% chance of getting the fifth element compared to 78.12% probability if the measurement is performed after just one iteration. On the other side, if another iteration of step 3 and 4 is performed, the resulting state becomes:

$$|\Psi\rangle = \frac{-7}{16\sqrt{2}}\Big(|000\rangle + |001\rangle + |010\rangle + |011\rangle - \frac{13}{7}\,|100\rangle + |101\rangle + |110\rangle + |111\rangle\Big) \tag{4.8}$$

and the probability of getting the marked element upon measurement is only 67% in this case. Therefore, it is important to choose the number of iterations for Grover's algorithm very carefully. The number of iterations "T" required to get the maximum probability of measuring the marked element is approximated as $T = (\pi/4)\sqrt{N}$.

General n-qubit case

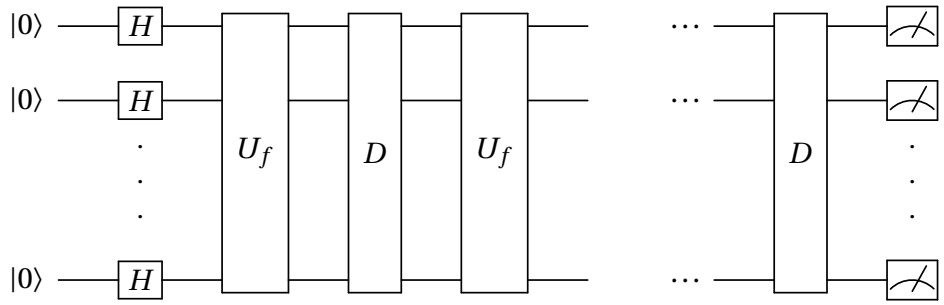
Given a function $f : \{0,1\}^n \to \{0,1\}$ with the promise that $f(x_0) = 1$ for a unique element x_0, the problem is to find this x_0. We use a quantum circuit on n qubits with initial state $|0\rangle^{\otimes n}$. Let H denote the Hadamard gate, and let U_0 denote the n-qubit operation which inverts the phase of $|0^n\rangle$: $U_0\,|0^n\rangle = -\,|0^n\rangle$, $U_0\,|x\rangle = |x\rangle$ for all $x \neq 0^n$. The first step of the algorithm is to apply Hadamard gate on all n qubits. Next, repeat the following operations T times for some T to be determined later.

1. Apply U_f, where $U_f\,|x\rangle = (-1)^{f(x)}\,|x\rangle$.

2. Apply D, where $D = -H^{\otimes n}U_0H^{\otimes n}$

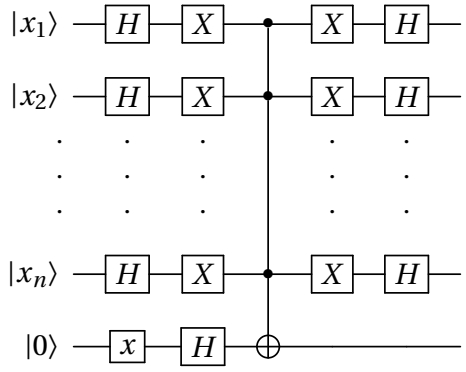
Finally, measure all the qubits and output the result.

In circuit diagram form, Grover's algorithm appears like:



Construction of gate D and U_f

The gate D in the algorithm applied on n-qubits can be implemented in the following manner. This shows that gate D requires $2n+1$ Hadamard (H) gates, $2n+1$ Pauli X gates and an n-control Toffoli gate.



The construction of gate U_f depends upon the function $f(x)$ being addressed by the algorithm.

4.2 Quantum fourier transform

Quantum Fourier transform (QFT) is the quantum analogue of discrete Fourier transform. It thus converts periodic functions to their conjugate domain (for example, time → frequency, position → momentum etc). Crucially it acts on quantum states. For example the QFT acting on two qubits is given by the transformation matrix

$$QFT = \frac{1}{2}\begin{pmatrix} \omega_N^0 & \omega_N^0 & \omega_N^0 & \omega_N^0 \\ \omega_N^0 & \omega_N^1 & \omega_N^2 & \omega_N^3 \\ \omega_N^0 & \omega_N^2 & \omega_N^4 & \omega_N^6 \\ \omega_N^0 & \omega_N^3 & \omega_N^6 & \omega_N^9 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & i \end{pmatrix} \tag{4.9}$$

This can be generalised to n qubits, which requires a 2^n dimensional matrix with its elements given by

$$QFT_{jk} = \omega_N^{jk} = \exp^{2\pi i jk/N} \tag{4.10}$$

where we count the first columns and rows of the matrix from 0. The Quantum Fourier Transform can be constructed out of more fundamental quantum gates, a Hadamard gate and a controlled phase gate.

$$H = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad R_n = \begin{pmatrix} 1 & 0 \\ 0 & \omega_{2^n} \end{pmatrix} \tag{4.11}$$

This requires $n(n-1)/2$ controlled phase gates in total and n Hadamard gates.

The QFT is not an algorithm *per se* but is crucial in many algorithms, such as Shor's algorithm as described in 4.3.

4.3 Shor's algorithm

Shor's algorithm is a procedure to factor a large number. However, most of the steps in Shor's algorithm are in fact classical, and phrase the problem to be solved as one of finding the periodicity of a function. It uses a result from Euler that states that the The algorithm is given in box

Shor's algorithm

1. Pick a number $a < N$
2. Calculate the greatest common divisor of this number a and N, $\gcd(a, N)$. If we obtain a number other than 1, then we have found a factor of N, so output $\gcd(a, n)$ as our factor for N.
3. Otherwise, find the period of the function $f(x) = a^x \mod N$. Label this period r
4. Since $a^0 \mod N = a^r \mod N = 1$, then $a^r - 1 \mod N = 0$. Factoring, $a^r - 1 \mod N = (a^{r/2} - 1)(a^{r/2} + 1) \mod N = 0$. Thus $(a^{r/2} - 1)(a^{r/2} + 1) = lN$ for some integer l. If we found r to be odd then this method has failed: return to 1.
5. Output either $\gcd(a^{r/2} - 1)$ or $\gcd(a^{r/2} + 1)$ as the factors of N.

However, step 3 is not efficient classically. We can however use a quantum algorithm to provide a polynomial speed-up. We shall now explain to explain the machinery behind this process, attempting to develop an intuition for the quantum effects over the fine detail of this process. The diagram demonstrates the processes which occur.

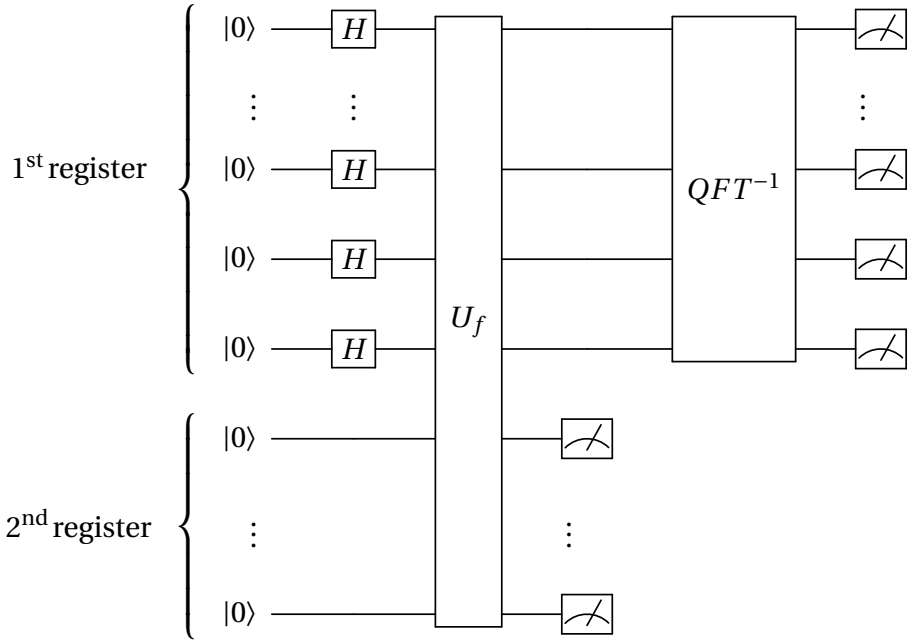


Figure 4.2: This circuit implements Shor's algorithm.

[DIAGRAM OF AMPLITUDES AND PHASES GO HERE, FOLLOWED BY EXPLANATION]

The circuit shown below implements Shor's algorithm. The 1^{st} register uses Hadamard gates to put the qubits in a uniform superposiiton. The second step uses an Oracle operation U_f for a periodic function $f = a^x \mod N$ which has the following effect on the first and second registers:

$$U_f |x\rangle |0\rangle = |x\rangle \left|f(x)\right\rangle U_{a^x \mod N} |x\rangle |0\rangle = |x\rangle \left|a^x \mod N\right\rangle \tag{4.12}$$

measuring the second register then collapses it to a single $f_0 = f(x_0)$, where $f(x_0) = f(x_0 + r) = f(x_0 + 2r) = ...$ for the periodicity r of the function. Thus the first register, through entanglement, is now in the a superposition of the states $\left|x_0 + jr\right\rangle$ for all integers j allowed for $x_0 + jr$ to be within the register. Therefore we have a periodic superposition of a state, generating peaks in the spectrum as can be seen in [REF TO DIAGRAM TO BE ADDED]. This is a problem that can be clearly solved by the QFT - we have states that are periodic with period r in one (Hilbert) space, and applying the inverse QFT will obtain a single strong peak at r.

Algorithm name	Purpose	Number of qubits used	Number of gates required	Probability of success
Grover's algorithm	Unstructured search	n qubits for 2^n marked elements	$(2T+1)n$ Hadamard gates, T U_f gates, and T U_0 gates, where $T \approx \frac{\pi}{4}\sqrt{2^n}$	$\sin^2((2T+1)\arcsin\frac{1}{\sqrt{2^n}})$
Quantum period finding	Finding Periodicity			
Deutsch-Jozsa algorithm	Constant or Balanced function	$n+1$ qubits for 2^n elements	One U_f gate and $2n+1$ Hadamard gates	100%
Eigensolver				

Chapter 5

Programming a future universal quantum computer

Don't believe everything you read on the Internet

Abraham Lincoln

Three languages will be covered here:

- Quil
- QISKit
- Q#
- Project Q

Algorithms to implement:

- Shor's algorithm (as example)
- Deutsch's algorithm (as exercise)
- Grover's algorithm (as exercise) (Ankur and David have already organised this one, so it'd be nice :3)
- Deutsch's algorithm (as exercise/example)

5.1 Implementing Deutsch's Algorithm

5.1.1 Deutsch's Algorithm with pyQuil

Deutsch's Algorithm

Given a function $f : \{0,1\}^2 \rightarrow \{0,1\}$ promised to be either balanced or constant. We initialise the system in the state $|00\rangle$, we need to implement the gates $H^{\otimes 2}$ and then U_f and then $H^{\otimes 2}$ and then perform a measurement in the computational basis.

```
import numpy as np
# State initialisation
state0=np.array([[1],[0],[0],[0]])
# Gate Implementation
H=(1/np.sqrt(2))*np.array([[1,1],[1,-1]])
state1=np.dot(np.kron(H,H),state0)
U=np.array([[-1,0,0,0],
            [0,1,0,0],
            [0,0,-1,0],
            [0,0,0,1]])
state2=np.dot(U,state1)
state3=np.dot(np.kron(H,H),state2)
# Measurement: defining basis
ket0=np.array([[1],[0]])
ket1=np.array([[0],[1]])
ket00=np.kron(ket0,ket0)
ket01=np.kron(ket0,ket1)
ket10=np.kron(ket1,ket0)
ket11=np.kron(ket1,ket1)
# Measurement: projectors P00,P01,P10,P11
P00=np.dot(ket00,ket00.T)
P01=np.dot(ket01,ket01.T)
P10=np.dot(ket10,ket10.T)
P11=np.dot(ket11,ket11.T)
# Probability of obtaining: 00,01,10,11
prob00=np.trace(np.dot(P00,np.dot(state3,np.conj(state3).T)))
prob01=np.trace(np.dot(P01,np.dot(state3,np.conj(state3).T)))
prob10=np.trace(np.dot(P10,np.dot(state3,np.conj(state3).T)))
prob11=np.trace(np.dot(P11,np.dot(state3,np.conj(state3).T)))
print(prob00,prob01,prob10,prob11)
```

Listing 5.1: Deutsch's algorithm implemented in Python only

Now implementing this with pyQuil. We first need to learn how to add our own gates.

```
# Defining our own gates
Ufma=np.array([[-1,0,0,0],
               [0,1,0,0],
               [0,0,-1,0],
               [0,0,0,1]]);
p.defgate("Uf",Ufma);
```

```
p.inst(("Uf",0,1));
```

In Listing 5.2

```
import numpy as np
from pyquil.quil import Program
from pyquil.api import QVMConnection
from pyquil.gates import X,Z,Y,H, I
# Invoking and renaming
qvm=QVMConnection()
p=Program()
# Gate implementation
p.inst(H(0),H(1))
# Assuming the given function gives
Ufma=np.array([[-1,0,0,0],
                [0,1,0,0],
                [0,0,-1,0],
                [0,0,0,1]]);
# Adding matrix as a gate
p.defgate("Uf",Ufma);
# Applying new gate and Hadamards
p.inst(("Uf",0,1));
p.inst(H(0),H(1))
# Measurements
p.measure(0,0)
p.measure(1,1)
# Running the program
cr=[]
results=qvm.run(p,cr,4)
print(results)
```

Listing 5.2: Deutsch's algorithm with pyQuil

5.1.2 Deutsch's Algorithm in Qasm (IBM Q-experience)

The Deutsch-Jozsa algorithm can be coded on this language as well. We begin by setting the bits in the register

5.1.3 Deutsch's Algorithm with Project Q

5.1.4 Deutsch's Algorithm with Q#

5.1.5 Deutsch's Algorithm Exercises

Implement the following algorithms in the language of your choice.

Exercise 2: Deutsch-Josza Algorithm

As in Deutsch algorithm but with $n > 2$.

Exercise 3: Bernstein-Vazirani Algorithm

As in DJ with $n = 3$, and given a function f that is a parity function. Checking that the code indeed identifies the parity function

5.2 Implementing Shor's algorithm

Shor's algorithm one of the most well-known quantum algorithms, due to it's implications for the safety of the commonly used RSA encryption scheme. It is also interesting, as it demonstrates the way classical and quantum computing can be used together. A large part of the algorithm is classical, but an essential part uses the quantum computer for significant speed up. The algorithm is discussed in more detail in section 4.3.

This section will go into detail how the quantum parts of the algorithm are written in the different languages, with the full code being provided in the Appendix/Github.

5.2.1 Shor's algorithm with pyQuil

The first thing to do is import the libraries required, Python has libraries which help with calculating the greatest common divisors and the continued fraction expansion convergents. From pyQuil, we need to import the functions that allow us create and run the program as discussed in section 2.1. We can also take advantage of an implementation of the Quantum Fourier Transform (QFT) in Grove.

```
import numpy as np
from math import gcd
from fractions import Fraction

from pyquil.quil import Program
from pyquil.api import QVMConnection
from grove.qft.fourier import qft
```

The first step is to randomly select an integer $1 < a < N$ and if the greatest common divisor of a and N is not 1, then we have found a factor. Otherwise, we continue onto the quantum part of the algorithm to find the order of $f(x) = a^x \bmod N$. First, the size of the registers need to be defined.

```
    m = int(np.ceil(np.log2(N**2)))   # Size of first register
    M = 2**m   # M is the smallest power of 2 greater than N^2
    n = int(np.ceil(np.log2(N-1)))   # Size of second register - need to represent 0 to N-1
```

Next we can set up the connection to the QVM and initialise the program structure. pyQuil only labels qubits in one large register and since it is more helpful for us to have two registers, we can specify which qubits we wish to use for each register.

```
    # Set up connection and program
    qvm = QVMConnection()
    p = Program()
    # Labels for register 1 and 2
    reg_1 = range(m)    # First m qubits
    reg_2 = range(m,m+n)    # Last n qubits
    reg_all = range(m+n)    # Label for both registers
```

Applying the QFT to the first register is now simple.

```
    # Apply QFT to first register
    p.inst(qft(reg_1))
```

A more difficult task is creating the bit oracle $O_f |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle$. pyQuil does not have a function that allows you to construct arbitrary mathematical operations between registers, so the bit oracle must be constructed as a matrix. The function below constructs this matrix (need to explain in more detail the indexing and change oracle to SWAP gates since this code is currently not functioning!).

```
def bit_oracle(a, N, m, n):
    mat_size = 2**(m+n)     # There are m + n qubits
    oracle_matrix = np.zeros([mat_size,mat_size])
    for x in range(2**m):
        f = pow(a,x,N)    #a**x % N
        for y in range(2**n):
            col = 2**n*x + y
            row = 2**n*x + y^f
            oracle_matrix[row][col] = 1
    # Return the matrix
    return oracle_matrix
```

A new gate using this unitary matrix can be defined using the function `defgate`, and applied to both registers.

```
    # Apply the oracle to both registers - need superposition of |x>|f(x)>
    oracle_matrix = bit_oracle(a, N, m, n)
    p.defgate("oracle", oracle_matrix)
    p.inst(("oracle",) + tuple(reg_all))
```

The next step in the algorithm calls for measuring the second register. pyQuil does not allow you to measure more than one qubit at a time, so the measurement is done inside a loop and a classical address to store the measurement result at must be specified. The code below measures qubit i and stores it in classical address i.

```
    # Measure the second register
    for i in reg_2:
        p.measure(i,i)
```

Once the QFT has been applied to the first register and that register measured, the program to find the approximate periodicity of $f(x)$ is complete and can now be run on the QVM. The classical addresses to store the results in and the number of trials must be specified.

```
    # Run the approximate peridicity algorithm
    classical_addresses = list(reg_all)
    results = qvm.run(p, classical_addresses, trials = 1)
```

Converting the output y of the approximate periodicity part of the algorithm from binary into an integer is done using the code below. It is important to keep in mind that qubit 0 gives the least significant bit.

```
    # Determine output y of algorithm
    y = 0
    for i in reg_1:
        y = y + 2**i*results[0][i]
```

The next steps of the algorithm - determining the order r from the output y and testing for failure cases - is classical and does not need to be expanded upon here. The full code is located in the Appendix/Github.

5.2.2 Shor's algorithm with QISKit

Within the framework of the QISKit SDK, there are no built in functions to call from either QISKit or ACQUA libraries, thus the quantum Fourier transform and the bit-wise oracle operations must be programmed in as separate functions for calling into the main program.

This implementation of Shor's will be implemented using IBM's Q QASM 32-qubit simulator.

To begin, we start my importing all of the relevant packages for running the program. These are called from QISKit and generic Python libraries:

```
import numpy as np
import math

from math import gcd
from fractions import Fraction

from qiskit import ClassicalRegister, QuantumRegister, QuantumCircuit
```

5.2.3 Shor's algorithm with ProjectQ

We can follow the same steps as for pyQuil and QISKit since all these languages are Python based. The code for the classical part of the algorithm stays the same and so does the size of the registers. This time the functions that need to be imported from ProjectQ are,

```
from projectq import MainEngine
from projectq.ops import All, Measure, QFT, BasicMathGate
```

Since ProjectQ is run on a local simulator, your computer, it is good practice to make sure that the number of qubits required for the computation can actually be simulated to avoid crashing the computer. Most modern computers have 8GB of RAM which is exactly enough memory for 29 qubits, but there are other processes being run on the computer so you would only be able to simulate 28.

```
    # Check we can simulate on our local simulator
    if m + n > 28:   # 28 qubits = 4GB of RAM
        print('Number of qubits required =', m+n, 'which is too large to simulate.')
        return 0
```

Now we can load the simulator by calling `MainEngine` and allocate the two registers.

```
    engine = MainEngine()
    # Initialise registers
    reg_1 = engine.allocate_qureg(m)
    reg_2 = engine.allocate_qureg(n)
```

The QFT is a standard operation in ProjectQ and the syntax for applying it is,

```
    # Apply QFT to first register
    QFT | reg_1
```

Next, we have to define and apply the oracle. This is fairly straightforward in ProjectQ since there is a class `BasicMathGate` which allows you to define an arbitrary mathematical operation on a number of registers. The oracle can be defined as a function acting on two registers and this function can be passed to `BasicMathGate`.

```
    # Define the oracle O_f|x>|y> = |x>|y + f(x)> where + is bitwise XOR
    def O_f(x,y) : return (x, y^pow(a,x,N))
    # Apply oracle to both registers
    BasicMathGate(O_f) | (reg_1, reg_2)
```

The following step, to measure the second register can be done in one line as the function *All* allows you to do the same operation on multiple qubits at once.

```
    # Measure the second register
    All(Measure) | reg_2
```

After applying the QFT and measuring the first register, to execute the algorithm all the instructions need to be sent to the simulator using the `flush` command and the output determined.

```
    # Run the approximate peridicity algorithm
    engine.flush()

    # Determine output y of algorithm
    y = 0
    for i in range(m):
        y = y + 2**i*int(reg_1[i])
```

This completes the quantum part of Shor's algorithm in ProjectQ.

5.2.4 Shor's algorithm with Q#

There are three main operators required for Shor's algorithm: the Hadamard (H), the Quantum Fourier Transform (QFT) and the operation U_f. The first, H is a primitive gate in Q# and for the second, the QFT, one can use an operation from the library which can be applied to multiple qubits at once. The third operation however, is a bit more complicated, as there is no straightforward way to implement it in Q#. In this example we have used an implementation which is given in [**?**]. In figure 5.1 the corresponding circuit can be seen. Now, instead of an operation mapping $|x\rangle \rightarrow |(a^x) mod N\rangle$ we only need an operation mapping $|x\rangle \rightarrow |(ax) mod N\rangle$. This is easier to do, especially in Q#, as the standard library already contains such an operation.

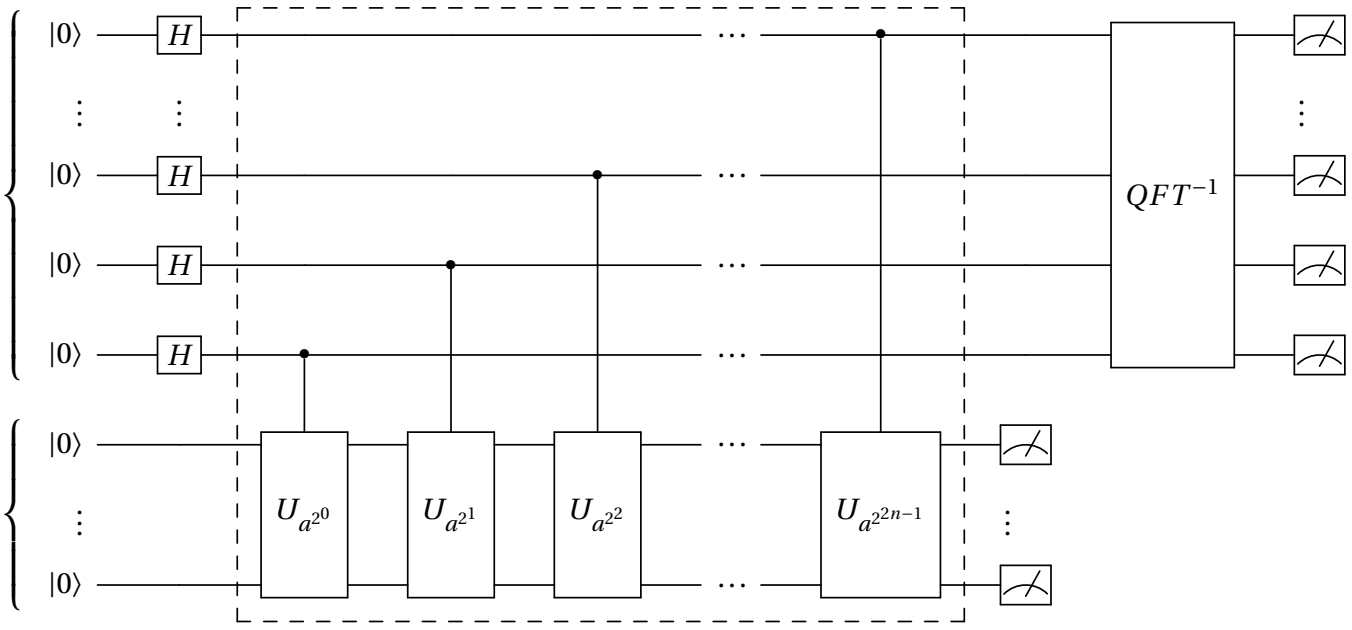


Figure 5.1: Circuit for approximate periodicity finding in Shor's algorithm. The operation U_i is given by the map $|x\rangle \rightarrow |(xi)modN\rangle$. Together, the operations in the dashed box implement the bit oracle U_f with $f(x) = (a^x)modN$. Adapted from [**?**].

In the case of Shor's algorithm it is interesting to see which part is best done by the CPU and which best by the QPU. CPUs are most likely much more optimised for many calculations, as they've have been around for much longer. Therefore, it makes sense to only use the QPU for the elements that quantum algorithms can speed up drastically. For Shor's algorithm this means that only the third step of the algorithm needs the QPU and the rest can be written in, e.g., C#.

To start we specify the quantum namespaces we will be using and create the namespace we will be working in, `Quantum.Shor`. Then we define a new class, `Shor`, and inside it a method in which we will implement Shor's algorithm, `Factor`.

```
1  using Microsoft.Quantum.Simulation.Core; // to use Q#
2 using Microsoft.Quantum.Simulation.Simulators; // to use Q#
3
4 namespace Quantum.Shor{
5     class Shor {
6         static int Factor(int N) {
7         }
8     }
9 }
```

Next we will implement Shor's algorithm in the `Factor` method step by step, leaving the implementation of the quantum algorithm part until last. Given N we need to check if it is even (i.e. 2 would be a factor), and otherwise find a random integer a coprime to N, smaller than N, but larger than 1. The method `GDC`, calculating the greatest common divisor of two integers, has to be imple-

mented, too, but is omitted here. It can be found in the complete code for this implementation given in the appendix.

```
41          static int Factor(int N) {
42
43              // Check if N is even
44              if (N % 2 == 0) {
45                  return 2;
46              }
47
48              // Find random integer a such that 2<a<N
49              Random rnd = new Random();
50              int a = rnd.Next(2, N);
51
52              // Check if a and N are coprime
53              int b = GCD(a, N);
54              if (b != 1) {
55                  return b;
56              }
```

The next step uses the quantum algorithm, but we will only write the call for now and implement the Q# code later. The quantum algorithm does not find the order of a modulus N, but it finds the approximate periodicity of $f(x) = (a^x) mod N$, which can then be used to find the order in the classical part of the code. We let `ApproximatePeriodicity` be the quantum operation, which takes the values for a and N as input. As always, we must also pass the simulator to the operation together with the arguments of `ApproximatePeriodicity` using the method `Run`. Finally we retrieve the `Result` property, which is the outcome of the quantum operation. In the next line we cast this outcome to `int`.

```
58              // Quantum algorithm to find approximate periodicity
59              int y;
60              using (var sim = new QuantumSimulator()) {
61                  var outcome = ApproximatePeriodicity.Run(sim, N, a).Result;
62                  y = (int)outcome;
63              }
```

Now that we have the approximate periodicity, we need to find the order, which we do by finding the convergents of the continued fraction expansion of the fraction $z = \frac{y}{M}$, where y is the approximate periodicity and M is the smallest power of 2 bigger than N^2. The method `FindConvergents` must again be implemented separately and example code can be found in the appendix.

```
67              // M is smallest power of 2 larger than N^2
68              double Nsquared = Math.Pow((double)N, 2.0);
69              int M = (int)Math.Pow(2.0, Math.Ceiling(Math.Log(Nsquared) / Math.Log(2.0)));
70
71              // Find the convergents of the continued fractions expansion of y/M
72              List<(int, int)> convergents = FindConvergents(y, M);
```

In the last step we go through the convergents and check for the cases that the denominator of the convergent is smaller than N, even and the convergent is within $1/(2N^2)$ of z. These convergents are then used to factor N using the `GCD` method. If the outcome of this is N or 1, the algorithm has failed. Otherwise we have found a factor of N.

```
74              int r;
75              double z = (double)y/ (double)M;
76              double maxDiff = 1 / (double)(2 * N ^ 2);
77              foreach ((int num, int denom) in convergents){
78                  double fraction = (double)num / (double)denom;
79                  if (Math.Abs(fraction-z)<=maxDiff && denom < N) {
80                      r = denom;
81
82                      // Check if r is even
83                      if (r % 2 != 0) {
84                          continue;
85                      }
86
87                      // Find the greatest common divisor between a^(r/2)-1 and N
88                      int s = GCD((int)Math.Pow(a,(r / 2)) - 1, N);
89
90                      if (s == 1 || s == N) {
91                          continue;
92                      }
93
94                      return s;
95                  }
96              }
```

Now we have a chance that the method ends without returning a factor of N. As a method must return a valid return value at every possible end point in the code, we must let a program using this method know that the algorithm failed by throwing an exception, if we reach this point in the program. To show that this failure of the algorithm is caused by the non-deterministic nature of the quantum algorithm, we can create our own type of exception:

```
189     public class QuantumAlgFailedException : Exception {
190
191         public QuantumAlgFailedException(string message) : base(message) {
192         }
193     }
```

We can then throw this exception if we reach the end of the method without finding a factor.

```
98              throw new QuantumAlgFailedException
99                  ("no suitable order r could be found: either r was odd or s was 1");
```

Now that the classical programming section of the factoring method is done, we can write the Q# code. Starting again with the essentials of the code we have the definition of the namespace we are working in, `Quantum.Shor`, the namespaces we are using and the bare skeleton of the function.

```
1  namespace   Quantum.Shor{
2
3     open Microsoft.Quantum.Primitive;
4     open Microsoft.Quantum.Canon;
5   open Microsoft.Quantum.Extensions.Math;
6   open Microsoft.Quantum.Extensions.Convert;
7   operation ApproximatePeriodicity(N: Int, a: Int) : (Int){
8
9         body{
```

```
10      }
11    }
12  }
```

Inside the `body` we can now implement the approximate periodicity algorithm. First we need to determine how many qubits we will need. The algorithm needs to have space for the value $N = 2^n$ and for a value of maximum size $M = 2^m$. Before we allocate the qubits, we need to create the mutable variable `otucome1`, which will hold the value that our operation returns.

```
38      let len1 = Ceiling(Log(ToDouble(N^2))/Log(ToDouble(2))); // length register 1
39      let len2 = Ceiling(Log( ToDouble(N) )/Log(ToDouble(2))); // length register 2
40
41      mutable outcome1 = 0;
```

Next, we create the two registers. First all the qubits get allocated together and then they are separated into two registers.

```
44      using (qubits = Qubit[len1+len2]){
45
46        // Create the two registers
47        mutable indexReg1 = new Int[len1];
48        for (i in 0..len1-1){
49          set indexReg1[i] = i;
50        }
51        let reg1 = Subarray(indexReg1, qubits);
52
53        mutable indexReg2 = new Int[len2];
54        for (i in 0..len2-1){
55          set indexReg2[i] = len1+i;
56        }
57        let reg2 = Subarray(indexReg2, qubits);
```

The next steps are implementing the gates that can be seen in the quantum circuit diagram in figure 5.1. To implement the U_i operation, we use the operation `ModularMultiplyByConstantLE` from the Q# library. However, this is not quite the operation we want, yet. We can now write our own operation `ModularQubitMultiplyByExp` which will be exact the U_f operation.
One of the main things we have to be careful with, when using a multi-qubit operation is the endianness which is assumed by the operation. Endianness is concerned with which (qu)bit is the most significant: e.g. if `01` means $0 \times 2^0 + 1 \times 2^1 = 2$ or $0 \times 2^1 + 1 \times 2^0 = 1$. In the first case the first (qu)bit represents the smallest value, which is also denoted "little-endian". The second case gives the largest, i.e. most significant value, first, so it is also called "big-endian".
In Q# some operations assume one endianness while others use the other and many are defined for both and the library documentation generally tells you what is endianness is used. The library operation we are going to use, uses the register in the little-endian ordering. This is denoted in the arguments in the operation of the type `LittleEndian`, which is derived from `Qubit[]` and therefore lets us cast directly to this type. The casting syntax in Q# is different than in C#: Assuming we have a variable `register` of type `Qubit[]`, we can cast to `LittleEndian` using the syntax `LittleEndian(register)`.
Now, we can define our new operation `ModularQubitMultiplyByExp`, which takes a qubit register of `LittleEndian` type, a base value and a power value for the exponential, and the modulus value. Using one of the maths functions from the library we can calculate the exponent and then

use `ModularMultiplyByConstantLE` to create the wanted operation. However, with just defining the body of the operation we are not yet done. As this operation does not contain any measuring operations, it can be easily inverted, creating the so-called "adjoint" operation. This can be done simply with the statement `adjoint auto`. Similarly, we can create a controlled version of the operation and even the controlled inverted equivalent. This allows the operation to be used in four different ways with writing just one implementation.

```
100 operation ModularQubitMultiplyByExp
101       (stateIn: LittleEndian, expBase: Int, power: Int, modulus: Int):(){
102
103   body{
104     ModularMultiplyByConstantLE(ExpMod(expBase, power, modulus), modulus, stateIn);
105   }
106
107   adjoint auto
108   controlled auto
109   controlled adjoint auto
110 }
```

Going back to the main operation `ApproximatePeriodicity` we can now easily implement the algorithm, as we have all the building blocks. First we apply Hadamard gates to all qubits in the first register. To apply a one-qubit operation to all qubits in a register we can use the library operation `ApplyToEach` and pass both the operation we want to apply and the qubits to it.

```
60        ApplyToEach (H, reg1);
```

Now, we can apply our `ModularQubitMultiplyByExp` to the first register. As you could see in the circuit these operations are controlled on qubits in the first register. Using a loop we can easily implement this, as we ensured that we created a controlled version of this operation. Using the syntax `(Controlled Operation)([ctrl], (arg1, arg2, ...))` one can ensure that operation `Operation` is controlled by `Qubit ctrl` and receives the required arguments `arg1`, etc.

```
63        for (i in 0..len1-1){
64          (Controlled ModularQubitMultiplyByExp)
65                      ([reg1[i]], (LittleEndian(reg2), a^2, i, N));
66        }
```

Next, we measure the second register, where we use the library method `MeasureInteger`, which takes a register, measures each qubit and returns the equivalent integer value to all the outcomes in little-endian ordering.

```
69        let outcome2 = MeasureInteger(LittleEndian(reg2));
```

Now we have to apply the inverse Quantum Fourier Transform. The Q# library contains a Quantum Fourier Transform operation, specifically one for a little-endian ordered register: `QFELE`. We can use the `Adjoint` keyword to get the inverse operation.

```
72        (Adjoint QFTLE)(LittleEndian(reg2));
```

Now we can measure the first register to get value we want. As `outcome1` is a mutable variable, we have to use the `set` keyword to assign the value to it, instead of the `let` keyword.

```
75          set outcome1 = MeasureInteger(LittleEndian(reg1));
```

The last step is "cleaning" the qubits, i.e. resetting them to the $|0\rangle$ state before releasing them. This can be done by writing another operation, this time an operation that takes one `Qubit` and sets it to $|0\rangle$.

```
14      operation SetZero (q1: Qubit) : (){
15
16          body{
17
18              let current = M(q1);
19
20              if (current != Zero){
21                  X(q1);
22              }
23          }
24      }
```

As this operation takes one `Qubit` as input and returns nothing, we can use it in combination with the `ApplyToEach` operation, passing our operation and all the qubits together to it. After cleaning them, we release them when exiting the `using` block.

```
78          ApplyToEach(SetZero, qubits);
```

Now we can return our measured integer to the classical program.

```
82        return outcome1;
83      }
```

5.2.5 Shor's Algorithm Exercise(s)

5.3 The universal quantum computer

Chapter 6

Implementations

Unperformed experiments have no results

Asher Peres

6.1 Not another introduction!

The vast majority of tasks facing modern programmers require little or no knowledge of how computers really work. There is a wealth of of programming languages and development tools which remove implementation details and allow the engineer to focus on the important aspects of the task at hand.

However, in the general quest to develop a quantum computer, it pays to take a closer look at the internal structure of computers. The language surrounding quantum algorithms has been heavily influenced by classical computers (the most important example being the qubit, which is analogous to a classical bit), and it makes sense that the implementation of a large scale quantum computer might be similarly influenced by its classical counterpart. Furthermore, due to the lack of large scale quantum computers, complex software stacks and quantum development environment which hide implementation details simply do not exist.

We therefore consider it necessary to understand at least the basics of how quantum computers might work. In this section, we discuss possible architectures of future large scale quantum computers. In particular, we consider the similarities and differences between classical and quantum computers, and how this might lead to differences between classical and quantum programming languages.

The section is divided into two main parts. The first focuses on quantum computer hardware and the physical systems which represent qubits and implement quantum operations. The second is focused on the implementation of quantum algorithms, from the instruction set level up to high level languages.

6.1.1 The development of classical computers

6.2 Quantum computer hardware architecture

Quantum architecture is to do with the structures that make up a quantum computer: there need to be qubits, unitary operations (data processing), memory (data storage), control (instruction execution), 'buses' (getting enough connectivity between qubits), input/output (quantum measurement, initialising qubits), error correction (preventing decoherence of quantum states). Several architecture features are achieved by using ancilla qubits (qubits required to implement certain useful operations).

In this section we would like to compare the architecture of a quantum computer with the architecture of a classical computer. Most high level elements are analogous, but there are some differences: e.g. error correction is not thought of in the same way classically. Most of the elements are subtly different: quantum I/O involves quantum measurements; memory basically involves swapping qubits since copying is not allowed.

Quantum instruction sets (a higher level type of architecture) can be made analogous to classical instruction sets in terms of their purpose (to expose fundamental units of data processing and control to a compiler).

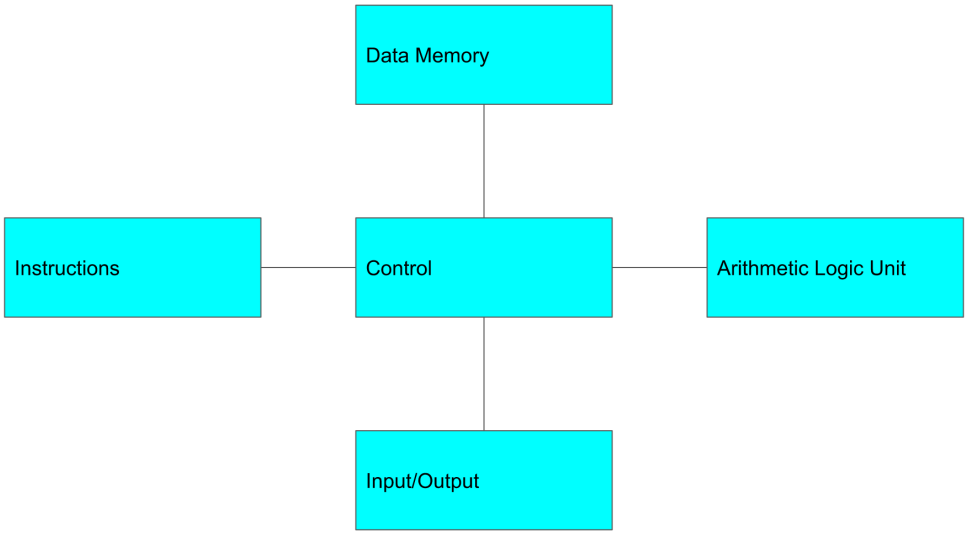


Figure 6.1: Harvard Architecture

The over-arching structure for quantum computers can be separated into two parts, there is the architecture and the platform. A quantum architecture has a different meaning to the traditional use of the word. Within the community, architectures is the distinction between the gate model, one-way quantum computing or topological quantum computing. A platform is the physical system that is used to build the computer such as, photonics, trapped ions, semi-conductor qubits, superconducting qubits.

We now discuss some of the current physical approaches to building a quantum computer. In

reality the quantum architectures and platforms are not completely separate, some physical systems are much better suited to specific architectures. For example the gate model is a natural choice when using trapped ions or superconducting qubits whereas photons are much better suited to one-way quantum computing.

There are two distinct types of qubits, stationary (e.g. Trapped ions or superconducting qubits) or flying qubits (photons). Here we will only discuss trapped ions and photons **Poisson Bullets are the best!!1!** as we believe these systems are the easiest to visualise conceptually.

A good way to think of a quantum computer is a very large, noisy machine which is incredibly sensitive to its environment. The aim is trying to control it just to keep the machine coherent. Most of the resources used are for error correction, keeping the whole thing coherent, a very small part of the whole machine is the information processing.

6.2.1 What are the qubits?

6.2.2 What are the operations?

6.2.3 Trapped Ions

Trapped ions are a remarkably stable physical system (once an ion is trapped it remains trapped with a very high probability). Electric fields are used to trap the ions, by changing the voltages in the x,y,z components in the electric field it is possible to shuttle the individual ions around. This enables a great deal of control over the dynamics of the ions.

The quantum gates are performed on the qubits using either microwaves which use atomic transitions or optical pulses for electronic transitions. The microwave gates are appealing as it opens the possibility of addressing multiple qubits at once. This will be necessary when taking into account error correction as one logical X operation could correspond to 100s or 1000s of X operations on physical qubits to produce one logical X.

The [**?**] blueprint suggests using a modular approach of packing tiles shown in Fig. 6.2 on a 2D surface. Each tile would contain one ion and have a dedicated loading zone, an entangling zone with adjacent tiles each with only one ion on and a detection zone for readout. Entangling gates or operations are performed by bringing two qubits close together. This is relatively easy for trapped ions as of the good control available.

6.2.4 Superconducting qubits

6.2.5 Linear optical quantum computing

linear optical quantum computing is great!

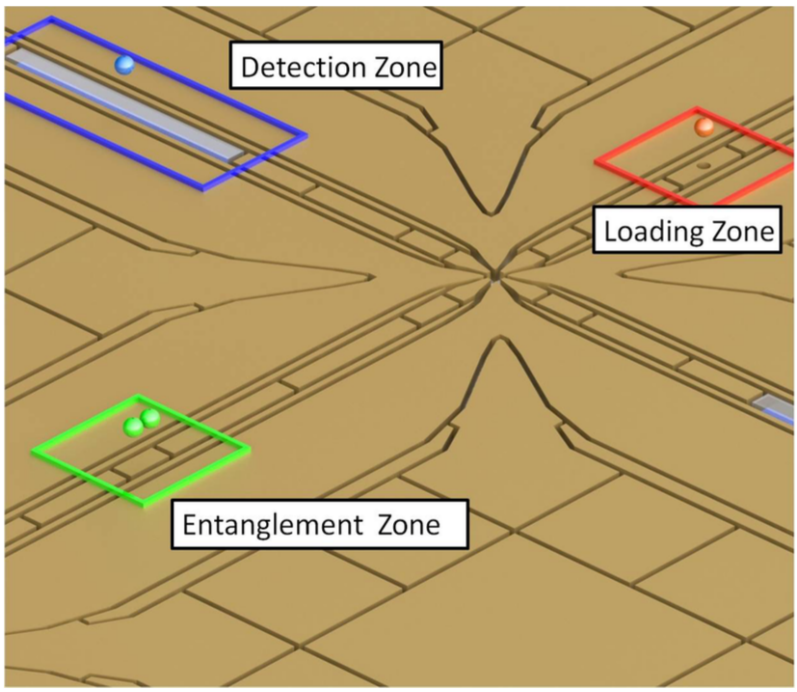


Figure 6.2: Caption [**?**]

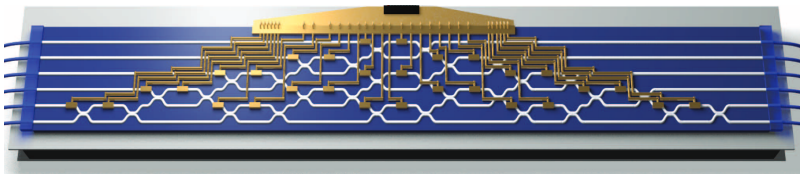


Figure 6.3: Caption [**?**]

6.2.6 Error Correction

Full fault tolerance is beyond the scope of this guide however we will briefly discuss error correction. The basic idea of error correction is to use redundancy to counteract errors. Increasing the resources devoted to storing the logical information should in principle make the computer more resilient to errors.

6.3 Quantum software architecture

Computer languages are influenced by the structure of the computer. In classical computers, languages developed historically in order to increase abstraction and ease the task of the programmer. To begin with, assembly language replaced machine code because it is easier to remember a mnemonic like ADD rather than a string of ones and zeros representing the same thing. The C programming language was invented to make common programming constructions such as if-statements and while-loops easier to program, using machine independent syntax. C is a compiled language, meaning that it needs to be converted into machine code before it can run. There is a different compiler for each

computer architecture, which converts C into the specific instructions that can be executed on that machine.

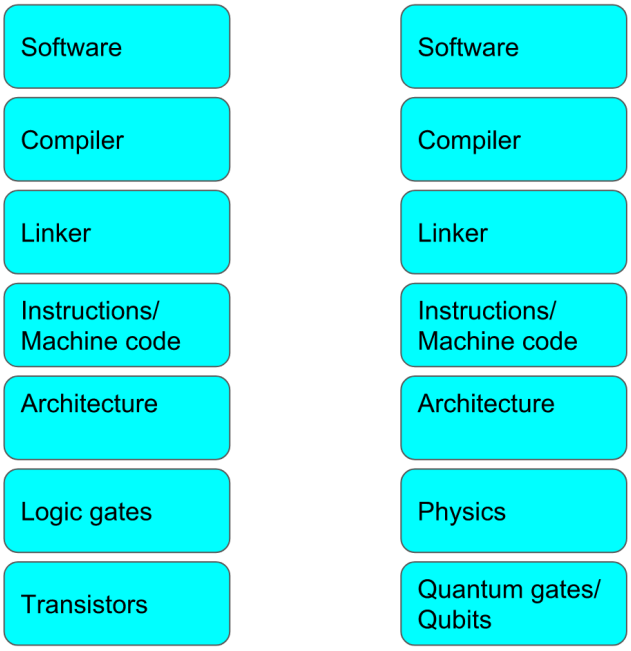


Figure 6.4: Caption

Other levels of abstraction have subsequently been introduced into programming. Some relate to code structure (object oriented programming, functional programming, and other paradigms), and others to the portability of the code. For example, Java runs in an environment called a virtual machine. This is a program which runs a file containing Java source code on a particular computer. The critical difference between this and a compiler is that many computers have a Java virtual machine, so that the end user can just run the same program without having to worry about whether there is a compiler present.

Quantum programming languages may eventually develop in a similar way. However, there are some important differences. For example, quantum programming languages are being developed without there existing a large scale quantum computer to run programs on. This is due to the hindsight afforded by classical programming languages, and the assumption that quantum programming languages will be essentially similar. Their role is currently to understand the structure of quantum algorithms better and to provide a platform for simulating quantum computers, rather than to make programming quantum computers easier.

Here, we will consider how the implementation of a large scale quantum computer might inform the structure of quantum languages. For example, most quantum programming languages currently emphasise single qubit manipulation. While this works when we only have a few qubits, it clearly becomes unfeasible in a quantum computer with a million qubits. There, it will be necessary to organise qubits into registers, variables and types depending on their role. What should those types be?

How should it be informed by the structure of the quantum computer?

Part of the issue is the current lack of understanding of large scale quantum computer architectures. A typical classical computer CPU architecture is shown in Fig. 6.1. It contains modules for memory, processing and arithmetic, buses for moving data around, and input/output ports to connect to other devices. The computer is controlled by an instruction set, which is read and decoded in the control section.

Each instruction performs a basic operation such as moving data from external memory into working memory, adding variables together, etc. There are also control instructions for deciding what to do next, for example branches and sub-routine calls. Memory is often organised into registers, which might be 8 bits wide in a small microcontroller, or 64 bits wide in a modern Intel processor. These instructions are the lowest level of processing available to a classical programming language. By analogy, it is likely that a quantum computer will have a similar architecture containing an instruction set (ADD, MOV, etc.). We will consider the possible structure of such an instruction set in subsection 6.3.1. We will then consider the simplest type of abstraction above quantum assembly language – the analogue of C (QLang+-!). We will consider the necessity of quantum compilers and linkers, and how they differ from their classical counterparts in (section).

6.3.1 Quantum instruction sets

An instruction set is a collection of operations designed to meet two competing requirements. The instructions must contain enough operations to allow access to the full functionality of the computer, without exposing implementation details. Further, it must be simple to express a program in terms of the instruction set, so the instructions should be as high level as possible. However, the instruction set should not be overly large or contain complex operations, otherwise it would be difficult to implement in hardware.

Candidate instruction sets could therefore tend towards two opposite extremes. On the one hand, the instruction set could be very large and implement nearly everything a program might want to do: matrix multiplication, Fourier transforms, database operations, etc. Every operation would be optimised in hardware and all possible tasks would be accounted for. This would make programming and compiling very simple and programs would tend to be very efficient. However, the instruction set would be difficult to implement, and inflexible in the face of new applications and tasks.

On the other end of the scale, the instruction set might be the single operation NAND. This would be universal for classical computation, so all programs could be realised. The design of the computer would be very simple, with only one instruction to implement. However, simple programming tasks such as addition would become very difficult, and complicated programs would be nearly impossible to write.

Classical instructions sets sit between these two extremes. They contain general operations such as arithmetic (ADD, MUL, etc.), copy operations (MOV) and control flow (BRANCH, CALL, etc.) and they also contain some lower level details such as bit manipulation.

In the rest of this section we consider the instructions which might form the basis of a full quantum computer.

Qubit manipulation

Despite the need for full quantum computers to contain operations on registers, it is likely that they would retain single qubit manipulation. In classical computers single bits can be accessed using bit manipulation instructions. For example, in the Microchip 16-bit dsPIC family of microcontrollers [**?**], bits can be set to zero or one using the BCLR and BSET instructions.

```
1    CLR     w0       ; clear register w0
2    CLR     w1       ; clear register w1
3    BSET    w0, #2   ; set bit #2 of register w0 to 1
4    BTG     w1, #6   ; toggle bit #6 of register w1
5    BCLR    w1, #5   ; clear bit #5 of register w1
```

Listing 6.1: Single bit operations

This code initialises the two registers w0, w1 to zeroes, puts a one in the second bit of register w0, and a zero in the fifth bit of register w1. The BTG instruction toggles bit six of w2, swapping its value from one to zero or vice versa. It performs the classical NOT operation.

In quantum computers, there are many more possibilities for single qubit operations other than just setting and clearing bits. As mentioned above **REF**, a quantum computer should be able to perform all single qubit operations, which are indexed by two continuous parameters. The quantum instruction set may implement a small set of commonly used operations, such as X, Y, and Z. Then the assembly language may look like:

```
1    CLR     w0       ; Quantum registers should be initialised to zero
2    CLR     w1
3    QBTX    w0, #2   ; Perform an X gate on bit #2 or w0
4    QBTY    w1, #5
```

Listing 6.2: Quantum X and Y operations

which performs an X gate on the second qubit of register w0, and performs the Y gate on the fifth qubit of register w1. However, the quantum computer would also need to implement arbitrary rotations and phase shifts, as follows:

```
1    QBTX    w0, #2, #30 ; Perform a 30 degree X rotation on bit #2 of w0
2    QBTY    w1, #5, #45
3    QPHA    w1, #0, #45 ; Perform a 45 degree phase shift on qubit #0 of w1
```

Listing 6.3: Arbitrary single qubit rotations

Now, the code executes a 30° X-rotation on the second qubit of w0, and a 45° Y-rotation on the fifth qubit of w1.

The instruction set would need to contain a few 2-qubit operations, such as CNOT:

```
1    CNOT    w0, #0, #1  ; Perform a CNOT between qubits #0
2                        ; (control) and #1 (target) of w0
3    CNOT    w0, w2      ; Perform CNOT between all qubits of the
4                        ; w0 (control) and w2 (target) register
5    CNOT    w0, w2, #1  ; Perform CNOT between qubit 1 of w0 and w2
```

Listing 6.4: CNOT operations

The CNOT operation and single qubit unitaries allow the implementation of any unitary operation on qubits. Therefore it would not be necessary to include any more two- or N- qubit gates. However, they may be included for convenience, or because they are simple to implement on a particular architecture. For example, the SWAP gate (which swaps the computational basis states) could replace the CNOT gate if it were simpler to implement a SWAP operation.

A generalisation of the SWAP operation is the QMOV operation, which move the state of one quantum register into another:

```
1    QMOV    w0, w1       ; Move the quantum state of w0 to w1
2    QMOV    0x1111, w1   ; Move the state stored at address 0x1111 to w1
```

Listing 6.5: QMOV operations

This operation is more subtle than its classical counterpart. Copying is not allowed in quantum computing, so the operation would have to be realised as a swap, where the states in w0 and w1 are interchanged. Alternatively, the state might be teleported, which is a process involving measurement of register w0. The implementation of the QMOV is unimportant; the key feature is that the state which was in w0 ends up in w1.

This is actually a critical instruction of a full quantum computer. It is likely that the computer will be implemented in such a way that only a small set of register (the working registers $w_0 ... w_n$) support the full range of quantum operations. Therefore quantum data will need to be copied back and forth between these registers and quantum RAM, which is a large block of quantum registers whose only purpose is to store quantum states. A typical sequence of operations might be as follows:

```
1    QMOV    0x11FD, w1       ; Move the quantum state at address 0x11FD to w1
2    QMOV    0xFFFF, w2       ; Move the quantum state at address 0xFFFF to w2
3    CNOT    w1, w2           ; Perform a CNOT operation across all qubits
4    QMOV    w1, 0x11FD       ; Move the states back
5    QMOV    w1, 0xFFFF       ; Move the states back
```

Listing 6.6: Adding

Coherent classical operations

Any operation which can be expressed in a classical computer has a quantum version, which we will call the 'coherent' operation. A coherent operation is one that can be performed on superpositions of states. For example, addition can be implemented coherently in a quantum computer. Suppose that there are two quantum registers $|a\rangle$ and $|b\rangle$ whose contents can be interpreted as numbers. For example $|0101\rangle$ and $|1001\rangle$ represent the numbers 5 and 9. Then there is a quantum version of addition, which we will denote QADD, whose action on the states above is

$$\mathrm{QADD}\big[\,|0101\rangle\,,|1001\rangle\,\big] = |1110\rangle\,.$$

In other words, it produces the state which corresponds to the result of the sum of $5+9=14$. However, it also acts on superpositions, so that $5+9=14$ and $6+9=15$ could be performed simultaneously:

$$\mathrm{QADD}\left[\frac{1}{\sqrt{2}}\big(|0101\rangle+|0110\rangle\big),|1010\rangle\right] = \frac{1}{\sqrt{2}}\big(|1110\rangle+|1111\rangle\big).$$

Now, the two possible results 14 $|1110\rangle$ and 15 $|1111\rangle$ exist in superposition in the output.

This type of operation intrinsically acts on a register containing many qubits. Otherwise it is not possible to interpret the register as a number.

The above example, QADD, is not a valid quantum operation. This is because all quantum operations must be realised as unitary operations acting on a fixed number of qubits. In particular, there must be the same number of qubits in the output of QADD as there are in the input. Implementing this process involves a complicated set of tradeoffs to do with qubit allocation and gate optimisation, as we considered in the section hardware architecture above **section**.

A valid version of QADD may work as follows:

```
1    QADD    w0, w1, w2  ; Add the contents of w0 and w1 and place it in w2
2                        ; w0 and w1 don't change but are now entangled with w2
```

The important fact about this instruction, which makes it different from the classical version ADD, is that all three registers are considered both inputs and outputs at the same time. The QADD instruction is a unitary operations performed on the registers w0, w1 and w2. This means that w0 and w1 become entangled with w2 after the instruction has been performed.

More complicated classical operations can also be implemented coherently. Consider the function $f(x)$ of the variable x, which takes a bitstring of length N to one of length M. We want a quantum operation Qf which can performs the same function as f, but which also acts on superpositons. For example, if $f(x) = x^2$, then Qf would act as follows on the superposition of $|2\rangle = |0010\rangle$ and $|0\rangle = |0000\rangle$ as follows:

$$Qf\left[\frac{1}{\sqrt{2}}\big(|0010\rangle + |0000\rangle\big)\right] = \frac{1}{\sqrt{2}}\big(|0100\rangle + |0000\rangle\big).$$

This corresponds to the fact that $2^2 = 4$ and $0^2 = 0$, and those two computations can be performed in superposition. As in the case of QADD, this operation must be implemented as a unitary operation acting on a fixed number of qubits. One way of achieving this is defining the unitary map

$$|x\rangle\,\big|y\big\rangle \xmapsto{Qf} |x\rangle\,\big|f(x)\oplus y\big\rangle,$$

where $\oplus$ is bitwise XOR. Here, x is a register containing the input N bit number, and y is a n bit register. Then setting $y = |0\rangle$ results in the the value $f(x)$ being written into the y register:

$$|x\rangle\,|0\rangle \xmapsto{Qf} |x\rangle\,\big|f(x)\big\rangle.$$

This operation is often called the bit oracle.

This form of Qf can also be used to implement a coherent function of two variables $g(r,s)$. Suppose that r and s are N bits long and the output is again M bits long. By concatenating r and s, we get a single $2N$ bit long number, which we will call x. Then f can be seen as a function of a single variable x, which can be implemented using Qf above.

For example, consider the implementation of QADD for 2-bit input arguments. The classical function is addition, i.e. $g(r,s) = r+s$. Concatenating r and s gives a new variable x. The truth table for the output g for different values of r and s is shown in Table 6.1. Here, it is clear that the variables r and

Table 6.1: Truth table for 2-bit addition

x		$f(x)$
r	s	$r+s$
00	00	0000
00	01	0001
00	10	0010
00	11	0011
01	00	0001
01	01	0010
01	10	0011
01	11	0100

x		$f(x)$
r	s	$r+s$
10	00	0010
10	01	0011
10	10	0100
10	11	0101
11	00	0011
11	01	0100
11	10	0101
11	11	0110

s can be concatenated into a single variable x, and that the truth table in Table 6.1 can be interpreted as a function of a single variable x which maps 4-bit numbers to 4-bit numbers.

Clearly a quantum computer will need enough instructions to implement all coherent operations of interest. One way to achieve this would be to implement an instruction QFUN which implements an arbitrary reconfigurable coherent operation. The truth table of the classical function could be loaded into a dedicated block of registers $f0,...,fn$.

For example, consider the function $f(x)$ on a 2 bit variable x defined by the truth table in Table 6.2. The function $f(x)$ is the classical XOR operation on 2 bits, but could also be interpreted as the 1-bit half adder (addition modulo 2). This function could be loaded an implemented coherently as follows:

Table 6.2: Truth table for 1 bit addition

x	$f(x)$
00	0
01	1
10	1
11	0

```
1    ; Set the function f(x)
2    MOV     #0, f0     ; Load the truth table for f(x)
3    MOV     #1, f1     ; by moving the truth table outcomes
4    MOV     #1, f2     ; into the registers f0...f3
5    MOV     #0, f3
6    ; Perform the coherent operation
7    QFUN    w0, w1       ; Here, w0 contains x. The output
8                         ; y will be stored in w1.
```

This approach suffers from two key drawbacks. The first is its implementation, which may be very difficult. The hardware would need to decide how to implement the coherent operation, which is a non-trivial resource allocation problem. It would likely be quite difficult to come up with general purpose hardware which produces optimised results for all classical functions f. The second more serious problem is the lack of extensibility. Here, the size of the function f (i.e. the number of rows in

its truth table) is limited by the number of dedicated registers $f0...fn$. This is especially true of truth tables which are sparse, where most of the dedicated registers represent wasted space.

A better approach would be to devise a minimal set of instruction which could be used to realise all coherent classical operations. For example, suppose that QADD and QMUL were implemented as follows:

```
1    MOV    #2, w0        ;
2    MOV    #3, w1
3    QADD   w0, w1, w2   ; Add 2 by 3 and put the result (5) in w2
4    QMUL   w0, w1, w3   ; Multiply 2 by 3 and put the result (6) in w3
5    QADD   w2, w3, w4   ; Add the contents of w2 and w3
```

These kind of manipulations can be used to build up any polynomial coherent operation.

An interesting implementation of the quantum adder is presented by Draper [**?**]. The circuit makes use of the Quantum Fourier Transform to turn addition operations into phase operations, and then uses controlled phase gates to implement the operation. This approach avoids using the carry qubits which are involved in implementing addition based on classical models of addition. The same type of approach is extended by Beauregard [**?**] to obtain the coherent multiplication operation.

Quantum conditionals

One of the most important operations in classical computing is the if-statement. This is a control flow operation, which checks whether a condition is true and executes one block of code or another accordingly.

The prototypical conditional operation in many instruction sets is the bit test. For example, BTSC (bit-test skip-if-clear) can be used to implement the classical controlled-NOT gate, with the truth table Table 6.3. The operation is performed by testing whether the control bit A is one, and if it is

Table 6.3: Truth table classical CNOT

x	$f(x)$
00	0
01	1
10	1
11	0

toggle bit B (using BTG). If A is zero, the BTSC causes the BTG line to be skipped.

```
1    BTSC    w0, #0  ; comment
2    BTG     w0, #1
```

In quantum instruction sets, there will almost certainly be classical branching operations, which perform quantum operations that are conditioned on the value of classical bits. However, it is conceivable that there are also quantum conditional operations, where the BTSC instruction above might be generalised to depend on a qubit. For example, the QBTSC instruction would execute the next line of code if the test qubit is in the state one, otherwise the next line would be skipped.

What if the test bit is in a superposition of zero and one? Then the outcome of the program should be to leave the quantum computer in a superposition of executing and not executing the next line of code.

For example, the CNOT gate could be implemented as follows:

```
QBTSC   w0, #0
QBTG    w0, #1
```

Here, the target (qubit 1 of w0) is toggled depending on the value of the control (qubit 0 or w0). If the control is in the state

$$\frac{1}{\sqrt{2}}\big(|0\rangle + |1\rangle\big),$$

then the target would be in a superposition of non-toggled and toggled – exactly the action of the CNOT gate.

Measurement instructions

One of the most important features of quantum computing is the ability to measure qubits and collapse the state to one of a particular set of outcomes. Measurements can be made in arbitrary bases, but these can always be reduced to measurement in the computational basis by a suitable choice of single qubit unitary operations.

The following example shows some possible measurement instructions

```
QMEAS   w0, cw1     ; Measure quantum register w0, place result in classical register
cw1
QMEAS   w1, #0, cw1 ; Measure qubit 0 of w0, place the result in cw1
```

Listing 6.7: Quantum register measurement

Classical operations

It makes sense for the instruction set of a quantum computer to contain classical operations as well. Many quantum algorithms (e.g. Shor's factoring algorithm, the variational quantum eigensolver), contain both quantum and classical elements, and so it would be benificial to be able to access both types of operation in the same machine using a consistent syntax.

This entails the encorporation of many instructions which are standard on classical computers: for example, arithmetic (ADD, MUL), control flow (CALL and BRA), bit manipulation, etc. As discussed in the hardware section **above**, these would be implemented by classical logic in the classical part of the quantum computing architecture.

In addition to purely classical operations, some instructions could have hybrid quantum/classical dialects. For example, the addition instruction QADD could add two quantum registers or add a quantum register to a classical register:

```
QADD    w0, w1, w2  ; Add quantum registers w0 and w1 and
                    ; place result in w2
QADD    cw0, w0, w1 ; Add classical register cw0 to quantum
```

```
4                       ; register w0 and place the result in w1
```

Listing 6.8: Different dialects of QADD

This kind of thing, where a single instruction can take many different types of arguments, is standard in classical instruction sets. Behind the scenes, the instructions might be implemented differently depending on their arguments. For example, as described by Beauregard [**?**], it is possible to reduce the number of quantum gates required for adding a classical register to a quantum register, compared to the full operation of adding two quantum registers.

Quantum types

The most important of these is a quantum type. Classical types typically include things like 'double' (for integers), 'char' (for letters), 'float' (for floating point number), etc. and possible more complex types like point (which contain references to variables). In a quantum computer, there would have to be a types for a quantum registers. For example, 'qdouble' might refer to a quantum register whose contents is interpreted as an integer. A 'qbitmap' might refer to a register whose contents is not to be interpreted, i.e. all the qubits should be treated independently.

There would probably not be a qubit type in quantum languages for long-term quantum computers. This is because there would presumably be enough resources that a quantum register (say 8 qubits wide) could be used as a single qubit, but ignoring the other seven qubits. This is similar to the lack of a bit type in modern classical programming languages, for the same reason.

Quantum operations

There would need to be fundamental quantum operations. The types of quantum operations would be dictated by the quantum type of the variables which are operands. For example, two variables of type qdouble could be coherently added, but it would not make sense to coherently add two variables of type qbitmap.

There would need to be some facility for implementing quantum measurements. There would need to be some variant of a 'measure' keyword, which could be made to either measure registers of single qubits. This might depend on type, so that by default qdouble are measured register wide but qbitmaps require an argument specifying which bit to measure.

There is also the possibility of including quantum conditional operations, such as a quantum if statement. Ignoring for the moment the (extreme) difficulty of implementing such a construction, there are several ways the syntax might work. One is to have a standard if statement which automatically becomes as classical or quantum if depending on the type of the argument. Alternatively, there may be two keywords, cif (classical-if) and qif (quantum-if), which make the distinction obvious in the source code.

6.3.2 Higher level languages

The purpose of higher level languages is to provide a readable syntax for common language constructions. These can then be translated by a compiler into assembly language for a particular machine. This removes the need for the programmer to understand machine code, and also makes the language portable across platforms with different instruction sets.

In classical computing, high level programming languages contain control flow statements (if, for, while, etc.), type checking (e.g. whether a register is interpreted as an integer or a character), syntax for defining functions, etc.

In a quantum programming language, many of these remain. It would still be of interest to perform classical operations such as classical if-statements and classical function calls. However, there would need to be new features to account for the new quantum elements that are accessible to the programmer, as we describe below.

6.3.3 High- and low- level quantum languages

functional vs object oriented. High level languages is all about adding abstraction: language constructs are not closely related to computer hardware features, but should be easier/more intuitive to use, and should work across different hardware implementations.

Low level languages:more quantumness, harder to use, but more closely related to hardware and consequently not portable. E.g. assembly based on the instruction set architecture, C constructs are based closely on instructions (if, while, do, based on branch instructions. arithmetic, bit manipulation, variable storage and access are all cpu instructions).

6.3.4 Examples of different types of languages

Discuss the languages included in the language section: Q#, Quil, Quiskit, Scaffold [1].

The majority of currently available quantum languages are (high level languages with object oriented structure such as Python and C# but are used as low level languages). As discussed in section **REF** the languages reduce to listing quantum gate instructions, building up the logic circuit for the algorithm by essentially by hand. This can pose problems for algorithms which make use of a quantum oracle which is a unitary $U(f(x))$ which depends on a classical function $f(x)$ as calculating an optimal reversible version of $f(x)$ is hard classically to compute **REF (p polling)**.

Python is an interpreted and C# uses JIT compilation, Scaffold resembles the most compiled language present in our discussion.

The current languages discussed here use individual qubits focusing on the ability to address each qubit individually. We hope that future languages will avoid doing this as it poses problems for many qubit machines.

---

[1] C

6.3.5 Compilers

Compilers break down programming language code into the instruction set of the machine. The compiler is more or less important depending on the amount of abstraction in the language. The difficulty of implementing the compiler depends directly on how much abstraction is already contained in the instruction set. By definition, assembly language does not require a compiler because it is already written in terms of machine instructions. At the moment the machine code of all existing quantum information processors is their gate set, meaning that the compiler must perform gate synthesis (see **??** for a more detailed discussion on gate synthesis).

This is contrary to the implementation of classical computers, whose instruction set is a much higher level than logic gates. We anticipate that large scale quantum computers will eventually have instruction sets that contain many more quantum operations in addition to quantum gates, meaning that the burden on the compiler will be greatly reduced.

review compiler progress with references. Include the 'compiler' for pyquil. comment on the range of meanings 'quantum compiler' seems to have. Are there any actual compilers? Is the scaffold compiler legit?

language design what about quantum computing as a whole influences the design of the language for future languages. How does the quantumness show up at different levels of abstraction (i.e. gates at the bottom, quantum-if statements a bit of the way up, ..., languages with no explicit references to quantum stuff?)

6.3.6 Linkers

Oli's fault not finished. connectivity, memory allocation (qubit allocation),

6.4 The future

6.4.1 QLANG +− ™ the Quantum programming language

We introduce the QLANG+−™ language qubytes™ and quibbles™.

Bibliography

[1] Will Zeng, Blake Johnson, Robert Smith, Nick Rubin, Matt Reagor, Colm Ryan, and Chad Rigetti. First quantum computers need smart software. *Nature*, 549(7671):149–151, sep 2017.

[2] http://www.qusoft.org/quantum-software-manifesto/.

[3] Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition*. Cambridge University Press, 2010.

[4] http://www.iop.org/cs/page_43644.html.

[5] Stephen G Odaibo. A quantum mechanical review of magnetic resonance imaging. *arXiv preprint arXiv:1210.0946*, 2012.

[6] John Bardeen and Walter Hauser Brattain. The transistor, a semi-conductor triode. *Physical Review*, 74(2):230, 1948.

[7] Tim Cross. After Moore's law. *The Economist Technology Quarterly*, 2016.

[8] https://www.dwavesys.com/home.

[9] https://en.wikipedia.org/wiki/Orders_of_magnitude_(mass).

[10] John Preskill. Quantum computing and the entanglement frontier, 2012.

[11] Richard P Feynman. Simulating physics with computers. *International journal of theoretical physics*, 21(6-7):467–488, 1982.

[12] IM Georgescu, Sahel Ashhab, and Franco Nori. Quantum simulation. *Reviews of Modern Physics*, 86(1):153, 2014.

[13] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 549(7671):195, 2017.

[14] Martin Schaden. Quantum finance. *Physica A: Statistical Mechanics and its Applications*, 316(1-4):511–538, 2002.

[15] https://research.google.com/teams/quantumai/.

[16] https://research.googleblog.com/2018/03/a-preview-of-bristlecone-googles-new.html.

[17] https://www.research.ibm.com/ibm-q/.

[18] https://newsroom.intel.com/press-kits/quantum-computing/.

[19] https://www.rigetti.com.

[20] Andrew Steane. The ion trap quantum information processor. *Applied Physics B: Lasers and Optics*, 64(6):623–643, 1997.

[21] Daniel Loss and David P DiVincenzo. Quantum computation with quantum dots. *Physical Review A*, 57(1):120, 1998.

[22] https://newsroom.intel.com/news/intel-sees-promise-silicon-spin-qubits-quantum-computi

[23] Terry Rudolph. Why i am optimistic about the silicon-photonic route to quantum computing. *APL Photonics*, 2(3):030901, 2017.

[24] https://phys.org/news/2018-06-ornl-summit-supercomputer.html.

[25] Patrick J. Coles, Stephan Eidenbenz, Scott Pakin, Adetokunbo Adedoyin, John Ambrosiano, Petr Anisimov, William Casper, Gopinath Chennupati, Carleton Coffrin, Hristo Djidjev, David Gunter, Satish Karra, Nathan Lemons, Shizeng Lin, Andrey Lokhov, Alexander Malyzhenkov, David Mascarenas, Susan Mniszewski, Balu Nadiga, Dan O'Malley, Diane Oyen, Lakshman Prasad, Randy Roberts, Phil Romero, Nandakishore Santhi, Nikolai Sinitsyn, Pieter Swart, Marc Vuffray, Jim Wendelberger, Boram Yoon, Richard Zamora, and Wei Zhu. Quantum algorithm implementations for beginners, 2018.

[26] Nathan Killoran, Josh Izaac, Nicolás Quesada, Ville Bergholm, Matthew Amy, and Christian Weedbrook. Strawberry fields: A software platform for photonic quantum computing, 2018.

[27] Dawid Kopczyk. Quantum machine learning for data scientists, 2018.

[28] http://dkopczyk.quantee.co.uk/category/quantum_computing/.

[29] Ryan LaRose. Quantum machine learning for data scientists, 2018.

[30] James R. Wootton. Benchmarking of quantum processors with random circuits, 2018.

[31] Lucien Hardy. Quantum theory from five reasonable axioms. *arXiv preprint quant-ph/0101012*, 2001.

[32] https://www.rigetti.com/forest.

[33] Henry Maurice Sheffer. A set of five independent postulates for boolean algebras, with application to logical constants. *Transactions of the American mathematical society*, 14(4):481–488, 1913.

[34] https://commons.wikimedia.org/wiki/File:NAND_ANSI_Labelled.svg.

[35] https://commons.wikimedia.org/wiki/File:NOR_ANSI_Labelled.svg.

[36] Rigetti Computing. pyquil documentation, 2018. https://media.readthedocs.org/pdf/pyquil/latest/pyquil.pdf.

[37] Robert S Smith, Michael J Curtis, and William J Zeng. A practical quantum instruction set architecture, 2016.

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

[39] https://rebootingcomputing.ieee.org/rebooting-computing-week/industry-summit-2017.

[40] Damian S. Steiger, Thomas Häner, and Matthias Troyer. Projectq: An open source software framework for quantum computing. *arXiv:1612.08091*, 2018.

[41] http://vergil.chemistry.gatech.edu/notes/hf-intro/hf-intro.pdf.

[42] Lov K. Grover. A fast quantum mechanical algorithm for database search. In *Proceedings of the Twenty-eighth Annual ACM Symposium on Theory of Computing*, STOC '96, pages 212–219, New York, NY, USA, 1996. ACM.

[43] Stephane Beauregard. Circuit for shor's algorithm using 2n+3 qubits. *Quantum Info. Comput.*, 3(2):175–185, March 2003.

[44] B Lekitsch, S Weidt, AG Fowler, K Mølmer, SJ Devitt, C Wunderlich, and WK Hensinger. Blueprint for a microwave trapped-ion quantum computer. *arXiv preprint arXiv:1508.00420*, 2015.

[45] Jacques Carolan, Christopher Harrold, Chris Sparrow, Enrique Martín-López, Nicholas J Russell, Joshua W Silverstone, Peter J Shadbolt, Nobuyuki Matsuda, Manabu Oguma, Mikitaka Itoh, et al. Universal linear optics. *Science*, 349(6249):711–716, 2015.

[46] *dsPIC30F/33F Programmer's Reference Manual*, 2008.

[47] Thomas G Draper. Addition on a quantum computer. *arXiv preprint quant-ph/0008033*, 2000.

[48] P Mittelstaed P Busch, PJ Lahti. The Quantum Theory of Measurement. In *The Quantum Theory of Measurement*. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996.

Appendix A

A gentle introduction to quantum theory

No fair! You changed the outcome by measuring it!

Hubert Farnsworth

In this section we cover some supplementary background quantum theory. Although a deeper knowledge of quantum theory would serve to further your understanding of quantum computing, it should not be necessary for sections 1 to (?). The following textbook is referenced throughout this Appendix which we recommend to interested readers: *Quantum Computation and Quantum Information* by M. A. Nielsen and I. L. Chuang [**?**]. You have seen in Section 1.1 that the state of a system governed by quantum mechanics can be represented by a sum of basis vectors that correspond to binary representations of information. Typically, quantum mechanics is captured by a slightly different notation called Dirac notation, invented by one of the four-fathers of Quantum physics, Paul Dirac. In this section we will introduce Dirac notation and cement the description given in Section 1.1 into a more formal language that will help you to probe deeper into more advanced literature.

A.1 Quantum mechanics

A.1.1 Quantum States

In quantum mechanics, we describe the state of a system simply with labels. These labels we assign to the system, such as '0','1', aim to give some intuition about the state of the system. We could equally have used 'open', 'closed' if appropriate. Equally this should be viewed as assigning information to a state. For example, if a state is labelled '110' then in binary representation, the state carry's the information '6'. Formally these labels are called quantum numbers and in general are not restricted to be one of two values.

For example, imagine the 4 of spades was chosen from a deck of cards, a good choice of label to describe the card would be '4' or '♠'. Equally in a quantum system we would say that the card is

in the state '4' or '♠' which we write formally as $|4\rangle$, or $|\spadesuit\rangle$. In this scenario the quantum number '4' could have been one of 13 values, therefore the set of states needed to fully describing the system (the system here being a randomly chosen card) are $\{|A\rangle, |2\rangle, |3\rangle, \ldots, |K\rangle\}$ with quantum numbers $\{A, 2, 3, \ldots K\}$. If the set of quantum numbers are unique and their associated states describe all possible values a properties can take then they are said to form a Hilbert space $\mathscr{H}$ of the system [**?**] *p66*. Formally we say the *complete* set of *independent* and *orthonormal* states describing a system span a Hilbert space:

$$\mathscr{H} := span\{|A\rangle, |2\rangle, |3\rangle, \ldots, |K\rangle\} \tag{A.1}$$

The Hilbert space should loosely be thought of as a vector space. It's purpose is to mathematically define all the possible states a system can occupy. This is a very useful tool when we start to describe the evolution of a system because it had better be the case that our description of a system remains physically possible, i.e. remains within the Hilbert space. For example, it would make no sense to talk about the state $|A\rangle$ evolving to the state $|\spadesuit\rangle$. In that sense, the Hilbert space helps define the boundaries of a system.

In Dirac notation, $|\ldots\rangle$, used to describe a state, is called a 'ket' and for every 'ket' there is a 'bra' conversely written as $\langle\ldots|$. The names originates from the first and second halves of the word 'bra(c)ket', which when placed together resemble the most important operation in quantum computation: the inner product. For a formal introduction to Dirac notation *see p13* of [**?**]. The inner product is a simple function that does the following on basis states in the Hilbert space:

if (state $|a\rangle$ *is the same as state* $|b\rangle$*)*
return 1
else
return 0

In Dirac notation this operation is performed by turning the ket $|a\rangle$ into a bra, $\langle a|$ (this process is described more formally in section (?) but for now can be thought of as just a bracket swap). The 'bra' $\langle a|$ and 'ket' $|b\rangle$ are then used to form the word 'bra(c)ket' and is equal to 1 or 0 depending on whether the state a is equal to the state b.

For example, returning to our deck of cards, the inner product of the states $|A\rangle$ with $|5\rangle$ is written as:

$$\langle A|5\rangle = 0 \tag{A.2}$$

Conversely, the inner product of the states $|J\rangle$ with $|J\rangle$ would be:

$$\langle J|J\rangle = 1 \tag{A.3}$$

When a set of states are unique in this way, they are said to be orthonormal. The inner product

is important as it allows you to check that all the states that form your Hilbert space are orthonormal (remember this is a key requirement for a set of states to form a Hilbert space) and will become very useful when we introduce superposition in the next section.

To draw a parallel with Section 1.1, the state $|5\rangle$ (or $|101\rangle$ as we are free to choose the label) can be mathematically represented by the vector $\begin{pmatrix}0\\0\\0\\0\\1\\0\\0\\0\end{pmatrix}$ in which each element corresponds to a binary number. In this case, taking the inner product simply involves performing the dot product between two vector representations.

It should be noted that the dimension of a Hilbert space is equal to the number of basis states that span it. In the case of our deck of cards, the Hilbert space of card values has dimension 13, however, our mathematical representation only allows for values 0 (000) to 7 (111) and so cannot fully capture the Hilbert space. This is a reflection of the fact that binary representation of information only increase in dimension size by powers of 2. This means that strictly speaking, the full Hilbert space cannot be represented in this way unless a power of two.

You may be wondering why we bother with such a representation if in general it cannot always fully capture the system, and the answer is because typically in quantum computing we only deal with qubits which on there own have 2 dimensional Hilbert spaces spanned by $|0\rangle$ and $|1\rangle$. As a result, when two qubits are combined, their joint Hilbert space has 2^2 basis states spanned by $|00\rangle, |01\rangle, |10\rangle, |11\rangle$. In this way, it will always be possible to capture the Hilbert space of n qubits with a binary representation of 2^n labels. Composite systems of Hilbert Spaces will be discussed further in Section (?).

A.1.2 Superposition

In quantum mechanics, using the states that form the basis of our Hilbert space to describe the system is not enough. Observation of quantum systems tell us that when we prepare a state multiple times and measure it, the outcome will not always be the same but instead follow some probabilistic statistics. At this point its crucial to point out that the formalism does not try to explain why this is the case, but only provides a way of describing the system. I highlight this fact for the following reason. Typically when confronted with a new phenomena, we turn to the mathematical description to gain some intuition of its causes. With quantum mechanics, the formal description should not be used to find a "logical" explanation of how superposition works but only used to help fully describe the observed physics of the system.

To understand how we can describe this phenomena formally, lets make our deck of cards a quan-

tum deck of cards that now obeys the laws of quantum mechanics and see how our observations are captured within the mathematical formalism.

The first thing to note is that the act of measuring appears to perform an operation on the system that takes it from its superposition state to a basis state of the Hilbert space. The pre-measurement 'superposition state' contains information about which outcomes we will attain and with what probability we will attain each measurement outcome.

For example, lets say you are given a face down card that when flipped multiple times is sometimes a queen and sometimes a 4 (strange I know, but totally allowed within quantum mechanics). Before performing the operation of turning it over the state of the card should be thought of as being in a superposition of $|Q\rangle$ and $|4\rangle$. That is to say, the state used to describe the system before the measurement operation is not just one basis state but two, containing some probabilities that describe how often we get each outcome. Only under the measurement operation (in this case the act of flipping the card) does the state become one of the two basis states. In general we may not be aware of how likely each outcome is and so to build up a picture of the superposition state we must re-prepare the experiment and repeating the same measurement many times. Since all we have to describe the systems pre-measurement state is the probabilities of getting each state after measurement, this is what's use in the mathematical description.

Formally, given a state $\big|\psi\big\rangle$ that is said to be in a superposition of the states $|a\rangle$ and $|b\rangle$ where the probability of getting the state $|a\rangle$ is $|\alpha|^2$ and the probability of getting the state $|b\rangle$ is $\big|\beta\big|^2$ then we describe the state as:

$$\big|\psi\big\rangle = \alpha\,|a\rangle + \beta\,|b\rangle \tag{A.4}$$

In general, both α and β can take complex values and so to ensure that the probabilities are real we take the absolute value squared [**?**] *p80*. The significance of having complex valued amplitudes will be discussed on the next page but for now its sufficient to consider them as real.

Returning to our example, say we get a queen one third of the time and a four two thirds of the time. We would describe the pre-measurement superposition state as:

$$\big|\psi\big\rangle = \frac{1}{\sqrt{3}}\,|Q\rangle + \frac{2}{\sqrt{6}}\,|4\rangle \tag{A.5}$$

where

$$\left|\frac{1}{\sqrt{3}}\right|^2 + \left|\frac{2}{\sqrt{6}}\right|^2 = 1 \tag{A.6}$$

as expected since the probabilities of getting a queen or a four must sum to one. It is true that $\big|\psi\big\rangle \in \mathscr{H}$ since any linear combination of the basis states lives within the span of those basis states.

Aside: Why do we use complex amplitudes?

In general the amplitudes α and β are used not only to keep track of the probabilities of possible measurement outcomes, but also to account for a second important property, namely, the relative phase between each state. The need for a phase, as well as a magnitude, originates from our observations of wave-like interference between two superposition states. From experimental observations we know that quantum mechanical particles (even particles with mass) have inherent wave like properties such as wavelength and phase that must be reflected within the mathematical description. Each basis state in the superposition has a relative phase that helps describe how the overall state will constructively or destructively interfere when combined with another state. For a review of the general postulates of quantum mechanics see [**?**] *p96*.

Complex numbers are useful because they naturally contain a phase. Each complex number can be parameterised as $\alpha = |\alpha|e^{i\theta}$ where the angle θ is the phase angle that can take any value between 0 and 2π. To help see the significance of the phase let's look at the difference between two states with different phases such as: $|\psi_a\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ and $|\psi_b\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. Both are superposition's of the state 0 and 1 with equal probabilities but with different relative phases preceding the 1 state (note that $e^{i\pi} = -1$). The relative phase difference will manifest itself when the two states interfere with each other, the result of which can be seen by taking the inner product between them just as before, except now the states are superposition states with complex amplitudes instead of basis states. Lets take this opportunity to introduce a more formal definition of the inner product of two arbitrary states and practice how to perform such a product.

To compute the inner product we take the 'bra' of $|\psi\rangle$ as before in Example A.2 except now the 'bra' of $|\psi\rangle$ is given by the complex conjugate of the amplitudes multiplying the basis states within $|\psi\rangle$. For example, given $|\psi\rangle = (\alpha|a\rangle + \beta|b\rangle)$ and $|\phi\rangle = (\gamma|a\rangle + \rho|b\rangle)$ where $\{\alpha, \beta, \gamma, \rho\} \in \mathbb{C}$ then $\langle\psi| = (\overline{\alpha}\langle a| + \overline{\beta}\langle b|) \equiv (|\psi\rangle)^\dagger$. More formally the process described above is called 'taking the adjoint' of the state and is signified by the dagger. The complex conjugate of a complex number (denoted by an overhead bar) is given by simply negating the complex part. For example, if $\alpha = a + ib$ then the complex conjugate is given by $\overline{\alpha} = a - ib$. The 'bra' is then used with the 'ket' to form 'braket' just as before in Example A.2. Since the inner product is a distributive operation, the 'braket' can be expanded out just as in normal multiplication making sure each 'bra' is combined with every 'ket':

$$\begin{aligned}\langle\psi|\phi\rangle &= (\overline{\alpha}\langle 0| + \overline{\beta}\langle 1|)(\gamma|0\rangle + \rho|1\rangle) \\ &= (\overline{\alpha}\gamma\langle 0|0\rangle + \overline{\alpha}\rho\langle 0|1\rangle + \overline{\beta}\gamma\langle 1|0\rangle + \overline{\beta}\rho\langle 1|1\rangle) \\ &= (\overline{\alpha}\gamma + \overline{\beta}\rho)\end{aligned} \tag{A.7}$$

For example, using $|\psi_a\rangle$ and $|\psi_b\rangle$ given above, the inner product is as follows:

$$\begin{aligned}\langle\psi_a|\psi_b\rangle &= \frac{1}{2}(\langle 0| - \langle 1|)(|0\rangle + |1\rangle) \\ &= \frac{1}{2}(\langle 0|0\rangle + \langle 0|1\rangle - \langle 1|0\rangle - \langle 1|1\rangle) \\ &= \frac{1}{2}(1+0+0-1) \\ &= 0\end{aligned} \tag{A.8}$$

We can see above that due to the minus phase on the basis state $|1\rangle$ in the superposition state $|\psi_a\rangle$ the inner product is 0, or in other words, when combined the overlap of the two superposition states destructively interfere giving a zero inner product i.e. the two states are orthogonal to each other. Had the phase been different, the inner product could have taken a non-zero complex value signifying some constructive interference between them. For a more in-depth discussion of superposition see [**?**] *p13*.

A.1.3 Operators

So far we have discussed how to formally describe the state of a quantum mechanical system and how to calculate the probability of a measurement outcome on a superposition state. We also introduced the notion of a measurement being a type of operation that maps superposition states to basis states. In general, all measurements are represented by *Hermitian* operators that have some specific properties. General quantum mechanical measurement is beyond the scope of this guide and instead we will restrict our discussion to the computational basis state $\{|0\rangle, |1\rangle\}$ used in quantum computation to describe qubits (see [**?**] for a more general discussion of measurement in quantum mechanics). In this restricted case, a measurement determines the value of the qubit (0 or 1) with some probability determined by the state it's in.

An operator can be thought of as a mapping of one state to another, like applying a BIT-FLIP to the basis states in a superposition or even the mapping of a basis state into a superposition state. These types of simple operation form the basis of all quantum computation, much like AND, OR and COPY in classical computation, except now the rules are different. One key difference is that there is no COPY operation since in order to copy a state, it must first be measured. The issue is that measuring collapses (or projects) a state to a basis state so we can never know the exact form of the pre-measurement state and therefore cannot make a true copy. We can express the idea of a general operation by the following equation:

$$|\phi\rangle = \hat{O}|\psi\rangle \tag{A.9}$$

The most trivial operator we can imagine is the identity operator $\hat{I}$ that just maps a state $|\psi\rangle$ to itself. In order to remain physically reasonable the operator $\hat{O}$ has a number of important restrictions that are listed and explained below.

1. $\hat{O}: \mathscr{H} \rightarrow \mathscr{H}$. The operator must map states of a Hilbert space to other states in the same Hilbert space. Remember that the Hilbert space defines the boundaries of our physical system which must be respected as we are considering only closed systems.

2. The operator must preserve the inner products between the basis states. This is the same as saying that an operator cannot change the fundamental nature of the basis states. For example, the 0 and 1 state must always be orthonormal to each other otherwise they will no longer form the Hilbert space and the boundaries of our system will then change. This results in the equality $\hat{O}^\dagger \hat{O} = \hat{I}$ (we will see what the adjoint of an operator is shortly). Interestingly, for the above to be true the operator must have an inverse and therefore must be reversible.

Operators are mathematically represented in the computational basis by a string of back to back 'kets' and 'bras'. This form allows the mapping of one state to another via use of the inner product. When operating on a qubit state $|\phi\rangle$ from the left the operator 'bras' combine with the 'kets' of the state replacing them the 'kets' from the operator. This results in a mapping of a state to another state. For example, given an operator $\hat{X}$ that performs a BIT-FLIP mapping 0 to 1 and 1 to 0. We represent this operator in the computational basis as:

$$\hat{X} = |0\rangle\langle 1| + |1\rangle\langle 0| \tag{A.10}$$

To see that the operator acts as we expect, let's act it upon the state $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$:

$$\begin{aligned} \hat{X}|\phi\rangle &= (|0\rangle\langle 1| + |1\rangle\langle 0|)(\alpha|0\rangle + \beta|1\rangle) \\ &= \alpha|0\rangle\langle 1|0\rangle + \beta|0\rangle\langle 1|1\rangle + \alpha|1\rangle\langle 0|0\rangle + \beta|1\rangle\langle 0|1\rangle \\ &= \beta|0\rangle + \alpha|1\rangle \end{aligned} \tag{A.11}$$

The resulting state is as we would expect with the basis states 0 and 1 flipped. This is just one example of an important operator in quantum computation. From point two above, we know that every operator must have an adjoint form, just like the states of a system. The origin of adjoint operators follows from standard linear algebra and will not be discussed in this guide. For a more detailed explanation of the form operators and their adjoints, see (REF).

An operator can also be expressed in matrix form whereby each column maps a basis state to a new combination. For example, take the Hadamard operator H that maps each basis sate $|0\rangle$ and $|1\rangle$ to each superposition with opposite relative phase. This is a key operator is quantum computation because it allows functions to be applied to both basis states at the same time since there are both basis states present in the superposition. To represent H in matrix notation we write:

$$\hat{H} = \frac{1}{\sqrt{2}}\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{A.12}$$

The first column maps $|0\rangle$ to the state $\frac{1}{\sqrt{2}}(|0\rangle+|1\rangle) \equiv |+\rangle$ and the second column maps $|1\rangle$ to $\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle) \equiv |-\rangle$. The state $|+\rangle$ and $|-\rangle$ are frequently used and so given their own names. The Dirac form of $\hat{H}$ is:

$$\hat{H} = |+\rangle\langle 0| + |-\rangle\langle 1| \tag{A.13}$$

Single qubit operators are useful for state manipulation and are used frequently in lager systems of qubits. In general, any operation on a system of qubits can be decomposed into single and two qubit operators (or gates). It should be noted that the mapping of a large unitary operation to a sequence of single and two qubit gates in nontrivial and typically theoretical descriptions of quantum algorithms are presented as larger operators, often acting on the entire Hilbert space, with the assumption that there exists a decomposition that can be physically applied. To understand the formalism behind this we must generalise our description to beyond the single qubit.

A.1.4 Beyond the Single Qubit

We have already mentioned that a system of n qubits has a Hilbert space of dimension 2^n, but how do we go about finding what form the basis states take. Formally we combine two or more closed systems via the tensor product of their Hilbert spaces. We introduce the tensor product of two qubits that live in $\mathscr{H}_1$ and $\mathscr{H}_2$ respectively, denoted $\mathbb{C}_1^2 \otimes \mathbb{C}_2^2$, by taking the tensor of each combination of basis states where the first state in the tensor is from $\mathscr{H}_1$ and similarly for the second. This process enables us to build the basis states of the combined Hilbert space one at a time:

$$\begin{aligned} |0\rangle \otimes |0\rangle &\equiv |00\rangle \\ |0\rangle \otimes |1\rangle &\equiv |01\rangle \\ |1\rangle \otimes |0\rangle &\equiv |10\rangle \\ |1\rangle \otimes |1\rangle &\equiv |11\rangle \end{aligned} \tag{A.14}$$

$$\mathscr{H}_{1+2} := span\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\} \tag{A.15}$$

We have conveniently chosen to labelled the four basis states of the joint Hilbert space with quantum numbers that give us information about the basis states that form them. We could then assign 00 to 0, 01 to 1, 10 to 2 and 11 to 3 in a binary encoding of the integers 0-3. Note that sometimes in the literature, authors will write $|0\rangle|1\rangle$ instead of $|01\rangle$, however they are equivalent.

Thus an arbitrary superposition state living in $\mathscr{H}$ can now take the form:

$$|\psi\rangle = a_{00}|00\rangle + a_{01}|01\rangle + a_{10}|10\rangle + a_{11}|11\rangle = \sum_{i,j=0}^{1} a_{ij}|ij\rangle \tag{A.16}$$

In general, if two qubits are in the state $|\psi\rangle_1 = |+\rangle$ and $|\psi\rangle_2 = |-\rangle$ respectively, then there joint state is described by:

$$\begin{aligned}\big|\psi\big\rangle_1 \otimes \big|\psi\big\rangle_2 &= \big(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\big)_1 \otimes \big(\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\big)_2 \\ &= \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)\end{aligned} \tag{A.17}$$

The tensor product generalises to any number of qubits and the number of basis states scale by $O(2^n)$. To perform operations on some or all of the qubits in a system, the dimension of your operator must match that of the Hilbert space that it operates on. For example, say we have a system of 4 qubits and we want to act with $\hat{H}$ on qubit 2 and $\hat{X}$ on qubit 4. The total operator we act on the state will be given by: $\hat{I} \otimes \hat{H} \otimes \hat{I} \otimes \hat{X}$ where the identity acts on qubit 1 and 3. We can depict this operation by a circuit model diagram as described in Section (?).

A.1.5 An Intro to Quantum Information Theory

In this section we will used what we have learnt in Section A to examine a basic example of a quantum algorithm.

The first thing to note is that the size of you Hilbert space (ie the number of qubits you have) ultimately determines the domain over which a computation can be performed. This is intuitive because each (orthogonal) basis state provides an opportunity to attach a label that is unique to all others in the system. For example we could provide a database entry to each basis state that can then be fed into a computation. When dealing with a large number of qubits, its often easier to bunch them together into registers with each register having a specific role. Due to the inability to copy states and the need for each operation to be reversible, it is often necessary to introduce 'redundant' groups of qubits into your system to help perform computations. These extra qubits are known as ancilla qubits. To help demonstrate one of the needs for ancilla qubits, lets examine how to perform a simple AND (which in general is not reversible operation) between two qubits.

Given two qubits $|x_1\rangle$ and $|x_2\rangle$ where $x_1, x_2 \in 0, 1$, the AND operation should have the following truth table:

$\|x_1\rangle \otimes \|x_2\rangle$	$\|x_1 \wedge x_2\rangle$
00	0
01	0
10	0
11	1

(A.18)

Since the outcome 0 is degenerate, the operation cannot be reversed. Instead, the outcome is added modulo 2 to a third ancilla qubit $|z\rangle$ (whose initial value is usually set to 0). This will then give the follow truth table:

$\|x\rangle_1 \otimes \|x\rangle_2 \otimes \|z\rangle$	$z \oplus x_1 \wedge x_2$
000	0
010	0
100	0
110	1

(A.19)

To check that this process is reversible, let's perform the same operation where z is now the the output of the first AND above. This should map back to the original blue0 states.:

$\|x\rangle_1 \otimes \|x\rangle_2 \otimes \|z \oplus x_1 \wedge x_2\rangle$	$(z \oplus x_1 \wedge x_2) \oplus x_1 \wedge x_2$
000	0
010	0
100	0
111	0

(A.20)

To obtain universal deterministic classical computation, it is sufficient to be able to implement the NOT and AND gates. For quantum computation, it's sufficient to be able to perform arbitrary single qubit control, i.e. the ability to map either basis state to any superposition, and the CNOT two qubit gate which performs a controlled not on the second qubit based on the value of the first.

A.2 Exercises?

Appendix B

A gentle tutorial to Python

Let's put a nice quote here!

Andres

Please, if someone can do this? we are promising in the preface that we were gonna have something like this xD.

There are two main versions of Python used, 2.7 and 3.5/3.6. Throughout this guide we use Python3 and refer to it as Python.

has a large amount of documentation available

libraries vectors lists matrices multiplication Kronecker products complex numbers

- Syntax
- Linear algebra mostly

B.1 Unix install

B.2 MacOS install

B.3 Windows install

Appendix C

Answers to Exercises

Not a priority right now, but to keep it in mind! :3

Answers for Chapter 2

Answers for Chapter 4

Appendix D

Complete code examples

Shor

D.0.1 Shor in Q#

```
using Microsoft.Quantum.Simulation.Core; // to use Q#
using Microsoft.Quantum.Simulation.Simulators; // to use Q#
using System; // to use several inbuilt classes
using System.Collections.Generic; // to allow the use of List


namespace Quantum.Shor{

    /**
     * Main class containing the factoring method and a small program using it.
     */
    class Shor {

        /**
         * Entry point of program
         */
        static void Main(string[] args){

            System.Console.WriteLine("Please enter an integer you would like to factor:");
            int toBeFactored = Int32.Parse(System.Console.ReadLine());

            try {
                int factorVal = Factor(toBeFactored);
                System.Console.WriteLine(String.Format
                    ("A factor of {0:d} is {1:d}.", toBeFactored, factorVal));
            } catch (System.FormatException sfExp) {
                System.Console.WriteLine("The value you input was not a valid integer.");
            } catch (QuantumAlgFailedException qExc) {
                System.Console.WriteLine(String.Format
                    ("Algorithm failed, because {0}", qExc.Message));
            }

            System.Console.WriteLine("Press any key to continue...");
            System.Console.ReadKey();
        }

        /**
        * Computes a factor of a given integer using Shor's algorithm.
        * Throws a QuantumAlgFailedException in case the quantum algorithm fails.
        */
        static int Factor(int N) {

            // Check if N is even
            if (N % 2 == 0) {
                return 2;
            }

            // Find random integer a such that 2<a<N
            Random rnd = new Random();
            int a = rnd.Next(2, N);

            // Check if a and N are coprime
```

```
53              int b = GCD(a, N);
54              if (b != 1) {
55                  return b;
56              }
57
58              // Quantum algorithm to find approximate periodicity
59              int y;
60              using (var sim = new QuantumSimulator()) {
61                  var outcome = ApproximatePeriodicity.Run(sim, N, a).Result;
62                  y = (int)outcome;
63              }
64
65              // Determine order r of a modulo N from quantum outcome with continued
    fractions
66
67              // M is smallest power of 2 larger than N^2
68              double Nsquared = Math.Pow((double)N, 2.0);
69              int M = (int)Math.Pow(2.0, Math.Ceiling(Math.Log(Nsquared) / Math.Log(2.0)));
70
71              // Find the convergents of the continued fractions expansion of y/M
72              List<(int, int)> convergents = FindConvergents(y, M);
73
74              int r;
75              double z = (double)y/ (double)M;
76              double maxDiff = 1 / (double)(2 * N ^ 2);
77              foreach ((int num, int denom) in convergents){
78                  double fraction = (double)num / (double)denom;
79                  if (Math.Abs(fraction-z)<=maxDiff && denom < N) {
80                      r = denom;
81
82                      // Check if r is even
83                      if (r % 2 != 0) {
84                          continue;
85                      }
86
87                      // Find the greatest common divisor between a^(r/2)-1 and N
88                      int s = GCD((int)Math.Pow(a,(r / 2)) - 1, N);
89
90                      if (s == 1 || s == N) {
91                          continue;
92                      }
93
94                      return s;
95                  }
96              }
97
98              throw new QuantumAlgFailedException
99                  ("no suitable order r could be found: either r was odd or s was 1");
100         }
101
102         /**
103          * Computes greatest common divisor between x and y, assuming x and y are not 0.
104          * Uses the Euclidean algorithm.
105          */
```

```
106         static int GCD(int x, int y) {
107 
108             // Ensure x > y
109             if (x < y) {
110                 int swap = y;
111                 y = x;
112                 x = swap;
113             }
114 
115             // Apply the Euclidean algorithm.
116             int remainder;
117             while (y != 1) {
118                 remainder = x % y;
119                 if (remainder == 0)
120                     return y;
121 
122                 x = y;
123                 y = remainder;
124             }
125 
126             return y;
127         }
128 
129         /**
130          * Computes the convergents of the continued fraction expansion of a given fraction
    .
131          */
132         static List<(int, int)> FindConvergents(int num, int denom) {
133 
134             List<int> denominators = new List<int>();
135             int wholeNumber;
136             int remainder;
137 
138             // Find the denominator values for the continued fraction
139             while (denom % num != 0) {
140                 wholeNumber = denom / num;
141                 remainder = denom % num;
142 
143                 denom = num;
144                 num = remainder;
145 
146                 denominators.Add(wholeNumber);
147             }
148             denominators.Add(denom / num);
149 
150             // Find the actual convergents: use the method a+b/c = (a*c+b)/c sequentially
151             List<(int, int)> convergents = new List<(int,int)>();
152             convergents.Add((1, denominators[0]));
153             int swap;
154             for (int i = 1; i < denominators.Count; i++) {
155                 denom = denominators[i];
156                 num = 1;
157 
158                 for (int j = i; j>0; j--) {
```

```
                    wholeNumber = denominators[j - 1];
                    num = num + wholeNumber * denom;

                    swap = num;
                    num = denom;
                    denom = swap;
                }

                convergents.Add(SimplifyFraction(num, denom));
            }

            return convergents;
        }

        /**
         * Simplifies a given fraction.
         * E.g. when given (3,9) it will return (1,3)
         */
        static (int,int) SimplifyFraction(int num, int denom) {

            int commonFactor = GCD(num, denom);
            return (num / commonFactor, denom / commonFactor);
        }


    }

    /**
     * An Exception to handle failures in the algorithm.
     */
    public class QuantumAlgFailedException : Exception {

        public QuantumAlgFailedException(string message) : base(message) {
        }
    }

}
```

```
namespace  Quantum.Shor{

  open Microsoft.Quantum.Primitive;
  open Microsoft.Quantum.Canon;
 open Microsoft.Quantum.Extensions.Math;
 open Microsoft.Quantum.Extensions.Convert;


 /// # Summary
 /// Sets the given qubit to the |0> state in the computational basis
 /// # Input
   /// ## q1
   /// The qubit to be operated on
   operation SetZero (q1: Qubit) : (){

       body{
```

```
17
18              let current = M(q1);
19
20              if (current != Zero){
21                  X(q1);
22              }
23          }
24      }
25
26  /// # Summary
27  /// Given N and coprime a, calculates the approximate period of f(x) = (a^x) mod N
28  /// # Input
29    /// ## N
30    /// An integer, the modulus in f(x)
31    /// ## a
32    /// A random integer coprime to N, the base of the exponential in f(x)
33    /// # Output
34    /// Int: the approximate period
35  operation ApproximatePeriodicity(N: Int, a: Int) : (Int){
36
37        body{
38      let len1 = Ceiling(Log(ToDouble(N^2))/Log(ToDouble(2))); // length register 1
39      let len2 = Ceiling(Log( ToDouble(N) )/Log(ToDouble(2))); // length register 2
40
41      mutable outcome1 = 0;
42
43      // Allocate all qubits required
44      using (qubits = Qubit[len1+len2]){
45
46        // Create the two registers
47        mutable indexReg1 = new Int[len1];
48        for (i in 0..len1-1){
49          set indexReg1[i] = i;
50        }
51        let reg1 = Subarray(indexReg1, qubits);
52
53        mutable indexReg2 = new Int[len2];
54        for (i in 0..len2-1){
55          set indexReg2[i] = len1+i;
56        }
57        let reg2 = Subarray(indexReg2, qubits);
58
59        // Apply H to first register
60        ApplyToEach (H, reg1);
61
62        // Apply O_f for f(x) = a^x mod N-> use method from paper of Stephane Beauregard
63        for (i in 0..len1-1){
64          (Controlled ModularQubitMultiplyByExp)
65                      ([reg1[i]], (LittleEndian(reg2), a^2, i, N));
66        }
67
68        // Measure 2nd register
69        let outcome2 = MeasureInteger(LittleEndian(reg2));
70
```

```
71          // Apply Q_N to first register
72          (Adjoint QFTLE)(LittleEndian(reg2));
73
74          // Measure first register -> outcome k
75          set outcome1 = MeasureInteger(LittleEndian(reg1));
76
77          // Clean qubits
78          ApplyToEach(SetZero, qubits);
79
80        }
81
82        return outcome1;
83      }
84    }
85
86    /// # Summary
87    /// Gives mapping |x> -> |(x*a^b) mod N)
88    /// # Input
89      /// ## stateIn
90      /// A Qubit[] in little endian format: |x>
91      /// ## expBase
92      /// The base of the exponential: a
93      /// ## power
94    /// The power used in the exponential: b
95    /// ## modulus
96    /// The value in which modulus everything is calculated: N
97    /// N and a should be coprime
98      /// # Output
99      /// ()
100   operation ModularQubitMultiplyByExp
101         (stateIn: LittleEndian, expBase: Int, power: Int, modulus: Int):(){
102
103     body{
104       ModularMultiplyByConstantLE(ExpMod(expBase, power, modulus), modulus, stateIn);
105     }
106
107     adjoint auto
108     controlled auto
109     controlled adjoint auto
110   }
111 }
```