Quantum Meta-Programming with templates, by Dummies, for Dummies, with a weak focus on Poisson Bullets, loosely in the style of a book (Deprecated since August 2018).

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 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

August 30, 2018

Contents

Preface

Quantum programming, as you might expect, requires understanding of both quantum theory and programming. we have designed this guide to cover from the basics of quantum theory and programming, to the implementation of quantum programs in already existing quantum computers. 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 believe that you might find something useful out of this guide.

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 software, 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 current quantum programming languages/libraries, example programs and exercises implemented in different quantum languages/libraries, 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 our quantum grand challenge on quantum programming. We are addressing the 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 engineers might find useful.

Quantum regards!

QE-CDT Cohort 4

September, 2018

0.1 Things to be careful with and TODO list

- Python 3!
- Check same spelling of pyQuil, Project Q, Qiskit and Q#
- Check Use of Active voice
- Please use autoref instead of ref alone

- no adjoints!
- reversible NOT unitary
- move all sections to be consisten sizes. NO SECTION* only subsection

Chapter 1

Structure of this guide

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!

1.0.1 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 **??**, and then come back to **??**. From here you can easily follow chapter two which uses mostly Python.

1.0.2 Profile 2: The quantum theoretician

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

1.0.3 Profile 3: The Enthusiast

We recommend first a fast course on quantum mechanics. **??** then **??**. Then our basics for programming, . After this, you can comfortably go to **??**.

If you wanna explore more on Q theory references. Describing these references [1], [**?**], [**?**]
If you wanna explore more on programming references. Python course [**?**] what else here?

Abbreviations

Q^{TM} Quantum

Chapter 2: Quantum Software

QVM Quantum Virtual Machine

QPU Quantum Processing Unit

CPU (Classical) Central Processing Unit

API Application Programming Interface

QUIL Quantum Instruction Language

QDK Quantum Development Kit

QISKit Quantum Information Science Kit

QASM Quantum Assembly Language

Chapter 3: Algorithms

QFT Quantum Fourier Transform

Chapter 4: NISQ Algorithms

NISQ Noisy Intermediate-Scale Quantum

FTLSQ Fault-Tolerant Large-Scale Quantum

QUBO Quadratic Unconstrained Binary Optimisation

QAOA Quantum Approximate Optimisation Algorithm

VQE Variational Quantum Eigensolver

QSDP Quantum Semidefinite Programming

Chapter 6: Implementations

SC Superconducting

LOQC Linear Optical Quantum Computing

CV Continuous Variables

Chapter 7: Architecture

QLANG^{TM} Quantum CLANG

Chapter 2

Introduction

> The purpose of computing is insight, not numbers
>
> *Richard Hamming, Numerical Methods for Scientists and Engineers*

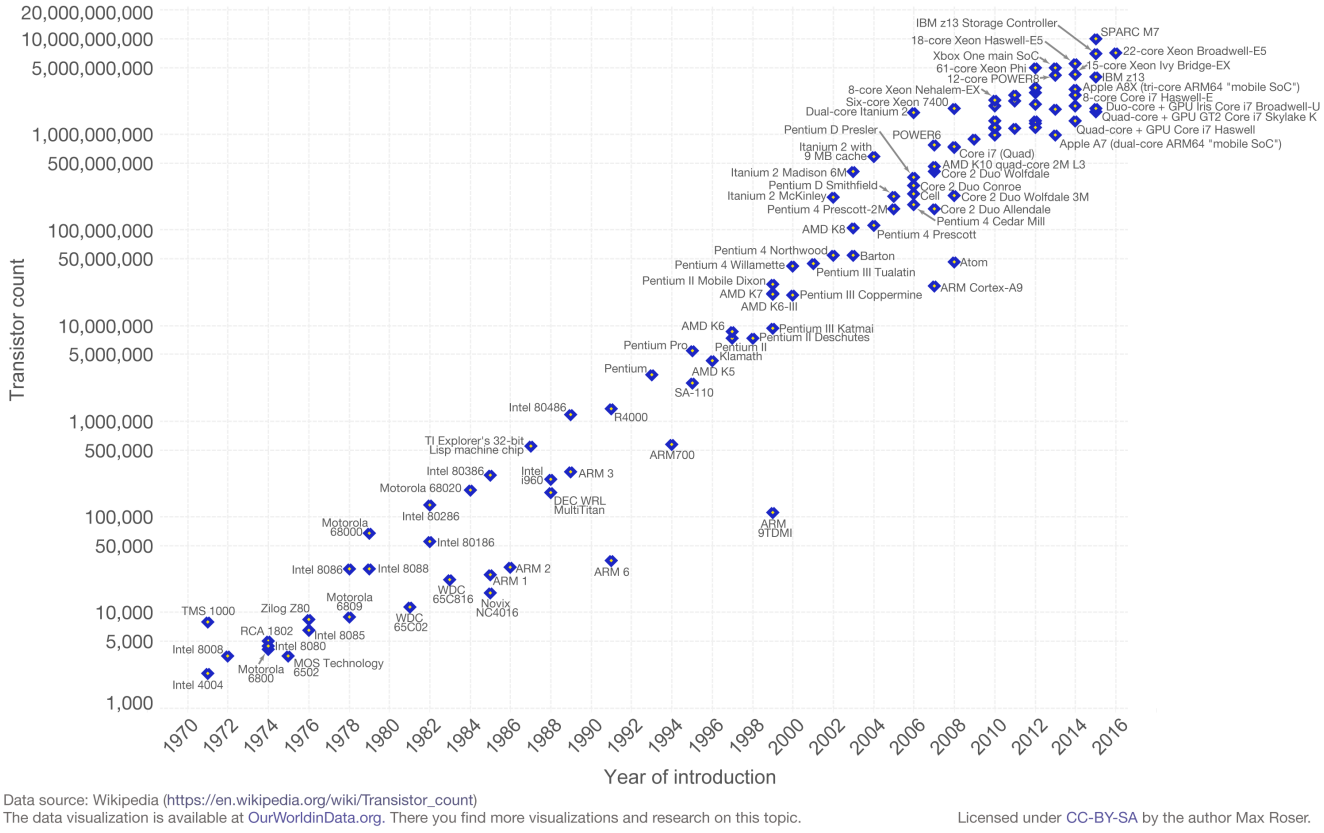
2.0.1 Motivation for this guide/ the need for quantum programmers

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 incompatible of our a human's experience of the world.

There are two main facets which differentiate quantum from classical theory. The first is the quantisation of properties of a particle such as charge, angular momentum, and energy, from which the field derives its name. 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) [2, 3], perhaps the greatest impact has been made through the manipulation of semiconductor materials. Since the invention of the first transistor in 1947 [4], semiconductor technology has laid the groundwork for scalable computing, bringing us into the the current information age.

We are reaching 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. 2.1. This was driven predominantly by the cost and power consumption per transistor going down as feature sizes decreased [5]. 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 have insufficient energy to reach.

This example of tunnelling is related to the so-called 'wave-particle duality' - this is the second



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

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 some form of 'superposition' of the two, giving many more possible values which can be processed. Currently for computationally difficult tasks supercomputers must be used, which require massive amounts of space and are costly to build and run. The current state of the art is the Summit supercomputer which is shown in figure Fig. 2.2 with 200 petaflops of computing power. With an exponential scaling in computational power, quantum computers are capable of performing tasks with fewer resources compared to classical [1]. Thus adding a single extra qubit could double the computing power. Combined with entanglement to allow our qubits to influence each other in these superposition states, we can harness a form of parallelism that results from the wave-like nature 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 [6]).

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, these combined features of superposition and entanglement give quantum computers significant potential to outperform conventional computers at certain tasks. At the point when quantum computers are able to outperform classical supercomputers at a task, the so-called 'quantum supremacy' will have been achieved [7].

These tasks range from aiding the fields of medicine, chemistry and materials with applications including creating more powerful simulations[2] [9]; providing possible speedups for AI and machine learning [10]; assisting with modelling complex logistics problems; and improving financial models [11].

However, achieving this potential does not come without significant engineering difficulties. Readout or detection of the information in the qubit (measuring it) removes the quantum aspect of the information contained within, resulting in it reverting to the classical bit values with differing probabilities. This probabilistic nature is a fundamental (and irremovable) part of quantum theory. Furthermore, the technology is still young and undeveloped. On the hardware side, D-wave has many 'quantum annealers' which can solve specific problems using quantum tunnelling, but the race to build a universal quantum computer is in an early stage. Academic institutions, large corporations (including Google [12, 13], IBM, [14] and Intel [15]) and smaller start-ups (Rigetti [16], Xanadu [17]) alike have invested heavily in hardware. There are a wide range of platforms and architectures, including but not limited to superconducting qubits [13], ion traps [18], quantum dots [19], spin qubits in silicon [20], and silicon photonics [21].

One area that remains comparatively underdeveloped is software. Algorithms exist for many of the applications above, but there may be many more as yet undiscovered. Furthermore, 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 of 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.

This area has recieved significant attention recently, with a slew of documents addressing the issue of quantum software implementations [23–28]. However, many of these documents focussed on one or two languages. In this guide, we intend to keep a broad overview of the current state of the languages out there. Furthermore, in the manner of conventional programming guides we provide 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

[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 [8].

Figure 2.2: The summit supercomputer, currently set to be the worlds fastest supercomputer, taking up the area of two tennis courts. [22]

state of the field. Initially we explain in-depth the working principles behind quantum computers and their differences to classical computers in Chapter 3, and then introduce some of the current programming languages in We examine the programming languages that will be able to interface between the algorithms and the quantum computer in Chapter 4. We go on to discuss a selection of algorithms and their applications in Chapter 5.

Since in the short-term, quantum computers are likely to be noisy, error-prone and limited in scale, we discuss how they can be used in this regime in Chapter 6. While the long term software implementations are discussed in Chapter 7. Then we go on to consider and hardware-specific implementations and hardware/software architectures in detail in Chapter 8.

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

Chapter 3

Quantum theory background

> I think I can safely say that nobody understands quantum mechanics
>
> *Richard Feynman, The Character of Physical Law*

3.1 The Qubit

In this section we will introduce quantum mechanics and its basic unit of information, the qubit. In general a qubit is the name given to well defined two level systems that are governed by the laws of quantum physics. Any system that obeys the mathematical description of a qubit given in this chapter can in principle be used to perform quantum computation. The set of physical systems that fall into this category is diverse meaning there are many competing physical platforms being developed that are capable of quantum computation. The theory of quantum mechanics is mathematically described using linear algebra and so we advice that the reader should first review basic linear algebra concepts, such as basis vectors, matrices and tensor products, before continuing with this chapter.

In classical computing 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. In quantum information theory, the fundamental unit of information is the qubit. The qubit, much like the bit, is a description of the state of a system. In the case of the bit, the system being described is a classical two level system, such as the transistor. This is also the case for the qubit, except now, the dynamics are dominated by quantum physics instead of classical physics, resulting in fundamentally different behaviour. To understand the differences in behaviour this brings about, we will first introduce the notions of quantum measurement and superposition.

Quantum mechanics is a measurement based theory in which we use our observations of a system to describe the state of that system in a mathematically precise way. In this regard, we describe the state of a qubit by measuring its value and noting down the outcome. This is the same as in the

classical case expect now, the same state can give different measurement outcomes upon repeated measurement. As a result of this, the state description must capture the probabilistic nature of the measurement outcomes if it is to fully represent the qubit. Since the qubit is a two level system, the possible outcomes are binary valued '0' or '1', but crucially, if we prepare a qubit in the same state multiple times its value might not be consistent upon measurement. A qubit that gives non-consistent measurement outcomes is said to be in a *superposition*. Formally, we describe the qubit ψ by its state $|\psi\rangle$. The notation $|...\rangle$ for the state of a system is standard in the field and was invented by one of the fore-fathers of Quantum physics, Paul Dirac. For a more in depth introduction to Dirac notation please see Appendix A or *p13* of [1].

Given a qubit, $|\psi\rangle$, that when measured always gives the outcome 0, we say that ψ is in the basis state $|0\rangle$:

$$|\psi\rangle = |0\rangle \equiv \begin{pmatrix} 1 \\ 0 \end{pmatrix} \tag{3.1}$$

This state is not in a superposition and will always give '0' upon measurement. The states $|0\rangle$ and $|1\rangle$ are known as basis state and are used in linear combinations to formally describe superposition state. This terminology comes from linear algebra in which basis vectors are used to represent arbitrary vectors in a vector space. In the same way, basis states are used to form arbitrary superposition states of a qubit. Given a second qubit $|\phi\rangle$, now in a superposition of basis states $|0\rangle$ and $|1\rangle$, we say that:

$$|\phi\rangle = \alpha|0\rangle + \beta|1\rangle \equiv \alpha\begin{pmatrix} 1 \\ 0 \end{pmatrix} + \beta\begin{pmatrix} 0 \\ 1 \end{pmatrix} \tag{3.2}$$

The parameters α and β preceding the basis state, known as amplitudes, are related to the probabilities of obtaining the corresponding values 0 or 1. More precisely, the probability of getting the state $|0\rangle$ is $|\alpha|^2$ and the probability of getting the state $|1\rangle$ is $|\beta|^2$. To continue drawing parallels with vector formalism, each basis state can be thought of as an orthonormal basis vector spanning the space of possible superpositions. Crucially, each basis state is independent of all others i.e. the inner product between $|0\rangle$ and $|1\rangle$ is 0.

In general, α and β can take complex values and so to ensure that the probabilities are real we take the absolute value squared [1] *p80*. It is also the case that the probabilities sum to 1 since we must always get '0' or '1' upon measurement. The significance of having complex valued amplitudes is discussed briefly in Appendix A. Taking the vector representation of both $|\phi\rangle$ and $|1\rangle$, then the probability of measuring $|\phi\rangle$ and obtaining the outcome 1, collapsing the superposition state $|\phi\rangle$ to the basis state $|1\rangle$, is given by the absolute value squared of their inner product.

In order to perform useful computations with qubits, we need to be able to control the state of a qubit. This is done by single qubit gates that take input states and map them to other input states. These gates are described further in (REF) but an example of a particularly important gate, known as the Hadamard gate, is given below.

A gate (or operator as they are sometimes called) can 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 equal superposition with opposite relative phase. This is a key operator is quantum computation because it maps basis states to superposition states. To represent $\hat{H}$ in matrix notation we write:

$$\hat{H} = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \tag{3.3}$$

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 states $|+\rangle$ and $|-\rangle$ are frequently used and so given their own names plus and minus. A gate can also be written in Dirac form in the following way:

$$\hat{H} = |+\rangle\langle 0| + |-\rangle\langle 1| \tag{3.4}$$

To see how this form is applied to a general state $|\psi\rangle$, see Appendix A.

So far, our system is only capable of represent two unique points of information which we have assigned '0' and '1'. In practice, we need a system that has more than two basis states, giving us the facility to represent larger amounts of information. For this, we must generalise our description of a qubit to a multi-qubit system.

3.2 Registers

In order to encode large amounts of information into a system, the system must contain an orthonormal basis states for each information point. To access a larger space of basis states we can combine several qubits into so called registers, with each register going on to performing a specific task in a computation algorithm. Before we discuss how the capacity of a system scales with register size, let's introduce how to combine to qubits to form a 2-qubit register.

We combine qubits via the tensor product of each of the qubits individual basis states which combine to form a new set of basis states. This process enables us to build the basis states of the combined 2-qubit register one at a time. Let the state subscript denote qubits 1 (blue) and 2 (red) in the register, then the basis states of the combined register are given as follows:

$$\begin{aligned} |0\rangle_1 \otimes |0\rangle_2 &\equiv |00\rangle \\ |0\rangle_1 \otimes |1\rangle_2 &\equiv |01\rangle \\ |1\rangle_1 \otimes |0\rangle_2 &\equiv |10\rangle \\ |1\rangle_1 \otimes |1\rangle_2 &\equiv |11\rangle \end{aligned} \tag{3.5}$$

We have conveniently chosen to labelled the four basis states with quantum numbers that give us information about the component basis states that form them. We can now 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 of the 2-qubit register can now take the form:

$$\left|\psi\right\rangle = a_{00}\left|00\right\rangle + a_{01}\left|01\right\rangle + a_{10}\left|10\right\rangle + a_{11}\left|11\right\rangle = \sum_{i,j=0}^{1} a_{ij}\left|ij\right\rangle \tag{3.6}$$

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

$$\begin{aligned}\left|\psi\right\rangle_1 \otimes \left|\psi\right\rangle_2 &= \left(\frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)\right)_1 \otimes \left(\frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)\right)_2 \\ &= \frac{1}{2}(|00\rangle - |01\rangle + |10\rangle - |11\rangle)\end{aligned} \tag{3.7}$$

From Equation 3.7 above, we can read out the probabilities of finding qubits 1 and 2 in each basis state. For example the probability of measuring the register and finding qubit 1 in the state '0' and qubit 2 in the state '1' is the absolute value squared of the amplitude preceding the basis state $|01\rangle$, in this case $\left|\frac{1}{2}\right|^2 = \frac{1}{4}$. In fact, the probability distribution across each of the basis states is constant and equal to $\frac{1}{4}$ because the 2-qubit register is in an equal superposition.

The tensor product generalises to any size register and the number of basis states scale by $O(2^n)$ where n is the number of qubits within the register. The power of quantum computation is a direct result of this scaling, combined with the fact that we can prepare a system to be in an equal superposition of all basis states at the same time. For example, if we were to assign each basis state to a specific entry in a large database, then we could prepare a large quantum system to be in a superposition of all the basis states and then perform computations on all data entries simultaneously.

3.3 Quantum Computation

To perform arbitrary operation on multi-qubit systems, it's possible to show that all one requires is single qubit control and two qubit gates that can be strung together into a sequence of operations. Therefore, a computation is mapped onto an equivalent string of single and two qubit gates performed on a register of qubits. In general, quantum algorithms can be broken down into the following key steps:

1. Determine the number of qubits required to represent the domain of your computation and assign each basis state to an information point.

2. Input the all '0' basis state in which all qubits are in the state '0' and create an equal superposition across all basis states.

3. Convert the computation into a string of gates that typically allows the evaluation of your computation on all basis states simultaneously or in a resource efficient manor i.e. with relatively few gates.

4. The result of each basis state computation is then encoded into the amplitudes of the corresponding basis state in such a way that suppresses the amplitude of all wrong answers and enhances the amplitude of all the correct answers.

5. The overall register is then measured giving the correct answer with high probability, due to its enhanced amplitude.

The following is an example of how to apply multiple gates to a system. Say we have a system of 2 qubits and we want to act with both $\hat{H}$ on qubit 2 and $\hat{H}$ on qubit 2. The total operator we act on the 2-qubit register will be given by: $\hat{H} \otimes \hat{H}$ where the Hadamard acts on each qubit. This operation will map the input state $|00\rangle$ to an equal superposition across the registers basis state (as is requires in step 2 above). The corresponding matrix for this operation is given by the tensor product of each single gate matrix:

$$\hat{H} \otimes \hat{H} = \begin{matrix} & |0\rangle_1 & |1\rangle_1 \end{matrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \otimes \begin{matrix} & |0\rangle_2 & |1\rangle_2 \end{matrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} = \begin{matrix} & |00\rangle & |01\rangle & |10\rangle & |11\rangle \end{matrix} \frac{1}{\sqrt{2}} \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \tag{3.8}$$

From the above we can see how each of the 2-qubit basis states will map under the operation $|H\rangle \otimes |H\rangle$ by looking at the corresponding column. For example, $(\hat{H} \otimes \hat{H})|00\rangle = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)$. This is exactly the state needed in many quantum algorithms and serves as a starting point for computation. The operation described above is the implement ion of two single qubit gates, one on each qubit within the register. In order to perform computation, 2-qubit gates is required whose operation on one qubit depends upon the value of the other. A key example of this is the CNOT gate which performs a controlled-NOT operation on the second qubit, controlled by the value of the first. The matrix representaion of the CNOT is as follows:

$$\hat{CNOT} = \begin{matrix} & |00\rangle & |01\rangle & |10\rangle & |11\rangle \end{matrix} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \tag{3.9}$$

Let's act the CNOT gate on our equal superposition state to see how each basis state in the equal superposition is mapped:

$$\hat{CNOT}\left(\frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle)\right) = \frac{1}{2}(|00\rangle + |01\rangle + |11\rangle + |10\rangle) \tag{3.10}$$

We can see that the third basis state, $|10\rangle$, is mapped to $|11\rangle$ and conversely the fourth term, $|11\rangle$, is mapped to $|10\rangle$. Since the value of the first qubit is '1' in both these cases, a NOT is performed on the value of the second qubit. Coincidentally, this results in the state remaining the same, however,

this would not have been the case if the amplitudes of each basis state had not been the equal. The CNOT is necessarily a reversible gate, as is required by quantum mechanics (see Appendix A). A list of important gates, along with a description of the gate circuit model, is given in the following section.

3.4 Quantum gates and Measurement

introduce in text and table Cnot coherent classical XOR non separable can't write as a tensor product. Clifford + T gateset

This section briefly reviews the gate model for circuit based quantum computing and discusses the similarities 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 [15] IBM [14], Google [12] and Rigetti [29]* 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. 3.1.

(a) Digital operation

(b) Quantum operation

Figure 3.1: 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.

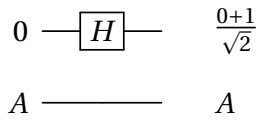


Figure 3.2: Hadamard gate acting on the top qubit and no gate performed on the bottom qubit

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.

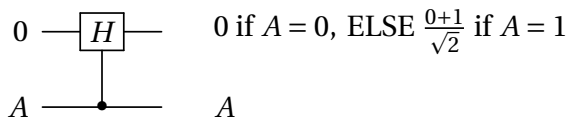


Figure 3.3: Hadamard gate acting on the top qubit and no gate performed on the bottom qubit

3.5 Gate cheat sheet

Gate	Description	Matrix representation	Dirac notation
$\boxed{X}$	Bit flip	$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$	$\|0\rangle\langle 1\| + \|1\rangle\langle 0\|$
$\boxed{Z}$	Phase flip	$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}$	$\|0\rangle\langle 0\| - \|1\rangle\langle 1\|$
$\boxed{T}$	Phase gate	$\begin{bmatrix} 1 & 0 \\ 0 & \exp\{i\pi/8\} \end{bmatrix}$	$\|0\rangle\langle 0\| + \exp\{i\pi/8\} * \|1\rangle\langle 1\|$
$\boxed{H}$	Superposition	$\frac{1}{\sqrt{2}}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$	$\|0\rangle\langle 0\| + \|0\rangle\langle 1\| + \|1\rangle\langle 0\| - \|1\rangle\langle 1\|$

Table 3.1: Single qubit gates

Gate	Description	Matrix representation	Dirac notation
$\boxed{R_X(\theta)}$	Rotation around the X axis	$\begin{bmatrix} cos(\theta/2) & -isin(\theta/2) \\ -isin(\theta/2) & cos(\theta/2) \end{bmatrix}$	No.
$\boxed{R_Z(\theta)}$	Rotation around the Z axis	$\begin{bmatrix} \exp\{i\theta/2\} & 0 \\ 0 & \exp\{-i\theta/2\} \end{bmatrix}$	$\exp\{i\theta/2\}\|0\rangle\langle 0\| + \exp\{-i\theta/2\}\|1\rangle\langle 1\|$

Table 3.2: Single qubit arbitrary rotation gates

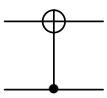
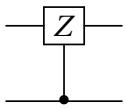
Gate	Description	Matrix representation	Dirac notation
$\boxed{CNOT}$ or 	Controlled Bit flip	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}$	$\|00\rangle\langle 00\| + \|01\rangle\langle 01\| + \|01\rangle\langle 10\| + \|10\rangle\langle 11$
$\boxed{CPHASE}$ or  $\|10\rangle\langle 10\| - \|11\rangle\langle 11\|$	Controlled Phase flip	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}$	$\|00\rangle\langle 00\| + \|01\rangle\langle 01\| +$
$\boxed{SWAP}$ or 	Swap qubits	$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$	$\|00\rangle\langle 00\| + \|01\rangle\langle 10\| + \|10\rangle\langle 01\| + \|11\rangle\langle 11$

Table 3.3: Two qubit (control) gates

Chapter 4

Introduction to Quantum Software

Now that we have introduced some quantum mechanics and how it can be connected to the idea of a computation using circuits and gates, we can start looking at how to write quantum programs. There are already various software packages available which are specifically designed to make interaction with a quantum computer easier, even though the quantum computers themselves are only just starting to get developed. Because of this, most software also includes simulators, which can either run on your computer at home or use more powerful computers via cloud computing. Next to that, some of the software can used with the quantum chips that have been developed by IBM, Google, Rigetti and D-Wave. However, the simulators are usually free, whereas access to these chips is often not. Furthermore, with the exception of D-Wave, which is a very specialised quantum computer, the chips are still very primitive and the simulators are just as suitable for you to develop a good understanding of quantum programming principles.

In this guide six different software packages will be used: pyQuil, QISKit, ProjectQ, Q#, D-Wave Ocean and 1QBit. The first four are designed to be used for universal quantum computing, whereas the last two are more focused on optimisation problems.

In this chapter we will introduce these software packages, explaining their basic syntax and giving a minimal example quantum program: Create a qubit, put it in a superposition and then measure it, see example 1. Another example is used for D-Wave Ocean and 1QBit, as their general purpose is different. A brief overview of quantum software that is not more extensively considered in this guide is given is section 4.8. At the end, we will include a short comparison between the software packages that have been introduced in this chapter.

In general quantum programs have three main steps:

1. Allocate qubits
2. Apply gates to single or multiple qubits
3. Measure the final state of the qubits

This structure is not 100% accurate, as one may allocate additional qubits for certain steps of an algorithm, or measure part of the qubits before applying more gates to the leftover qubits before measuring them. However, one must always allocate some qubits to get started and the final step

should always be a measurement, as the human-computer interaction will always be classical and we therefore need to get classical information from the qubits to be able to use to result of a quantum computation in any way.

Another thing one has to keep in mind is that quantum computers are very good at some specific tasks, but there are also many tasks where classical computers are just as good or even better. Classical computers have been around for a very long time and benefit from many optimisations that have been developed in this time. Therefore, when programming we should try and use classical and quantum computers together, each for the tasks they are best at, respectively. Some software packages do this explicitly, where different parts of your program apply to the classical CPU and the quantum processor (QPU). Other packages sort this out themselves and let the user just use qubits in the main program.

4.0.1 Simple example

Example 1: Generating and measuring a single-qubit state

Generate the one-qubit pure state

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

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

4.1 Rigetti - pyQuil

pyQuil is a Python based programming language created by Rigetti [30] as part of their quantum programming toolkit for their hardware. In this section we address the main features of pyQuil and the description of its syntax with a simple example.

4.1.1 Quantum Programs with pyQuil

We will now do a simple demonstration in pyQuil to create and measure the superposition $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$. We will need to initialise a qubit in the $|0\rangle$ state, then apply a Hadamard gate and measure the result.

```
# 1. Calling Libraries
from pyquil.quil  import Program
from pyquil.api   import QVMConnection
from pyquil.gates import H
# 2. Initialising the program
qvm = QVMConnection()
p = Program()
# 3. Applying gates
p.inst(H(0))
# 4. Performing measurements
p.measure(0,0)
# 5. Executing the program
results = qvm.run(p, [], 4)
print(results)
```

1. **Calling libraries** - We call an object Progam which will contain the instructions of our program, a connection to the QVM and the single-qubit gate Hadamard H.

2. **Initialising the program** - When allocating a program (p in this case) we implicitly have allocated n qubits in the state $|00...0>$. The amount of available qubits will depend on whether it's being run on a QVM or a QPU. The object program will contain the list of instructions to be run later on.

3. **Applying gates** - Considering the method isnt we apply from 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 Hadamards for two qubits the notation is this. The complete set of gates can be found in HERE. and u can define your own gates in the following way:

4. **Performing measurements** - We add the instruction measuring qubit zero and assigning it to classical register 0 and similarly for qubit 1.

5. **Executing the program** - The object program p is a list of instructions which are now being run. We print the results.

4.2 IBM - QISKit

4.2.1 Quantum Programs with QISKit

Our first example using QISKit is to generate a single qubit superposition, $\left|\psi\right\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$.

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. 4.1.

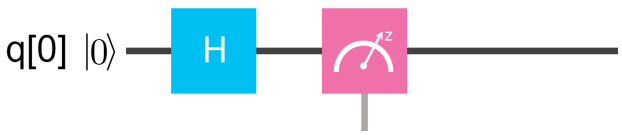


Figure 4.1: The composer view of the code that we can use to simulate the generation of $H|0\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$.

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.

To create this state, and measure it on IBM's Q QASM Simulator,

```
# 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(1)
c = ClassicalRegister(1)
qc = QuantumCircuit(q, c)

# Preform a Hadamard on the qubit in the quantum register to create a superposition
qc.h(q[0])
# 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 measurement basis and corresponding counts
print("simulation: ", sim_result)
print(sim_result.get_counts(qc))
```

When run, the output produced is as follows:

```
Local backends:  ['local_qasm_simulator', 'local_statevector_simulator',
 ↪  'local_unitary_simulator']
simulation:  COMPLETED
{'0': 528, '1': 496}
```

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 **??** has been generated, and measured.

4.3 ProjectQ

ProjectQ is an open source domain specific language embedded in Python. 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 and QISKit.

4.3.1 Quantum Programs with ProjectQ

We will now do a simple demonstration in ProjectQ to create and measure the superposition $\frac{|0\rangle+|1\rangle}{\sqrt{2}}$. We will need to initialise a qubit in the $|0\rangle$ state, then apply a Hadamard gate and measure the result.

```
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))
```

Walking through this example step by step,

- **Initialising the program** - 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 (your computer) as default, this can also be set up to be the IBM back-end or something else such as a circuit drawer.
- **Allocating qubits** - Qubits can now be allocated, either individually (as shown above) or as a register, from the `engine`. Creating a single qubit is done using `allocate_qubit` and a register by `allocate_qubit(n)`.
- **Applying gates** - The syntax to do an operation is `Gate | Qubits`.
- **Performing measurements** - Measurements have the same syntax as applying gates but using the command `Measure`. It is really important to measure all your qubits at the end of the program to avoid errors in your code.

- **Executing the program** - The gates are sent to the back-end by use of the `flush` command. If you wish to display the result, the qubit can be converted to type integer after measurement.

4.4 Microsoft - 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 a 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 computers having this framework installed. This is no longer restricted to Windows computers, but a version of the .NET framework has recently become available for Mac and Linux computers as well. The easiest way to get started is probably by programming with Visual Studio, as it has inbuilt .NET support. 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 4.1. 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 chang-

ing 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 4.1: Primitive types in Q#

4.4.1 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. A namespace is way to define a scope that can contain different elements in a program. The namespace can span multiple files and, as in our case, can be used by different languages.

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.

```
using Microsoft.Quantum.Simulation.Core;
using Microsoft.Quantum.Simulation.Simulators;

namespace Quantum.Superposition{

    class Driver{
        static void Main(string[] args){

            using (var sim = new QuantumSimulator()) {
                Result outcome = Superposition.Run(sim).Result;
                System.Console.WriteLine
                        ($"Measuring equal superposition state gave outcome {outcome}.");
            }

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

Listing 4.1: 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 register
            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]);
              }
            }

      return res;

      }
    }
}
```

Listing 4.2: A simple program in Q creating and measuring an equal superposition.

4.5 D-Wave-Ocean

As the universal quantum computer is hard to achieve in the short term requiring high quality and scalable qubits, the technologies presented here make use of existing qubits as an intermediate step towards the goal of universal quantum computing. One of these technologies is quantum annealing.

Different from universal quantum computer which can implement algorithms by gate, annealing type quantum computer is ansatz based computer designed for specific optimisation problem. Annealing is a process that searches for the minimum energy of a state. Classically this is done by the thermal annealing (simulated annealing) changing the configuration slowly and accepting the configuration when lower energy state is achieved. Quantum annealing is expected to be implemented more efficiently with the assistance of entanglement and tunneling as showed in Fig. 4.2.

D-wave has its own software stack:

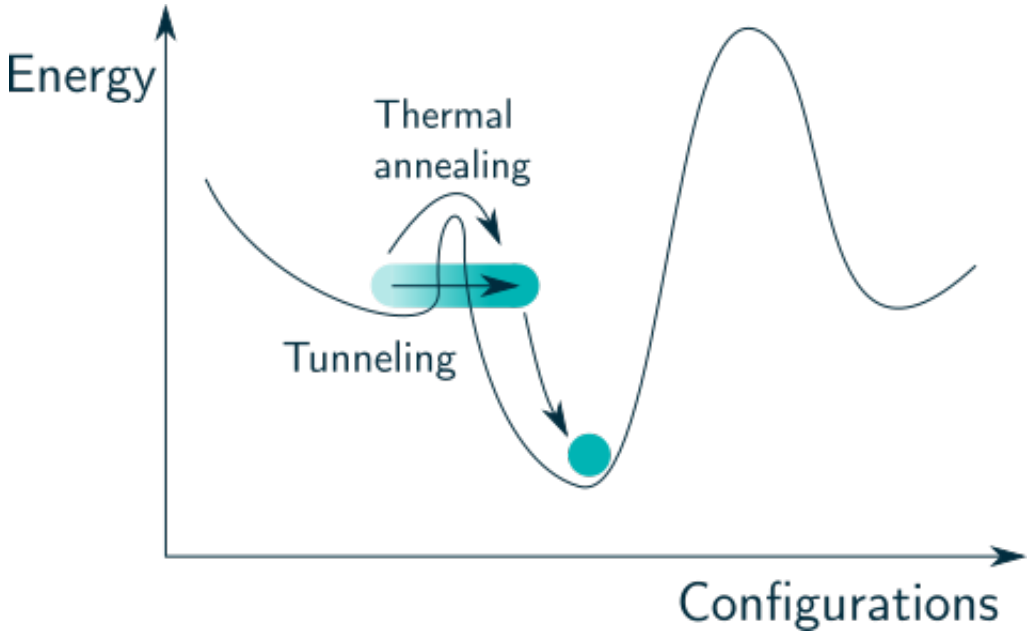


Figure 4.2: Caption

- **D-Wave Ocean**- D-Wave toolkits to help solve problems on D-Wave system.
- **qbsolv**- Open source hybrid optimization solver for partitioning quadratic unconstrained binary optimization (QUBO) problems which is ran either on tabu classical solver or D-Wave system. By default, the classical solver is used. To use D-Wave's machine, separate arrangements should be included.
- **QPU**- Quantum processing unit. An API key is needed to run the program on the quantum computer.

Here as an example we will consider getting the configuration to minimise the quadratic polynomial $-x_0 - x_1 + 2x_0x_1$ for $x_i \in \{0,1\}$

```
import dwave_qbsolv

# Create a quadratic polynomial -x_0 - x_1 + 2x_0x_1
Q = {(0, 0): -1, (1, 1): -1, (0, 1): 2}

# Create the local solver
solver = dwave_qbsolv.QBSolv()

# Solve for the minimum energy configuration
response = solver.sample_qubo(Q, num_reads=300) # Sample 300 times
sol = list(response.samples())
energy = response.data_vectors['energy']
print(sol)
print(energy)
```

The output gives

```
[{0: 1, 1: 0}, {0: 0, 1: 1}]
[-1. -1.]
```

It can be seen that x_0 and x_1 will always take different value. This is equivalent to the NOT logic. Generally, the Boolean gate can be implemented as the QUBO form.

4.6 1QBIT-QUANTUM-READY SDK

Alternatively, 1Qbit has developed the 1QBIT QUANTUM-READY™ software development kit targeting hardware independent languages. Currently it uses a local classical solver or interface with D-Wave's system via an API key. Both the 1QBIT SDK and D-Wave Ocean stack can be run with Python. We will introduce the 1QBIT QUANTUM-READY™ SDK in this section, as an example of the language used to interface with D-Wave's machine. Currently, only online Jupyter Notebook versions of the language are available *(see http://qdk.1qbit.com/)*. By default a classical solver is used. To use D-Wave's machine, separate arrangement should be made.

```
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)
qubo = builder.build_polynomial()

# Create a local solver
solver = DWaveSolver()

# Get configuration of best solution
solver.solver.num_reads = 300 # Sample 300 times
sol = solver.minimize(qubo)
print(sol)
```

The output is

```
Energy = -1, Frequency: 152, Configuration = {x0 = 0, x1 = 1}
Energy = -1, Frequency: 148, Configuration = {x0 = 1, x1 = 0}
```

4.7 Comparison of Languages

We now compare some of the implementation dependent features of the languages and give a comparison between them.

pyquil qubits are denoted by integers which can be unclear sometimes.

qiskit

General syntax comments

PyQuil has nice syntax handling of gates `program.inst(gate_1(qubit_1, gate_2(qubit_2)`

Qiskit syntax is ok. the documentation is rather difficult to find in some of the examples we mention here [1][2]

Project Q has a heavy focus on making the programs written using it to look (syntactically) like operators acting on states. The emphasis on Dirac notation is clear as the operation perform an Pauli X on qubit 1 is given by `X | q[1]`.

Qubit management

The pyQuil qubit management is dynamic, qubits are allocated dynamically which does not require the users input but do have to allocate classical registers.

The qiskit library is heavily based on using quantum circuits. You specify quantum and classical registers and ancilla qubits.

Project Q also requires the user to manually allocate qubits and the QASM resembles dynamically allocated qubits with allocate and deallocate at the start and end of the instruction set.

Q# using to allocate qubits but don't have to declare classical variable but c# is typed.

Compilers/Gate simplification

All of the libraries have access to the standard gate set in their python environments[3].

In pyQuil the user can also use the `defgate()` method to define their own gate either in terms of compositions of existing gates or by specifying a matrix representation of the gate. Qiskit also can define gates. Project Q has the option to define gates?

The pyQuil compiler takes one of Rigetti's quantum devices as an argument and compiles their Quill QASM into device specific gate-set and remaps the qubits to match the connectivity of the machine.

IBM has no working compiler[4] but we can return the Open QASM (IBM's Quantum Assembly format),

The Project Q compiler is designed to have a modular structure, so the user can pick and choose the level of optimisation required. It works quite well although you have to manually specify the

[1] Can we say this?

[2] The documentation is impossible, there are random links everywhere to depreciated methods...

[3] X,Y,Z,H,Phase gates and arbitrary rotations

[4] there is a compiler but it does nothing, specify x only and the program crashes, specify x and H and the program uses Z. U(x,y,z)(θ).

available gate-set you have access to and the connectivity of the machine. They are working on adding in backend support for the IBM quantum devices.

Backends

Rigetti then offers a remote Quantum Virtual Machine (QVM) which runs simulations of up to 26 qubits requiring an API key to use. They do not provide a local simulator which runs on a personal computer. IBM do provide a local simulator and access to their devices and remote simulator require and API key. Project Q comes with a local simulator, written in c++ and is built on your machine on install using python wrappers.

The project Q simulator is best of the local simulators however currently there is no way to compile project Q code into IBM's OASM to run on IBM's machines or compile to Quil to run on Rigetti's machines. Although IBM and Project Q existed as part of the same project [5] we are unsure of when the project Q to IBM backend will be implemented.

One slightly annoying feature of pyQuil the software package is the lack of local quantum simulator. This can cause problems when trying to execute large programs as we often found the connection timed out before the program finished.

Device mappings

Rigetti's Quil compiler takes a dictionary of one of their devices which then creates a map of the positions of the qubits and the connectivity of the machine.

QISKits' `get remote backends` method doesn't return any valid devices to compile to, when using one of their device names an error saying the device is not valid is produced. It may be possible to manually add device mappings but even then the *compiler* didn't compile to the given gate-set or it would produce an error for some gate-sets.

Project Q lets you specify the connectivity using a Python dictionary and gate-set by importing the gates you want/have access to. The compiler works quite well.

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? Table 4.2

- **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.
- **how beginner-friendly is the syntax?**
- **particular conveniences** easy to implement oracles? qft?!!!!! what other conveniences?
- what other general characteristics should we highlight?

[5] see git change-log

4.8 Further languages

There are a few languages and python libraries that haven't been mentioned in this guide due to differences in the computation model they are based on and their usefulness for immediate quantum programming. Some companies that aim to deliver full stacks of quantum software resources have not released any documentation, yet. Examples of these kinds of languages, libraries and resources are included below.

- **Scaffold** is a language that resembles the C and C++ programming languages and aims to provide structure in order to use classical variables and quantum registers in the same code, hence its name. Potential issues with memory allocation are reduced within the language, while allowing users to compose their relevant variables together. Several design decisions are included in the documentation to make programming easier. These include an imperative programming model (rather than a functional one) and rigorous error checks. Further reading can be found in the Scaffold publication [31].

- **Cirq** is an open-source project from Google that has recently been developed. The aim is to make quantum algorithms for existing quantum computers easier to code. Based on a python library, it allows simple tasks to be run on a simulated quantum processor, that will be available soon. Future access to Google's recently announced Bristlecone processor [32] will form part of this development package [33]. Existing documentation and tutorials can be found at [34].

- **Quipper** is a functional programming language for quantum computation that uses a high-level structure for gate organisation and programmable transformations of quantum circuits. As well as including all well-known quantum algorithms (like the ones we have mentioned), its distribution also includes seven operations from literature that are uncommon in other quantum software distributions. A full list of documentation and updates can be found on the Quipper website [35].

- **Strawberry Fields**, a full-stack continuous variable quantum software platform using Python. It relies on a completely different model of quantum computing where continuous variables replace discrete variables. The strawberry fields Python library allows access to the various elements of the stack. The quantum programming language named Blackbird is used to write quantum circuits inside of strawberry fields library. Blackbird uses similar syntax to Project Q but adapted to a continuous variable implementation. An example using both of these found in the quantum teleportation tutorial on the Xanadu website (see [36]). The strawberry fields stack also includes an interactive model for its quantum algorithm construction. More detail on these aspects can be found in the Xanadu whitepaper [24].

As mentioned before, a GitHub page has been dedicated to this topic that includes links to the relevant pages associated with quantum software: [37].

	pyQuil	QISKit	ProjectQ	Q#	Cirq	Ocean	1QBIT SDK	Scaffold	Straw
Company	Rigetti	IBM	-	Microsoft	Google	Dwave	1qbit	-	X
Platform	SC	SC	Agnostic	Agnostic	SC	Annealing	Annealing	Agnostic	
Type	Lib	Lib	Lib	Lan	Lib	Lib	Lib	Lan	
QVM Local (# qubits)		5	28	27					
QVM Cloud (# qubits)	26			>40					
QPU Access?	Yes*	Yes*	Yes*	No	Future	Yes*	Y$^{s:}$	e	
QPU (# qubits)	num	num	num	-	num	num	num		
What else?	-	-	-	-	-	-	-		

Table 4.2: The mother of all tables. Superconducting qubits (SC), Continuous Variables (CV). Library (Lib). Language (Lan)

4.9 Exercises

Implement the following examples in the language of your choice.

Exercise 1: Creating a two-qubit pure state (Bell state)

a) An important state in quantum information is this Bell state

$$\left|\psi\right\rangle = \frac{1}{\sqrt{2}}\left(|00\rangle + |11\rangle\right),$$

which can be constructed for two qubits by applying a Hadamard to the first qubit followed by a CNOT controlled on the first qubit. Construct this state in a language of your choosing. Measure both qubits, what correlations do you observe?

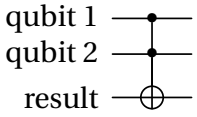
b) There are three other Bell states

$$\frac{1}{\sqrt{2}}\left(|00\rangle - |11\rangle\right), \frac{1}{\sqrt{2}}\left(|01\rangle + |10\rangle\right) \text{ and } \frac{1}{\sqrt{2}}\left(|01\rangle - |10\rangle\right).$$

Can you construct these using a similar method, using extra X or Z gates?

Exercise 2: Boolean AND

We can construct a quantum gate that performs a Boolean AND operation. Classically this operation takes two bits and returns one bit. As a quantum gate it needs to be reversible which requires keeping the input states, therefore performing an AND gate between two qubits requires three qubits. The quantum circuit for the AND gate is:



The result of the Boolean AND between qubit 1 and 2 is output in the third qubit after doing a doubly controlled NOT.
Construct this operation in a language of your choice.

Exercise 3: Minimize a polynomial

Get the configuration to minimize the quadratic polynomial $3x_2 + x_0x_1 - 2x_0x_2 - 2x_1x_2$ for $x_i \in \{0,1\}$ using the preferred library

Chapter 5

Quantum Algorithms and Applications

A quantum computation is like a symphony—many lines of tones interfering with one another

Seth Lloyd, Programming the Universe: A Quantum Computer Scientist Takes on the Cosmos

5.1 Introduction

In this chapter we review some of the well-known quantum algorithms that are thought to provide a speed-up over classical algorithms. This chapter should help to give the reader a sense of what future full scale quantum computer will be used for before continuing with more current short term applications in Chapter 6. In 1992, David Deutsch and Richard Jozsa proposed a quantum algorithm that is exponentially faster than any known deterministic classical algorithm [38]. Although the algorithm doesn't solve any problem of practical relevance, it was one of the first algorithms that demonstrated an exponential speed-up over classical algorithms. Soon after that, in 1994, Peter Shor proposed his famous factoring algorithm that sparked a huge interest in quantum computing and quantum cryptography [39]. A few years later in 1996, Lov Grover came up with an algorithm for the unstructured search problem which provides a quadratic speed-up over it's classical counterpart [40]. Even though the speed-up for Grover's algorithm isn't exponential, it can be applied to a wide variety of problems including NP-complete problems. Since these algorithms were first proposed, a lot of research has been aimed at implementing them across a wide variety of different platforms. Even though quantum algorithms provide a speed-up over classical algorithms, it is worth noting that quantum algorithms are probabilistic in nature and may not always provide the correct answer. It is therefore necessary to evaluate the speed-up along with the probability of success when comparing quantum algorithms to their classical counterparts.

Most quantum algorithms exploit a unitary operation called the 'oracle' operation. It is important to first understand the role the oracle operation plays in each algorithm and it therefore leads our

discussion. We then go on to explain Deutsch-Jozsa's algorithm in section 5.3, included because of its historical importance as the first quantum algorithm. This is then followed by Grover's algorithm in section 5.4, presented as a useful example for developing some intuition about the probabilistic nature of quantum algorithms. It will also provide some insight into the use of superposition as a tool for quantum parallelism and amplitude amplification as a method for increasing the success probability of algorithms. In section 5.5 we give a brief introduction to Shor's algorithm, followed by quantum annealing in section 5.6 and the variational eigensolver algorithm in section 5.7. The explanation of each algorithm is accompanied by examples to help make them more comprehensible and intuitive to understand. Finally, the algorithms, their resource requirements and success probabilities have been summarised in a table at the end of this chapter in Table 5.8.

5.2 The Oracle Operation

The oracle operation is used in each algorithm as a way of encoding the result of each computation into the quantum state which we can then extract through measurement. It should be thought of as a call to a function that we wish to learn about. It takes a state labelled x and encodes the answer $f(x)$ into the resulting state. For example, suppose there is a function $f(x)$ on a one-bit domain range x such that $f : \{0,1\}^n \mapsto \{0,1\}$. We can create a state $|x,0\rangle$ consisting of two registers, one for the domain x and the other which we will use to store the answer $f(x)$ initially set to 0. The oracle function would then encode the result of each bit string from the domain register into the range register. The operation transforms the state as follows:

$$|x,0\rangle \xrightarrow{O_f} \big|x, 0 \oplus f(x)\big\rangle$$

where $\bigoplus$ indicates addition modulo 2. This is important since it will not always be the case that our range register is initialised to 0 and in that case would be encoded via modulo 2 addition with $f(x)$, as seen in the next example. It is well known that this operation is unitary and can be simulated by an appropriate sequence of gates. This operation is usually referred to as the 'bit oracle'. More importantly, if this operation is applied to the state $|x,-\rangle = |x\rangle\,(|0\rangle - |1\rangle)/\sqrt{2}$, where the range resister is now initialised in the minus state $|-\rangle$ (see Eq. 3.7) instead of the state $|0\rangle$, then the output state is given by:

$$\begin{aligned} U_f\Big(\frac{1}{\sqrt{2}}|x\rangle\left(|0\rangle - |1\rangle\right)\Big) &= \frac{1}{\sqrt{2}}|x\rangle\left(\big|0\oplus f(x)\big\rangle - \big|1\oplus f(x)\big\rangle\right) \\ &= (-1)^{f(x)}\frac{1}{\sqrt{2}}|x\rangle\left(|0\rangle - |1\rangle\right) \end{aligned}$$

This operation is a special case of the bit oracle and is generally known as the 'phase oracle', denoted U_f instead of O_f. It is an important part of many quantum algorithms as it allows the result of a computation to be encoded into the phase of a state which directly links the probability of each measurement outcome x to its value $f(x)$.

5.3 Deutsch-Jozsa algorithm

The Deutsch-Jozsa algorithm is one of the first algorithms to demonstrate "quantum parallelism". In very simple terms, quantum parallelism is a feature of quantum computing which allows it to evaluate a function $f(x)$ simultaneously for many different values of x via the use of superposition. 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 Deutsch-Jozsa algorithm determines with certainty whether $f(x)$ is constant or balanced. 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 the 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 the implementation of the oracle which is not shown in the box below.

Deutsch-Jozsa Algorithm

1. Start with the state $|0\rangle\,|0\rangle\,|1\rangle$.
2. Apply Hadamard gate to all the qubits which leads to the state:

$$|\Psi\rangle = \frac{1}{2\sqrt{2}}\big(|0\rangle + |1\rangle\big)\big(|0\rangle + |1\rangle\big)\big(|0\rangle - |1\rangle\big).$$

3. Apply the oracle 'U_f' which transforms the state $|\Psi\rangle$:

$$U_f\,|\Psi\rangle = \frac{1}{2\sqrt{2}}\big(|0\rangle - |1\rangle\big)\big[(-1)^{f(00)}\,|00\rangle + (-1)^{f(01)}\,|01\rangle + (-1)^{f(10)}\,|10\rangle + (-1)^{f(11)}\,|11\rangle\big]$$

4. Apply the Hadamard to the first two qubits. 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 steps above can be extended to the n-qubit case and the circuit diagram for Deutsch-Jozsa algorithm for a general n-qubit case is given in Fig. 5.1. Crucially the algorithm calls $f(x)$ once and evaluates it across the entire domain of f at the same time. This is the power of superposition leading to inbuilt quantum computing parallelism.

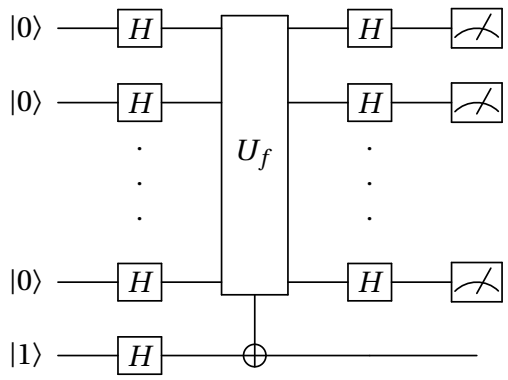


Figure 5.1: Circuit implementation of Deutsch-Jozsa algorithm for n-qubit input.

Deutsch-Jozsa Algorithm (n-qubit case)

1. Start with the state $|0\rangle^{\otimes n}|1\rangle$, where $|1\rangle$ is the ancilla qubit required for the oracle.
2. Apply the Hadamard gate to all the qubits in both registers.
3. Apply the oracle 'U_f' to the n dimensional working register.
4. Apply the Hadamard to the n dimensional working register and measure them.
5. If all the qubits are in the state $|0\rangle$, then the function $f(x)$ is constant, otherwise it is balanced.

5.4 Grover's algorithm

One of the earliest algorithms that was designed to use quantum resources is described in a 1996 paper by Lov Grover [40]. 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), suppose we have the following truth table with the third element as marked.

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

(5.1)

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 call $f(x)$ at most 3 times, since

it could always end up being the last element applied to $f(x)$. The number of attempts scales as expected, with order N, $O(N)$. Specifically, it takes at most N attempts to find the marked element out off N possible elements.

However, using the principle of superposition, we can explore the whole space of elements simultaneously. To do this we need two gates. The first one flips the phase of the marked element leaving all other the same. For the marked element being '10' as above, we have

$$U_f = \begin{array}{c} \begin{matrix} |00\rangle & |01\rangle & |10\rangle & |11\rangle \end{matrix} \\ \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{array} \tag{5.2}$$

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

$$D = \frac{1}{2}\begin{pmatrix} -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \end{pmatrix} \tag{5.3}$$

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 $|00\rangle \rightarrow |00\rangle + |01\rangle + |10\rangle + |11\rangle$ (we have ignored normalisation for simplicity). This can be represented by the matrix transformation:

$$H \otimes H\,|00\rangle = \frac{1}{2}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) \tag{5.4}$$

In the next step of the algorithm, we apply the phase oracle U_f (see section 5.2). This picks out the marked element, giving it a minus sign. This U_f gate is in fact exactly the phase oracle previously described:

$$U_f\,|x\rangle = (-1)^{f(x)}\,|x\rangle\,, f(x) = \begin{cases} 1 & \text{if x is marked} \\ 0 & \text{if x is unmarked} \end{cases} \tag{5.5}$$

Applying this gate gives:

$$U_f \frac{1}{2}(|00\rangle + |01\rangle + |10\rangle + |11\rangle) = \frac{1}{2}\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 1 \\ 1 \\ -1 \\ 1 \end{pmatrix} = \frac{1}{2}(|00\rangle + |01\rangle - |10\rangle + |11\rangle) \tag{5.6}$$

The final step is to apply D. The application of gate D can be understood as an operation which increases the amplitude of the phase shifted element in the previous step, suppressing the amplitude of the others. This can be thought of as a constructive interference on the marked element and destructive interference on all of the other elements.

$$\frac{1}{2}D(|00\rangle + |01\rangle - |10\rangle + |11\rangle) = \frac{1}{2}.\frac{1}{2}\begin{pmatrix} -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \end{pmatrix}\begin{pmatrix} 1 \\ 1 \\ -1 \\ 1 \end{pmatrix} = \frac{1}{4}\begin{pmatrix} 0 \\ 0 \\ 4 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} = |10\rangle \tag{5.7}$$

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 superposition inevitably leads to success with a non-unity success rate, as demonstrated in the next section, where we look at the three qubit case described using Dirac notation instead of matrices, as now we would have to use $2^3 \times 2^3$ matrices. Hence the transition to Dirac notation is a natural progression for larger system sizes.

Three qubit case

Here we consider the problem on $N = 8$ elements, where the fifth element is marked. Crucially, this change to a larger number of qubits means that the problem is no longer solved with certainty. The algorithm can be applied with a minimum of 3 ($2^3 = 8$) qubits in the following manner. Here we have let the marked element be the 5th element '100':

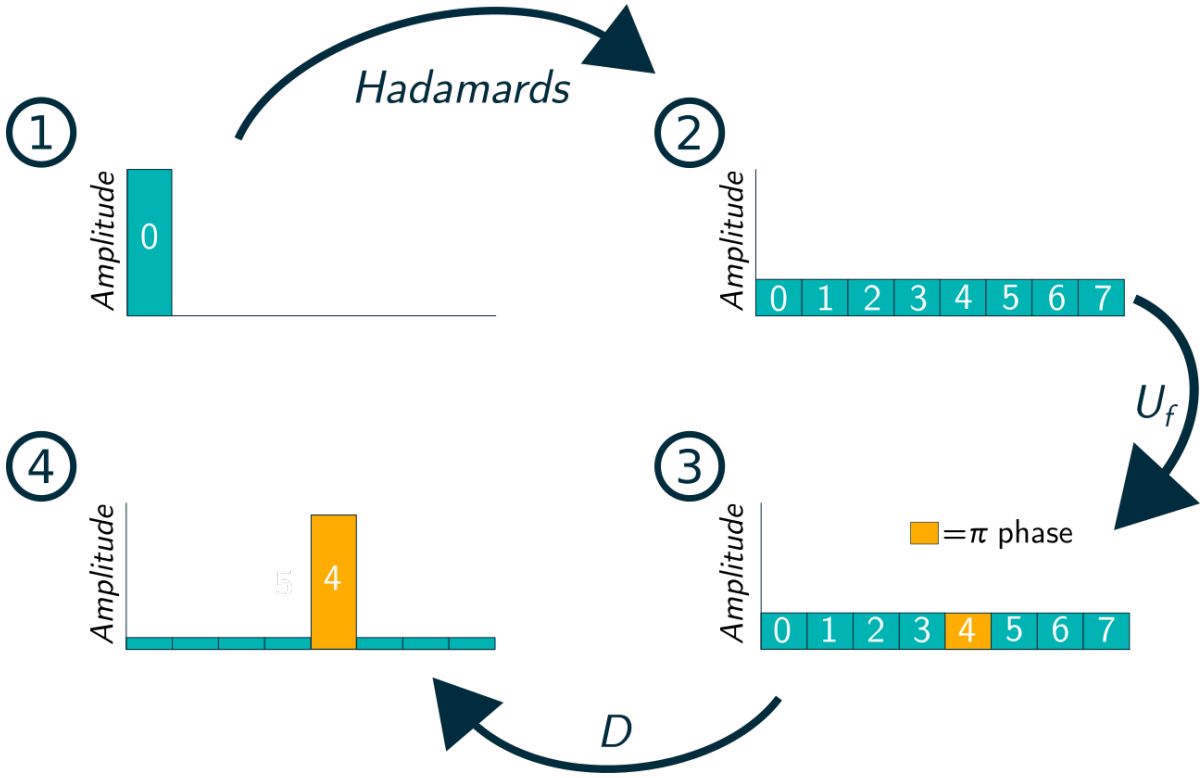


Figure 5.2: Pictorial demonstration of the steps of Grover's Algorithm

Grover's Algorithm on three qubits

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 (adds a π phase shift) 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 $|\Psi\rangle$.

Whereas for two qubits applying the combination of $U_f D$ once was sufficient to find the marked element with certainty, with more qubits we must be careful about the number of times this combi-

nation is applied. Repeating these two gates on the state gives an oscillating probability for finding the correct element. For clarification, if we carry out another iteration of steps 3 and 4 in the above example, we get the state:

$$|\Psi\rangle = \frac{-1}{8\sqrt{2}}\big(|000\rangle + |001\rangle + |010\rangle + |011\rangle - 11\,|100\rangle + |101\rangle + |110\rangle + |111\rangle\big). \tag{5.8}$$

Now if a measurement is performed on this state, there is a 94.5% chance of getting the fifth element compared to 78.1% probability after just one iteration. If yet another iteration of step 3 and 4 is performed, the resulting state becomes:

$$|\Psi\rangle = \frac{-7}{16\sqrt{2}}\big(|000\rangle + |001\rangle + |010\rangle + |011\rangle - \frac{13}{7}\,|100\rangle + |101\rangle + |110\rangle + |111\rangle\big) \tag{5.9}$$

and the probability of getting the marked element upon measurement decreases again to 67%. Therefore, it is important to choose the number of iterations for Grover's algorithm carefully. The number of iterations "T" required to get the maximum probability of measuring the marked element is approximately given by:

$$T \approx \frac{\pi}{4}\sqrt{N} - \frac{1}{2} \tag{5.10}$$

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 in the initial state $|0\rangle^{\otimes n}$.

Grover's Algorithm on n-qubits

1. Start with the state $|0\rangle^{\otimes n}$.
2. Apply the Hadamard gates to each qubit result in the state: $|\Psi\rangle = \frac{1}{\sqrt{n}}\sum_{x\in\{0,1\}^n}|x\rangle$
3. Apply $U_f\,|x\rangle = (-1)^{f(x)}\,|x\rangle$, which reverses the sign of the marked element.
4. Apply D where $D = -H^{\otimes n}U_0H^{\otimes n}$.
5. Repeat steps 3 and 4 T times, where $T = \frac{\pi}{4}\sqrt{N} - \frac{1}{2}$.
6. Measure the state $|\Psi\rangle$.

The circuit implementation of Grover's algorithm is given in Fig. 5.3.

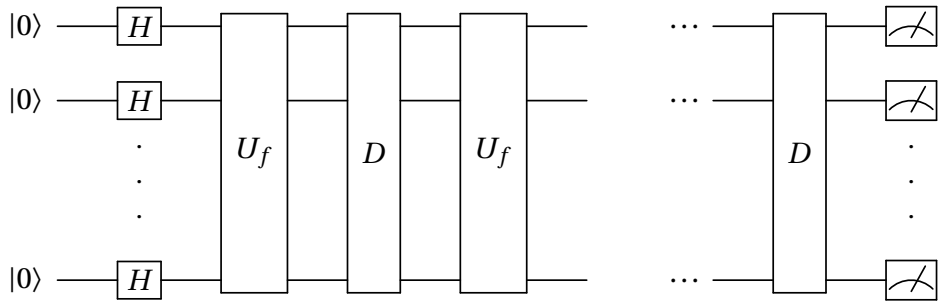


Figure 5.3: Circuit implementation of Grover's algorithm.

Construction of gate D and U_f

The gate D in Grover's algorithm applied on n-qubits can be can be decomposed into smaller gates as shown in Fig. 5.4. This shows that gate D requires $2n+1$ Hadamard (H) gates, $2n+1$ Pauli X gates and an n-control Toffoli gate.

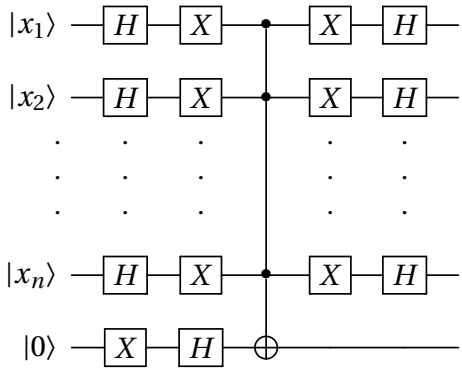


Figure 5.4: Decomposition of gate 'D' into Hadamard, Pauli X and n-control Toffoli gates.

The construction of gate U_f is not as straight forward as for the D gate above and in general is dependent on which algorithm being used. We have coded the oracle function required for Grover's algorithm in several languages in Chapter 8.

5.5 Shor's algorithm

Shor's algorithm describes a procedure for factorising large numbers. Given an n-digit integer N, Shor's algorithm determines whether N is prime and in the case it is not, the algorithm outputs a non-trivial factor of N. The algorithm consists of mainly classical ingredients with one computationally heavy step outsourced to a quantum computer. The first part of Shor's algorithm consists of a classical algorithm that determines the greatest common divisor (gcd) of two integers. This is efficiently achieved by using Euclid's algorithm. The second part of the algorithm involves finding the

periodicity of a discrete periodic function. This is achieved by employing a period finding algorithm and it is this step that utilises a quantum algorithm. The box below describes the Shor's algorithm in more detail.

Shor's algorithm for factorising N

1. Check if N is even, if so output 2. Else, Pick an initial guess $a < N$.
2. Calculate the greatest common divisor of 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. If we found r to be odd then this method has failed: return to step 1.
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.
5. Output either $\gcd(a^{r/2} - 1)$ or $\gcd(a^{r/2} + 1)$ as the factors of N.

Currently no efficient classical algorithm is known to solve step 3. However, using a quantum algorithm the problem can be solved exponentially faster than the best known classical algorithms. We shall now explain the machinery behind this process, aimed more at developing an intuition for the quantum effects rather than the fine detail of the process.

Fig. 5.7 shows a circuit to implement Shor algorithm. The 1^{st} register is acted upon by Hadamard gates to put the qubits in a uniform superposiiton. The second step uses the bit oracle operation O_f for the periodic function $f(x) = a^x \mod N$ which has the following effect on the second registers:

$$U_{a^x \mathrm{mod} N} |x\rangle |0\rangle = |x\rangle \left|a^x \mathrm{mod} N\right\rangle \tag{5.11}$$

Measuring the second register then collapses the state to basis states equal to $f_0 = f(x_0)$, where $f(x_0) = f(x_0 + r) = f(x_0 + 2r) = ...$ where r is the periodicity of the function. Thus the first register is now in 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 states, generating peaks in the probability distribution of measurement outcomes that correspond to the values x_0, $x_0 + r$, $x_0 + 2r...$ as can be seen in step 4 of Fig. 5.5. We can then extract the period with the quantum fourier transform.

5.5.1 Quantum Fourier Transform

One of the most useful transformations in mathematics is the discrete Fourier transform. The quantum Fourier transform (QFT) is exactly the same transformation apart from the fact that QFT is applied to quantum states. For example the QFT acting on two qubits is given by the transformation matrix:

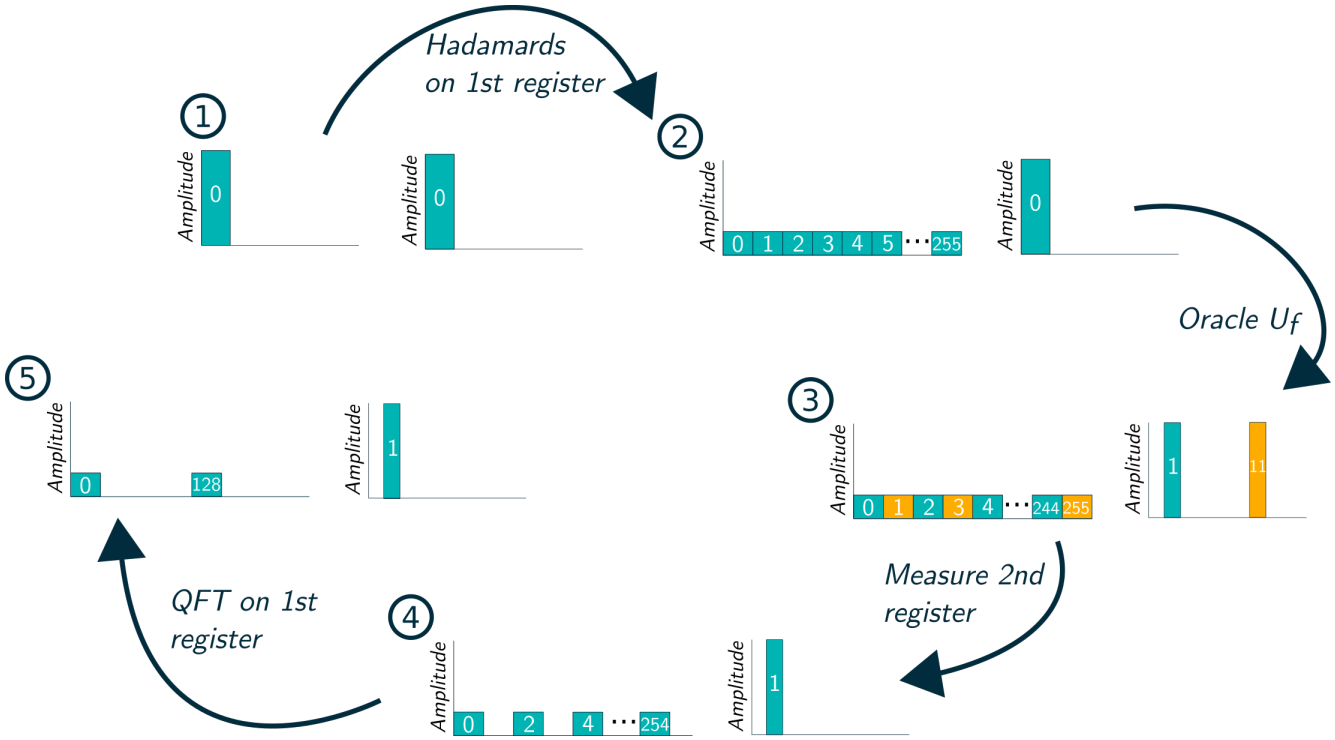


Figure 5.5: Diagram of the quantum aspects of Shor's algorithm to factor 15, using $a = 11$. In step 3, the different colours show which qubit values in the 1st and 2nd registers are associated to each other, a demonstration of entanglement. When the first register is measured, there is a 50% chance of success - if 128 is measured and divided by $2^8 = 256$, we can extract the correct value of $r = 2$ from the resulting fraction $l/r = 1/2$

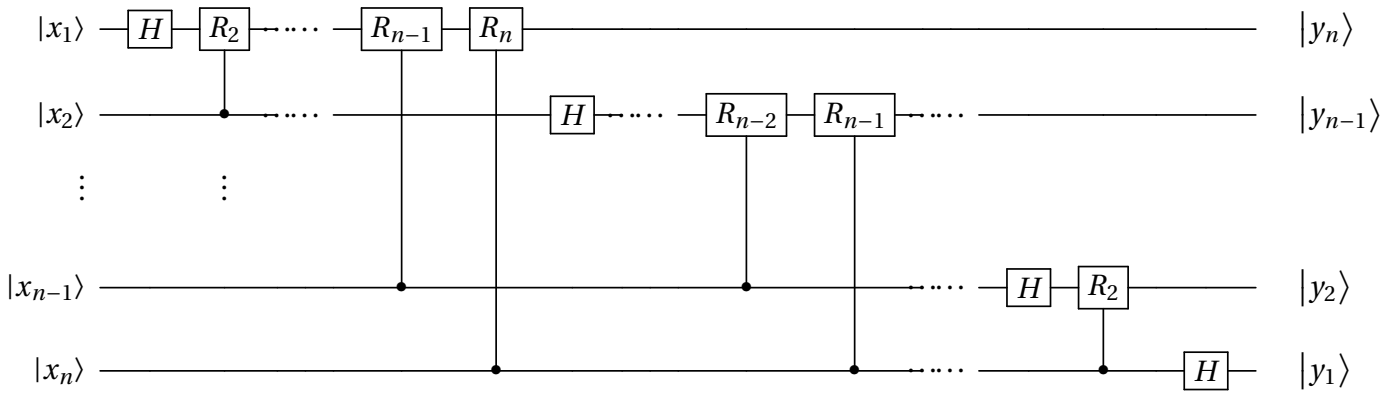


Figure 5.6: Efficient circuit for the quantum Fourier transform.

$$QFT = \frac{1}{2}\begin{pmatrix} \omega_N^0 & \omega_N^0 & \omega_N^0 & \omega_N^0 \\ \omega_N^0 & \omega_N^1 & \omega_N^2 & \omega_N^3 \\ \omega_N^0 & \omega_N^2 & \omega_N^4 & \omega_N^6 \\ \omega_N^0 & \omega_N^3 & \omega_N^6 & \omega_N^9 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & i & -1 & -i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & i \end{pmatrix} \tag{5.12}$$

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

$$QFT_{jk} \equiv \omega_N^{jk} = \exp^{2\pi i jk/N} \tag{5.13}$$

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

$$H = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}, \quad R_k = \begin{pmatrix} 1 & 0 \\ 0 & \omega_{2^k} \end{pmatrix} \tag{5.14}$$

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

The circuit diagram given by Fig. 5.6 shows how QFT is implemented on an n-qubit state $|x_1 \ldots x_n\rangle$ using Hadamard and controlled phase gates. The QFT outputs a state in the form $|y_n \ldots x_1\rangle$ and swap gates are used at the end of QFT to correct the order of the output state.

5.6 Quantum annealing

Quantum annealers are a type of optimisation algorithm. They can directly solve Quadratic unconstrained binary optimization (QUBO) problems. Given the coefficients Q_{ij}, the QUBO problem is to determine the minimum value of

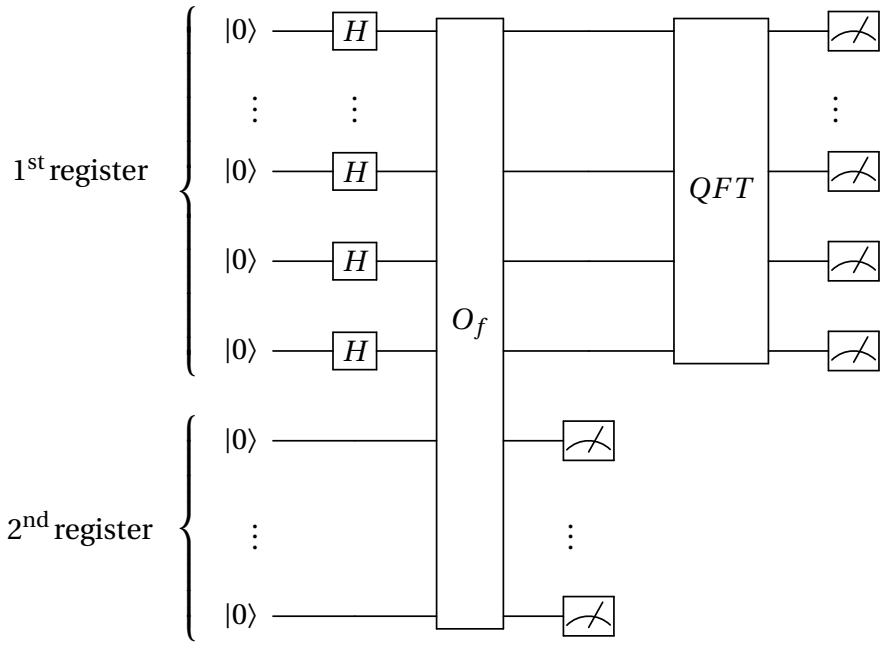


Figure 5.7: Circuit implementation of Shor's algorithm.

$$C = \sum_{i=1}^{N} Q_{ii} x_i + \sum_{i<j}^{N} Q_{ij} x_i x_j \; x_i \in \{0,1\}. \tag{5.15}$$

This type of problem is essential in the application ranging from graph theory [41] to machine learning [42],[43].

To give a better idea of what is QUBO problem, we will use the example of the light switching game taken from D-Wave's quantum computing primer [44]. Imagine there are several lights with certain bias marked for each of them and certain weights assigned between every two of them. For each switch, "ON" scores 1 point and "OFF" scores 0 point. The total score is the sum of the bias multiplied by the light configuration score. Furthermore, there is an additional score contribution if adjacent lights are both on. The goal is to set the light configuration such that the total score is minimised. If we have a positive bias for all of the lights and connections, it is intuitive to see such a configuration is setting all the light to "OFF". However, it is notoriously complicating to take the weights between the lights into account as well.Here is where the quantum annealer comes about.

5.6.1 Coloring

One of the NP problems which can be implemented in QUBO form is the vertex coloring problem. Given the graph G(V, E) with the set of vertices V and the set of edges E, the problem 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:

$$H = \sum_{n=0}^{N-1} (1 - \sum_{k=0}^{K-1} x_{n,k})^2 + \sum_{(u,v)\in E} \sum_{k=0}^{K-1} x_{u,k} x_{v,k} \tag{5.16}$$

$x_{n,k}$ with node n and color k is 1 only if the node n has color k and 0 otherwise. N is the total number of nodes and K is total number of colors 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 problems can be solved by current quantum annealers, Equation 5.16 must be reduced to a one dimension problem by reordering the index $n, k \rightarrow nK + k$. This results in

$$H = \sum_{n=0}^{N-1} (1 - \sum_{k=0}^{K-1} x_{nK+k})^2 + \sum_{(u,v)\in E} \sum_{k=0}^{K-1} x_{uK+k} x_{vK+k} \tag{5.17}$$

Further expansion and reduction using $x_i^2 = x_i$ gives

$$H = \sum_{n=0}^{N-1} (1 + \sum_{k_1=0}^{K-1} \sum_{k_2=0}^{K-1} x_{nK+k_1} x_{nK+k_2} - \sum_{k=0}^{K-1} x_{nK+k}) + \sum_{(u,v)\in E} \sum_{k=0}^{K-1} x_{uK+k} x_{vK+k} \tag{5.18}$$

This can be input directly into D-Wave system.

5.7 Variational Quantum Eigensolver (VQE) Algorithm

The dynamics of physical systems (examples materials and molecules) is modelled by Hamiltonians which are represented by Hermitian (self-adjoint) matrices. The modelling of these hamiltonians is important for the material science and developments of new drugs which involves working with huge of these Hermitian matrices representing complex molecules. Observables associated to these molecules, particularly the Energy of the system, and furthermore, the ground state energy of the system. So the problem here is an eigenvalue problem, given a (potentially huge) Hermitian matrix, finding the lowest eigenvalue. We can do this classically, but for huge matrices it requires a lot of resource (example here), so here is where the VQE offers speed ups REF.

The problem:

Given a physical system (say a molecule) represented by a n-qubit Hamiltonian H. The matrix H is an Hermitian matrix and therefore by the Spectral decomposition theorem it can be diagonalised in terms of its eigenvalues $\{\lambda_i\}$ eigenvectors $\{|\lambda_i\rangle\}$ as:

$$H = \sum_{i=0}^{n} \lambda_i |\lambda_i\rangle \langle\lambda_i|, \tag{5.19}$$

These eigenvectors form a basis. without loss of generality we can order the eigenvalues in increasing order $\lambda_0 < \lambda_1 < ... < \lambda_n$, so that the lowest eigenvalue λ_0 and associated eigenvector$|\lambda_0\rangle$ and $H|\lambda_0\rangle = \lambda_0|\lambda_0\rangle$. These eigenvalue and eigenvector are also called ground energy and ground state. We are interested in finding the lowest eigenvalue.

Classical Solution: Eigenvalue problem

Classically, and by virtue of the spectral decomposition theorem, the diagonalisation of H can, in principle, always be done, but here the question is if it can be done *efficiently*.

Quantum Solution: Variational Quantum Eigensolver (VQE) algorithm

In the VQE the situation is the following [45–51]:

1. Take the state of the system to an Ansatz state $\left|\psi(\theta)\right\rangle$.
2. Implementing the matrtix H as a gate, and applying it getting the state $H\left|\psi(\theta)\right\rangle$.
3. We can measure the system in a base containing the ansatz state $\left|\psi(\theta)\right\rangle$, and therefore we can obtain the expected value for the energy of the system when in state $\left|\psi(\theta)\right\rangle$ which is given by $\lambda_{\psi(\theta)} := \langle\psi(\theta)|H|\psi(\theta)\rangle$. This expected value is always greater or equal than the ground energy (See aside for a derivation):

$$\lambda_{\psi(\theta)} \geq \lambda_0, \quad \forall \left|\psi(\theta)\right\rangle. \tag{5.20}$$

4. With the help of a classical minimisation optimiser to move over the parameter(s) θ and after sampling enough times one should expect to be considerable close to the ground energy λ_0, and therefore the algorithm will output a λ_0' as a guess for the ground energy of the system. The classical optimiser will propose a new set of parameters θ' and we start the loop again. This is repeated an amoun of times which depend on the classical optimiser.

In Fig. 5.8 we have a diagram schematically representing the steps of the VQE.

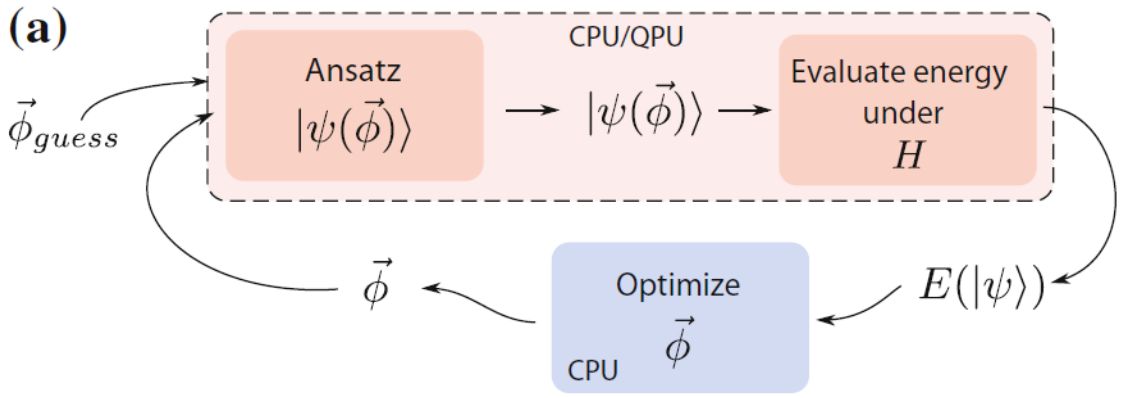


Figure 5.8: Someone pls could draw a pic like this? This is from Shadbolt's Thesis.

Aside: Deriving inequality Eq. 5.20

We have that we can write any state in terms of the energy basis $\{|\lambda_i\rangle\}$, the Ansatz in particular as:

$$\big|\psi(\theta)\big\rangle = \sum_{i=0}^{n} \psi_i \,|\lambda_i\rangle\,, \qquad \psi_i \in \mathbb{C}, \qquad \sum_{i=0}^{n} |\psi_i|^2 = 1.$$

Considering the expected value for the energy of the system in that state we have:

$$H\big|\psi(\theta)\big\rangle = \sum_{i=0}^{n} \psi_i H\,|\lambda_i\rangle = \sum_{i=0}^{n} \psi_i \lambda_i\,|\lambda_i\rangle\,.$$

Considering the expected value for the energy in the state $\big|\psi(\theta)\big\rangle$:

$$\lambda_{\psi(\theta)} = \langle\psi(\theta)|H|\psi(\theta)\rangle = \sum_{i=0}^{n} |\psi_i|^2\lambda_i \geq \sum_{i=0}^{n} |\psi_i|^2\lambda_0 = \lambda_0 \sum_{i=0}^{n} |\psi_i|^2 = \lambda_0.$$

Therefore we get the inequality:

$$\lambda_{\psi(\theta)} = \langle\psi(\theta)|H|\psi(\theta)\rangle \geq \langle\lambda_0|H|\lambda_0\rangle = \lambda_0.$$

5.8 Summary of algorithms covered in this chapter

Algorithm	**Purpose**	**Number of qubits required**	**Number of gates required**	**Success probability**
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				
Shor's Algorithm	Factor a large number N	$4\ln(N)/\ln(2) \leq n < 4\ln(2N)/\ln(2)$	$n < 4\ln(2N)/\ln(2)$ H gates, U_f gate, QFT gate	

Table 5.1: Table of Quantum Algorithms

Chapter 6

Noisy Intermediate-Scale Quantum (NISQ) Algorithms

Attaining quantum supremacy and exploring its consequences will be among the great challenges facing 21st century science, and our imaginations are poorly equipped to envision the scientific rewards of manipulating highly entangled quantum states, or the potential benefits of advanced quantum technologies.

John Preskill [7]

Quantum algorithms were first devised by David Deutsch in 1985 with the proposal of what nowadays is known as Deutsch's algorithm [52]. The development of quantum algorithms exploded during the nineties with the proposal of several other quantum algorithms [53, 54] as the famous Shor's and Grover's algorithms. Althought the problems addressed by quantum algorithms can be *simulated* by classical computers, they cannot be *efficiently simulated*, as in classical computers would require ridiculously long periods of time to do so. The appealing feature of quantum algorithms is that for particular types of problems, they offer speedups in comparison with their classical counterparts.

The great majority of quantum algorithms (if not all) were initially developed to be of a practical use when implemented in a Fault-Tolerant Large-Scale Quantum (FTLSQ) computer. A *fault-tolerant* quantum device refers to a device having low enough noise such that is able to implement an algorithm with a decent (number?) circuit depth. A *large-scale* quantum device refers to a device being able to control a number of $\sim 10^6$ physical qubits (how many logical?). Even though the construction of FTLSQ devices is a long-term goal which is believed to be at least several decades away [55], it has recently entered to the so-called era of: Noisy Intermediate-Scale Quantum (NISQ) devices. A quantum device is considered *noisy* when is able to implement quantum algorithms with a circuit depth of less than 1000 gates. A *intermediate-scale* quantum device refers to a device controlling a number of $\sim 50-100$ physical qubits [56]. Smaller devices < 20 have also been called *quantum processors* [57].

Quantum algorithms are expected to outperform their classical counterparts when implemented in FTLSQ computers and they are not expected (in principle) to do so when implemented in quantum

processors or NISQ devices. In fact, even though several of these quantum algorithms have already been implemented in quantum processors and NISQ devices, *all* of these instances can be simulated by classical computers. In order to achieve the breaking point of having quantum devices that can efficiently run an algorithm which cannot efficiently be classically simulated, we need to keep further scaling up current quantum hardware and particularly, keep developing algorithms suitable to be run on NISQ devices. This breaking point has been coined as *quantum supremacy*[1].

Quantum supremacy refers to the implementation of a quantum algorithm that cannot be efficiently implemented in a classical computer. Achieving quantum supremacy is a milestone for quantum computing which has not been reached yet, and it is a very active area of research. The term was first introduced by Preskill in [7] and further formalised later by Harrow and Montanaro [61]. Quantum supremacy through Boson sampling seems rather further away from what was expected [62] and theoretical works predict a minimum amount of 90 qubits in order to achieve it [63]. In Fig. 6.1 we have the research direction taken by Google in terms of quantum computing, specifying: NISQ era, FTLSQ era and the breaking point of quantum supremacy. It is fair to say that this is the general trend that most companies working on quantum computing are currently following.

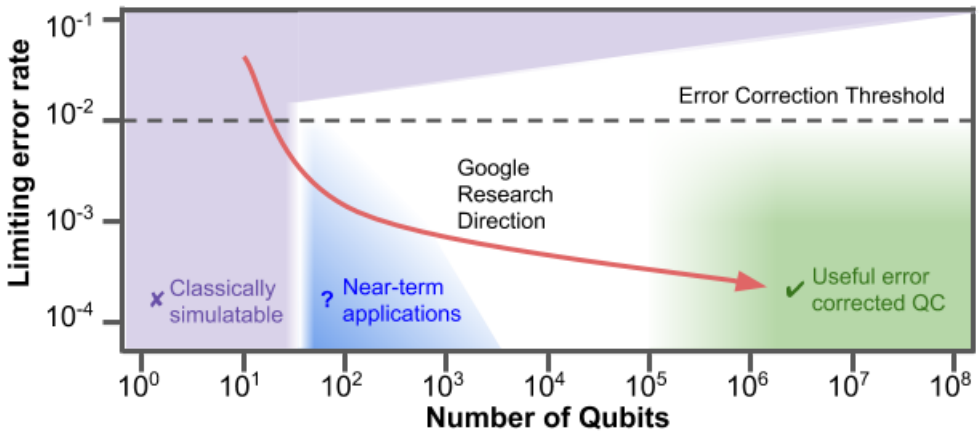


Figure 6.1: Google quantum computing research direction. Quantum devices with 50 or less qubits can be simulated with classical computers. NISQ devices are being considered as those with $50-100$ qubits. Near-term devices are those containing from 50 to a few thousands qubits. FTLSQ devices will contain 10^5 qubits. Quantum supremacy refers to an implementation that cannot be classically simulated. Image taken from [32].

There are several candidates algorithms in the race for quantum supremacy and a not comprehensive list includes the following: Quantum optimisers such as the Quantum Approximate Optimisation Algorithm (QAOA) [64–66] particularly the Variational Quantum Eigensolver (VQE) [45, 46], adiabatic quantum computing such as quantum annealing [67], quantum deep and machine learning [68, 69], quantum matrix inversion [70], quantum recommendation systems [71], quantum semidefinite programming (QSDP) [72, 73], quantum simulation [74], quantum games [28], and quantum

[1] In regards of the use of the term "Quantum Supremacy", we would like to quote Ref [58]:
The use of the word "supremacy"—which denotes "the state or condition of being superior to all others in authority"—was criticised in [59] because the syntagm 'white supremacy' is associated with the racial segregation and discrimination of the apartheid regime of South Africa. Proposals like "quantum advantage" or "quantum superiority" have been discussed [60], but to date none has gained ground.

kitchen sinks [75]. In this chapter we are going to address some basic examples for quantum annealing and the variational quantum eigensolver (VQE).

6.1 Adiabatic quantum computing & quantum annealers

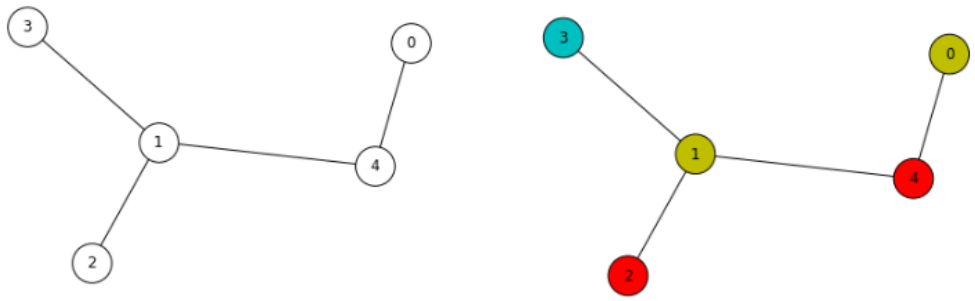
In this section, we will present the example of vertex coloring problem using both D-Wave ocean stack as well as 1QBIT QUANTUM-READY™ SDK following the algorithm discussed in the section 5.6.

6.1.1 Vertex Coloring with 1QBIT QUANTUM-READY™ SDK

Example: Vertex Coloring

Let's solve the vertex coloring problem for the configuration shown in Figure 6.2a

Figure 6.2: Problem setting and sample solution of vertex coloring problem



(a) Problem setting of the Vertex Coloring (b) Final configuration after coloring

First we want to import the `networkx` and `matplotlib` library for the graph plotting.

```
import networkx as nx
import matplotlib.pyplot as plt

from qdk.binary_polynomial import *
from qdk.common_solver_interface import *
```

The configuration of the coloring program is set by setting the nodes and neighbours. The graph of the confifuration can be drawn.

```
# Set the graph configuration by setting nodes and neighbours
nodes = [0,1,2,3,4]
neighbours = [(0,4),(1,2),(1,3),(1,4)]
# Draw the graph configuration
```

```
g = nx.Graph()
g.add_nodes_from(nodes)
g.add_edges_from(neighbours)
pos = nx.spring_layout(g)
nx.draw(g, pos=pos, with_labels=True, node_color='w', node_size=1000)
plt.show()
```

As only one dimensional QUBO problem can be solved, the equation should be reduced to one dimension by reordering the index $n, k \rightarrow nK + k$.

```
# Turn the two dimension index to one dimension
def d2tod1(n, k, K):
    return n * K + k
```

Define the function to transfer the final solution to the colour label on the graph.

```
# Function to color the final configuration
pallete = {0: 'r', 1: 'c', 2:'m', 3:'y'}
def get_color(n, sol, K):
    z = d2tod1(n, 0, K)
    color = 0
    for k in range(K):
        if sol[z + k]:
            break
     return pallete[k]
```

Build the quadratic equation for vertex colouring problem:

$$H = \sum_{n=0}^{N-1} (1 - \sum_{k=0}^{K-1} x_{nK+k})^2 + \sum_{(u,v)\in E} \sum_{k=0}^{K-1} x_{uK+k} x_{vK+k} \tag{6.1}$$

```
# Construct the quadratic equation
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, d2tod1(n,k,K))
    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, d2tod1(u,k,K), d2tod1(v,k,K))
        qubo.sum(builder.build_polynomial())
print (qubo)
```

We then set up the local solver and sample for 300 times

```
solver = DWaveSolver()
solver.solver.num_reads = 300
sol = solver.minimize(qubo).peek_minimum_energy_solution().configuration
```

Finally we can draw the final configuration using the node_color function defined.

```
# Draw the final configuration with color applied
nx.draw(g, pos=pos, with_labels=True, nodelist=g.nodes(),
node_color=[get_color(n, sol, K) for n in g.nodes()], node_size=1000)
plt.show()
```

6.1.2 Vertex Colouring with Ocean

Following really similar steps, the same colouring problem can be implemented on D-Wave Ocean.

```
# Import the library
import matplotlib.pyplot as plt
import networkx as nx
import dwave_qbsolv

# Set the graph configuration by setting nodes and neighbours
nodes = [0,1,2,3,4]
neighbours = [(0,4),(1,2),(1,3),(1,4)]

# Turn the two dimension index to one dimension
def d2tod1(n, k, K):
    return n * K + k
pallete = {0: 'r', 1: 'c', 2:'m', 3:'y'}

# Function to color the final configuration
def get_color(n, sol, K):
    z = d2tod1(n, 0, K)
    k = 0
    for k in range(K):
        if sol[z + k]==1:
            break
    return pallete[k]

# Construct the quadratic equation
N = 5
K = 4
qubo = BinaryQuadraticModel.empty(dimod.BINARY)

for n in range(N):
    qubo.add_offset(1)
    for k1 in range(K):
        for k2 in range(K):
            if k1!=k2:
```

```
                qubo.add_interaction(d2tod1(n,k1,K),d2tod1(n,k2,K),1)
    for k in range(K):
        qubo.add_variable(d2tod1(n,k,K),-1)
for (u,v) in neighbours:
    for k in range(K):
        qubo.add_interaction(d2tod1(u,k,K), d2tod1(v,k,K),1)
print(qubo)

# Set up the solver
solver = dwave_qbsolv.QBSolv()
# Solve for the minimum energy configuration
response = solver.sample(qubo, num_reads=300)
sol = list(response.samples())

# Draw the final coloring configuration
# Multiple possible minimum configurations are given as the output, here we take one of them
 ↪  to draw the graph
s = sol[0]
print(s)
g = nx.Graph()
g.add_nodes_from(nodes)
g.add_edges_from(neighbours)
pos = nx.spring_layout(g)
nx.draw(g, pos=pos, with_labels=True, nodelist=nodes,
node_color=[get_color(n,s,K) for n in nodes], node_size=1000)
plt.show()
```

6.2 Implementation of the Variational Quantum Eigensolver (VQE) Algorithm

In this section we are going to implement a simple example of the VQE algorithm, using the libraries provided by pyQuil and QISKit. This algorithm can also be implemented in Project Q and Q# but it needs to be done from scratch, so it is left as an exercise to the interested reader.

6.2.1 VQE Algorithm in pyQuil

In this subsection we are going to use the VQE algorithm from Grove's library to address the following simple VQE problem.

Example: A simple VQE algorithm in pyQuil

Consider the one-qubit Hamiltonian $H = \sigma_z$ and an ansatz given by: $\left|\psi(\theta)\right\rangle = \cos\theta/2\,|0\rangle - i\sin\theta/2\,|1\rangle = \mathrm{RX}(\theta)\,|0\rangle$. Find the ground state energy of the system.

Spoiler alert, this Hamiltonian has eigenvalues $+1,-1$, so the ground state energy is -1 which corresponds to the eigenvector $\left|\psi(\theta=\pi)\right\rangle = i\,|1\rangle$. However, we now want a quantum computer to do this by sampling different Ansatzes. This is interesting because for huge matrices, this VQE does better than classical methods at finding the ground eigenvalue.

```
# 1. Calling Libraries
from pyquil.quil   import Program
from pyquil.api    import QVMConnection
from pyquil.gates  import RX
from pyquil.paulis import sZ,PauliSum,PauliTerm

# Calling Grove Library and optimiser
from grove.pyvqe.vqe import VQE
import numpy as np
from scipy.optimize  import minimize

# 2. Initialising
qp = Program()
qvm = QVMConnection()
vqe = VQE(minimizer=minimize, minimizer_kwargs={'method': 'nelder-mead'})

# 3. Defining ansatz
def ansatzv(theta): # input vectors.
    qp1 = Program()
    qp1.inst(RX(theta[0],0))
    return qp1

# 4. Defining Hamiltonian
hamiltonian = sZ(0)

# 5. Running the VQE
initial_angle = [0.0]
result = vqe.vqe_run(ansatzv, hamiltonian, initial_angle, None, qvm=qvm)
print(result)
```

1. **Calling libraries** - We start by stating the libraries that we are going to use, quil, api, gates and now we also need paulis. We also need to call the vqe from grove and minimize from scipy.

2. **Initialising** - We initialise an object program, the connection to the QVM and a vqe object, setting the classical optimiser as being the Nelder Mead.

3. **Defining the Ansatz** - In pyQuil we are allowed to address the Ansatz. In this case we are directly implementing the required Ansatz as program in terms of an input angle.

4. **Defining Hamiltonian** - The Hamiltonian should be input as in terms of the pauli matrices and the paulis dictionary.

5. **Running the VQE** - We run the VQE with inputs being the ansatz program, a vector with initial the initial angle and the Hamiltonian of interest.

Printing results we first obtain the value of the parameter for which the minimum has been achieved, followed by the minimum.

```
{'x': array([3.1415625]), 'fun': -0.9999999995453805}
```

We are obtaining the expected answer as ground energy of -1 with ground state $\left|\psi(\theta)\right\rangle = \cos\theta\,|0\rangle - i\sin\theta/2\,|1\rangle = \mathrm{RX}(\theta/2)\,|0\rangle$, with $\theta = \pi$.

6.2.2 VQE algorithm in QISKit

Applications that can run on ibmqx4 include simulation of quantum state evolution and solving for the lowest energy of the system using a VQE. QISKit Aqua provides tools that can be used to make an eigensolver for the ground state energy of simple systems.

The first example we will explore is the case of finding the lowest eigenvalue for the Z Pauli matrix. By inspection, we know that the value is -1. We can confirm that QISKit can also figure this out using the code seen in Listing 6.1. The backend is specified as the local qasm simulator, which means this is being run on the local CPU. This initial step allows us to define our operator matrix in terms of the Paulis. The coefficients specify multipliers of these matrices. The next part details how we will run this code, and where it will be run (the backend). `algo_input` represents the input of the code that is described by the operator made of Paulis in the pauli dictionary. The algorithm input is then further specified as the instance of the energy of the system, labelled as EnergyInput in the QISKit nomenclature.

```
from qiskit_aqua import Operator, run_algorithm
from qiskit_aqua.input import get_input_instance
# Defines the dictionary of pauli operations
pauli_dict = {"paulis": [
    { "coeff": { "imag": 0.0, "real": 1.0 }, "label": "Z"}]}
algo_input = get_input_instance("EnergyInput")
algo_input.qubit_op = Operator.load_from_dict(pauli_dict)
params = {
  "algorithm": { "name": "VQE" },
  "optimizer": { "name": "SPSA" },
  "variational_form": { "name": "RY", "depth": 5 },
  "backend": { "name": "local_qasm_simulator" }}
# Runs a local simulation to produce the energy result
result = run_algorithm(params, algo_input)
print(result["energy"])
```

Listing 6.1: The example code that uses the VQE to find the lowest eigenvalue of the Z pauli operator. Code modified from the one that can be found at [76]

After a few minutes of running, this code produces the result of -1.0. This is exactly what we would expect to see.

The next example provided on the IBM QISKit Aqua site details a basic working code that displays the energy result of an operator that we can define in terms of tensor products of the Pauli matrices X and Z, as seen in Eq. 6.2. These matrices and their corresponding alpha coefficients specify an operator from which we can extract energy eigenvalues. This is labelled by the pauli dictionary in the code Listing 6.2. Once again, the calculation is done on the local simulator.

$$O = \alpha_1 (Z \otimes X) + \alpha_2 (Z \otimes Z) + \alpha_3 (X \otimes Z) \tag{6.2}$$

```
from qiskit_aqua import Operator, run_algorithm
from qiskit_aqua.input import get_input_instance
# Defines the dictionary of tensor products of pauli operations and the alpha coefficients we
 ↪  described in the operator equation
pauli_dict = {"paulis": [
    { "coeff": { "imag": 0.0, "real": 0.5 }, "label": "ZX" },
    { "coeff": { "imag": 0.0, "real": -1.0 }, "label": "ZZ" },
    { "coeff": { "imag": 0.0, "real": 0.5 }, "label": "XZ" }]}
algo_input = get_input_instance("EnergyInput")
algo_input.qubit_op = Operator.load_from_dict(pauli_dict)
# Defines the algorithm input in terms of the pauli dict and that we want the energy value
params = {
  "algorithm": { "name": "VQE" },
  "optimizer": { "name": "SPSA" },
  "variational_form": { "name": "RY", "depth": 5 },
  "backend": { "name": "local_qasm_simulator" }}
# Runs a local simulation in qasm to produce the energy result
result = run_algorithm(params, algo_input)
# Prints the final result of the energy of the system
print(result["energy"])
```

Listing 6.2: The example code that utilises the VQE as presented in the QISKit Aqua documentation that prints the result of the energy of the system defined in it. Modified code originally found at [76].

The result function is imported at the start from qiskit aqua and takes the parameters (params) dictionary that specifies the algorithm being run, the python optimizer (of which there are several) used and the backend used to run the code plus the two-qubit system defined by the pauli dict list as arguments. The results printed in the first three runs (and a few minutes of patiently waiting) of this code were: -1.4560546875, -1.4287109375 and -1.3896484375. All the decimal points have been kept to illustrate the slight changes that occur between iterations. This method is sensitive to the changes in the coefficients applied to the matrices used to define the operator.

6.3 TODO Summary and Outlook

6.4 TODO Exercises

Exercise 1 with QISKIT Aqua VQE
Try and modify the coefficients $\alpha_{1,2,3}$ in the VQE code and change their corresponding operators to the following: $Z \otimes I$, $I \otimes X$ and $X \otimes I$. How does this affect the output from the code? How much does each answer vary from the mean output value if you run the code multiple times? You can simplify the arithmetic by automating the process in your code if you'd like.

Exercise 2 with QISKIT Aqua VQE
Can we simulate a molecule in QISKit? We can compose a simulation of the ground-state energy of the Helium hydride molecular ion (HeH^+) in the VQE by using the approximations described in . We can use the supplementary information of that publication to add the additional operators and their matching coefficients. After running the new code, you should find that your value(s) is twice the value computed in the publication. Why could this be?

Chapter 7

Programming a future universal quantum computer

> There's plenty of room at the bottom...
>
> Richard Feynman

At present, the current power of quantum devices is fundamentally limited by the number of qubits we can engineer in the same physical system. To date, the most qubits achieved on a physical chip is 72 by Google's Bristlecone architecture. However, for a true quantum advantage, most theorise (taking into account the number of qubits required for error correction) that at least one million qubits are required for useful, universal quantum computation.

From the previous chapter, we learned about useful short-term applications of quantum enhanced computation. These applications, VQE's, adiabatic quantum annealers, will become ever more useful with an increase in resource size, most notably not just simulating the ground states of larger and larger molecules, but also interactions between molecules for catalyst, superconducting material, and drug discovery research.

Aside from increasing resource size for NISQ applications, in the long term increasing the number of error-corrected qubits in a system will allow us to preform much more complex calculations, such as the NP-hard problem of integer factorisation utilising Shor's algorithm, or more efficient unstructured search algorithms via Grover's algorithm. Any problem that can be mapped to these algorithms can benefit from a quantum computational speed-up.

It is for this reason, that in this section we will be concentrating on the implementation of Shor's and Grover's algorithms (where currently possible) on a multitude of different quantum software platforms. Here we hope to deliver an intuition as to how to program truly useful algorithms on quantum processors in the future.

7.1 Implementing Grover's algorithm

In this section we will implement Grover's algorithm as discussed in section 5.4. The steps that we need to implement Grover's algorithm are as follows

1. Import/initialise libraries
2. Create the equal superposition
3. Define the oracle operations U_f and D described in section 5.4
4. Perform the oracle operations T times
5. Measure the register to obtain the output

For our implementations the unique bit string to be found is given by either the user or a random number generator, rather than being given a black box with a single marked element. Therefore this just serves as a simple example for how Grover's algorithm will work.

We will aim to discuss the most important sections of the code which are the quantum parts. For the full code listings please check out our Appendix/Github.

7.1.1 Grover's Algorithm with pyQuil

Initialise libraries

We start by calling the libraries that we are going to use

```
1 import numpy as np
2
3 from pyquil.api import QVMConnection
4 from pyquil.quil import Program
5 from pyquil.gates import H
```

and initialise the program

```
18     # Invoking and renaming Program and Connection
19     qvm = QVMConnection()
20     p = Program()
```

Create the equal superposition

We can create the equal superposition by looping through all of the qubits and applying a Hadamard to each of them.

```
    # Step 1: Start with qubits in equal superposition
    for i in range(n):
        p.inst(H(i))
```

Create the oracle operations

Defining the Oracles matrices and adding them to be gates

```
# Defining Oracle matrices: U0 and Uf
vec0=np.ones((2**n,), dtype=np.int)
vecf=np.ones((2**n,), dtype=np.int)
vec0[0]=vec0[0]*(-1)
vecf[nf]=vecf[nf]*(-1)
U0ma=(-1)*np.diag(vec0)
Ufma=np.diag(vecf)
# Defining Oracle gates
p.defgate("U0",U0ma)
p.defgate("Uf",Ufma)
```

Apply the oracle operations

Calculating T and applying the operator of interest T times.

```
# Calculating T for the loop
T=((np.pi/4)*(1/np.arcsin(1/np.sqrt(2**n))))-0.5 # N=2**n
T=np.rint(T)
T=int(T)
# Applying Oracle loop T times
for i in range(1,T+1,1):
    p.inst(("Uf",)+tuple(range(n)))
    for ii in range(0, n, 1):
        p.inst(H(ii))
    p.inst(("U0",)+tuple(range(n)))
    for ii in range(0, n, 1):
        p.inst(H(ii))
```

Measure the register to obtain the output

Measuring all the qubits and running the program.

```
# Measurement stage
for ii in range(0, n, 1):
    p.measure(ii,ii)
# Running the program
results=qvm.run(p,[],5)
guess=results[0]
print("""Grover's algorithm is guessing that the marked string is""",guess)
```

The algorithm ends up printing a guess for the single-marked string.

7.1.2 Grover's Algorithm with QISKit

Initialise libraries

Firstly, as with most regular programming languages, we need to import the libraries that we intend to make use of:

```
# Program written using QISKit to demonstrate the way in which to implement Grovers search
# algorithm and then simulate the measurement on IBM's Q QASM simulator

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

# Import relevant library functions
import numpy as np
```

Here we define out function, the length of the marked element inputted by the user, the length of said marked element, and the number of qubits required to preform a search algorithm that will find said marked element:

```
def grover(marked_element):

        # Determine the number of qubits needed
        n = len(marked_element)
        N = 2**n
        print('Number of Qubits required is', n)
        print('Number of Ancillas required is', n)
```

Due to the way in which QISKit handles inputted strings, for the output of our function to make sense it is necessary to flip the values of the use inputted bit string:

```
        # Flip bitstring
        marked_element = marked_element[::-1]
```

Here we add in protection against user input that would require too many qubits to execute successfully. Via QISKit's local Q QASM simulator, we have the equivalent of 32 qubits available to us to

preform simulations. In these lines the function returns 0 is the used input requires more than 32 qubits:

```
        # Check if we can simulate on our local simulator (QASM Local simulator has 32
        # qubits)
        if n > 32:
                print('Number of qubits required =', 2*n, 'which is too large to simulate.')
                return 0
```

The code here shows how we calculate the number of iterations necessary to identify the user inputted bit string:

```
        # Determine the number of times to iterate
        T = int(round(np.pi*np.sqrt(N)/4 - 0.5))
        print('Number of iterations T =',T)
```

Now comes the QISKit element. Below we show how to define our n-qubit register, an $n+1$ classical readout register, $n-1$ ancilla qubits for two qubit operations, and a target register to aid in the implementation of an n-qubit CNOT gate. We then combine all of these resources into a single quantum circuit we can manipulate:

```
        # Initialise n qubit register, classical readout register and ancilla qubit
        q = QuantumRegister(n, 'ctrl')
        c = ClassicalRegister(n+1, 'meas')
        a = QuantumRegister(n-1, 'anc')
        t = QuantumRegister(1, 'tgt')

        # Combine resources into a quantum circuit
        qc = QuantumCircuit(q, a, t, c)
```

Create the equal superposition

Here we manipulate entire registers by looping through and applying unitary gates to each qubit in the register:

```
        # Step 1: Start in an equal weighted superposition state on the quantum register
        for i in range(n):
                qc.h(q[i])

        # Put the ancilla qubit into the - state
        qc.x(a[0])
        qc.h(a[0])
```

Apply the oracle operations

From any quantum computation literature it is known that to execute Grover's successfully, the operations D and U_f must be repeated sequentially T times. This is shown in the framework of QISKit below:

```
        # Step 2: Repeat applications of U_f and D
        for i in range(T):
                # Apply U_f
                for j in range(n):
                        if marked_element[j] == '0':
                                qc.x(q[j])

                # Implement an n-qubit CNOT gate
                qc.ccx(q[0], q[1], a[0])
                for i in range(2, n):
                        qc.ccx(q[i], a[i-2], a[i-1])
                # Copy
                qc.cx(a[n-2], t[0])
                # Uncompute
                for i in range(n-1, 1, -1):
                        qc.ccx(q[i], a[i-2], a[i-1])
                qc.ccx(q[0], q[1], a[0])

                for j in range(n):
                        if marked_element[j] == '0':
                                qc.x(q[j])

                # Apply D

                # Apply H
                for j in range(n):
                        qc.h(q[j])
                        qc.x(q[j])

                # Apply U_0
                # Implement an n-qubit CNOT gate
                qc.ccx(q[0], q[1], a[0])
                for i in range(2, n):
                        qc.ccx(q[i], a[i-2], a[i-1])
                # Copy
                qc.cx(a[n-2], t[0])
                # Uncompute
                for i in range(n-1, 1, -1):
                        qc.ccx(q[i], a[i-2], a[i-1])
                qc.ccx(q[0], q[1], a[0])

                # Apply H
                for j in range(n):
                        qc.x(q[j])
                        qc.h(q[j])
```

Measure the register to obtain the output

Coming to the end of the program, we 'measure' our quantum register onto a classical register:

```
87                  # Measure our quantum register via our classical register
88                  for i in range(n):
89                          qc.measure(q[i], c[i])
```

Finally, we execute our program on the local Q QASM simulator:

```
90                  # Execute the quantum circuit on the local simulator
91                  job = execute(qc, 'local_qasm_simulator')
92                  result = job.result()
93                  print('The results of the simulation shots are:', result.get_counts(qc))
```

Here we end the function by defining what type of input the user can submit to the function we just defined:

```
94  # Define input type for def
95  str_input = input("Input a bit string to find: ")
96  grover(str_input)
```

7.1.3 Grover's Algorithm with ProjectQ

The first thing to do is to import the functions from ProjectQ that will be required to create and run the program.

```
3  from projectq import MainEngine
4  from projectq.ops import All, Measure, H, X
5  from projectq.meta import Control
```

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.

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

Now the local simulator can be loaded by calling `MainEngine` and our qubits allocated. The number of qubits required is the length of the bit string provided by the user plus an ancilla qubit. The ancilla is required because we will use it to implement the oracle U_f and the gate D.

```
    # Initialise qubits
    engine = MainEngine()
    qubits = engine.allocate_qureg(n)
    ancilla = engine.allocate_qubit()
```

The main qubits need to start in an equal superposition, so Hadamards can be applied to all of them using the function `All`. The ancilla needs to start in the $|-\rangle$ state, and since qubits are initialised as $|0\rangle$, we can achieve this with an X followed by a H gate.

```
    # Step 1: Start with qubits in equal superposition and ancilla in |->
    All(H) | qubits
    X | ancilla
    H | ancilla
```

Now we come to the part of the algorithm where we must alternate between the gates U_f and D for T iterations where T is the closest integer to $\frac{\pi}{4}\sqrt{N}-\frac{1}{2}$. ProjectQ does allow you to define new gates from their matrix representations using `BasicGate`, but this only works well for one or two qubit gates. Therefore we need a different, more efficient way of doing the oracle.

To perform U_f, we can use the function `Control` from ProjectQ. This controls on all the qubits being in the state $|1\rangle$ - so it essentially functions as a quantum `if` statement. Suppose that the marked bit string is '101', U_f should flip the phase of $|101\rangle$. But since we can only control on all the bits being '1', we need to flip the second bit using an X gate. Once we have done this, we can use an X gate on the ancilla to introduce this phase, then flip the second bit back to normal. Our code can now generalise to any bit string.

```
        # Apply U_f
        for j in range(n):
            if marked_element[j] == '0':
                X | qubits[j]
        with Control(engine, qubits):
            X | ancilla
        for j in range(n):
            if marked_element[j] == '0':
                X | qubits[j]
```

Now we can do a similar construction to implement the gate $D = H^{\otimes n}U_0H^{\otimes n}$. This time flipping all the qubits with X since we need to control on all the qubits being $|0\rangle$.

```
        # Apply D
        # Apply Hadamards
        All(H) | qubits
        # Apply U_0
        All(X) | qubits
        with Control(engine, qubits):
            X | ancilla
        All(X) | qubits
        # Apply Hadamards
        All(H) | qubits
```

Finally, after we exit the loop, we must measure all the qubits using the `Measure` command and use `flush` to send all the instructions to the simulator. The result can then be converted to an integer from a qubit type using `int` and displayed as a string.

```
    # Step 3: Measure all the qubits and output result
    All(Measure) | qubits
    Measure | ancilla
    engine.flush()

    res = ''.join(str(int(qubit)) for qubit in qubits)
```

An example of the input and output is shown below. Up until around 16 qubits, the code takes less than a minute to run. Beyond that, not only do we have more qubits to simulate, but we also have to apply more gates as the number of iterations required increases. For example, trying to find a 20-bit string takes around 30 minutes.

```
Input a bit string to find: 10101
Number of iterations T = 4
Element found = 10101
```

7.1.4 Grover's Algorithm with Q#

We start Grover's algorithm as any other Q# program, by declaring the namespace we are working in, `Quantum.Grover` and open the namespaces we will be using.

```
namespace Quantum.Grover{

  open Microsoft.Quantum.Primitive;
  open Microsoft.Quantum.Canon;
  open Microsoft.Quantum.Extensions.Math; // to use maths functions
  open Microsoft.Quantum.Extensions.Convert; // to allow conversions between types
}
```

Programming quantum algorithms

When implementing a quantum program, an important step is to check the individual gates that are required for the program and ensure you know how to program them. In the case of Grover's algorithm we need various gates: first of all we need the Hadamard (H) gate, which is a primitive gate and provided by Q#. The other two main gates are more complex. U_f, the phase oracle dependent on the function $f(x)$ which marks the element we search for with Grover's and the gate $D = H^{\otimes n} U_0 H^{\otimes n}$ are not standard gates in Q# and will need to be coded up. Both operations require phase oracles. One can implement these using a bit oracle and an additional qubit, the ancilla qubit.

Let us have a look at the operation U_f, the general phase oracle. The operation takes three inputs: the main register of qubits `reg`, the ancilla qubit `ancilla` and the marked element in this search in a bit string equivalent representation `markedElement`. To implement the phase oracle with a bit oracle we then simply apply a NOT (X) gate on the ancilla controlled on the main register being in the marked state.

Q# has a build in mechanism to allow controlled operations. Using the syntax `(Controlled Operation)([ctrl], (arg1, arg2, ...))` one can ensure that operation `Operation` is controlled by all qubits in `Qubit[] ctrl` and receives the required arguments `arg1`, etc. However, the controlled operation is always selected on the $|11...1\rangle$ state, which is why we flip all states in the register where the bit string of the marked element is zero.

To ensure that we exit the operation having only changed the ancilla qubit, we have to flip those qubits in the main register back that we flipped before the controlled NOT on the ancilla.

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 with `controlled auto`, as all basic operation are controllable, too, and even the controlled inverted equivalent with `controlled adjoint auto`. This allows the operation to be used in four different ways with writing just one implementation.

```
67  operation PhaseOracle (reg: Qubit[], ancilla: Qubit[], markedElement: Int[]) : () {
68
69    body{
70
71      // Apply X gate to all elements which are zero in the bit string
72      for (i in 0..Length(markedElement)-1){
73        if (markedElement[i] == 0){
74          X(reg[i]);
75        }
76      }
77
78      // X gate on ancilla controlled on main register
79      (Controlled X)(reg, (ancilla[0]));
80
81      // Reapply X gate to all elements which are zero in the bit string
82      // to return to original state.
83      for (i in 0..Length(markedElement)-1){
84        if (markedElement[i] == 0){
85          X(reg[i]);
86        }
87      }
88    }
```

```
89
90      adjoint auto
91      controlled auto
92      controlled adjoint auto
93    }
```

Next we implement the D operation. We start with applying the Hadamard to every bit in the main register, using the library function `ApplyToEachCA`. This applies a one qubit operation to every qubit in the given register. The "CA" at the end indicates that this function is the controllable and adjointable version, so we can make the operation `D` controllable and adjointable again.

Then we perform U_0, in this case a controlled NOT (X) gate on the ancilla selecting on the $|00..0\rangle$ state by flipping all qubits in the main register. After flipping all these back, we perform the Hadamard again on all qubits as the last step of D.

```
95    operation D (reg: Qubit[], ancilla: Qubit[]) : (){
96
97      body{
98
99        // Apply Hadamard on all qubits in main register
100       ApplyToEachCA(H, reg);
101
102       // Apply U_0
103       ApplyToEachCA(X, reg);
104       (Controlled X)(reg, (ancilla[0]));
105       ApplyToEachCA(X, reg);
106
107       // Apply Hadamard on all qubits in main register
108       ApplyToEachCA(H, reg);
109     }
110
111     adjoint auto
112     controlled auto
113     adjoint controlled auto
114   }
```

Now we have all the elements we need to implement Grover's algorithm. We start by calculating the length of the qubit register we will need and creating a random number that will represent the marked element and the integer array form that our U_f requires.

The bit string representation is in so-called little-endian form. 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". Q# has various operations which use either of these orderings, which will be important in the implementation of Shor's algorithm.

```
8   operation Grover (size: Int) : (Int, Int) {
9
10    body{
11      // "size" is the number of elements in the database. "regLen" is the
12      // number of (qu)bits required to be able to represent all of them.
13      let regLen =   Ceiling(Log(ToDouble(size))/Log(ToDouble(2)));
```

```
14
15      // Choose a value at random for the marked element and create an
16      // Int array representing the equivalent bit string (Little Endian).
17      let markedVal = RandomInt(size);
18      mutable markedValBits = new Int[regLen];
19      mutable workingVal = markedVal;
20      for (i in regLen-1..-1..0){
21        set markedValBits[i] = workingVal/(2^(i));
22        set workingVal = workingVal%(2^(i));
23      }
```

Next we allocate the qubits we will use with a `using` statement, where we allocate the main register and the ancilla together. Next we have to split these into two arrays, the main register `reg` and the ancilla qubit `ancilla`. On a side note: the variable `outcome` will hold the result of the quantum algorithm. It needs to be declared outside the `using` block, as all variable declared inside the block are local and the return statement, which must be outside the block, otherwise cannot access it.

```
28      mutable outcome = 0;
29      using (qubits = Qubit[regLen+1]){
30
31        // Separate main register and ancilla qubit
32        mutable indexReg = new Int[regLen];
33        for (i in 0..regLen-1){
34          set indexReg[i] = i;
35        }
36        let reg = Subarray(indexReg, qubits);
37        mutable indexAncilla = new Int[1];
38        set indexAncilla[0] = regLen;
39        let ancilla = Subarray(indexAncilla,qubits);
```

Now we can start implementing the algorithm. We apply the Hadamard to all qubits in the main register and set the ancilla qubit to the required $|-\rangle$ state. Note that we use a different version of the `ApplyToEach` operation here. As we are in the main operation, which also contains measurements, we do not have to worry about the operation being adjointable or controllable.

Next we apply the U_f and D operations $T = \frac{\pi}{4}\sqrt{N}$ times to get to the required state and finally we measure the main register to find the marked element with high probability.

```
41        // Apply Hadamard to each qubit in register
42        ApplyToEach(H, reg);
43
44        // Set ancilla qubit to |-> state
45        X(ancilla[0]);
46        H(ancilla[0]);
47
48        // Loop U_f and D the required number of times
49        let repetitions = Round(PI()/4.0*Sqrt(ToDouble(size)));
50        for (i in 1..repetitions){
51          PhaseOracle(reg, ancilla, markedValBits);
52          D (reg, ancilla);
53        }
54
55        // Measure final state of register
56        set outcome = MeasureInteger(LittleEndian(reg));
```

The last steps of the operation are cleaning the qubits with the `ResetAll` operation, i.e. set them all back to the $|0\rangle$ state, and releasing them by exiting the `using` block. Cleaning qubits is good practise and should always be done before releasing the qubits you used. Finally, we return both the measured outcome of the quantum algorithm and the marked element we randomly chose in the beginning to allow the user to check if the algorithm worked.

```
59          ApplyToEach(SetZero, qubits);
60      }
61
62      return (outcome, markedVal);
```

You may have noticed, that though we implemented the operations `PhaseOracle` and `D` to allow adjoint and controlled use, we never actually used them this way in Grover's algorithm. We find that it is generally good practise to implement operations such that they could be adjointable and controllable, as this allows for better reuse of code, while requiring very little effort due to the automated creation of the adjoint and controlled versions of the operation with just a few keywords.

7.2 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. We have not always implemented the most efficient version of Shor's algorithm, but tried to do the version that is most easily understood. The algorithm is discussed in more detail in section 5.5.

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.

7.2.1 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 subsection 4.1.1. We can also take advantage of an implementation of the Quantum Fourier Transform (QFT) in Grove.

```
1 import numpy as np
2 from fractions import Fraction, gcd
3
4 from pyquil.quil import Program
5 from pyquil.api import QVMConnection
6 from pyquil.gates import H
7 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 $f(x) = a^x \bmod N$.

```
    # Step 1: Chose 1 < a < N uniformly at random
    a = np.random.randint(2,N-1)
    a = 2
    # Step 2: Compute b = gcd(a,N). If b > 1, output b and stop
    b = gcd(a,N)
    if b > 1:
        print('Factor found by guessing:', b)
        return b
```

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
```

Following the circuit diagram in Fig. 5.7, we apply Hadamards to all the qubits in the first register - this is equivalent to applying the QFT to the all zero state but more computationally efficient.

```
    # Apply QFT to first register
    for i in reg_1:
        p.inst(H(i))
```

A more difficult task is creating the bit oracle $O_f |x\rangle |y\rangle = |x\rangle |y \oplus f(x)\rangle$ where $|x\rangle$ is the first register and $|y\rangle$ the second register. 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.

```
# Define matrix for bit oracle O|x>|y> = |x>|y+f(x)>
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):
            row = x + (2**m)*y
            col = x + (2**m)*(y^f)
            oracle_matrix[row][col] = 1
    # Return the matrix
    return oracle_matrix
```

It is important here to get the indexing of the matrix correct. pyQuil does not allow you to create separate qubit registers - we have artificially created two registers by labelling which qubits we have in each of the registers. Since pyQuil enumerates bitstrings such that qubit 0 represents the least significant bit, the first register is the least significant and converting between binary and decimal is done as normal. However, to access the decimal values of the second register, we must multiply by 2^m. This is why the rows and columns have been labelled in this way.

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)
```

Applying the QFT to the first register is now simple - we can use the one coded in Grove.

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

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 - are classical and there are useful Python functions that help us to do this.

```
    # Use continued fraction expansion of z = y/M to find r
    r = Fraction(y,M).limit_denominator(N).denominator

    # If r is odd the algorithm fails
    if r % 2 == 1:
        print('Order r found is odd: algorithm failed')
        return 0

    # Step 4: Find factor of N
    s = gcd(a**(r/2)-1, N)
    if s == 1 or s == N:
        print('Factor found is 1 or', N,': algorithm failed')
        return N

    print('Factor found by Shor\'s algorithm is', s, 'using', m+n, 'qubits')
    return s
```

The smallest number that we can run Shor's algorithm with is $N = 15$, this requires 12 qubits which is well within the 26 that can be simulated by Rigett's QVM. Although we can view the Quil code (the gates we apply), this code does not actually run. We get a time-out error connecting to Rigetti's server. This is most likely due to the construction of the oracle as a matrix using `defgate`, for 12 qubits it is a 4096 x 4096 matrix.

7.2.2 Shor's algorithm with QISKit

The structure of QISKit prevents directly constructing an operator that performs the period finding algorithm in the same format that we programmed Grover's search algorithm. At the time of writing QISKit lacks the functionality to decompose a given unitary operation into a sequence of allowed operations. The problem of finding an efficient circuit in terms of a set of logic gates (e.g. Hadamard, Phase and CNOT) that implements a given unitary is an active area of interest. This is known as gate synthesis. The best known circuit for Shor's algorithm requires $2n+3$ qubits and $O(n^3 \log(n))$ logical operations to factor an n bit number[77]. Implementing this from first principles in beyond the scope of this guide, although we encourage the dedicated reader to try it for themselves.

7.2.3 Shor's algorithm with ProjectQ

We can follow the same steps as for pyQuil since they are both 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, BasicMathGate, QFT, H, Swap
```

After checking that we can simulate the number of qubits required, 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)
```

We can apply Hadamards to all the qubits in the first register using the function `All`.

```
    # Apply QFT to first register
    All(H) | 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
```

Now the QFT needs to be applied to the first register. ProjectQ has an inbuilt function for this called `QFT`, but we need to be careful using it since it does not do the final Swap gates required for the QFT circuit as shown in Fig. 5.6.

```
    # Apply the QFT to the first register
    QFT | reg_1
    for i in range(int(m/2)):
        Swap | (reg_1[i], reg_1[m-i-1])
```

After 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.

```
    # 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 and we can now test it. This code runs fairly fast considering the number of qubits it has to simulate, for example two digit numbers can be factored in less than 30 seconds. Of course this does not take into account the times that the algorithm will fail, so often the code has to be run a number of times. Below is an example of one of the largest numbers that can be factorised using this code, this took around 20 minutes to run.

```
Input an integer to factorise: 405
Factor found by Shor's algorithm is 27 using 27 qubits
```

7.2.4 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 [77]. In figure 7.1 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.

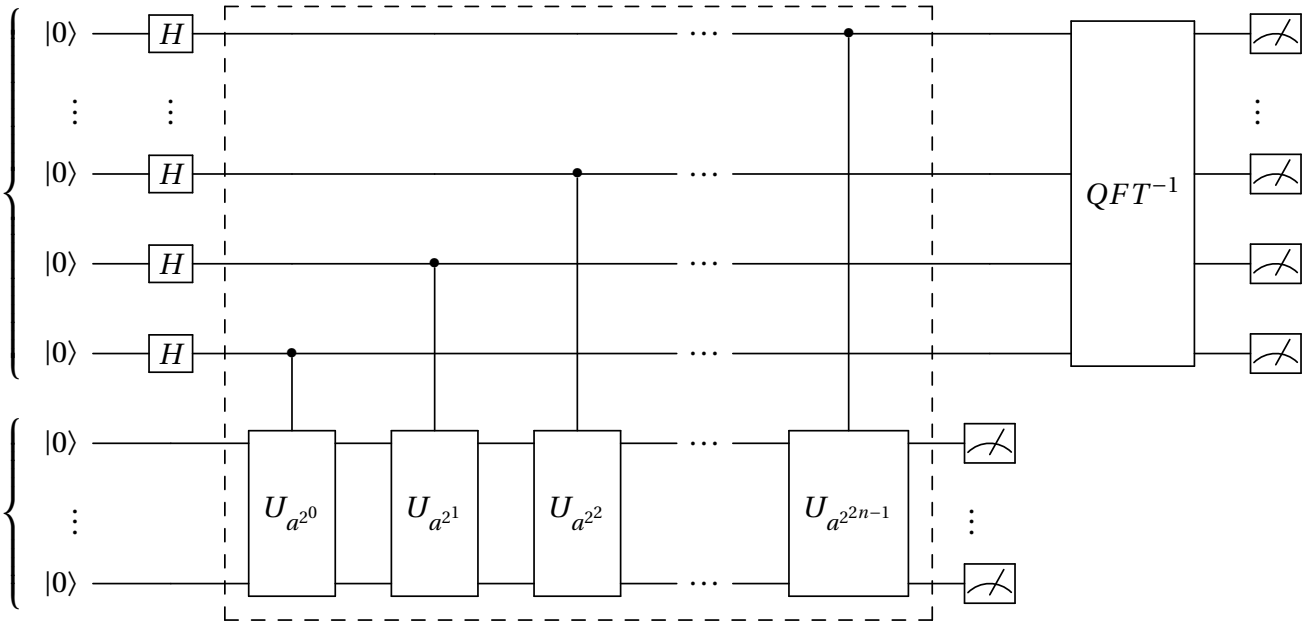


Figure 7.1: Circuit for approximate periodicity finding in Shor's algorithm. The operation U_i is given by the map $|x\rangle \rightarrow |(xi) mod N\rangle$. Together, the operations in the dashed box implement the bit oracle U_f with $f(x) = (a^x) mod N$. Adapted from [77].

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; // 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 imple-

mentation 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 implemented, too, but is omitted here. It can be found in the complete code for this implementation given in the **??**.

```
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 7.1. 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. 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 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.

```
operation ModularQubitMultiplyByExp
      (stateIn: LittleEndian, expBase: Int, power: Int, modulus: Int):(){

  body{
    ModularMultiplyByConstantLE(ExpMod(expBase, power, modulus), modulus, stateIn);
  }

  adjoint auto
  controlled auto
  controlled adjoint auto
}
```

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.

```
        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.

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

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.

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

Now we have to apply the inverse Quantum Fourier Transform. The Q# library contains a Quantum Fourier Transform operation, where we have to select the one with the right endianness. In this case we need the one with big endian ordering, `QFT`. We can use the `Adjoint` keyword to get the inverse operation.

```
        (Adjoint QFT)(BigEndian(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.

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

The last step is "cleaning" the qubits, i.e. resetting them to the $|0\rangle$ state before releasing them when exiting the `using` block.

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

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

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

7.3 TODO Summary and Outlook

Won't someone write me?

Comparing the implementations of these algorithms with the Q software available?

comments? comparison?

7.4 Exercises

Exercise 1: Experimenting with the number of iterations

For a number of qubits and marked string of your choice, experiment with how changing the number of iterations T changes how likely the algorithm is to succeed.
Plot a graph of T versus the probability of success by running Grover's 100 times for each T.

Exercise 2: Grover's algorithm for multiple marked elements

Grover's algorithm can be generalised to multiple marked elements. Now instead of flipping the phase of one state, the phase of all the marked states must be flipped - this is done by applying the phase oracle U_f for each marked element. Another difference is that the optimal number of iterations is

$$T \approx \frac{\pi}{4}\sqrt{\frac{N}{M}} - \frac{1}{2}$$

where M is the number of marked elements.
Code this algorithm in a language of your choice.

Exercise 3: Deutsch-Josza Algorithm

The Deutsch-Josza algorithm distinguishes between a constant and a balanced function, as explained in section 5.3. This exercise will walk you through implementing it in a language of your choice.

a) **Constructing U_f** U_f is a representation of the function $f(x)$. For example for two qubits if $f(00) = 1$, U_f takes $|00\rangle$ to $-|00\rangle$ and if $f(00) = 0$ we do nothing. Flipping the phase is done in the same way as in Grover's algorithm.
Construct U_f to flip the phase on the associated states $|x\rangle$ when $f(x) = 1$.

b) Now code the rest of the algorithm as described in section 5.3 using the U_f constructed above.

Chapter 8

Implementations

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

8.1 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 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, due to the lack of large scale quantum computers, complex software stacks and quantum development environments 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.

The section is arranged as follows. We begin with a historical overview of classical computers, and a brief description of classical architectures. We then contrast this with the multitude of current approaches to quantum computation. We look at quantum computer hardware and the physical systems which represent qubits and implement quantum operations.

We then discuss how the languages we've included in the guide can be turned into programs that run on quantum hardware. We conclude by comparing this process with the steps needed to control a classical computer.

8.1.1 Classical hardware

In the past there have been many different approaches to classical computation. Some devices such as the abacus or slide rule are simple mechanical devices that are operated manually, but which speed

up certain operations (e.g. arithmetic). Charles Babbage's difference engine uses a complicated system of cogs and gears to perform more general purpose operations. In an analog computer, voltages and currents are used to represent variables. Analog computers can solve differential equations and model physical systems.

During the middle and latter half of the 20^{th} century, there was a substantial move to replace all these methods with general purpose computers based on digital electronics. Digital electronics uses two voltage levels, for example 0 V and 5 V, to encode a bit. Bits can be manipulated using digital logic circuits. One of the first digital computers was the Electronic Numerical Integrator and Computer (ENIAC), which was completed in 1945. ENIAC was re-programmable and Turing-complete. However, the thousands of valves required made it bulky and expensive.

The large scale nature of these systems made it necessary to think carefully about the architecture of large scale computers. The Electronic Discrete Variable Automatic Computer (EDVAC), which followed the ENIAC, prompted Von Neumann to write a document outlining the high level operation of general operation of digital computers [78]. His proposed architecture for a digital computer became known as the Von Neumann architecture. It outlines the type of resouces that a classical computer would need, and how they would be connected together.

Following the invention of the transistor in the late 1940s, it became much easier to realise basic operations such as AND and OR gates, which could be used as the basis for digital computers. Transistors are much smaller, more reliable and more robust that valves. However, it was the invention of the integrated circuit in the 1950s which really kick started the digital computing revolution [79]. Integrated circuits make it possible to squash electronics onto silicon chips, which dramatically reduces both the size of the resulting electronics and the time required to assemble circuits, which would previously have been hand soldered. The first computer to use integrated circuits was the IBM 360 in 1964 [80].

These computers were bulky and expensive, comprising of a large mainframe, and small input output devices called terminals, which have a keyboard and a screen that allow users to interact with the mainframe. However, as discussed in Chapter 2, computers have become smaller and smaller, culminating in laptops, tablets and smartphones which can outperform the mainframes of the 60s.

As a result of all this success, microelectronics is the only classical computing 'platform'. Normally, microprocessors share the following common features:

- **Memory**: processors require blocks of internal or external random access memory (**RAM**) which can store data and programs.
- **Working registers**: these are used for performing operations on data, e.g. the processor moves data to the working registers to perform an operation such as addition
- **Arithmetic and logic unit (ALU)** and **floating point unit (FPU)**: for performing logical and mathematical operations.
- **Control**: this manages all the resources of the processor. The control unit is responsible for reading, decoding and executing instructions. Instructions are the simplest blocks of processing a classical computer can do

- **Buses**: shared data lines which can be used to move data between different parts of the processor.

- **Input/output**: for interacting with external devices. For example, a modern processor in a laptop will connect to external memory (e.g. more RAM, a hard drive), dedicated graphics hardware, a keyboard and display, etc.

The classical architecture of a microprocessor is the precise manner in which the above building blocks are implemented and connected together. Fig. 8.1 shows the Harvard architecture, where the memory for storing data is different to the memory used for storing the program. The von Neumann architecture (Fig. 8.2) discussed above uses the same physical memory for both the data and the program. Other architectural decisions include the width of the memory (e.g. 32 bit or 64 bit), the size of the instruction set (Intel chips for computers use large instruction sets; ARM devices for embedded applications have smaller instruction sets), and input/output capabilities (Intel chips might interact with graphics cards, whereas a Microchip microcontrollers might have a motor driver for use in automation applications.)

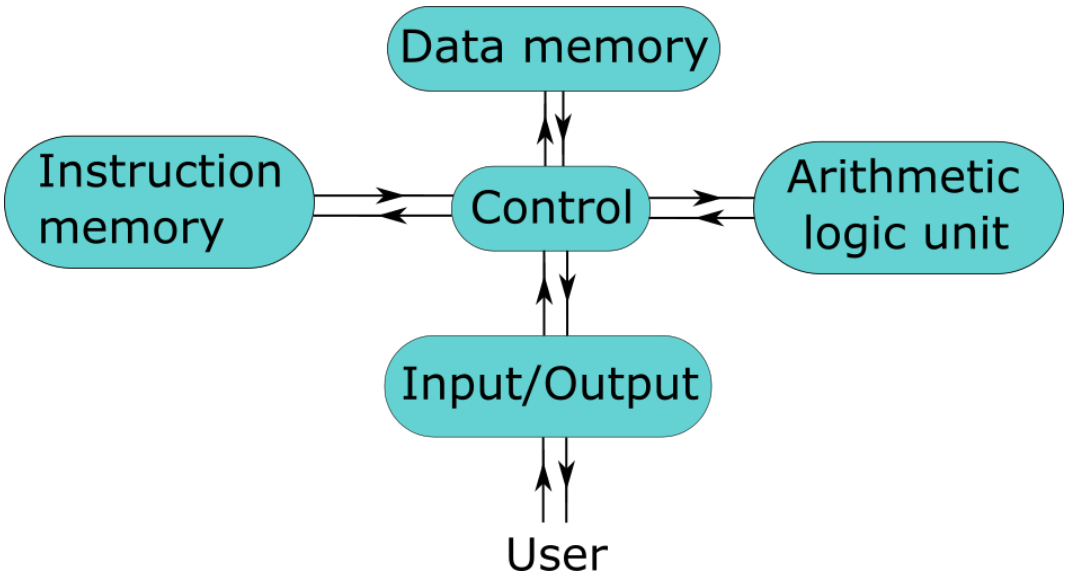


Figure 8.1: Harvard Architecture

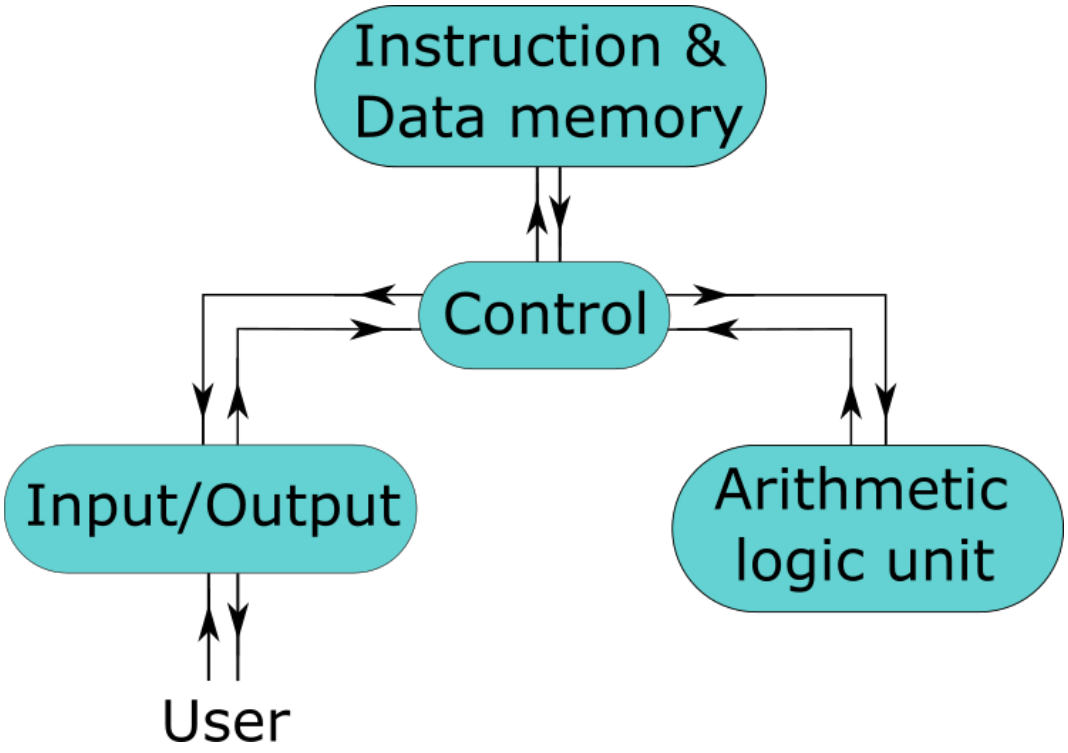


Figure 8.2: Von Neuman

8.1.2 Quantum hardware

In contrast with classical computers, there are comparatively few components in a quantum computer:

- **Qubits**: fundamental units of quantum information which can be in the state 0 or 1 like bits, but can also be in a superposition state. It must be possible to initialise a qubit in at least the zero state.
- **Quantum gates**: these are quantum operations which change the state of qubits. Typically gates which act on one or two qubits are considered most important, because it is possible to build up other more general gates from just these types
- **Measurements**: quantum computers must contain a system for making measurements on qubits – these always return either 0 or 1 and allow useful information to be extracted from the computer

There are also speculative requirements such as quantum memory that may eventually be needed. It is unclear at this stage exactly what components a large scale quantum computer will contain.

More stuff here plz.

There is currently no large scale quantum computer.

Cloud based quantum computing

Cloud based quantum computing is likely to remain the most realistic method of getting access to a quantum computer in the near future. Just like the early years of digital computing, access to a computer was only possible through the introduction of mainframes and terminals, due to the huge cost and space needed for current small scale quantum devices this is the most feasible approach. However, with access to the internet the 'mainframe computer' can be accessed from all over the world.

pyQuil, Qiskit and Project Q are all cloud based quantum libraries, with the appropriate access instructions for each given in Chapter 4, code can be ran on the companies devices. Executing code on the quantum devices costs credits which you can apply for on the IBM Qiskit website.

Like all of the languages discussed in this guide they use the approach of focusing on short-term quantum devices and adopt the dedicated quantum processor and classical processor structure.

8.2 Quantum platforms

The over-arching physical realisation of the components for quantum computers can be separated into two parts: the architecture and the platform. The platform is the physical system that is used to build the computer. Examples include photonics, trapped ions, semi-conductor qubits, and superconducting qubits.

Quantum architecture has a more restricted meaning than in the classical case, due to the lack of large scale structure in a quantum computer. It refers to the manner in which the physical platform is used to realise qubits and gates. This is analogous to the different ways transistors can be used to create logic gates, for example transistor-transistor logic (TTL) or complementary metal oxide semiconductor (CMOS). As a result, the quantum architectures and platforms are not completely independent; 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 photonics is 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 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 main difficulty is just trying to control it; most of the resources are used are for error correction, keeping the whole thing coherent. The remaining small part of the machine is for the information processing.

8.2.1 Trapped Ions

Trapped ions are a relatively stable physical system, which is a desirable feature for a quantum computer. Electric fields are used to trap the ions and individual ions can be shuttled around by changing the voltage in the direction you want the ion to move. 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. Microwave gates are appealing as it opens the possibility of addressing multiple qubits at once.

The [81] blueprint is a scheme which uses a modular approach of packing tiles shown in Fig. 8.3 on a 2D surface. Each tile contains the following: one ion, a dedicated loading zone, an entangling zone which interlinks ions on adjacent tiles and a detection zone for readout. Entangling gates or operations are performed by bringing two qubits close together.

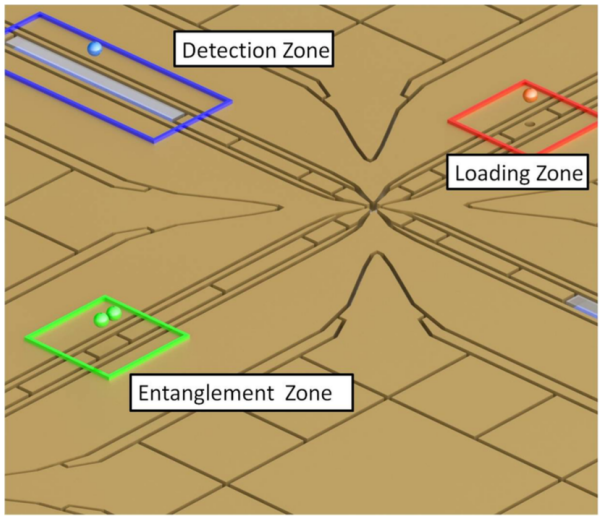


Figure 8.3: An X-junction ion trap array with different trapping regions for entanglement, loading and detection, taken from [81].

8.2.2 Superconducting qubits

Superconducting qubits (also called flux or charge qubits) are currently being worked on the most by private companies. The platform requires very cold temperatures in order to reach the superconducting regime. This will be problematic for future larger devices which will require larger and more expensive cryostats. Eventually we wont be able to fit the device inside a single cryostat placing a limit on the capacity of a single chip. It is generally believed within the field that cryostats will have to be interconnected for large scale computing which presents its own technical challenges. The cryogenic temperatures also pose a problem for control electronics[1]. A potential benefit to overcoming these engineering challenges is that the platform is very quick which is indicative of a high clock speed in future devices. Fig. 8.4 is a 2 qubit superconducting chip, the gold wires around the device are for the control electronics for performing gates and measurement readout.

[1] It is not well understood how traditional electronics behaves at cryogenic temperatures

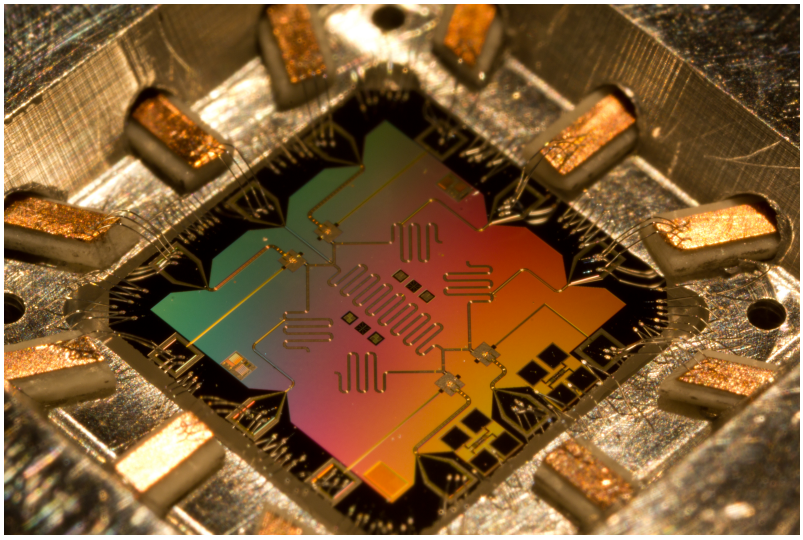


Figure 8.4: A superconducting chip which contains 2 qubits [82] *photo by Erik Lucero* [2]

8.2.3 Linear optical quantum computing

Linear optical quantum computing uses photons and waveguides (commonly referred to as modes) as the qubit. Using light for information processing seems like a natural choice as it is the fastest thing in the universe. Unlike trapped ions and superconducting qubits where their main problem is stopping the qubits interacting with the environment, photons interact very weakly with their environment. This is a positive as we then only have to worry about losing the photons rather than it decohering to a classical object. However, it also means that the two-qubit entangling gates are difficult to perform as photons not only interact weakly with the environment but also do not interact with each other.

Gates are currently performed using electrical heaters which heat locally a small section of the waveguide which in turn changes the phase of the photon[3]. Fig. 8.5 is a fully re-configurable 3 qubit integrated optical chip meaning it can perform any 3 qubit operation. The calculation is performed by using the electrically controlled phase-shifters and interference between the photons to change the probability distribution of which waveguides the photons exit the chip from. The answer to the computation is given by the which waveguides the photons exit.

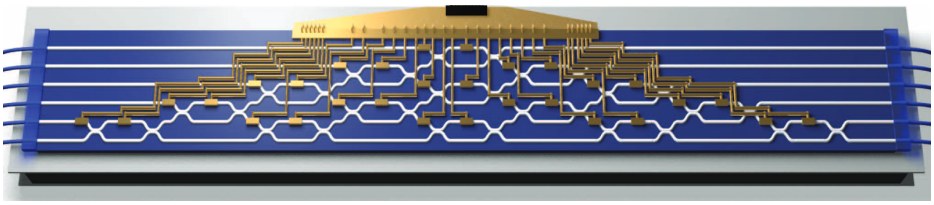


Figure 8.5: Caption [83]

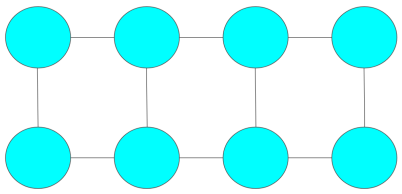
[2] http://web.physics.ucsb.edu/~martinisgroup/

[3] You can think of the phase of the photon in the same way as the phase of the electric field

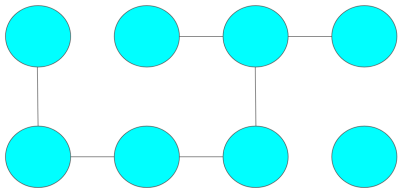
8.3 Physical gate-sets and connectivity restrictions

Irrespective of the physical platform used to build the quantum computer we can describe the device [4] with just the gate-set and the connectivity of the device. For example the different platforms above will naturally have access to different gates. In the linear optical platform it turns out to be much easier to perform a Z gate than an X gate. Whereas we could imagine a different system where the opposite is true.

Connectivity describes which qubits are able to interact with each other[5]. Ideally to have the most flexible and re-configurable quantum computer we would like to have full connectivity meaning all qubits are able to interact with every other. There are a number of physical implementation reasons why having many qubits connected is hard but we will not cover them here. Instead we highlight the following important fact, for a device to be universal only nearest neighbour connections are needed Fig. 8.6a. Connectivity is illustrated in Fig. 8.6. Currently, none of the devices built have achieved complete nearest-neighbour connectivity. A more likely connectivity is depicted in Fig. 8.6b. This is due to imperfect fabrication of the devices.



(a) Full lattice type nearest-neighbour connected qubits



(b) Partially connected qubits

Figure 8.6: An abstraction of 8 qubit device where the lines illustrate 'connections' between qubits.

In the same way that different platforms may be suited to different gate-sets, they may also be suited to different connectivities. For example, the trapped ion platform naturally has nearest neighbour connectivity as a direct result of the tiling scheme. Compare this to LOQC or superconducting qubits where there is no intuitive design for high connectivity due to the restriction that each qubit must be connected to control electronics. Therefore, as the connectivity scales up, so does the density of control lines required, presenting serious electrical engineering challenges.

[4] Assuming it is perfect and error-free

[5] There isn't really an analogy with digital computing as the architectures are so complex we don't worry about which memory addresses are connected to each other, this is the job of the linker

Equivalence between different platforms

Given this information we would like to be able to run a quantum program on any of the equivalent quantum devices (e.g. with the same gate-sets and connectivity).

We can make a list of possible scenarios for different implementations of two quantum computers:

Gate-set	Connectivity	Fix
Same	Same	Depends on what the language you have chosen to use returns, easy if both devices use the same quantum Assembly language [7]
Same	Different	Use a linker to remap the qubits on the devices
Different	Same	Use a compiler to perform gate synthesis
Different	Different	What classical digital compilers and linkers do

Table 8.1: Required software features to convert between different quantum devices

8.3.1 Minimal quantum program examples

We have chosen a minimal program which uses only classical physics, we take the binary representation of the letter 'Q' [8] and write the value to 8 qubits which represent a qubyte. We then measure the qubits to recover the string. Even for this simple example program we can see some differences between what goes on behind the scenes when using the built in compilers[9] and we comment on the quantum instruction set languages used.

Like the other quantum programs above the majority of the quantum hello world program, **??** is classical control. A small part of the algorithm uses the quantum computer. This example is novel in that there are no quantum effects used here, however it is more to illustrate the current progress of existing devices. A nice feature we have access to in the quantum case is that we can use either the bit value or phase to store information, instead of X gates an equivalent program could be written which uses H and Z gates.

?? shows the python input to the quantum library which passes in a bit string, the quantum simulator (or processor) is then run for each byte in the bitstring as this only requires 8 qubits. The results of each computation (measurement results) are stored in a python list and then when all of the bytes have been run on the quantum processor the bitstring is translated back into ASCII and printed using python.

We are using this example to show that quantum computers are able to do tasks that classical computers can do, a requirement if we are ever to think of building fully quantum machines (not using the co-processor model).

[7] We use Assembly here to denote the lowest level language which consists of listing instructions and which qubits the instructions act on

[8] There is nothing special about Q other than the unwritten rule of including q's when naming anything quantum related

[9] A better description would be linker but we will get to that

Even for this simple example there are a number of interesting features that all of the libraries discussed here feature:

- The majority of the algorithm is classical classical control
- The simulation has to be run many times to build up a an output distribution
- We are currently very limited by existing devices

Translating pyQuil to quantum hardware

To use pyQuil on a quantum processor, the source code needs to be be translated into Quil (Quantum Instruction Language) which lists the gates which must be applied to the qubits in the processor. Acts as an intermediate step between pyQuil and the actual instructions that will go to the quantum computer. The Quil compiler [10][11] restricts the quantum gates to the quantum devices gate-set and looks at the connectivity of the machine to make sure (2-qubit) control gates are acting on qubits that are physically connected.

We will give an example of the compilation process in pyQuil using a simple Hello World program, shown below in **??**. We specify the 8 qubit Agave device architecture which then maps the Quill code written to the device gate-set and connectivity.

Translating QISKIT (Quantum Information Science KIT) to hardware

The Qiskit library has to translate the python code into IBM's Open quantum assembly (OQASM) to be run on their devices. IBM has multiple quantum devices (3, 4, 5 and 16 qubit devices) that you can run quantum programs on. Qiskit also comes with a local simulator, similar to pyQuil which can simulate up to 32 qubits. An API key is needed to access the quantum devices but not the local simulator.

Here we give an example of the Qiskit python library code and the OQASM output [12] . We tried compiling the OQASM to a device specific arcitecture but couldn't get the program to reduce or simplify the gates.

Translating Project Q to hardware

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.

---

[10] http://docs.rigetti.com/en/stable/compiler.html

[11] We wouldn't strictly call this a compiler in the usual sense

[12] the user can write QASM embedded in Python and Qiskit will return the program as QASM [13]

[13] There is no spelling mistake here...

Project Q is also the only library discussed here that comes with a local simulator. You can also a specify machine architecture for IBM devices using the IBM compatible compiler [84] which will compile the project Q code into the gate-set and connectivity of the specified machine.

Translating Q# to hardware

It is not currently possible to convert code written in Q# to hardware. This is because there is only a simulator and no way to obtain the list of gates that are being simulated. We have therefore not included a minimal example of compiling Q# code to hardware level instructions.

Q# does however support a resource counter which estimates the resources required to run the quantum program on hardware. We give a minimimal example showing

```python
#!/usr/bin/env python3
from pyquil.quil import Program
from pyquil.api import QVMConnection
from pyquil.gates import X, Z, H
from compile_for_pyquil import
    ↪ compiletoquil

# Q in binary
bitstring = '01010001'

# Do quantum stuff now we have our bit
    ↪ string
qvm = QVMConnection()
qprog = Program()

# do X on q1, q3, q7
# remember HZH is X
qprog.inst(H(1), Z(1), H(1))
qprog.inst(X(3))
qprog.inst(X(7))

# do measurement over all 8 qubits
for i in range(0, 8):
    qprog.measure(i, i)

# store measurement outcomes
results = qvm.run(qprog)

# show info when compiled to 8 qubit
    ↪ AGAVE
compiletoquil(qprog)

print('# Result =', results[0])
```

Listing 8.1: Pyquil Hello Q program

```
# Original pyQuil program,

 H 1
Z 1
H 1
X 3
X 7
MEASURE 0 [0]
MEASURE 1 [1]
MEASURE 2 [2]
MEASURE 3 [3]
MEASURE 4 [4]
MEASURE 5 [5]
MEASURE 6 [6]
MEASURE 7 [7]
```

Listing 8.2: Quil code from pyQuil code

```
# Compiled quil code,

 PRAGMA EXPECTED_REWIRING "#(0 1 2 3 4 5
    ↪ 6 7)"
RX(-pi) 1
RX(-pi) 3
RX(-pi) 7
PRAGMA CURRENT_REWIRING "#(0 1 2 3 4 5 6
    ↪ 7)"
PRAGMA EXPECTED_REWIRING "#(0 1 2 3 4 5 6
    ↪ 7)"
MEASURE 0 [0]
MEASURE 1 [1]
MEASURE 2 [2]
MEASURE 3 [3]
MEASURE 4 [4]
MEASURE 5 [5]
MEASURE 6 [6]
MEASURE 7 [7]
PRAGMA CURRENT_REWIRING "#(0 1 2 3 4 5 6
    ↪ 7)"

# Result = [0, 1, 0, 1, 0, 0, 0, 1]
```

Listing 8.3: Compiled code using the 8 qubit AGAVE devce architecture

```
#!/usr/bin/env python3
from qiskit import ClassicalRegister,
 ↪  QuantumRegister, QuantumCircuit
from qiskit import available_backends, execute,
 ↪  compile
#from qiskit.wrapper._wrapper import *
from IBMQuantumExperience import
 ↪  IBMQuantumExperience
from qiskit import backends

# Q in binary
bitstring = '01010001'

# Allocate memory
q = QuantumRegister(8)
c = ClassicalRegister(8)
qprog = QuantumCircuit(q, c)

# do X on q1, q3, q7
# recall H Z H is X
qprog.h(q[1])
qprog.z(q[1])
qprog.h(q[1])

qprog.x(q[3])
qprog.x(q[7])

# Measure
qprog.measure(q, c)

gates = "x,h"
cp =
 ↪  compile(qprog,"local_qasm_simulator",basis_gates=gates)
print(cp['circuits'][0]['compiled_circuit_qasm'])

# print quantum assembly code
print(qprog.qasm())

# Submit the job to the Q QASM Simulator
job_sim = execute(qprog, "local_qasm_simulator")

# Fetch result
sim_result = job_sim.result()

# Print out the simulation measurements
print("# Result =", sim_result)
print(sim_result.get_counts(qprog))
```

Listing 8.4: Qiskit hello world

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q0[8];
creg c0[8];
h q0[1];
x q0[3];
x q0[7];
measure q0[0] -> c0[0];
measure q0[2] -> c0[2];
measure q0[3] -> c0[3];
measure q0[4] -> c0[4];
measure q0[5] -> c0[5];
measure q0[6] -> c0[6];
measure q0[7] -> c0[7];
U(0,0,3.14159265358979)
 ↪  q0[1];
h q0[1];
measure q0[1] -> c0[1];
```

Listing 8.5: OQASM for the qiskit hello world program

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q0[8];
creg c0[8];
h q0[1];
z q0[1];
h q0[1];
x q0[3];
x q0[7];
measure q0[0] -> c0[0];
measure q0[1] -> c0[1];
measure q0[2] -> c0[2];
measure q0[3] -> c0[3];
measure q0[4] -> c0[4];
measure q0[5] -> c0[5];
measure q0[6] -> c0[6];
measure q0[7] -> c0[7];

# Result = COMPLETED
{'10001010': 1024}
```

Listing 8.6: OQASM for the qiskit hello world program

```python
from projectq import MainEngine
from projectq.backends import CommandPrinter, Simulator
from projectq.setups import restrictedgateset
from projectq.ops import X, Z, H, Ry, Rz, CNOT, All,
    Measure

# set restricted gate-set
restricted_list = restrictedgateset.get_engine_list(
    one_qubit_gates=(Rz, Ry), two_qubit_gates=(CNOT, ),
        other_gates=())

# set engine
restricted_compiler = MainEngine(
    backend=CommandPrinter(accept_input=False),
        engine_list=restricted_list)

#  print qasm
printqasm =
    MainEngine(backend=CommandPrinter(accept_input=False))

# Run the simulator
simulate = MainEngine(backend=Simulator())

def qprogram(eng):
    q = eng.allocate_qureg(8)

    # Q in binary: '01010001'
    # recall H Z H is X
    H | q[1]
    Z | q[1]
    H | q[1]

    X | q[3]
    X | q[7]

    # measure all the qubits
    All(Measure) | q
    # execute the quantum program
    eng.flush()
    return q

# run for the compiler to see what the program does
qprogram(printqasm)
# run for restricted gate-set
qprogram(restricted_compiler)
# Simulate and print results
q = qprogram(simulate)

results = []
# the values of qubits is stored as the int value of q
for element in q:
    results.append(int(q[int(element)]))
print(results)
```

```
Allocate | Qureg[0]
Measure | Qureg[0]
Allocate | Qureg[1]
H | Qureg[1]
Z | Qureg[1]
H | Qureg[1]
Measure | Qureg[1]
Allocate | Qureg[2]
Measure | Qureg[2]
Allocate | Qureg[3]
X | Qureg[3]
Measure | Qureg[3]
Allocate | Qureg[4]
Measure | Qureg[4]
Allocate | Qureg[5]
Measure | Qureg[5]
Allocate | Qureg[6]
Measure | Qureg[6]
Allocate | Qureg[7]
X | Qureg[7]
Measure | Qureg[7]
Deallocate | Qureg[7]
Deallocate | Qureg[6]
Deallocate | Qureg[5]
Deallocate | Qureg[4]
Deallocate | Qureg[3]
Deallocate | Qureg[2]
Deallocate | Qureg[1]
Deallocate | Qureg[0]
```

```
Allocate | Qureg[0]
Measure | Qureg[0]
Allocate | Qureg[1]
Rz(3.14159265359) | Qureg[1]
Ry(1.570796326795) | Qureg[1]
Rz(6.28318530718) | Qureg[1]
Ry(1.570796326795) | Qureg[1]
Measure | Qureg[1]
Allocate | Qureg[2]
Measure | Qureg[2]
Allocate | Qureg[3]
Ry(9.424777960769) | Qureg[3]
Rz(3.14159265359) | Qureg[3]
Measure | Qureg[3]
Allocate | Qureg[4]
Measure | Qureg[4]
Allocate | Qureg[5]
Measure | Qureg[5]
Allocate | Qureg[6]
Measure | Qureg[6]
Allocate | Qureg[7]
Ry(9.424777960769) | Qureg[7]
Rz(3.14159265359) | Qureg[7]
Measure | Qureg[7]
Deallocate | Qureg[7]
Deallocate | Qureg[6]
```

Q# example

That's right. There is no example.

8.3.2 Summary

We now compare some of the implementation dependent features of the languages and give a comparison between them.

Qubit management

The pyQuil qubit management is dynamic, qubits are allocated dynamically which does not require the users input. The qiskit library is heavily based on using quantum circuits. You specify quantum and classical registers and ancilla qubits. Project Q also requires the user to manually allocate qubits and the QASM resembles dynamically allocated qubits with allocate and deallocate at the start and end of the instruction set.

Compilers/Gate simplification

All of the libraries have access to the standard gate set in their python environments[14].

In pyQuil the user can also use the `defgate()` method to define their own gate either in terms of compositions of existing gates or by specifying a matrix representation of the gate. Qiskit also can define gates. Project Q has the option to define gates?

The pyQuil compiler takes one of Rigetti's quantum devices as an argument and compiles their Quill QASM into device specific gate-set and remaps the qubits to match the connectivity of the machine.

IBM has no working compiler[15] but we can return the Open QASM (IBM's Quantum Assembly format),

The Project Q compiler is designed to have a modular structure, so the user can pick and choose the level of optimisation required. It works quite well although you have to manually specify the available gate-set you have access to and the connectivity of the machine. They are working on adding in backend support for the IBM quantum devices.

Backends

Rigetti then offers a remote Quantum Virtual Machine (QVM) which runs simulations of up to 26 qubits requiring an API key to use. They do not provide a local simulator which runs on a personal computer. IBM do provide a local simulator and access to their devices and remote simulator require

[14] X,Y,Z,H,Phase gates and arbitrary rotations

[15] there is a compiler but it does nothing, specify x only and the program crashes, specify x and H and the program uses Z. U(x,y,z)(θ).

and API key. Project Q comes with a local simulator, written in c++ and is built on your machine on install using python wrappers.

The project Q simulator is best of the local simulators however currently there is no way to compile project Q code into IBM's OASM to run on IBM's machines or compile to Quil to run on Rigetti's machines. Although IBM and Project Q existed as part of the same project [16] we are unsure of when the project Q to IBM backend will be implemented.

One slightly annoying feature of pyQuil the software package is the lack of local quantum simulator. This can cause problems when trying to execute large programs as we often found the connection timed out before the program finished.

Device mappings

Rigetti's Quil compiler takes a dictionary of one of their devices which then creates a map of the positions of the qubits and the connectivity of the machine.

QISKits' `get remote backends` method doesn't return any valid devices to compile to, when using one of their device names an error saying the device is not valid is produced. It may be possible to manually add device mappings but even then the *compiler* didn't compile to the given gate-set or it would produce an error for some gate-sets.

Project Q lets you specify the connectivity using a Python dictionary and gate-set by importing the gates you want/have access to. The compiler works quite well.

8.4 Comparing classical and quantum compilers

In the previous section we showed how quantum compilers work. They take source code (for example pyQuil) and turn it into a low level set of operations (for example Quil) which can be executed on a quantum device or simulator. The compiler performs certain optimisation operations (for example simplifying HZH to X) but does not otherwise substantially modify the source code.

Here we show, by way of comparison, the compilation steps that turn a classical program (Hello world, written in C) into an executable file. Our aim is to show that there are many more steps in the process, including optimisation and linking, which are not really present in the quantum case. We will discuss in detail what each of these steps are, why they are necessary and how they occur in a classical computer. We will then speculate on which of how quantum compilation may eventually develop as the complexity of quantum programs increases.

8.4.1 Compile, Assemble, link, execute

As shown in Fig. 8.7, there are three steps which turn a classical compiled language such as C into a program which can be executed on a classical computer. We will demonstrate this process in the case

[16] see git change-log

of a simple *Hello World!* program written in C. The program prints the string "Hello World!" and exits, returning 0 to the operating system. This is shown in Listing 8.7.

```c
#include <stdio.h>

int main(){
  printf("Hello World!\n");
  return 0;
}
```

Listing 8.7: Hello world! in C

The compiler takes this C source file as an input, and outputs the assembly language file shown in Listing 8.8. The assembly file looks very different to the original source code, because it contains instructions like movl and call. These instructions performs a basic operations such as moving data around or calling subroutines. There is normally a large degree of optimisation that goes into this step, because the language constructs in C do not map directly to instructions in assembly language.

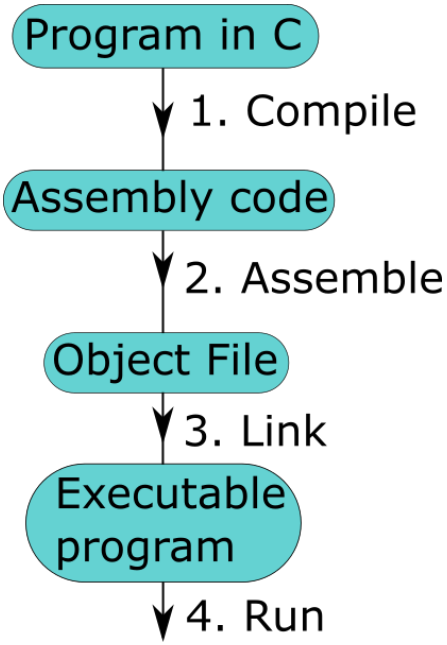


Figure 8.7: Compile, assemble, linking and running structure for running a program

```
main:
.LFB0:
        .cfi_startproc
        pushq   %rbp
        .cfi_def_cfa_offset 16
        .cfi_offset 6, -16
        movq    %rsp, %rbp
        .cfi_def_cfa_register 6
        movl    $.LC0, %edi
        call    puts
        movl    $0, %eax
        popq    %rbp
        .cfi_def_cfa 7, 8
        ret
        .cfi_endproc
```

Listing 8.8: Assembly file after compiling

The assembler then turns the assembly language file into an object file, shown in the left hand part of Listing 8.9. This is quite a simple step – it involves replacing all the instructions with strings of numbers called machine code which is readable by the target computer. The right hand column shows the instructions corresponding to the codes on the left. The critical feature of the object file is the lack of addresses. For example lines 3 and 4 contain placeholders $0x0 and e. These addresses are resolved by the linker, which assigns memory to all the variables. The linked file is shown in

Listing 8.10. Notice how the program itself has also been assigned a memory location – line 1 shows that the first instruction is at address 400526. In larger programs, each source file has its own object file. The linker combines all these files together to produce an executable file which can be run on the target computer.

```
40   0:    55                    push   %rbp
41   1:    48 89 e5              mov    %rsp,%rbp
42   4:    bf 00 00 00 00        mov    $0x0,%edi
43   9:    e8 00 00 00 00        callq  e <main+0xe>
44   e:    b8 00 00 00 00        mov    $0x0,%eax
45  13:    5d                    pop    %rbp
46  14:    c3                    retq
```

Listing 8.9: Object file before linking

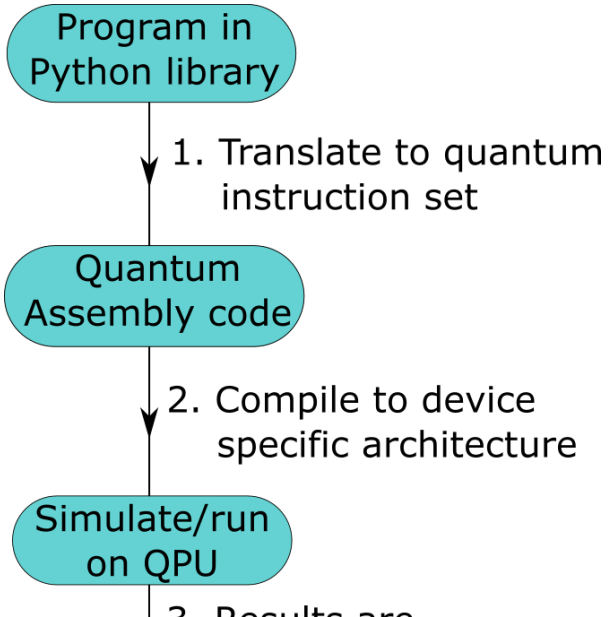
```
137 400526:    55                    push   %rbp
138 400527:    48 89 e5              mov    %rsp,%rbp
139 40052a:    bf c4 05 40 00        mov    $0x4005c4,%edi
140 40052f:    e8 cc fe ff ff        callq  400400 <puts@plt>
141 400534:    b8 00 00 00 00        mov    $0x0,%eax
142 400539:    5d                    pop    %rbp
143 40053a:    c3                    retq
```

Listing 8.10: Executable file after linking

8.4.2 Quantum compilation

Quil, OQASM and Project Q's quantum assembly have laid the ground work for building a quantum programming language. Although we suspect it will be some time before we get a full quantum language and not domain specific embedded languages, the work on quantum instruction sets has a long way to go.

The main feature we would like to highlight is that because Assembly is such a low-level language, the program is doesn't need to be compiled, the action of the assembler and linker is to rename the human readable code into machine code and allocate memory for the program. This is exactly the same as the quantum libraries discussed here



[17].

quantum computers have linkers at the moment. what a compiler is and should be for quantum computer. The assembler process which assigns temporary addresses to parts of the object file is similar to a lot of the resource counting steps that the current quantum libraries and languages have.

[17] see the cloud based section, the python code and *compiled* code are relabelings of each other

```
%% TODO IN A LATER VERSION. :(

%%%%%%%%%%%%%%%%%%%%%% RIP FORTRAN YOU MAGNIFICENT BASTARD %%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%% HELLO Q-FORTRAN THE QUANTUM FORMULA TRANSLATOR %%%%%%%%%%%%%%%%%%


                                                                                %%
  %%%%%%%%      %%%%%%   %%%%%      %%%%%%   %%%%%%%   %%%%%%          %%%       %%       %%
%          %     %        %      %     %      %      %      %      %       %  %%     % %     %%
%   %%     %     %%%%%    %      %     %%%%%%      %      %%%%%%      %%%%%%%    %   %    %%
%     %%   %     %        %      %     %     %%     %      %     %%    %       %%   %     % %%
  %%%%  %%      %          %%%%%      %      %%     %      %      %%  %         %%  %       %%%
           %%%                                      %                              %
               %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% this took longer than you'd think
```

Appendix A

A gentle introduction to quantum theory

In this chapter we cover some supplementary background quantum theory. Although a deeper knowledge of quantum theory would serve to further your understanding of quantum computing, everything you need to know for this guide is covered in Chapter 3. 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 [1]. You have seen in Section Chapter 3 that the state of a system governed by quantum mechanics can be represented by a sum of basis states that each correspond to binary representations of information. In this section we will give a more detailed introduction to Dirac notation and cement the description given in Section Chapter 3 into a more formal language that will allow you to describe more complex systems and will allow you to probe deeper into more advanced literature.

A.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. This labelling 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 can be anything you like.

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' of the chosen card could have been one of 13 possible values, therefore the set of states needed to fully describing the system (the system here being a randomly chosen card from a single deck of cards) 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 H of the system [1] *p66*. Formally we say the complete set of independent and orthonormal states describing a system span a Hilbert space of that system:

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

A Hilbert space is a vector space with a defined inner product. Its 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 $|18\rangle$ as there are now cards valued that high in the deck. In that sense, the Hilbert space helps define the boundaries of a system.

As described in Chapter 3, $|\ldots\rangle$, used to describe a state, is called a 'ket'. 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 [1]. 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 finding the associated bra of the state, $\langle a|$ (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.

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. Binary representation of information only increase in dimension size by powers of 2 meaning the full Hilbert

space would have dimension size that is also a power of two. 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 the following Section.

A.2 Superposition

Using the states that form the basis of our Hilbert space to describe the system is not sufficient. 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. We 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 quantum 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, 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 $|\psi\rangle$ 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 $|\beta|^2$ then we

describe the state as:

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

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 (see [1] *p80* for an example of how to take the absolute value). The significance of having complex valued amplitudes will be discussed on the next page but for now it's 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:

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

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 $\left|\psi\right\rangle \in H$ 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 [1] *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: $\left|\psi_a\right\rangle = \frac{1}{\sqrt{2}}(|0\rangle - |1\rangle)$ and $\left|\psi_b\right\rangle = \frac{1}{\sqrt{2}}(|0\rangle + |1\rangle)$. 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 [1] *p13*.

A.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 [85] 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. $\hat{O}: H \to H$. 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 $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$:

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

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 [1].

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:

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

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.

We have already mentioned that a system of n qubits has a Hilbert space of dimension 2^n and the process of finding the basis states of the combined system is described in Chapter 3. The tensor product of two qubits that live in H_1 and H_2 respectively, denoted $\mathbb{C}_1^2 \otimes \mathbb{C}_2^2$, gives s combined qubit space $H_{1+2} = H_1 \otimes H_2$.

$$H_{1+2} := span\{\left|00\right\rangle, \left|01\right\rangle, \left|10\right\rangle, \left|11\right\rangle\} \tag{A.14}$$

A.4 An Intro to Quantum Information Theory

In this section we will used what we have learnt in Section Chapter 3 & Appendix A to examine a basic example of a quantum algorithm.

The first thing to note is that the size of you Hilbert space (i.e. 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. Due to the inability to copy states and the need for each operation to be reversible, it is often necessary to introduce 'redundant' registers 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 a 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.15)

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.16)

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 0 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.17)

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.

Bibliography

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[17] https://www.xanadu.ai/.

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

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

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

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

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

[23] 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.

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

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

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

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

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

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

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

[31] https://www.cs.princeton.edu/research/techreps/TR-934-12.

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

[33] https://www.computerworld.com.au/article/644051/google-launches-quantum-framework-cirq

[34] https://github.com/quantumlib/Cirq.

[35] https://www.mathstat.dal.ca//~selinger/quipper/.

[36] https://strawberryfields.readthedocs.io/en/latest/algorithms/teleportation.html,.

[37] https://github.com/markf94/os_quantum_software.

[38] David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proc. R. Soc. Lond. A*, 439(1907):553–558, 1992.

[39] Peter W Shor. Algorithms for quantum computation: Discrete logarithms and factoring. In *Foundations of Computer Science, 1994 Proceedings., 35th Annual Symposium on*, pages 124–134. Ieee, 1994.

[40] 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.

[41] Andrew Lucas. Ising formulations of many np problems. *Frontiers in Physics*, 2, 2014.

[42] B. O'Gorman, R. Babbush, A. Perdomo-Ortiz, A. Aspuru-Guzik, and V. Smelyanskiy. Bayesian network structure learning using quantum annealing. *The European Physical Journal Special Topics*, 224(1):163–188, 2015.

[43] Steveni H. Adachi and Maxwell P. Henderson. Application of quantum annealing to training of deep neural networks. 2015.

[44] https://www.dwavesys.com/tutorials/background-reading-series/quantum-computing-primer.

[45] Alberto Peruzzo, Jarrod McClean, Peter Shadbolt, Man-Hong Yung, Xiao-Qi Zhou, Peter J. Love, Alán Aspuru-Guzik, and Jeremy L. O'Brien. A variational eigenvalue solver on a photonic quantum processor. *Nature Communications*, 5:4213 EP –, Jul 2014. Article.

[46] Jarrod R McClean, Jonathan Romero, Ryan Babbush, and Alán Aspuru-Guzik. The theory of variational hybrid quantum-classical algorithms. *New Journal of Physics*, 18(2):023023, 2016.

[47] P. J. J. O'Malley, R. Babbush, I. D. Kivlichan, J. Romero, J. R. McClean, R. Barends, J. Kelly, P. Roushan, A. Tranter, N. Ding, B. Campbell, Y. Chen, Z. Chen, B. Chiaro, A. Dunsworth, A. G. Fowler, E. Jeffrey, E. Lucero, A. Megrant, J. Y. Mutus, M. Neeley, C. Neill, C. Quintana, D. Sank, A. Vainsencher, J. Wenner, T. C. White, P. V. Coveney, P. J. Love, H. Neven, A. Aspuru-Guzik, and J. M. Martinis. Scalable quantum simulation of molecular energies. *Phys. Rev. X*, 6:031007, Jul 2016.

[48] Bela Bauer, Dave Wecker, Andrew J. Millis, Matthew B. Hastings, and Matthias Troyer. Hybrid quantum-classical approach to correlated materials. *Phys. Rev. X*, 6:031045, Sep 2016.

[49] Nicolas P. D. Sawaya, Mikhail Smelyanskiy, Jarrod R. McClean, and Alán Aspuru-Guzik. Error sensitivity to environmental noise in quantum circuits for chemical state preparation, 2016.

[50] Nicholas C. Rubin. A hybrid classical/quantum approach for large-scale studies of quantum systems with density matrix embedding theory, 2016.

[51] Jarrod R. McClean, Mollie E. Kimchi-Schwartz, Jonathan Carter, and Wibe A. de Jong. Hybrid quantum-classical hierarchy for mitigation of decoherence and determination of excited states. *Phys. Rev. A*, 95:042308, Apr 2017.

[52] D. Deutsch. Quantum theory, the church-turing principle and the universal quantum computer. *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, **400**(1818):97–117, 1985.

[53] Ashley Montanaro. Quantum algorithms: an overview. *Npj Quantum Information*, 2:15023 EP –, Jan 2016. Review Article.

[54] https://math.nist.gov/quantum/zoo/.

[55] T. D. Ladd, F. Jelezko, R. Laflamme, Y. Nakamura, C. Monroe, and J. L. O'Brien. Quantum computers. *Nature*, 464:45 EP –, Mar 2010. Review Article.

[56] John Preskill. Quantum computing in the nisq era and beyond, 2018.

[57] X. Qiang, X. Zhou, J. Wang, C. Wilkes, T. Loke, S. O'Gara, L. Kling, G. Marshall, R. Santagati, J. B. Wang, J. L. O'Brien, M. G. Thompson, and J. C. F. Matthews. A universal two-qubit photonic quantum processor. In *Conference on Lasers and Electro-Optics*, page FM1G.1. Optical Society of America, 2018.

[58] Cristian S. Calude and Elena Calude. The road to quantum computational supremacy, 2017.

[59] K. Wiesner. The careless use of language in quantum information, 2017.

[60] http://dabacon.org/pontiff/?p=11863.

[61] Aram W. Harrow and Ashley Montanaro. Quantum computational supremacy. *Nature*, 549:203 EP –, Sep 2017.

[62] Alex Neville, Chris Sparrow, Raphaël Clifford, Eric Johnston, Patrick M. Birchall, Ashley Montanaro, and Anthony Laing. Classical boson sampling algorithms with superior performance to near-term experiments. *Nature Physics*, 13:1153 EP –, Oct 2017.

[63] Alexander M. Dalzell, Aram W. Harrow, Dax Enshan Koh, and Rolando L. La Placa. How many qubits are needed for quantum computational supremacy?, 2018.

[64] Edward Farhi, Jeffrey Goldstone, and Sam Gutmann. A quantum approximate optimization algorithm, 2014.

[65] Edward Farhi and Aram W Harrow. Quantum supremacy through the quantum approximate optimization algorithm, 2016.

[66] Michel X. Goemans and David P. Williamson. Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming. *J. ACM*, 42(6):1115–1145, November 1995.

[67] Tameem Albash and Daniel A. Lidar. Adiabatic quantum computation. *Rev. Mod. Phys.*, 90:015002, Jan 2018.

[68] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 549:195 EP –, Sep 2017.

[69] Jeremy Adcock, Euan Allen, Matthew Day, Stefan Frick, Janna Hinchliff, Mack Johnson, Sam Morley-Short, Sam Pallister, Alasdair Price, and Stasja Stanisic. Advances in quantum machine learning, 2015.

[70] Aram W. Harrow, Avinatan Hassidim, and Seth Lloyd. Quantum algorithm for linear systems of equations. *Phys. Rev. Lett.*, 103:150502, Oct 2009.

[71] Iordanis Kerenidis and Anupam Prakash. Quantum recommendation systems, 2016.

[72] Fernando G. S. L. Brandao and Krysta Svore. Quantum speed-ups for semidefinite programming, 2016.

[73] Fernando G. S. L. Brandão, Amir Kalev, Tongyang Li, Cedric Yen-Yu Lin, Krysta M. Svore, and Xiaodi Wu. Quantum sdp solvers: Large speed-ups, optimality, and applications to quantum learning, 2017.

[74] Jonathan Olson, Yudong Cao, Jonathan Romero, Peter Johnson, Pierre-Luc Dallaire-Demers, Nicolas Sawaya, Prineha Narang, Ian Kivlichan, Michael Wasielewski, and Alán Aspuru-Guzik. Quantum information and computation for chemistry, 2017.

[75] C. M. Wilson, J. S. Otterbach, N. Tezak, R. S. Smith, G. E. Crooks, and M. P. da Silva. Quantum kitchen sinks: An algorithm for machine learning on near-term quantum computers, 2018.

[76] https://qiskit.org/aqua.

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

[78] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, (4):27–75, 1993.

[79] Jack S Kilby. Invention of the integrated circuit. *IEEE Trans. on Electoron Devices*, 23(7):651–652, 1976.

[80] Guniness World Records. First computer to use integrated circuits. http://www.guinnessworldrecords.com/world-records/first-computer-to-use-integrated-circuits, 2018.

[81] 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.

[82] http://web.physics.ucsb.edu/~martinisgroup/.

[83] 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.

[84] Alwin Zulehner, Alexandru Paler, and Robert Wille. Efficient mapping of quantum circuits to the ibm qx architectures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2018*, pages 1135–1138. IEEE, 2018.

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

https://github.com/ot561/qprogramming/tree/master