Quantum Meta-Programming for Dummies with a focus on Poisson Bullets

Cohort 4

Quantum Engineering CDT
University of Bristol

July 2, 2018

Preface

Won’t someone write me?

Contents

Introduction

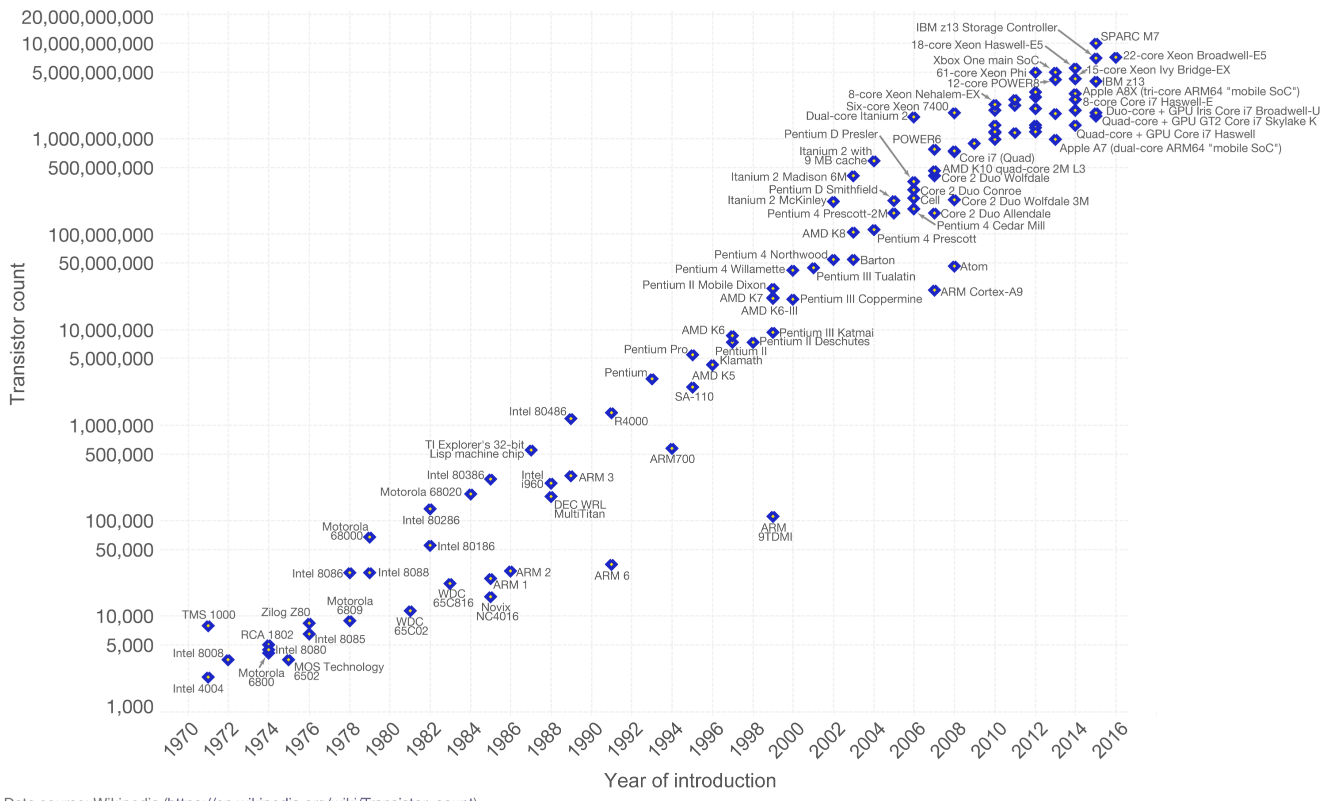
Quantum mechanics is one of the most well-tested theories in existence. It is also one of the most unintuitive, revealing aspects of nature at nanoscopic scales which are entirely incompatible of our a human's experience of the world.

There are two main facets which differentiate quantum from classical theory. The first, from which the field derives its name, is the quantisation of properties of a particle such as charge, angular momentum, and energy. This feature has already changed the world substantially over the course of the last 70 years. In addition to technologies such as the laser and magnetic resonant imaging (MRI) [1, 2], perhaps the greatest impact has been made through the manipulation of semiconductors. Since the invention of the first transistor in 1947 [3], semiconductor technology has laid the groundwork for scalable computing, bringing us into the the current information age.

The development of this technology comes at a critical time in the conventional silicon industry. The famous 'Moore's law', hypothesised in its current form in 1975, stated that the number of transistors per square inch would double every two years. This rule of thumb models the exponential scaling which computing power has followed since its conception incredibly well, as seen in Fig. 1. This was driven predominantly by the cost and power consumption per transistor going down as feature sizes decreased [4]. However, now increasingly small feature sizes have resulted in energy efficiencies and profitability are starting to plateau, while technical issues continue to increase. Some of these issues are in part due to quantum effects such as 'tunnelling', where an electron is able to access regions in space where, classically, it would be have insufficient energy to reach.

In fact, this feature is already used in so-called quantum annealers such as those built by D-Wave [5]. These machines use tunneling to find optimal solutions to a range of optimisation problems, where This example of tunnelling is related to the so-called 'wave-particle duality' which is the second key feature of quantum mechanics. The distinction between waves and particles, whose behaviour is well established in classical physics, becomes blurred. Every object in the universe, depending on their energy and confinement, will display both of these aspects to some degree.

The tantalising promise of quantum computing is hidden in this property. Particles such as electrons, which have comprised the backbone of electricity and classical information for the past century, have the ability to behave in a wave-like manner: they could contain not just the binary bit choices of 0 or 1, but one of an infinite number of continuous values, called 'qubits'. The ability to investigate the effect of a process on both 0 and 1 bits simultaneously means that quantum computers have the ability to scale exponentially (instead of polynomially) with the resources (i.e. bits) that are



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.

Licensed under CC-BY-SA by the author Max Roser.

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

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

While in general it is doubtful that a quantum computer will be universally 'faster' than a classical computer, and is much harder to engineer, there is significant potential to outperform conventional computers at certain tasks. While the amount of information processing in a conventional computer scales linearly with the number of bits, a quantum computer scales exponentially with the number of qubits for these tasks. Thus adding a single extra qubit could double the computing power. These are discussed, along with the quantum algorithms used to implement them, in chapter 3. At the point

[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 $\sim$ 100 times greater than the *current* global annual food production. (At $\sim 10^{12}$kg [6])

Figure 2: The summit supercomputer, currently set to be the worlds fastest supercomputer, taking up the area of two tennis courts. [20]

when quantum computers are able to outperform classical supercomputers at a task, the so-called ‘quantum supremacy’ will have been achieved.

These tasks range from aiding the fields of medicine, chemistry and materials with applications including creating more powerful simulations[2][8]; providing potential speedups for AI and machine learning [9]; assisting with modelling complex logistics problems; and improving financial models [10].

However, achieving this potential does not come without significant engineering difficulties. Readout or detection of the information in the qubit destroys the information contained within, resulting in us reverting to the classical bit values with an some probabilities. These probabilities are a fundamental (and unremovable) part of quantum theory, providing the link between . . . Furthermore, the technology is still very young and undeveloped. Algorithms exist for many of the applications above, but there may be many more as yet undiscovered. Academic institutions , large corporations (including Google [11, 12], IBM, [13] Intel [14]) and small start-ups (Rigetti, [15]) alike have invested heavily in hardware. There are a wide range of platforms and architectures, including but not limited to superconducting qubits [12], ion traps [16], quantum dots [17], spin qubits in silicon [18], and silicon poisson bullets [19].

[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 [7].

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

References for already available programming guides [21], [22], [23] also his blog [24], [25]. Our guide is gonna be different because these guides are all focused on a specific software whilst ours uses them all!! and also we are gonna be comparing them with some algorithms, though this has also been done [26]

Over the next few years and decades quantum computing is likely to become a reality, eventually becoming accessible to people from a range of disciplines via cloud services. It will be crucial when this becomes the case that people are able to understand how to use these machines in order to harness their applicability to the areas of mathematics, computer science, chemistry and finance. This guide is designed to be an introduction to the science of quantum computers and the current state of the field. Initially we explain in more depth how quantum computers work and their differences to classical computers in section 1.1. Since in the short-term, quantum computers are likely to be noisy, error-prone and limited in scale, we discuss how they can be used in this regime in chapter 2.

Once the engineering of quantum computers have been improved, then a host of more impressive applications can be demonstrated, which are considered in chapter 3. We examine the programming languages that will be able to interface between the algorithms and the quantum computer in chapter 4, and hardware-specific implementations and architectures in chapter 5.

A more complete description of quantum mechanics is given in chapter 6 for any interested party.

Chapter 1

Background

1.1 Weird Vector things

[Will add to and fix the horrible sized vectors soon soon]

In this section we will attempt to introduce quantum mechanics and its basic unit of information, the qubit, for those will little to no background in physics. Some knowledge of linear algebra may prove useful but is not necessary. In classical computing and information theory the fundamental unit of information is the familiar bit. Every bit is a binary number 0 or 1 that we use to represent false or true, or combine together to encode any information we wish. We can(for reasons that will become clear) represent a bit as 2-dimensional vector where,

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

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

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

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

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

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

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

This can be extended to find a matrix that allows out to change any basis state into any other. Adding together numbers in either their binary or decimal form is obvious, however this clearly does not correspond to simple addition in their vector representation.

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

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

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

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

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

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

[Some stuff here]

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

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

where α and β are real or complex number such that $|\alpha|^2 + |\beta|^2 = 1$. This is the reasonable requirement that probabilities should always sum to 1, but encapsulates the principle of superpositions that we must measure to find outcomes.

A complete mathematical description of quantum mechanics is given in 6.

Chapter 2

Short term quantum computing

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

2.1 Adiabatic quantum computing & quantum annealers

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

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

Quantum annealing computer can be programmed to solve Quadratic unconstrained binary optimization (QUBO) problem. QUBO problems are such that to determine the minimum value of

$$C = \sum a_i q_i + \sum b_{ij} q_i q_j \tag{2.1}$$

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

In Fig. 2.1 yadda yadda

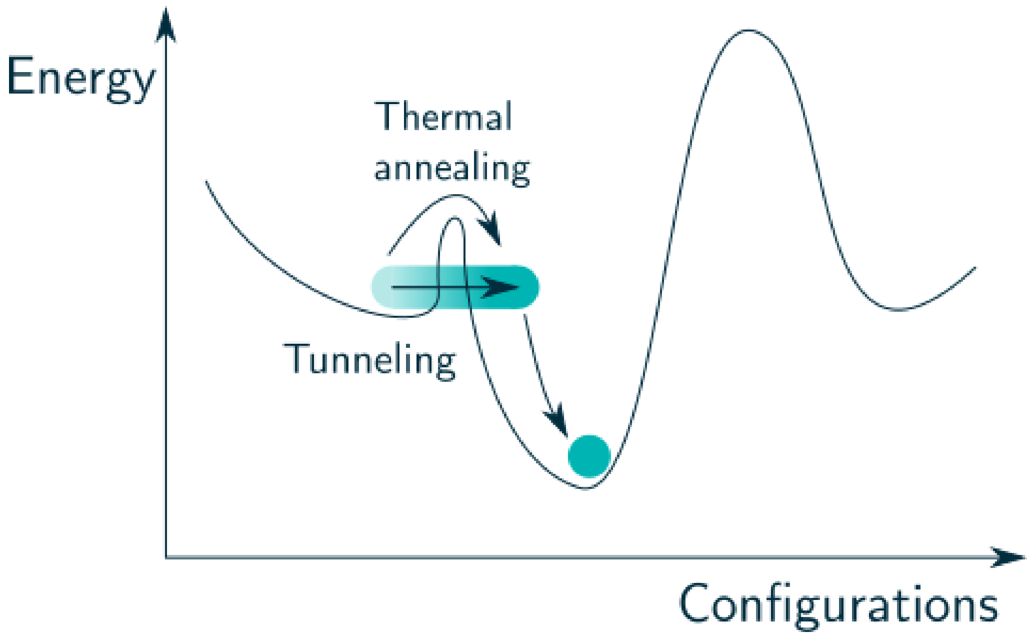


Figure 2.1: Caption

2.2 Rigetti- Forest

The Rigetti-Forest toolkit [27] for hybrid classical-quantum computing uses the language called Quil. The python library called pyQuil allows us to run simulations in their Quantum Virtual Machine (QVM) which allows the simulation up to 26 qubits, and to run simulations on their Quantum Processor Unit (QPU) which is the physical device called 19Q Acorn which have 20 physical qubits, of which 18 are logical. They recently made available a 100 pages pyQuil documentation [28] with examples and exercises to be implemented whether in the QVM or QPU. Access to the QVM can be granted to everyone by registering in their website (they send you an API key right away).

There are alrady programming guides addressing the use of pyQuil [23] and his blog [24], [25]. also citing benchmarking [26]. Our guide is similar to [25] as in we are going to be comparing specific algorithm with different languages, the differences are this and that also all code in jupyter notebooks.

In this section we address the main features of the Python library pyQuil from Rigetti following a simple example. We then address Deutsch's algorithm and an eigensolver algorithm.

2.2.1 Quantum Programs with pyQuil

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

Example 1

Generating the one-qubit pure state

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

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

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

```python
import numpy as np
# State initialisation
state0=np.array([[1],[0]])
# State manipulation
H=(1/np.sqrt(2))*np.array([[1,1],[1,-1]])
state1=np.dot(H,state0)
phi=np.pi/2
PHASE=np.array([[1,0],[0,np.exp(1j*phi)]])
state2=np.dot(PHASE,state1)
# Measurement: defining projectors P0,P1
ket0=np.array([[1],[0]])
ket1=np.array([[0],[1]])
P0=np.dot(ket0,ket0.T)
P1=np.dot(ket1,ket1.T)
# Probability of obtaining outcome 0
prob0=np.trace(np.dot(P0,np.dot(state2,np.
    conj(state2).T)))
# Probability of obtaining outcome 1
prob1=np.trace(np.dot(P1,np.dot(state2,np.
    conj(state2).T)))
```

Listing 2.1: Example with python only

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

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

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

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

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

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

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

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

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

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

3. Measurement: Yadda describing

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

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

```
# Running the program
cr=[]
results=qvm.run(p,cr,4)
print(results)
```

So in full our program looks like this Listing 2.2.

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

Listing 2.2: Example algorithm implemented with pyQuil.

Exercise 1: Arbitrary one-qubit pure state

Generating an one-qubit state of the form:

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

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

2.2.2 Deutsch's Algorithm

Deutsch's Algorithm

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

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

Listing 2.3: Deutsch's algorithm implemented in Python only

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

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

In Listing 2.4

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

Listing 2.4: Deutsch's algorithm with pyQuil

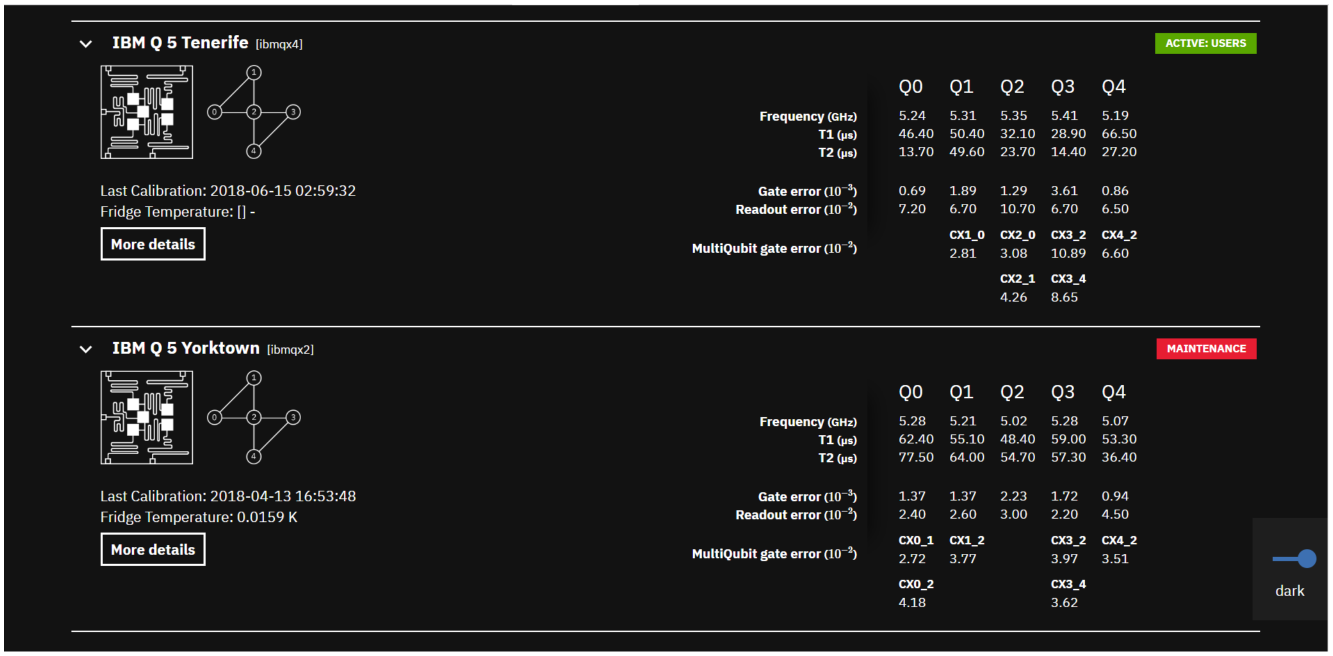
Exercise 2: Bernstein-Vazirani Algorithm

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

2.2.3 An Eigensolver Algorithm with pyQuil

2.3 IBM - Q Experience

IBM has launched the IBM Q experience that consists of a development environment called QISkit and a higher-level gate-building platform that allows users to compose their own algorithms. The devices used to perform the simulation and computation are based on a superconducting charge qubit implementation, which can be found described in more detail in chapter 5. There is ample space for development of algorithms in this environment as the visual arrays provide a clear structure to those familiar with quantum computation. The composer also displays the code on which it operates so that users interested in further development have the opportunity to learn how to code their own gates in QISkit. The associated documentation which is linked on the IBM Q website provides more detail to the structure [29]. Updates are regularly posted to the main site. Specifications of the device that is being used to perform these calculations are available on the website as seen in Fig. 2.2. In the most recent update, three processors can be accessed. Two of these have 5 qubits while the remaining processor has 16 qubits and is only available researchers on request. IBM recently published a report detailing their state-of-the-art prototype 50 qubit chip in the 2017 IEEE ICRC conference [30].


IBM Q 5 Tenerife [ibmqx4]

ACTIVE: USERS

Last Calibration: 2018-06-15 02:59:32
Fridge Temperature: [] -

More details

	Q0	Q1	Q2	Q3	Q4
Frequency (GHz)	5.24	5.31	5.35	5.41	5.19
T1 (μs)	46.40	50.40	32.10	28.90	66.50
T2 (μs)	13.70	49.60	23.70	14.40	27.20
Gate error (10^{-3})	0.69	1.89	1.29	3.61	0.86
Readout error (10^{-2})	7.20	6.70	10.70	6.70	6.50
		CX1_0	CX2_0	CX3_2	CX4_2
MultiQubit gate error (10^{-2})		2.81	3.08	10.89	6.60
			CX2_1	CX3_4	
			4.26	8.65	

IBM Q 5 Yorktown [ibmqx2]

MAINTENANCE

Last Calibration: 2018-04-13 16:53:48
Fridge Temperature: 0.0159 K

More details

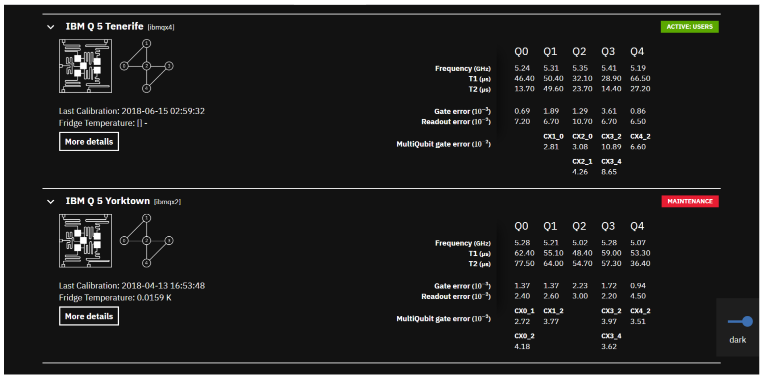
	Q0	Q1	Q2	Q3	Q4
Frequency (GHz)	5.28	5.21	5.02	5.28	5.07
T1 (μs)	62.40	55.10	48.40	59.00	53.30
T2 (μs)	77.50	64.00	54.70	57.30	36.40
Gate error (10^{-3})	1.37	1.37	2.23	1.72	0.94
Readout error (10^{-2})	2.40	2.60	3.00	2.20	4.50
	CX0_1	CX1_2		CX3_2	CX4_2
MultiQubit gate error (10^{-2})	2.72	3.77		3.97	3.51
	CX0_2			CX3_4	
	4.18			3.62	

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

Example Codes

Applications that can use the five-qubit register include simulation of quantum state evolution and computing the properties of a ground state. These applications are useful for quantum chemistry and solid state physics, which seek to understand systems of strongly correlated fermions and groups of molecules. The ground state problem can be understood as finding the solution to $H\left|\psi_g\right\rangle = E_g\left|\psi_g\right\rangle$, where H represents the Hamilton operator of the system and $|g\rangle$ and E_g represent the ground state and its associated energy, respectively.

Examples of composed codes in this environment are based on adding sequences of pre-defined gates. An example of two single-qubit operations being performed on bit 0 out of the 0-4 register. We


IBM Q 5 Tenerife [ibmqx4]

ACTIVE: USERS

Last Calibration: 2018-06-15 02:59:32
Fridge Temperature: [] -

More details

	Q0	Q1	Q2	Q3	Q4
Frequency (GHz)	5.24	5.31	5.35	5.41	5.19
T1 (μs)	46.40	50.40	32.10	28.90	66.50
T2 (μs)	13.70	49.60	23.70	14.40	27.20
Gate error (10^{-3})	0.69	1.89	1.29	3.61	0.86
Readout error (10^{-2})	7.20	6.70	10.70	6.70	6.50
MultiQubit gate error (10^{-2})		CX1_0 2.81	CX2_0 3.08	CX3_2 10.89	CX4_2 6.60
			CX2_1 4.26	CX3_4 8.65	

IBM Q 5 Yorktown [ibmqx2]

MAINTENANCE

Last Calibration: 2018-04-13 16:53:48
Fridge Temperature: 0.0159 K

More details

	Q0	Q1	Q2	Q3	Q4
Frequency (GHz)	5.28	5.21	5.02	5.28	5.07
T1 (μs)	62.40	55.10	48.40	59.00	53.30
T2 (μs)	77.50	64.00	54.70	57.30	36.40
Gate error (10^{-3})	1.37	1.37	2.23	1.72	0.94
Readout error (10^{-2})	2.40	2.60	3.00	2.20	4.50
MultiQubit gate error (10^{-2})	CX0_1 2.72	CX1_2 3.77		CX3_2 3.97	CX4_2 3.51
	CX0_2 4.18			CX3_4 3.62	

Figure 2.3: An example of the composition suite that allows gates to be strung together, mimicking the assembly of a classical computer. In this instance, the register is used to perform a basic

can look at an example of quantum chemistry in action.

2.4 The gate model and quantum circuits

This section briefly reviews the gate model for circuit based quantum computing and discusses the similarities between between digital and quantum computers. The gate model is one of the most popular architectures for quantum computation at the moment. A number of companies such as, *Intel [14] IBM [13], Google [11] and Rigetti [31]* 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. 2.4.

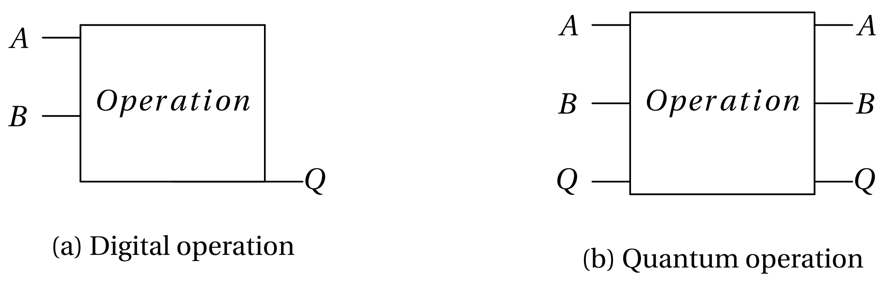
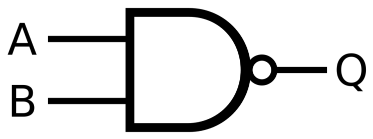


Figure 2.4: Digital and quantum logic circuits for implementing an arbitrary operation on bits A, B returning value Q

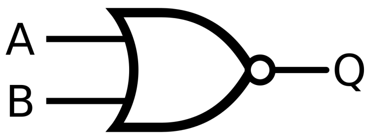
One of the main differences between the two figures is that in the quantum circuit the inputs $A\&B$ exist after the operation and the output Q is present before the operation. This is a feature of quantum computing being reversible (unitary).

Digital logic

Every digital computing operation can be built up from NAND logic gates [32]. We can call a NAND logic gate a universal gate for computation. The NOR gate is also universal in the same way any computation can be constructed from NOR gates.



(a) The NAND logic gate [33].



(b) The NOR logic gate [34].

Table 2.1: Global caption

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

(a) NOR gate truth table

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

(b) NAND gate truth table

Chapter 3

Algorithms and applications

We now discuss the three primary types of algorithms that take advantage of a quantum computer.

3.1 Quantum transforms

1

3.1.1 Quantum Fourier transform

Quantum Fourier transform is the quantum analogue of discrete Fourier transform.

3.1.2 Schur transform

3

3.2 Number theory algorithms

4

3.2.1 Shor's algorithm

5

3.2.2 Discrete Logarithm problem

6

3.3 Oracular algorithms

7

3.3.1 Grover's algorithm

One of the earliest algorithms that were designed to use quantum resources was described in 1996 paper by Lov Grover [35]. The algorithm attempts to solve the following problem: imagine you have a database of elements. We can represent them as bit strings, but we know that one of them is 'marked' by some function acting on that bit string. Examining the case where we have 4 numbers (2 bits), we have the following truth table.

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

(3.1)

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

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

$$U_f = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.2}$$

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

$$D = \frac{1}{2}\begin{pmatrix} -1 & 1 & 1 & 1 \\ 1 & -1 & 1 & 1 \\ 1 & 1 & -1 & 1 \\ 1 & 1 & 1 & -1 \end{pmatrix} \tag{3.3}$$

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

$$\frac{1}{2}\begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} = \frac{1}{2}\begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix} \tag{3.4}$$

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

$$\frac{1}{2}\begin{pmatrix}1&0&0&0\\0&1&0&0\\0&0&-1&0\\0&0&0&1\end{pmatrix}\begin{pmatrix}1\\1\\1\\1\end{pmatrix}=\frac{1}{2}\begin{pmatrix}1\\1\\-1\\1\end{pmatrix} \tag{3.5}$$

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

$$\frac{1}{2}.\frac{1}{2}\begin{pmatrix}-1&1&1&1\\1&-1&1&1\\1&1&-1&1\\1&-1&-1&1\end{pmatrix}\begin{pmatrix}1\\1\\-1\\1\end{pmatrix}=\frac{1}{4}\begin{pmatrix}0\\0\\4\\0\end{pmatrix}=\begin{pmatrix}0\\0\\1\\0\end{pmatrix} \tag{3.6}$$

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

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

3.3.2 Alternative representation: Dirac notation

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

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

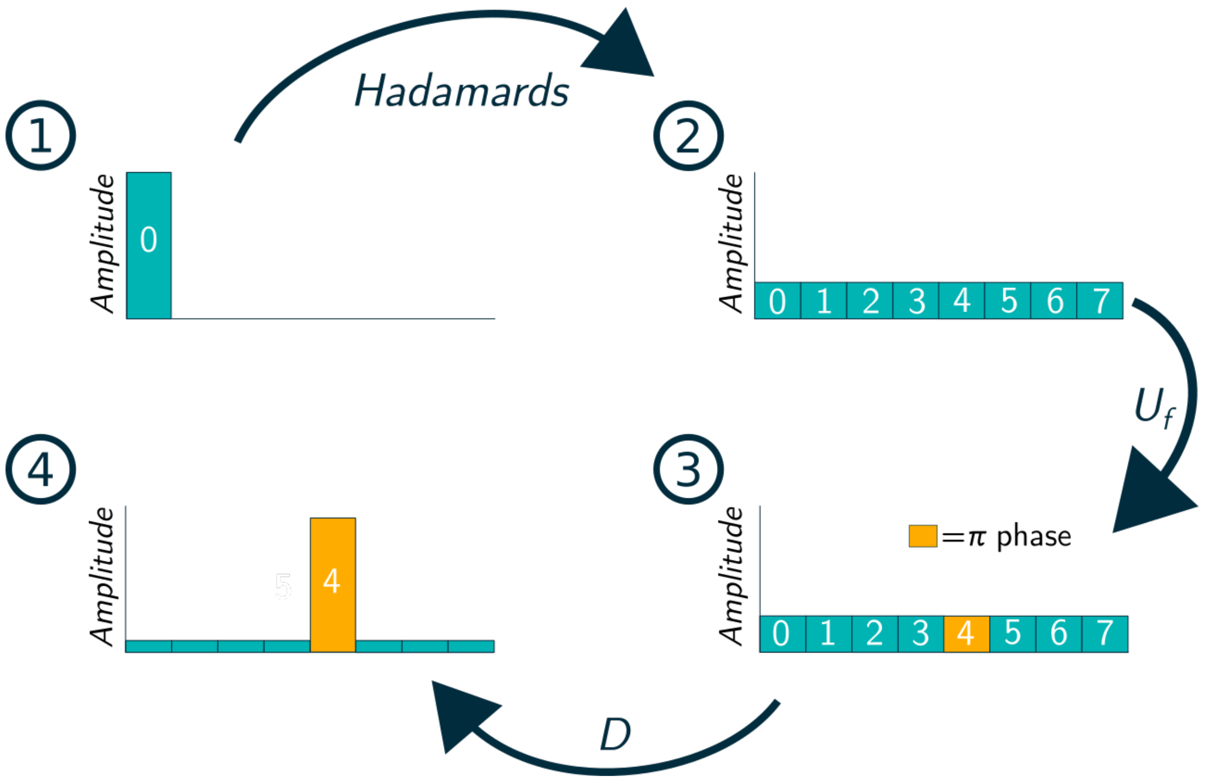


Figure 3.1: Pictorial demonstration of the steps of Grover's Algorithm

In the step 2 above, applying Hadamard gate to all the qubits leads to a state which is in superposition of all possible states (elements). It is important to start with the state $|000\rangle$ so that all the states in the superposition have the same initial phase. In step 3 we apply U_f gate which is dependent on $f(x)$ and is able to recognise the marked element. What it does is that it applies a π phase shift to the marked element but leaves all the other states unchanged. Next step is to apply gate D to the state obtained in step 3. The application of gate D can be understood as the operation which increases the amplitude of the phase shifted element in step 3. As mentioned in chapter 1, the measurement of a state leads to an output with a probability equal to the amplitude squared and thus, increasing the amplitude of the marked element state results in a higher probability of measuring that state. It is worth noting that repeating steps 3 and 4 a fixed number of times will increase the probability of detecting the marked element but after a certain point, the probability will start to decrease. For clarification, if we another iteration of step 3 and 4 in the above example, we get the state:

$$|\Psi\rangle = \frac{-1}{8\sqrt{2}}\big(|000\rangle + |001\rangle + |010\rangle + |011\rangle - 11\,|100\rangle + |101\rangle + |110\rangle + |111\rangle\big). \tag{3.7}$$

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

$$|\Psi\rangle = \frac{-7}{16\sqrt{2}}\big(|000\rangle + |001\rangle + |010\rangle + |011\rangle - \frac{13}{7}|100\rangle + |101\rangle + |110\rangle + |111\rangle\big) \tag{3.8}$$

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

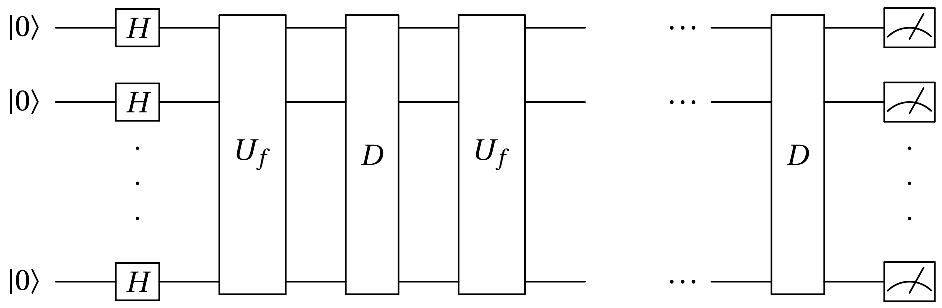
One of the important problems in computer science is the problem of unstructured search. Algorithm provided by Lov Grover (popularly known as Grover's algorithm) solves the unstructured search problem in $O(\sqrt{N})$ queries [**? ?**] compared to average number of $O(N)$ queries required by classical exhaustive search algorithms. The fact that Grover's algorithm can be applied to any problem in the complexity class **NP** makes it an important resource for the future of quantum computing. Grover's algorithm does not allow quantum computers to solve NP-complete problems in polynomial time but it provides a quadratic speedup over classical algorithms. This algorithm is based on the principle of selective phase-shifting of one or more states of a quantum system, which satisfy a certain condition, at each iteration. A simple case of Grover's algorithm for $N = 2^n$ elements with exactly one marked element has been described as following:

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

1. Apply U_f, where $U_f|x\rangle = (-1)^{f(x)}|x\rangle$.
2. Apply D, where $D = -H^{\otimes n}U_0H^{\otimes n}$

Finally, measure all the qubits and output the result.

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

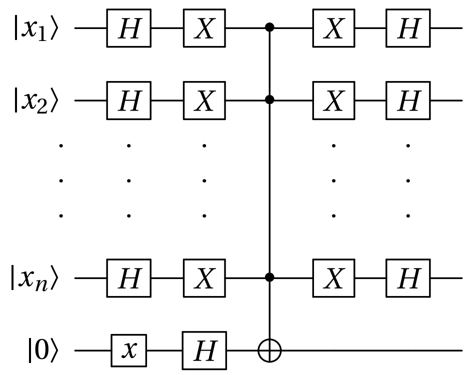


Time complexity of Grover's algorithm

Applying Grover's algorithm on an input $x \in \{0,1\}^n$ solves the unstructured search problem with a query complexity of $O(2^n)$. This leads to a time complexity of $O(2^{n/2}\text{poly(n)})$ [36], where the poly(n) overhead depends on the underlying hardware of the gates involved in the algorithm [37]. The circuit complexity of the gates U_f and D is the main concern while calculating this overhead as these gates are applied repeatedly in the algorithm until the desired state is found. The circuit complexity of gate U_f depends on the function $f(x)$ being solved by the algorithm while the circuit complexity of gate D is independent of the $f(x)$ but depends on the number of qubits required by the algorithm.

Construction of gate D

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



Algorithm name	Purpose	Number of qubits used	Number of gates required	Probability of success
Grover's algorithm	Unstructured search	n qubits for 2^n marked elements	$(2T+1)n$ Hadamard gates, T U_f gates, and T U_0 gates, where $T \approx \frac{\pi}{4}\sqrt{2^n}$	$\sin^2((2T+1)\arcsin\frac{1}{\sqrt{2^n}})$
Quantum period finding				

The hidden subgroup problem

9

3.4 Approximations & Simulating quantum systems

10

3.4.1 Approximating Matrix powers

11

3.4.2 Approximating Partition functions

12

Chapter 4

Programming a future universal quantum computer

Three languages will be covered here: Quil, QISKit and Q#.

4.1 Language features

For each language need to talk about the following:

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

4.1.1 pyQuil

Rough outline of what will be in this section, some of it is taken from Rigetti's website so will be changed!

- How is the program structured? (E.g. as a quantum processor/subprogram to regular programming language)

  - **pyQuil** - open source Python library developed that constructs programs (in Quil) for quantum computers
  - **Quil** – Quantum Instruction Language, represents quantum computations with classical feedback and control, assembly-like
  - **Forest** - Rigetti's entire quantum programming toolkit
  - **QVM** - Quantum Virtual Machine, runs simulations of up to 26 qubits, need API key
  - **QPU** – Quantum Processing Unit, actual chip with 19 qubits, need to request access
  - **Grove** - open source Python library containing quantum algorithms developed with pyQuil

  A program would be written in pyQuil, this gets translated to Quil which is like assembly language and once compiled is run on the QVM (or QPU). (Add diagram to make it clear?)

- Qubit, Ancilla bits, Classical bits
  Qubit (0/1) is referred to by its integer index. There are no ancilla bits. Classical bits (TRUE/-FALSE) can be used for classical control.

- How are gates applied?
  Gates are applied like: p = Program.inst(H(0), CNOT(0,1)). This applies a Hadamard to the zeroth qubit then a CNOT between the zeroth and first qubit. Available gates are: list these

- How are measurements handled?
  Measurements are done by: p.measure(0, 0) or p = Program(MEASURE(0,0)). This measures the zeroth qubit and stores the result in the zeroth classical register. Measurements are in the computational basis. It is also possible to look at the wavefunction before and after measurement.

- Available libraries
  Grove `http://grove-docs.readthedocs.io/en/latest/` is an open source Python library containing examples such as QFT, Grover etc.

- Available for which operating systems
  Available for all operating systems – you just need Python installed

- Use case of language and future of language
  Quil is an opinionated quantum instruction language — its basic belief is that in the near term quantum computers will operate as coprocessors, working in concert with traditional CPUs. This means that Quil is designed to execute on a Quantum Abstract Machine that has a shared classical/quantum architecture at its core. Not entirely sure what the future of the language will be, but I assume Rigetti will continue to develop it. Quil is more low-level and pyQuil can act as a more high-level language.

- Support available
  There is a lot of support available. PyQuil and Grove are open source with active githubs. There is documentation for pyQuil `http://pyquil.readthedocs.io/en/latest/index.html`, a white paper for Quil `https://arxiv.org/pdf/1608.03355.pdf` and an open Slack channel which can be joined so you can ask for help. There are also some videos and extra information on `https://www.rigetti.com/index.php/forest`.

- Flexibility of language (hardware, simulation, cost estimator, etc.)
  Quil acts as an assembly language so maybe could be used with any architecture but is currently only used with Rigetti's simulator and chip. There is a quantum virtual machine which is a simulator anyone can run code on and a superconducting chip with 19 qubits that requires special access.

4.1.2 QISKit

13

4.1.3 Q#

14

4.1.4 Further languages

TODO: short overview of other languages available.

4.2 Implementing Shor's algorithm

Shor's algorithm one of the most well-known quantum algorithms, due to it's implications for the safety of the commonly used RSA encryption scheme. It is also interesting, as it demonstrates the way classical and quantum computing can be used together. A large part of the algorithm is classical, but an essential part uses the quantum computer for significant speed up. The algorithm is discussed in more detail in section 3.2.1.

4.2.1 pyQuil

4.2.2 QISKit

4

4.2.3 Q#

When programming in Q#, the quantum computer is considered to be a coprocessor, a so-called QPU. Programs written in Q# control the QPU and need to be called by a driver program written in another language. The only requirement for this driver language is that it supports the .NET Framework. For the examples in this document C# is used, however other languages such as F# could be used instead without a problem.

```
// Placeholder code, also need to find specific "colouring" for C# instead of C
using Microsoft.Quantum.Simulation.Core;
using Microsoft.Quantum.Simulation.Simulators;

namespace Quantum.Bell {
    class Driver {
        static void Main(string[] args) {
            using (var sim = new QuantumSimulator()) {
                // Try initial values
                Result[] initials = new Result[] { Result.Zero, Result.One };
                foreach (Result initial in initials) {
                    var res = BellTest.Run(sim, 1000, initial).Result;
                    var (numZeros, numOnes, agree) = res;
                    System.Console.WriteLine(\$"Init:{initial,-4} 0s={numZeros,-4} 1s={
    numOnes,-4} agree={agree,-4}");
                }
            }
            System.Console.WriteLine("Press any key to continue...");
            System.Console.ReadKey();
        }
    }
}
```

4.3 The universal quantum computer

13

Chapter 5

Implementations

Thing

5.1 Not another introduction!

It is less important to understand the internal structure of computers today than it was in the past, given the wealth of programming languages and development tools available to the engineer.

However, in the general quest to develop a quantum computer, it pays to take a closer look at the internal structure of computers. The language surrounding quantum algorithms has been heavily influenced by classical computers (for example the qubit is analogous to a classical bit), and it makes sense that the implementation of a large scale quantum computer might be similarly influenced by its classical counterpart. Furthermore, software stacks do not yet exist for quantum computers, because of the lack of large quantum computers, and so it is not currently possible to get away with a high level understanding.

In this section, we discuss possible architectures of future large scale quantum computers. In particular, we consider the similarities and differences between classical and quantum computers, and how this might lead to differences between classical and quantum programming languages. We begin with a brief review of the historical development of computers and software, which we use as the starting point for our discussion of quantum computer architectures.

5.2 The development of classical computers

Are classical computers optimal? If it we had access to all the technology and hindsight we have now, would computers have developed differently? How would you optimally arrange a billion transistors into a classical computer if you could start from scratch?

There are many different approaches to computation. Some devices, such as the abacus or slide rule, are simple mechanical devices that speed up certain operations (performing arithmetic and finding logarithms) Others, such as Charles Babbage's difference engine, which computes polynomial approximates to logarithmic and trigonometric functions it also operates using mechanical means. However, the mechanism is automated so

Following the invention of the transistor in the late 1940s,

These are the problems we face in the design of a quantum computer today. We have an opportunity to consider these questions now, in anticipation of the existence of a billion qubit quantum computer.

The fundamental unit in the modern classical computer is the transistor. However, the transistor did not mark the beginning of the development of computers.

———— PLAN ————

Babbage difference engine – calculation of polynomial expressions, based on full adder operations realised analog computing – solving differential equations with op amps. The general purpose computer – turing machines, etc Von Neumann paper on computer architecture Digital computers using transistors

quantum computers as something different to a computer – perhaps it's wrong to draw any analogy at all.

Is it possible to learn from the development of computers? Make standards to avoid the back compatibility problems in classical; think about a universal optimal architecture, etc.

How did computers develop? Development of transistors, logic, the microcontroller, the minicomputer, etc. Comparison of different types of architectures: Harvard, Von-Neuman, transputers(!) (to make the point that some of them failed!). Suggestions of where we are at with quantum computer development – maybe transistors/logic gates.

5.3 Quantum Languages

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

5.3.1 High- and low- level quantum languages

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

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

5.3.2 Examples of different types of languages

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

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

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

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

If and QIf [2] statements

gate listing

5.4 Compilers

Compilers break down programming language code into the instruction set of the machine. The compiler is more or less important depending on the amount of abstraction in the language. The difficulty of implementing the compiler depends directly on how much abstraction is already contained in the instruction set. By definition, assembly language does not require a compiler because it is already written in terms of machine instructions. At the moment the machine code of all existing quantum information processors is their gate set, meaning that the compiler must perform gate synthesis (see section 5.6 for a more detailed discussion on gate synthesis).

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

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

[1] C

[2] Quantum if statement

5.4.1 Quantum oracles

One of the most used quantum tools is a bit oracle which is a function that maps

$$|x\rangle\left|y\right\rangle \mapsto |x\rangle\left|f(x)\oplus y\right\rangle$$

.

5.4.2 Linkers

connectivity, memory allocation (qubit allocation),

5.5 Quantum computer architecture

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

Hardware: the types of things included in the physical implementation: e.g. photons, trapped ions, etc. architecture

Compare the architecture of a quantum computer with the architecture of a classical computer. Most high level elements are analogous, but there are some differences: e.g. error correction does not really exist classically. Then most of the elements are subtly different: quantum I/O involves quantum measurements; memory basically involves swapping qubits since copying is not allowed.

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

Go through each element of the architecture talking about possible issues that might arise in its implementation.

5.6 Physical implementation

Quantum gate synthesis 7

Architecture is gate model, one-way quantum computing, topological.

platform is photonics, semi-conductor qubits, super-conducting qubits.

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

The gate model static and flying qubits The cluster state model Topological qubits

5.6.1 What are the qubits?

5.6.2 What are the operations?

- Classical If
- Quantum If QuIF

5.7 Error Correction

5.8 The future

5.9 QLANG $+-$ ™ the Quantum programming language

We introduce the QLANG$+-$™ language qubytes™ and quibbles™.

Chapter 6

Advanced topics

In this section we cover some supplementary background quantum theory. Although a deeper knowledge of quantum theory would serve to further your understanding of quantum computing, it should not be necessary for sections 1 to (?). You have seen in Section (?) that the state of a system governed by quantum mechanics can be represented by a binary 1 or 0, similar to that of a classical bit. However, in general the state of a bit can now be in a superposition of both 1 and 0, collapsing to one or the other when measured. In this section we will cement this description into a more formal language that will help you to probe deeper into more advanced literature.

6.1 Quantum mechanics

6.1.1 Quantum States

In quantum mechanics, we describe the state of a system simply with labels. These labels we assign to the system, such as '0','1', aim to give some intuition about the state of the system. We could equally have used 'open', 'closed' if appropriate. Formally these labels are called quantum numbers and in general are not restricted to be one of two values.

For example, imagine the 4 of spades was chosen from a deck of cards, a good choice of label to describe the card would be '4' or '♠'. Equally in a quantum system we would say that the card is in the state '4' or '♠' which we write formally as $|4\rangle$, or $|♠\rangle$. In this scenario the quantum number '4' could have been one of 13 values, therefore the set of states needed to fully describing the system (the system here being a randomly chosen card) are $\{|A\rangle, |2\rangle, |3\rangle, \ldots, |K\rangle\}$ with quantum numbers $\{A, 2, 3, \ldots K\}$. If the set of quantum numbers are unique and their associated states describe all possible values a properties can take then they are said to form a Hilbert space $\mathscr{H}$ of the system. Formally we say the complete set of independent and orthonormal states describing a system span a Hilbert space:

$$\mathscr{H} := span\{|A\rangle, |2\rangle, |3\rangle, \ldots, |K\rangle\} \tag{6.1}$$

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

The notation $|\ldots\rangle$, used to describe a state, is called a 'ket' and for every 'ket' there is a 'bra' conversely written as $\langle\ldots|$. The names originates from the first and second halves of the word 'bra(c)ket', which when placed together resemble the most important operation in quantum computation: the inner product. 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 quantum physics notation this operation is performed by turning the ket $|a\rangle$ into a bra, $\langle a|$ (this process is described more formally in section (?) but for now can be thought of as just a bracket swap). The 'bra' $\langle a|$ and 'ket' $|b\rangle$ are then used to form the word 'bra(c)ket' and is equal to 1 or 0 depending on whether the state a is equal to the state b.

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

$$\langle A|5\rangle = 0 \tag{6.2}$$

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

$$\langle J|J\rangle = 1 \tag{6.3}$$

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.

6.1.2 Superposition

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

To understand how we can describe this phenomena formally, lets make our deck of cards a 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 I know, but totally allowed within quantum mechanics). Before performing the operation of turning it over the state of the card should be thought of as being in a superposition of $|Q\rangle$ and $|4\rangle$. That is to say, the state used to describe the system before the measurement operation is not just one basis state but two, containing some probabilities that describing 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:

$$|\psi\rangle = \alpha\,|a\rangle + \beta\,|b\rangle \tag{6.4}$$

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. The significance of having complex valued amplitudes will be discussed on the next page but for now its sufficient to consider them as real.

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

$$\left|\psi\right\rangle = \frac{1}{\sqrt{3}}\left|Q\right\rangle + \frac{2}{\sqrt{6}}\left|4\right\rangle \tag{6.5}$$

where

$$\left|\frac{1}{\sqrt{3}}\right|^2 + \left|\frac{2}{\sqrt{6}}\right|^2 = 1 \tag{6.6}$$

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

Aside: Why do we use complex amplitudes?

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

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 $\left|\psi\right\rangle$ as before in Example 6.2 except now the 'bra' of $\left|\psi\right\rangle$ is given by the complex conjugate of the amplitudes multiplying the basis states within $\left|\psi\right\rangle$. For example, given $\left|\psi\right\rangle = (\alpha\,|a\rangle + \beta\,|b\rangle)$ and $\left|\phi\right\rangle = (\gamma\,|a\rangle + \rho\,|b\rangle)$ where $\{\alpha,\beta,\gamma,\rho\} \in \mathbb{C}$ then $\left\langle\psi\right| = (\overline{\alpha}\,\langle a| + \overline{\beta}\,\langle b|) \equiv (\left|\psi\right\rangle)^\dagger$. 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 6.2. 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':

$$\begin{aligned}\left\langle\psi\middle|\phi\right\rangle &= (\overline{\alpha}\,\langle 0| + \overline{\beta}\,\langle 1|)(\gamma\,|0\rangle + \rho\,|1\rangle)\\ &= (\overline{\alpha}\gamma\,\langle 0|0\rangle + \overline{\alpha}\rho\,\langle 0|1\rangle + \overline{\beta}\gamma\,\langle 1|0\rangle + \overline{\beta}\rho\,\langle 1|1\rangle)\\ &= (\overline{\alpha}\gamma + \overline{\beta}\rho)\end{aligned} \tag{6.7}$$

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

$$\begin{aligned} \langle \psi_a | \psi_b \rangle &= \frac{1}{2}(\langle 0| - \langle 1|)(|0\rangle + |1\rangle) \\ &= \frac{1}{2}(\langle 0|0\rangle + \langle 0|1\rangle - \langle 1|0\rangle - \langle 1|1\rangle) \\ &= \frac{1}{2}(1 + 0 + 0 - 1) \\ &= 0 \end{aligned} \tag{6.8}$$

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 [38] *p13*.

6.1.3 Operators

So far we have discussed how to formally describe the state of a quantum mechanical system and how to calculate the probability of a measurement outcome on a superposition state. We also introduced the notion of a measurement being a type of operation that maps superposition states to basis states. In general, all measurements are represented by *Hermitian* operators that have some specific properties. General quantum mechanical measurement is beyond the scope of this guide and instead we will restrict our discussion to the computational basis state $\{|0\rangle, |1\rangle\}$ used in quantum computation to describe qubits (see [39] 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.

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. We will see that one key difference is that there is no COPY operation. We can express the idea of an operation by the following equation:

$$|\phi\rangle = \hat{O}|\psi\rangle \tag{6.9}$$

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

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

2. The operator must preserve the inner products between the basis states. That is to say that the following must hold:

   ```
   if (state $\hat{O}|a\rangle$ is the same as state $\hat{O}|b\rangle$)
   return 1
   else
   return 0
   ```

   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 $hatX$ that performs a BIT-FLIP mapping 0 to 1 and 1 to 0. We represent this operator in the computational basis as:

$$\hat{X} = |0\rangle\langle 1| + |1\rangle\langle 0| \tag{6.10}$$

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

$$\begin{aligned}\hat{X}|\phi\rangle &= (|0\rangle\langle 1| + |1\rangle\langle 0|)(\alpha|0\rangle + \beta|1\rangle) \\ &= \alpha|0\rangle\langle 1|0\rangle + \beta|0\rangle\langle 1|1\rangle + \alpha|1\rangle\langle 0|0\rangle + \beta|1\rangle\langle 0|1\rangle \\ &= \beta|0\rangle + \alpha|1\rangle\end{aligned} \tag{6.11}$$

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. This duality of states/operators all having adjoints is a direct result of quantum mechanics fundamentally being a linear algebra formalism where points in a vector space are defined not only by their coordinates but also by the basis in use. For example, say I have a 3D Cartesian coordinate system and I specify a point (1,2,1), what this really means is $\hat{x}+2\hat{y}+\hat{z}$. The point is only unique once I have specified the basis vectors in use. Indeed it is intuitive to think of an operator, say a stretch, acting on the coordinates but we could equally have applied a squeeze to our definition of the basis states. Both perspectives must be considered in order to fully understand all possible routes to an outcome. In the case of quantum mechanics the adjoint state should be thought of as the basis and the state the coordinates with operators only acting on states and adjoint operators only acting on adjoint states. For a more detailed explanation of linear algebra formalism, see (REF). Let's now define the adjoint of an operator and take the adjoint of the BIT-FLIP operator, $\hat{X}^\dagger$, to see how it acts upon the adjoint state $|\phi\rangle^\dagger$.

NEED HELP WITH THIS AS I DONT THINK ITS CLEAR OR EVEN CORRECT

Let $\hat{O}$ be an operator represented in the computational basis that maps $|\psi\rangle$ to $|\phi\rangle$ with the following general form:

$$\hat{O} = \alpha |0\rangle \langle 0| + \beta |0\rangle \langle 1| + \gamma |1\rangle \langle 0| + \rho |1\rangle \langle 1| \tag{6.12}$$

where $\{\alpha, \beta, \gamma, \rho\} \in \mathbb{C}$. The adjoint of $\hat{O}$ is then given by:

$$\begin{aligned} \hat{O}^\dagger &= (\alpha |0\rangle \langle 0| + \beta |0\rangle \langle 1| + \gamma |1\rangle \langle 0| + \rho |1\rangle \langle 1|)^\dagger \\ &= (\alpha |0\rangle \langle 0|)^\dagger + (\beta |0\rangle \langle 1|)^\dagger + (\gamma |1\rangle \langle 0|)^\dagger + (\rho |1\rangle \langle 1|)^\dagger \\ &= \overline{\alpha}(|0\rangle \langle 0|)^\dagger + \overline{\beta}(|0\rangle \langle 1|)^\dagger + \overline{\gamma}(|1\rangle \langle 0|)^\dagger + \overline{\rho}(|1\rangle \langle 1|)^\dagger \end{aligned} \tag{6.13}$$

Things left to mention...

1. Linearity of operators
2. left and right acting
3. commutation
4. The +/- states
5. Expectation values

6.1.4 Beyond the Single Qubit

6.2 Quantum Information Theory

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

Bibliography

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

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

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

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

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

[6]

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[21] 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.

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

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

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

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

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

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

[28] Rigetti Computing. pyquil documentation, 2018.

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

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

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

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

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

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

[35] 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.

[36] Ashley Montanaro. Quantum algorithms: an overview. *npj Quantum Information*, 2:15023, 2016.

[37] Matthew Amy, Olivia Di Matteo, Vlad Gheorghiu, Michele Mosca, Alex Parent, and John Schanck. Estimating the cost of generic quantum pre-image attacks on sha-2 and sha-3. In Roberto Avanzi and Howard Heys, editors, *Selected Areas in Cryptography – SAC 2016*, pages 317–337, Cham, 2017. Springer International Publishing.

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

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