Hands-On
Quantum Information Processing with Python

Get up and running with information processing and computing based on quantum mechanics using Python

Dr. Makhamisa Senekane

BIRMINGHAM—MUMBAI

Hands-On Quantum Information Processing with Python

Copyright © 2021 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing or its dealers and distributors, will be held liable for any damages caused or alleged to have been caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

Group Product Manager: Kunal Parikh

Publishing Product Manager: Devika Battike

Senior Editor: Mohammed Yusuf Imaratwale

Content Development Editor: Sean Lobo

Technical Editor: Manikandan Kurup

Copy Editor: Safis Editing

Project Coordinator: Aishwarya Mohan

Proofreader: Safis Editing

Indexer: Priyanka Dhadke

Production Designer: Shankar Kalbhor

First published: January 2021

Production reference: 1280121

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-80020-115-6

www.packt.com

To my wife, Mosele Tsemane, for her outstanding support and words of encouragement throughout the writing of this book. To our daughter, Puleng Senekane.

Packt.com

Subscribe to our online digital library for full access to over 7,000 books and videos, as well as industry leading tools to help you plan your personal development and advance your career. For more information, please visit our website.

Why subscribe?

- Spend less time learning and more time coding with practical eBooks and Videos from over 4,000 industry professionals
- Improve your learning with Skill Plans built especially for you
- Get a free eBook or video every month
- Fully searchable for easy access to vital information
- Copy and paste, print, and bookmark content

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at packt.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at `customercare@packtpub.com` for more details.

At www.packt.com, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on Packt books and eBooks.

Contributors

About the author

Dr. Makhamisa Senekane is a lecturer at the National University of Lesotho. He has vast experience in the fields of quantum cryptography, quantum information processing, quantum computing, machine learning, and more.

About the reviewer

Srinjoy Ganguly is the founder and CEO of AdroitERA an EdTech firm and possesses double masters in Quantum Computing Technology from Technical University of Madrid, Spain and Artificial Intelligence from University of Southampton, UK, respectively. He has over 4 years of experience in Quantum Computing and 5 years of experience in Machine Learning, Deep Learning, AI . He is currently leading Quantum Machine Learning (QML) study space at QWorld and has authored a book on Quantum Computing with Silq Programming. He has given expert talk on QML at IEEE SPS. His research interests include Quantum Machine Learning, Quantum Computing, Quantum Image Processing, Machine Learning and Deep Learning.

Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit authors.packtpub.com and apply today. We have worked with thousands of developers and tech professionals, just like you, to help them share their insight with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Table of Contents

Preface

Section 1: Fundamentals of Quantum Information Processing

Chapter 1: Getting Started with Quantum Information Processing

Providing an overview of QIP

Subfields of QIP

Understanding the rationale for choosing Python for QIP

Installing Python and other packages

pipenv and creating the environment

Summary

Further reading

Chapter 2: Quantum States, Operations, and Measurements

Technical requirements

An introduction to linear algebra

Exploring vector spaces

Understanding the properties of a vector

Understanding the properties and types of matrices

Understanding the properties of a tensor product

Exploring the history of quantum information processing

Understanding the qubit

Qubit operations

Quantum measurement

Types of measurement

Dealing with multiple qubits

Quantum gates on multiple qubit systems

CNOT gate

Swap gate

Three-qubit gates

The quantum no-cloning theorem

Cluster-based quantum computing

Adiabatic quantum computing

Hybrid quantum computing model

Summary

Further reading

Section 2: Quantum Computers and Quantum Algorithms

Chapter 3: Entanglement and Quantum Teleportation

Technical requirements

Exploring the history of quantum entanglement

Understanding the Bell theorem and CHSH inequality

Understanding composite systems and entanglement

Understanding the CNOT gate – the entangling gate

Understanding Bell states

Understanding the entanglement of more than two quantum states

Greenberger-Horne-Zeilinger state (GHZ state)

The W state

Understanding entanglement as a resource – quantum teleportation

Quantum teleportation

Summary

Further reading

Chapter 4: Working with Quantum Circuits

Technical requirements

Introducing classical logic gates

Introducing single-qubit and multi-qubit gates

Introducing quantum circuits

Exploring quantum error correction

Exploring superdense coding

Summary

Further reading

Chapter 5: Quantum Algorithms

Technical requirements

Introducing Deutsch's algorithm

Exploring the Deutsch-Josza algorithm

Exploring the Bernstein-Vazirani algorithm

Introducing quantum Fourier transform and quantum phase estimation

Quantum Fourier transform (QFT)

Quantum phase estimation

Introducing Simon's algorithm

Exploring Shor's algorithm

Exploring Grover's algorithm

Summary

Further reading

Section 3: Deep Diving into Quantum Information

Chapter 6: Non-Local Quantum Games

Technical requirements

Understanding classical game theory

A brief history of game theory

Leaning about strategies in game theory

Exploring cooperative and non-cooperative games

Exploring zero-sum and non-zero-sum games

Understanding the prisoner's dilemma

Understanding the matching pennies game

Exploring the rock-paper-scissors game

Understanding quantum game theory

Understanding non-local quantum games

Exploring quantum strategies in non-local quantum games

Understanding the CHSH game

Understanding the GHZ game

Understanding the XOR game

Summary

Further reading

Chapter 7: Quantum Cryptography

Technical requirements

Introducing classical cryptography

A history of classical cryptography

Caesar's cipher

The one-time pad

A history of modern cryptography

Diffie-Hellmann key exchange protocol

Cryptographic primitives

Quantum cryptography

A history of quantum cryptography

Quantum cryptography primitives

Quantum key distribution protocols

Post-quantum cryptography

The NewHope key exchange scheme

The SPHINCS+ digital signature scheme

Summary

Further reading

Chapter 8: Quantum Machine Learning

Technical requirements

Understanding conventional (classical) machine learning

The three main categories of machine learning

Exploring artificial neural networks

Exploring SVMs

Understanding quantum machine learning

Data encoding

Quantum SVMs

Quantum variational classifier

Summary

Further reading

Chapter 9: Continuous-Variable Quantum Information Processing

Technical requirements

Introducing continuous-variable quantum information processing

Understanding the theory of continuous-variable quantum systems

Exploring continuous-variable quantum teleportation

Understanding continuous-variable quantum game theory

Continuous-variable quantum key distribution

Understanding continuous-variable quantum machine learning

Summary

Further reading

Chapter 10: Current Trends in Quantum Information Processing

Exploring current trends in quantum cryptography

Exploring current trends in quantum communication

Understanding current trends in quantum algorithm design

Exploring current trends in quantum machine learning

Understanding current trends in quantum computing hardware technologies

Exploring the future prospects of QIP

Summary

Further reading

Other Books You May Enjoy

Preface

Quantum information processing is a field of study that is a subclass of information processing. It exploits the laws of quantum mechanics to enable information processing in a manner that offers some advantages over conventional, non-quantum information processing. Quantum information processing has various sub-fields. These include quantum communication, quantum cryptography, quantum computing, and quantum error correction. This book provides a hands-on introduction to quantum information processing using the Python programming language.

In this book, you will find step-by-step explanations of essential concepts, practical examples, and self-assessment questions. You will begin by exploring the introductory notions of quantum information, including an overview of the prerequisite mathematical tools. Furthermore, you will learn how to use the Python programming language to implement some of the quantum information processing concepts. Finally, you will learn how to use various Python-based quantum information processing frameworks, such as QuTiP, Qiskit, Cirq, Strawberry Fields, and PennyLane.

By the end of this Hands-On Quantum Information Processing with Python book, you will have a deeper understanding and appreciation of the subject of quantum information and how various quantum information processing concepts can be implemented using Python.

Who this book is for

This book is for software developers, physicists, and mathematicians who are interested in quantum information processing. An understanding of the Python programming language would be beneficial. Furthermore, a basic understanding of mathematics, especially linear algebra, would be beneficial.

What this book covers

Chapter 1, *Getting Started with Quantum Information Processing*, provides a basic introduction to quantum information processing. It further provides information on the software tools and other requirements for you to get the most out of this book.

Chapter 2, *Quantum States, Operations, and Measurements*, provides the mathematics required for you to understand the subsequent chapters. Furthermore, this chapter introduces some quantum information processing concepts, such as qubits, quantum operations, the quantum no-cloning theorem, and models of quantum computing.

Chapter 3, *Entanglement and Quantum Teleportation*, discusses the concept of quantum entanglement. Furthermore, the chapter discusses one of the quantum communication protocols, namely the quantum teleportation protocol.

Chapter 4, Working with *Quantum Circuits*, introduces and discusses various quantum circuits. Additionally, this chapter covers quantum error correction. Finally, the chapter discusses one of the protocols of quantum communication, namely superdense coding.

Chapter 5, *Quantum Algorithms*, introduces various quantum algorithms. The quantum algorithms discussed in this chapter include the Deutsch algorithm, the Deutsch-Jozsa algorithm, Simon's algorithm, Grover's algorithm, and Shor's algorithm.

Chapter 6, *Non-local Quantum Games*, introduces various quantum nonlocal games that allow players to perform more optimally than if they did not use quantum resources. The quantum games covered in this chapter include the CHSH game, the GHZ game, and the XOR game.

Chapter 7, *Quantum Cryptography*, introduces the quantum information processing sub-field of quantum cryptography. It covers the implementation of quantum key distribution protocols, such as the BB84, B92, and E91 protocols, using the Python programming language. Finally, the chapter introduces the implementation of post-quantum cryptography using the Python programming language.

Chapter 8, *Quantum Machine Learning*, introduces quantum machine learning. It also covers the implementation of quantum machine learning algorithms using the Python programming language.

Chapter 9, *Continuous-Variable Quantum Information Processing*, covers the continuous-variable aspect of quantum information processing. Furthermore, it introduces implementations of continuous-variable quantum information processing using the Python programming language.

Chapter 10, *Current Trends in Quantum Information Processing*, concludes the book and covers current trends in the field of quantum information processing. It explores current trends in various sub-fields of quantum information processing. Finally, this chapter explores the future of quantum information processing.

To get the most out of this book

Software/hardware covered in the book	OS requirements
QuTiP	Windows, macOS X, or Linux (any)
Qiskit	Windows, macOS X, or Linux (any)
Cirq	Windows, macOS X, or Linux (any)
Strawberry Fields	Windows, macOS X, or Linux (any)
PennyLane	Windows, macOS X, or Linux (any)

If you are using the digital version of this book, we advise you to type the code yourself or access the code via the GitHub repository (link available in the next section). Doing so will help you avoid any potential errors related to the copying and pasting of code.

Download the example code files

You can download the example code files for this book from GitHub at https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python. In case there's an update to the code, it will be updated on the existing GitHub repository.

We also have other code bundles from our rich catalog of books and videos available at https://github.com/PacktPublishing/. Check them out!

Download the color images

We also provide a PDF file that has color images of the screenshots/diagrams used in this book. You can download it here: https://static.packt-cdn.com/downloads/9781800201156_ColorImages.pdf.

Conventions used

There are a number of text conventions used throughout this book.

`Code in text`: Indicates code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles. Here is an example: "The code uses the `qiskit` Python module."

A block of code is set as follows:

```
from qutip import *

v_00 = bell_state(state="00")
v_01 = bell_state(state="01")
v_10 = bell_state(state="10")
v_11 = bell_state(state="11")
```

Any command-line input or output is written as follows:

```
pip install pipenv
```

Bold: Indicates a new term, an important word, or words that you see onscreen. For example, words in menus or dialog boxes appear in the text like this: "This state is one example of entangled quantum states known as **Bell states**, **Bell basis states**, or **EPR states**."

TIPS OR IMPORTANT NOTES

Appear like this.

Get in touch

Feedback from our readers is always welcome.

General feedback: If you have questions about any aspect of this book, mention the book title in the subject of your message and email us at `customercare@packtpub.com`.

Errata: Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you have found a mistake in this book, we would be grateful if you would report this to us. Please visit www.packtpub.com/support/errata, selecting your book, clicking on the Errata Submission Form link, and entering the details.

Piracy: If you come across any illegal copies of our works in any form on the Internet, we would be grateful if you would provide us with the location address or website name. Please contact us at `copyright@packt.com` with a link to the material.

If you are interested in becoming an author: If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, please visit authors.packtpub.com.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions, we at Packt can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about Packt, please visit packt.com.

Section 1: Fundamentals of Quantum Information Processing

In this section, we will give you a jumpstart on the content and provide you with information regarding the necessary software tools and other requirements. We will cover the basic math required to understand the subsequent chapters, using Python as a tool for understanding.

This section comprises the following chapters:

- *Chapter 1*, *Getting Started with Quantum Information Processing*
- *Chapter 2*, *Quantum States, Operations, and Measurements*

Chapter 1: Getting Started with Quantum Information Processing

This chapter provides an introduction to **quantum information processing** (**QIP**). However, in order to understand QIP, it is important to revisit classical/conventional information processing, from which QIP is derived.

Therefore, this chapter will first provide a brief introduction to classical information processing. This will then be followed by a brief introduction to QIP.

The birth of classical information processing can be attributed to Claude Shannon, whose ideas in the 1940s on information theory paved the way for the information revolution that would later follow. Shannon's ideas made use of the laws of classical physics in order to enable information processing.

Furthermore, conventional information processing uses the concept of the **binary digit** (**bit**) as a unit of information. As we will see later in this book, this unit of information has its quantum counterpart, called a **quantum bit** (**qubit**). A qubit, as we will later learn, is the basic unit of quantum information. We will also later learn that a qubit is a more fundamental unit of information than a classical bit.

Another key aspect of conventional information processing is the notion of entropy, which was proposed by Shannon. In essence, entropy is a measure of uncertainty. Shannon's version of entropy can be used to quantify resources that are required to store information. For a random variable, X, and probability p, where the probability of an event, i, occurring is denoted by p_i, the Shannon entropy, $H(X)$, is given as follows:

$$H(x) = H(P_1 \cdots p_n) = -\sum_{x} p(x) log p(x$$

The following applies:

$$\lim_{x} x log x = 0$$

This chapter discusses the following topics:

- Providing an overview of QIP
- Understanding the rationale for choosing Python for QIP
- Installing Python and other packages
- `pipenv` and creating the environment

Providing an overview of QIP

The field of QIP arose in response to the limits imposed on conventional information processing by the fundamental laws of physics. These limits were initially investigated in the 1960s by Rolf Landauer. Landauer's work was later extended by his

colleague, Charles Bennett, in the 1970s.

Initially, the focus of QIP was to determine the physical limits that are imposed on conventional information processing. However, as time progressed, the scope of QIP was expanded in order to explore such ideas as quantum cryptography, quantum computing, and quantum communication. These ideas will be discussed later in this book.

Unlike classical/conventional information processing, which is based on the laws of classical physics, QIP uses a completely different paradigm. QIP makes use of quantum mechanical phenomena and principles in order to provide a more powerful way of processing information than is allowed by the laws of classical physics (quantum mechanics can be broadly defined as the study of nature at a sub-atomic level). This way, QIP is intended to provide some sort of advantage over classical information processing.

As already stated earlier, the basic unit of quantum information is a qubit (a qubit can be generalized to an arbitrary dimension and is called a **qudit** (for *d* dimensions). However, our focus in this book is limited to a two-dimensional space, and a qubit is for two-dimensional space). It is analogous to a bit. However, unlike a bit, which exists in either of two states (0 or 1), a qubit can exist in a superposition of states. Mathematically, a qubit is represented by a unit vector residing in a two-dimensional complex Hilbert space, $\mathbb{C}2$. Basically, Hilbert space is an example of vector space. More details about Hilbert space will be covered in *Chapter 2*, *Quantum States, Operations, and Measurements*.

Mathematically, a qubit, $|\psi>$, is given as follows:

$$|\psi\rangle = \alpha|0\rangle + \beta|1\rangle$$

Here, α and β are the probability amplitudes and are such that the following applies:

$$|\alpha^2| + |\beta^2| = 1.$$

The $|0>$ and $|1>$ states are the computational basis states analogous to 0 and 1 in classical information processing. A computational basis for both bits and qubits is made up of a pair of vectors that are linearly independent. A basic introduction to linear algebra will be provided in *Chapter 2*, *Quantum States, Operations, and Measurements*.

The computational basis state, $|0>$, can be represented as follows:

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

On the other hand, the computational basis state can also be represented as follows:

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

It is worth noting that any qubit, $|\psi>$, can be represented as the linear combination of the computational basis states.

A qubit can also be represented using a **Bloch sphere** (the Bloch sphere is named after the physicist and Nobel laureate *Felix Bloch*). This representation, which is a geometric representation of a qubit, provides a valuable means of visualizing a qubit.

A schematic diagram of a Bloch sphere is shown in *Figure 1.1*. Using a Bloch sphere, a qubit is represented as a point on the surface of a unit sphere. Therefore, a generic quantum state, $|\psi>$, is given as follows:

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

Here, $0 \le \theta \le \pi$ and $0 \le \phi \le 2\pi$. Also, θ, γ, and ϕ are real numbers. Since the global factor, $exp(i\gamma)$, has no observable effect, it can be omitted. Therefore, the preceding equation can be effectively written as follows:

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

Here is a graphical representation of a qubit using the Bloch sphere:

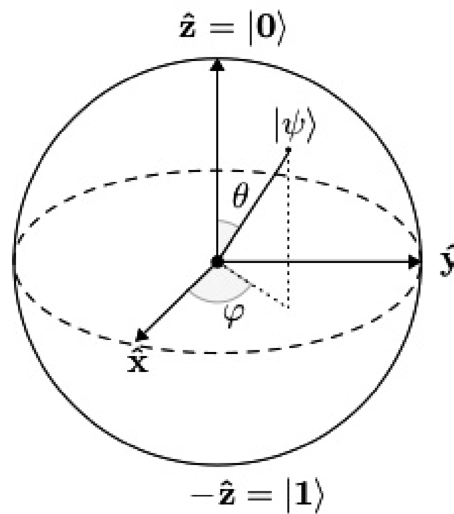


Figure 1.1 – A graphical representation of a qubit using the Bloch sphere

So far, we have only been focusing on a single qubit. However, qubits can also form composite systems of more than one qubit. Such systems are represented using a tensor product. A tensor is a generalization of a matrix and is widely used in QIP.

In a tensor product representation, an *n*-qubit quantum system is represented as follows:

$$|\psi\rangle = |\psi_1\rangle \otimes |\psi_2\rangle \otimes \cdots |\psi_n\rangle$$

As an example, consider a two-qubit composite system that is made up of qubits $|0\rangle$ and $|0\rangle$. This qubit is written as $|00\rangle$, which implies the following:

$$|00\rangle = |0\rangle \otimes |0\rangle$$

Vectorially, the same state (|*00*>) is given as follows:

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

This is because, as already stated, the state |*0*> is represented vectorially as follows:

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

|*1*>, meanwhile, is represented as follows:

$$\begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Therefore, |*00*> means the following:

$$\begin{bmatrix} 1 \\ 0 \end{bmatrix} \otimes \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

This looks as follows:

$$\begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

Now, let me briefly explain how the preceding tensor product is calculated. This is done by taking the first element of the vector on the left, then multiplying by all the elements (two elements in this case) of the vector on the right, and writing that result as the first two elements of the final product. The same is done for the second element of the vector on the left; it is multiplied by all the elements of the vector on the right, and the results of that are written as the last two elements of the product.

As an exercise for you, we request you to verify the following two elements:

$$|10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

$$|11\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

An additional exercise is for you to determine vectorial representations of the three-qubit quantum systems.

Based on the information provided in the second-last equation previously, it may seem natural to assume that any composite quantum system can be written as tensor products of the qubits. However, quite unfortunately, this is not the case. Composite quantum systems that can be represented in terms of the second-last equation previously shown are said to be separable.

On the other hand, those composite quantum systems that are not separable are said to be entangled. One example of an entangled quantum state is $(1/\sqrt{2})(|00> + |11>)$. This quantum mechanical phenomenon of entanglement, which *Albert Einstein* referred to as the *spooky action at a distance*, is one of the key resources of QIP.

The notion of *spooky action at a distance* will be further covered later in this book in *Chapter 3*, *Entanglement and Teleportation*. In essence, it refers to the correlated quantum effects that are observed by two quantum particles even though such quantum particles are separated by an arbitrarily long distance.

There are two basic quantum operations on a quantum system. These operations are **quantum logic gate operation** and **quantum measurement**. The former is in essence realized through the unitary transformation of a qubit. A qubit's evolution under a unitary

operation means that a quantum gate, which is a unitary transformation and hence reversible, is acting on a qubit. Under this quantum operation, the quantum system does not lose its **quantumness**.

The second quantum operation is the quantum basis measurement. Unlike the former quantum operation, this quantum operation on a qubit is irreversible. That is, once the quantum measurement operation is applied on a qubit, the qubit loses its quantumness and collapses to a classical state of either 0 or 1. The key difference between the two operations is that the output of the former is still quantum, while the output of the latter is classical.

The QIP paradigm discussed thus far is known as **discrete-variable QIP**. It uses discrete, finite-dimensional complex Hilbert space.

On the other hand, there is another paradigm of QIP that uses infinite-dimensional complex Hilbert space. This paradigm is known as **continuous-variable QIP**. By way of analogy, discrete-variable QIP can be thought of as *digital* QIP. On the other hand, continuous-variable QIP can be thought of as *analog* QIP.

So far in this section, we have provided an overview of **QIP**. In the next subsection, we will explore various subfields of QIP.

Subfields of QIP

Having introduced the basic principles of QIP, now our attention will turn to the subfields of QIP. We will do so by first exploring the quantum cryptography subfield.

Quantum cryptography

The first subfield of QIP that was pursued in earnest was quantum cryptography. The initial quantum cryptography concept was explored by *Stephen Wiesner* in the 1960s.

Wiesner's idea was about the use of quantum mechanics to make banknotes that would be impossible to counterfeit based on the laws of physics, and the use of these laws for an *oblivious transfer* cryptographic protocol.

Stephen Wiesner's idea of banknotes that are impossible to counterfeit inspired *Charles Bennett* (Bennett was a friend of Wiesner) and *Gilles Brassard* to propose the first ever **quantum key distribution** (**QKD**) protocol in 1984. This QKD protocol would later be known as the **BB84 protocol**.

In the early 1990s, another QKD protocol was independently proposed by *Artur Ekert*, and it was based on entanglement. This QKD protocol, which was later found to be mathematically equivalent to the BB84 protocol, came to be known as the **E91 protocol**. Currently, quantum cryptography is arguably one of the most successful subfields of QIP.

A few examples of real-world implementations of the QKD protocol are provided as follows. In 2007, it was used in Switzerland to secure election data. Furthermore, it was used in 2010 in Durban, South Africa, to secure communication data during the 2010 soccer World Cup competition. Lastly, just recently (in 2020), it was used to secure inter-continental communication, with communicating parties being in China and Austria.

Now that we have covered quantum cryptography, which is one of the subfields of QIP, the next step is to explore another subfield of QIP, namely quantum processing.

Quantum computing

Another subfield of QIP that has progressed rapidly is **quantum computing**. Quantum computing harnesses quantum mechanical concepts such as **entanglement** in order to efficiently solve some of the computational problems that are intractable to conventional computers.

Quantum computers are envisaged to provide capabilities that far exceed those of conventional computers. Thus, quantum computers should have a *super-advantage* over their conventional counterparts. The expression *quantum super-advantage* is preferred over the more offensive *quantum supremacy* that is sometimes used. I do believe that quantum super-advantage adequately conveys the message that quantum computing capabilities far exceed those of conventional computers. This it (quantum super-advantage) does without sounding offensive like quantum supremacy.

The history of quantum computing began in the 1980s, with the ideas of *Yuri Manin* in the **Union of Soviet Socialist Republics** (**USSR**) and *Richard Feynman* and *Paul Bernioff* in the **United States of America** (**USA**). These ideas were further extended by *David Deutsch*, who proposed the first ever universal quantum computer in 1985.

However, for the remainder of the 1980s, interest in quantum computing seemed to be waning. Quantum computing was just considered as an area of academic curiosity that was devoid of any practical relevance. The interest in quantum computing was revived in the 1990s. This revival was due to two key breakthroughs in quantum computing.

The first one was due to *Peter Shor's* quantum algorithm, which could be used to factor large numbers. Shor's algorithm demonstrated that in principle, a quantum computer can be used to break the security of conventional cryptosystems.

Another significant quantum computing breakthrough was a quantum search algorithm designed by *Lov Grover* in 1996. This algorithm, which was designed to search for an element in an unstructured database, was demonstrated to provide a quadratic speed-up compared to the best-known conventional algorithm of the time. This speed-up means that if a conventional algorithm would take at most n steps to search for an element in an unstructured database, the Grover algorithm would do so in at most $\sqrt{n}$ steps.

Although quantum computing promises to provide a quantum super-advantage over conventional computing, a fully functional quantum computer is yet to be realized. The major drawback with the realization of a full-scale quantum computer is that quantum systems are very fragile and isolating them from their environment (which they interact with) is very challenging.

Quantum systems' interaction with an environment might result in some errors that would need to be corrected. Since the 1990s, a plethora of quantum error correction techniques has been proposed.

Currently, a full-blown, highly scalable quantum computer has not yet been realized. However, smaller quantum computers are already being built. These small-to-medium-scale quantum computers, which typically do not use some of the complex error correction techniques, are referred to as **noisy intermediate-scale quantum** (**NISQ**) computers. NISQ computers are typically used for small-scale quantum computing, since they still do not have error correction capabilities. Therefore, NISQ computers are used mainly to demonstrate some limited capabilities of quantum computing, since they are not full-scale quantum computers.

Now that we have covered the quantum computing subfield, the next step is to explore another subfield of QIP, namely quantum games. Therefore, the design of quantum games is briefly discussed next.

Design of quantum games

Besides quantum cryptography and quantum computing, another subfield of QIP is the **design of quantum games**. In a conventional arrangement, game theory is concerned with rational decision making in a conflict environment. Quantum game theory generalizes conventional game theory to the quantum domain. Like other QIP subfields, quantum games harness quantum phenomena, especially entanglement, as a resource, and the objective is to outperform their classical counterpart.

So far, we have provided a brief introduction to QIP. We have introduced the subfields of quantum cryptography, quantum computing, and quantum games. The information provided in this chapter will make it possible for you to easily follow the information that will be provided in the later chapters of this book. Subsequent chapters of this book will further elaborate on what was introduced in this chapter.

The next subsection provides a rationale for using Python for QIP.

Understanding the rationale for choosing Python for QIP

The programming language of choice that will be used in this book is **Python**.

Python is arguably one of the most popular programming languages. This is due to the fact that Python is a very versatile, high-level programming language with a very active and robust developer and open source community.

Python's versatility and simplicity make it the language of choice for diverse applications such as embedded systems, gaming, website development, graphics design, data science, artificial intelligence, and complex scientific and numerical computing.

Another reason for choosing Python as the language of choice for this book is that the majority of the quantum information frameworks that will be deployed later in this book are based on Python. These QIP frameworks include those developed by prominent companies such as Google and IBM, together with some from lesser-known QIP start-up companies such as Xanadu in Canada (the official website of Xanadu is https://www.xanadu.ai).

Installing Python and other packages

Python is a cross-platform programming language in the sense that it can be used in at least three of the major operating systems, namely Microsoft Windows, Apple's macOS, and open source Linux.

Python's installer and documentation are freely available on Python's official website at https://www.python.org/ and are shown in *Figure 1.2*:

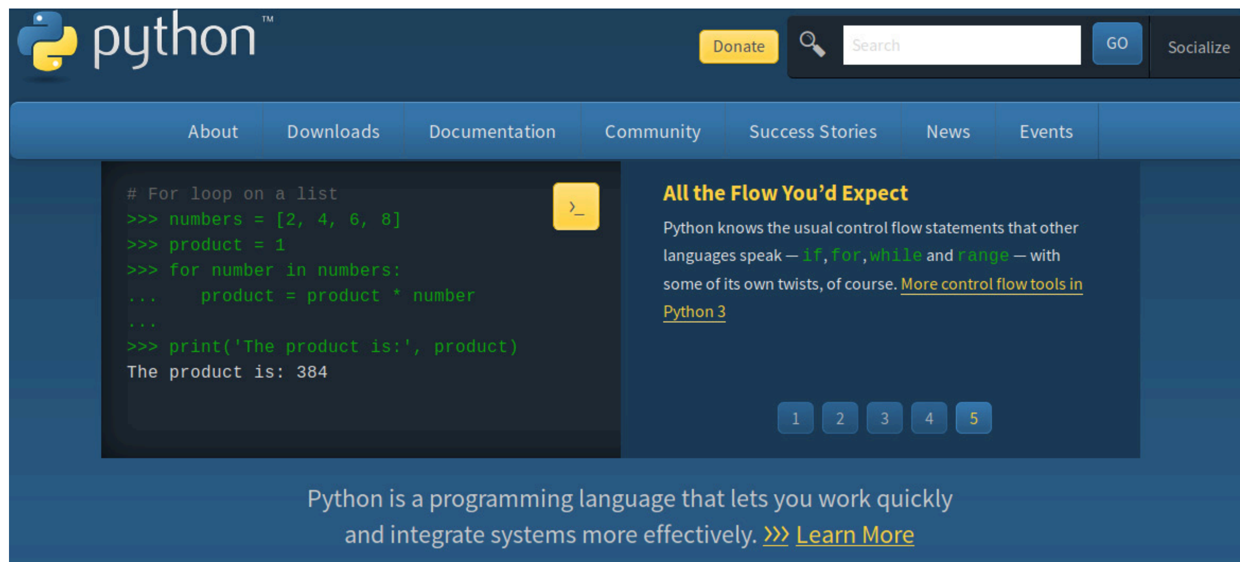


Figure 1.2 – The official website of the Python programming language

Once on the home page of the website, it is then easy to navigate to either the **Downloads** or **Documentation** tabs. For the purposes of this book, we recommend that you download *Python version 3.8.2*.

Another way of downloading the Python installer is via the use of Continuum Analytics' Anaconda cross-platform Python distribution. Unlike the official Python installer, the Anaconda installer comes with additional Python packages by default. This makes

package management less cumbersome than in the former case. The official website of Anaconda is https://www.anaconda.com/.

Once the appropriate Python installers are downloaded, they can then be installed on the target machine, depending on the type of operating system being used.

NOTE

In the preparation of this book, a 64-bit Ubuntu 20.04 LTS operating system was used.

Furthermore, in order to install additional Python packages, a **pip Python installer program** can be used. The `pip` command is run from the computer's terminal. The syntax for `pip` package installation is shown as follows:

```
pip install PackageName
```

In order to update an already-installed package using `pip`, the following command is used:

```
pip install PackageName --upgrade
```

The QIP frameworks that need to be included for use in this book include the following:

- **IBM's Qiskit**: https://qiskit.org/
- **Google's Cirq**: https://cirq.readthedocs.io/en/stable/
- **Xanadu's PennyLane**: https://pennylane.ai/
- **Xanadu's Strawberry Fields**: https://strawberryfields.readthedocs.io/en/stable/
- **QuTiP**: http://qutip.org/

As an alternative to installing Python packages on a local computer machine, Google's **Colaboratory** or **Colab** environment can be used. The official website of Google Colab is https://colab.research.google.com.

The command that is used to install Python packages on Google Colab is similar to the `pip` command, the exception being that unlike the `pip` command, the Google Colab command is preceded by the `!` symbol. Thus, the Google Colab command used to install a package called `PackageName` is as follows:

```
!pip install PackageName
```

The command used to update an already-existing Python package is as follows:

```
!pip install PackageName --upgrade
```

Now that we have understood the process of the installation of Python packages, let's explore `pipenv`.

pipenv and creating the environment

Python packages can be installed using installation tools such as `pip`. In some instances, it might be necessary to separate certain Python packages from the main Python installation. The reason for this might include the need to avoid some installation conflicts and to offer protection to the main Python installation against the installation of unstable packages. Examples of the tools that are used to isolate the main Python installation in this sense include `virtualenv` and `venv`.

Normally, in order to manage Python packages and dependencies, different Python libraries are used. For instance, `pip` is used for package installation, update, and uninstallation, while another tool such as `venv` is used to manage virtual environments. These packages are typically used separately. This in practice makes the management of a Python project very cumbersome. The solution to this is the use of a tool that can do the tasks of package management (installation, updating, and uninstallation), along with virtual environment creation and support. The appropriate tool for this is `pipenv`.

`pipenv` is a Python packaging tool. This tool is used to manage and simplify Python package dependencies.

It can be installed using the following command:

```
pip install pipenv
```

Once `pipenv` is installed, it replaces `pip` as a package management tool. Furthermore, it becomes responsible for all the package dependency tasks. It also creates two new files, namely the following:

- `pipfile`: Gives a list of all installed packages
- `pipfile.lock`: Used to manage complex dependencies

With that, we have reached the end of the chapter.

Summary

In this introductory chapter, we provided an overview of QIP. We introduced quantum cryptography, quantum computing, and quantum game design.

We also discussed the difference between discrete-variable QIP and continuous-variable QIP. Furthermore, we introduced the concept of a qubit, which is analogous to a conventional bit (binary digit) used by conventional computers.

Furthermore, we discussed the benefits of using the Python programming language for QIP. Additionally, we discussed the installation of the Python programming language and Python packages both on a local machine and in a cloud-based Google Colab environment. We also discussed how package management can be done by using the `pipenv` tool.

Having introduced QIP and all the necessary tools of QIP, the next chapter will cover the basic mathematics necessary to understand the subsequent chapters. Python will be used as a tool for the hands-on demonstrations of the concepts being discussed.

Further reading

- Nielsen, M. A., & Chuang, I. L. (2011). *Quantum computation and quantum information: 10th Edition*. New York, NY, USA: Cambridge University Press, 1107002176, 9781107002173.
- Wilde, M. M. (2017). *Quantum information theory*. Cambridge University Press.
- Khan, F. S., Solmeyer, N., Balu, R., & Humble, T. S. (2018). *Quantum games: a review of the history, current state, and interpretation*. Quantum Information Processing, 17 (11), 309.

Chapter 2: Quantum States, Operations, and Measurements

This chapter covers the basic mathematics that is required to understand the subsequent chapters. We will use Python as a tool for understanding the mathematical concepts covered in this chapter.

We will start with a brief introduction to linear algebra, which forms the basis of quantum information processing. Then we will provide a brief history of quantum mechanics, and link this to the definition of a qubit. Finally, we will cover the quantum operations; both unitary operations and quantum measurements (which are non-unitary).

We will cover the following main topics in this chapter:

- An introduction to linear algebra
- Exploring the history of quantum information processing
- Understanding the qubit
- Dealing with multiple qubits
- The quantum no-cloning theorem
- Quantum computing models – beyond the gate model

Technical requirements

The requirements for this chapter are the following:

- A basic understanding of the Python programming language
- Navigation of Google's Colab environment
- Elementary (post-secondary) mathematics

The GitHub link for this chapter can be found here:

https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter02

The next section provides a brief introduction to linear algebra. Linear algebra plays a central role in quantum information processing.

An introduction to linear algebra

Linear algebra is a field of mathematics that is concerned with the study and the manipulation of vectors. Alternatively, a vector can be thought of as an ordered sequence of numbers. Vectors can be added/subtracted together, or multiplied by a number (scalar), to produce another object of the same type, namely a vector.

A vector usually represents quantities that have both magnitude and direction. It consists of a tuple of one or more scalars, and these scalars are referred to as the components of a vector. For instance, an *n*-tuple vector, *x*, has *n* components and can be written as:

$$x = [x1x2 \cdots xn]$$

Vectors can be written either as rows (row vectors) or as columns (column vectors). The preceding vector x is a row vector. Written as a column vector, a vector y is written as:

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

A concatenation of a vector into two-dimensional space forms a matrix, while a concatenation of matrices and/or vectors into higher-dimensional space forms a tensor. A matrix $Anxm$ is a rectangular array of numbers (called **elements**) with n rows and m columns. The size of a matrix with n rows and m columns is nxm.

It can easily be seen that the preceding vector x is a matrix with 1 row and n columns. Thus, x is a $1xn$ matrix. On the other hand, the preceding vector y is a matrix with n rows and 1 column. Therefore, it is a $nx1$ matrix.

A vector is a one-dimensional array, while a matrix is a two-dimensional array. On the other hand, a tensor is a generalization of a matrix and a vector. It is an n-dimensional array with $n > 2$.

Vectors, matrices, and tensors can be created and manipulated in Python. One of the most prominent Python packages used for linear algebra is **NumPy**. The following code snippet shows how vectors, matrices, and tensors can be created using NumPy:

```
import numpy as np
x = np.array([1,2,3]) # a vector with three components.
y = np.array([4,5,6]) # another vector.
A = np.array([[1,2,3], [4,5,6], [7,5,6]]) # a 3x3 matrix.
B = np.array([[2,3,4], [1,2,0], [3,3,3]]) # another 3x3
    matrix.
v = np.kron(x,A) # a tensor product of vector x and matrix A.
w = np.kron(y,B) # a tensor product of vector y and matrix B.
```

The preceding code snippet can be explained as follows:

1. First, `numpy` is imported using the alias `np`.
2. Then, two vectors, `x` and `y`, are created using the `array()` function from `numpy`.
3. This is followed by the creation of two matrices, `A` and `B`.
4. Finally, two tensor products, `v` and `w`, are created using the `kron()` function from `numpy`.

   The former (tensor product `v`) is the tensor product of vector `x` and matrix `A`, while the latter (tensor product `w`) is the tensor product of vector `y` and matrix `B`.

Arithmetic operations, such as addition, subtraction, and scalar multiplication, can be performed on the vectors. The same is also true of matrices and tensors. The following code snippet shows some examples of arithmetic operations on vectors, matrices, and tensors:

```
# Arithmetic operations
#vectors
z = x + y # vector addition
z = x - y # vector subtraction
z = 2 * z # scalar multiplication
#matrices
C = A - B # matrix multiplication
C = x * A # a product of a vector and a matrix
#tensors
u = v + w # tensor addition
u = 5 * u # multiplication of a tensor by a scalar
```

The preceding code snippet shows basic operations that can be performed on vectors, matrices, and tensors. These operations include addition, multiplication, and scalar multiplication.

So far, we have covered the basics of linear algebra, focusing on vectors, matrices, and tensors. In the next section, we will cover vector spaces.

Exploring vector spaces

A **vector space** is a collection of vectors. Let V be a set of vectors on which two arithmetic operations of addition and scalar multiplication are defined. Then, for vectors x, y, z and scalars a, b, the following axioms hold for a vector space, V:

- $x + y \in V$
- $x + y = y + x$
- $x + (y + z) = (x + y) + z$
- V has a zero vector such that for $x \in V, x + 0 = x$
- For every $x \in V$, there is a vector in V denoted by $-x$ such that $x + (-x) = 0$
- $ax \in V$
- $a(x + y) = ax + ay$
- $(a + b)x = ax + bx$
- $a(bx) = (ab)x$
- $1x = x$

Another feature of a vector space is the concept called the **basis**. A set of vectors, B, forms a basis in vector space V, if the following conditions are satisfied:

- Every vector $x \in V$ can be represented as a linear combination of vectors in B.
- All the vectors in B are linearly independent.

Finally, the field of scalars in V is either the real numbers $(\mathbb{R})$ or the complex numbers $(\mathbb{C})$. A complex number is the number z that can be written in the following form:

$$z = a + ib$$

Here, both *a* and *b* are real numbers, and

$$i = -1 - - - \sqrt{}$$

The expression *ib* is called the *imaginary number*. The (complex) conjugate of a complex number is the same complex number with the + sign replaced by the – sign, or vice versa. Thus, the complex conjugate of $z = a + ib$

is $z1 = a - ib$

while the complex conjugate of $z = c - id$ is $z1 = c + id$.

Understanding the properties of a vector

Now let's discuss the properties of a vector. At this point, it is worth noting that a vector can either be a row vector or a column vector. In the former, the elements of a vector are represented by a row, while in the latter, the elements of such a vector are represented by a column:

- **Transpose**: One of the properties of a vector is a transpose. A transpose of a vector, denoted by *T*, transforms a column vector into a row vector, and a row vector into a column vector. Thus, for a vector *x* with *n* components, here are the two equations:

  Equation 1 can be written as follows:

$$[x_1 \quad x_2 \quad \cdots \quad x_n]^T = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

Equation 2 can be written as follows:

$$\begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}^T = \begin{bmatrix} x_1 & x_2 & \cdots & x_n \end{bmatrix}.$$

The first equation indicates that the elements of a row vector (represented by a row) are transformed into the column vector (represented by the column) when the transpose is applied to it. On the other hand, the second equation is the opposite of the first equation. In the second equation, the transpose is applied to the column vector in order to transform it into the row vector:

- **Dot product**: Another property of a vector is a dot product, which is also called an **inner product**. For vectors *x* and *y*, each with *n* components (elements), the dot product (denoted by $\langle x,y \rangle$) is given as:

$$\langle x, y\rangle = \sum_{i}^{n} x_i y_i = x_1 y_1 + x_2 y_2 + \cdots + x_n y_n$$

The output of a dot product is a scalar. The Python code for computing an inner product for two vectors using `numpy` is as follows:

```
import numpy as np
x = np.array([1,2,3]) # a vector with three components.
y = np.array([4,5,6]) # another vector.
# The output of an inner product of vectors is a scalar
a = np.inner(x,y)
print(a)
```

The preceding code snippet imports `numpy` (using the alias `np`). Then it creates two vectors, `x` and `y`. Finally, the inner product of these two vectors is calculated using the `inner()` function from `numpy`:

- **Outer product**: A vector property related to an inner product is an outer product. Unlike an inner product, whose output is a scalar, the output of an outer product is a matrix. Matrices will be covered later in this chapter. The Python code snippet for computing an outer product of two vectors is shown as follows:

  ```
  import numpy as np
  x = np.array([1,2,3]) # a vector with three components.
  y = np.array([4,5,6]) # another vector.
  # The output of an outer product of vectors is a matrix
  a = np.outer(x,y)
  print(a)
  ```

  This code uses `numpy` to create two vectors, `x` and `y`. Then it uses the `outer()` function from `numpy` in order to compute the outer product of vectors `x` and `y`.

- **Norm**: Another property of a vector is a norm. An *Lp*-norm of a vector *x*, denoted by $\|x\|_p$, is defined as:

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right), (1 \leqslant p \leqslant \infty)$$

A norm of a vector characterizes its length. There is a special class of vectors with the property that are shown in the following equation:

These vectors are called **unit vectors**.

- **Orthogonality**: The vectors have another property called **orthogonality**. Two vectors, *x* and *y*, are said to be **orthogonal** if their inner product is zero. Thus, vectors *x* and *y* are orthogonal if $\langle x,y \rangle = 0$. Geometrically, orthogonal vectors are perpendicular. Vectors that are both orthogonal and are unit vectors (have a length of 1) are said to be **orthonormal**.

Understanding the properties and types of matrices

A **matrix** is defined as a two-dimensional array of scalars, with at least one row and at least one column. An *nxm* matrix *A* is defined as a two-dimensional array with *n* rows and *m* columns, and is given as:

$$A_{n\times m} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ \vdots & \vdots & \cdots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix}$$

Now, let's discuss the properties of a matrix:

- **Determinant**: One property of a matrix is its determinant. Determinants of matrices are only defined for square matrices, where the number of rows is the same as the number of columns. The determinant of a matrix *A*, denoted by *det(A)* or $|A|$, is a scalar.

  The Python code snippet for computing the determinant of a matrix is given as follows:

```
import numpy as np
A = np.array([[1,2,3], [4,5,6], [1,5,1]]) # a 3x3 matrix.
det_A = np.linalg.det(A)
print(det_A)
```

  The preceding code uses the `linalg.det()` function from `numpy` in order to compute the determinant of the matrix `A`:

- **Trace**: Matrices also have a property called trace. For a matrix A, the trace of A, denoted by $T^r(A)$, is given as a sum of the diagonal elements of A.
- **Transpose**: Like vectors, matrices also have transposes. A transpose of a matrix A, denoted by A^T, swaps the row elements of A with its column elements. Hence, the following applies:

$$[A_{m\times n}]^T = [A_{n\times m}].$$

A Python code for a matrix transpose is shown as follows, using **`numpy`**:

```
import numpy as np
A = np.array([[1,2,3], [4,5,9], [5,2,1]]) # a 3x3 matrix.
det_A = np.linalg.det(A)
np.transpose(A)
```

The preceding code uses the **`transpose()`** function from **`numpy`** in order to compute the transpose of the matrix **`A`**.

- **Eigenvalues and Eigenvectors**: The last property of matrices worth discussing is that of eigenvalues and eigenvectors. Let V be the vector space over a field. Also, let A be a matrix. Then, a scalar, $\lambda \in \mathbb{F}$, is called an eigenvalue of A if there exists a non-zero vector, $x \in V$, such that:

$$Ax = \lambda x,$$

with vector x being called the eigenvector of A.

Now, let's focus on various types of matrices. These types are discussed as follows:

- **Identity matrix**: One of the examples of matrices is the identity matrix, denoted by I. An identity matrix has a property such that, for any matrix A, multiplication by I does not change A. That is:

$$A * I = I * A = A$$

The diagonal elements of an identity matrix are $1^{'s}$, while the off-diagonal elements are $0^{'s}$. Another example of a matrix is an inverse matrix:

- **Inverse matrix**: A matrix A has an inverse matrix, denoted by A-1, such that:

$$A * A - 1 = A - 1 * A = I$$

The inverse of a matrix A exists only if the determinant of A is not 0.

- **Diagonal matrix**: Finally, for a field in $\mathbb{R}$, a diagonal matrix Q is a matrix such that:

$$QT * Q = Q * QT = I$$

For a field in $\mathbb{C}$, there are three key matrices worth mentioning in this chapter. They are the **Hermitian adjoint**, **Hermitian matrix**, and **unitary matrix**. Let's talk about each of these in turn.

- **Hermitian adjoint**: A Hermitian adjoint of matrix *A*, denoted by *A†*, is a conjugate transpose of *A*.
- **Hermitian matrix**: On the other hand, a Hermitian matrix, denoted by *H*, is a square matrix (thus, the number of columns is the same as the number of rows) with the property that:

$$H \dagger = H$$

This means that the adjoint of a Hermitian matrix is the same as the Hermitian matrix itself.

- **Unitary matrix**: A unitary matrix, denoted by *U*, is the square matrix, *Unxn*, which consists of orthonormal columns. Additionally, a matrix *U* is unitary if, and only if:

$$U = eiH$$

Here, *H* is a Hermitian matrix. Another property of a unitary matrix is the following:

$$U \dagger U = UU \dagger = I$$

Understanding the properties of a tensor product

As already stated, a tensor is a generalization of a vector and a matrix to higher dimensions. For the purposes of this book, we are only interested in two properties of a tensor. These properties are the **trace** of a tensor product, and the **partial trace** of a tensor product.

For matrices *A* and *B*, a tensor product of *A* and *B*, denoted by *T*, is given as:

$$T = A \bigotimes B$$

A Python code for computing the tensor product of matrices *A* and *B* using both `qutip` and `numpy` is shown as follows:

```
from qutip import *
import numpy as np
A = np.array([[2,3], [4,5]])
B = np.array([[2,4], [1,0]])
A = Qobj(A); B = Qobj(B)
# computing tensor product using qutip
```

```
C = tensor(A,B); print(C)
#computing tensor product using numpy
D = np.kron(A,B); print(D)
```

The code uses two modules, namely, **`qutip`** and **`numpy`**. First, two matrices *A* and *B* are created using **`numpy`**. These matrices are also converted into **`qutip`** objects by using the **`Qobj()`** function. Then, the tensor product is computed using both **`qutip`** and **`numpy`**. For the former, the **`tensor()`** function is used, while for the latter, the **`kron()`** function is used:

- **Trace**: A trace of a tensor *T*, which is a tensor product of matrices *A* and *B*, is given as:

$$Tr(T) = Tr(A) * Tr(B)$$

- **Partial trace**: On the other hand, the partial trace of a tensor product traces over a part of the tensor product. Thus, for matrices *A* and *B*, which form the tensor product:

$$T = A \otimes B$$

either *A* or *B* can be traced out such that:

$$A = TrB(T) = TrB(A \otimes B) = A * Tr(B)$$

and

$$B = TrA(T) = TrA(A \otimes B) = B * Tr(A)$$

A Python code for calculating both the trace and the partial trace of a tensor product using **`qutip`** is shown in the following code block:

```
from qutip import *
import numpy as np
A = np.array([[2,3], [4,5]])
B = np.array([[2,4], [1,1]])
# convert the arrays A and B to qutip objects
A = Qobj(A); B = Qobj(B)
T = tensor(A,B);
print("The tensor product is", T)
print("The trace of T is", T.tr())
Tr_B = T.ptrace(0) # trace out B by selecting A; thus A Tr(B)
print("Tr_B is", Tr_B)
Tr_A = T.ptrace(1) # trace out A by selecting B; thus B Tr(A)
```

```
print("Tr_A is", Tr_A)
```

The code uses two Python modules, `qutip` and `numpy`. First, two matrices, `A` and `B`, are created using `numpy`. Then, these matrices are converted into `qutip` objects. This is followed by computing the tensor product of matrices `A` and `B`, and this tensor product is assigned to `T`. Then, the trace of `T` is computed using the `tr()` function from `qutip`. Finally, using the `ptrace()` function from `qutip`, the partial trace is computed.

So far, we have provided a brief introduction to linear algebra. We have introduced the vectors, the matrices, and the tensors. Furthermore, we have provided the properties of vectors, matrices, and tensors. In the next section, we will provide a brief history of quantum information processing.

Exploring the history of quantum information processing

In order to grasp the history of quantum information processing, it is imperative to explore the development of **quantum mechanics** (theory) first. This is due to the fact that quantum mechanics forms the theoretical basis for quantum information processing.

Quantum mechanics is a conceptual framework that is used to describe the microscopic physical reality. The development of quantum theory started at the beginning of the 20th century, and this development was led by physicists including Niels Bohr, Albert Einstein, and Max Planck.

The development of quantum mechanics was motivated in part by the failure of classical physics to adequately account for the behavior of sub-atomic objects such as electrons. The understanding of nature at a sub-atomic level led to the massive success of the electronics and conventional computing industries. This technological achievement is referred to as the **first quantum revolution**.

Quantum information processing is billed as the **second quantum revolution**, since it follows the success of the first quantum revolution just discussed.

Quantum information processing arose in response to the limitations imposed on conventional computers by the fundamental laws of physics. The first person to explore these limitations on conventional computing was Rolf Landauer, who was working at IBM at the time. He started these investigations in the early 1960s. Landauer's work focused on the application of the laws of thermodynamics to the field of computing.

In the 1970s, Landauer's work was extended by Charles Bennett, who was his (Landauer's) colleague at IBM. Bennett demonstrated that it is possible to perform a two-way reversible computation. A two-way reversible computation means that given the outputs of a computation, it is possible to use those outputs and the circuit used for computation in order to obtain the inputs of the computation.

It is worth noting that a two-way reversible computation implies that information is not lost during the computation. As we will see later, there is a set of quantum operations that is reversible. Also, the set of quantum operations that is not reversible is a quantum measurement. This will also be discussed later in this chapter.

Besides the focus on the limitations imposed on conventional information processing, the earlier stages of quantum information processing also focused on the use of quantum theory for applications in the security sector.

In the 1960s, Stephen Wiesner explored the use of quantum mechanics in order to make banknotes that would be impossible to counterfeit. Unfortunately, Wiesner's idea did not gain any acceptance within the scientific community. It was not until the 1980s that his idea was used as an inspiration to pursue another field of quantum information processing.

In the 1980s, inspired by Wiesner's idea of banknotes that cannot be counterfeited, Charles Bennett and Gilles Brassard developed a cryptographic scheme that uses the laws of quantum mechanics to securely communicate information. This security scheme later came

to be known as the **Bennett Brassard 1984 (BB84) quantum key distribution (QKD) protocol**. The QKD protocol is used to securely share the cryptographic key between the legitimate communicating parties, with the guarantee that the presence of an eavesdropper would be detected.

In 1991, quite independently from the work of Bennett and Brassard, Artur Ekert used quantum theory to develop a new QKD protocol, which was later to be known as the **E91 protocol**. Although the BB84 and E91 QKD protocols are based on the different aspects of quantum theory, it was established in 1992 by Charles Bennett, Gilles Brassard, and David Mermin that the two QKD protocols are mathematically equivalent.

The idea of a quantum computer was pursued in earnest in the 1980s by the likes of Yuri Manin, Paul Bernioff, and Richard Feynman. Due to his popularity, especially in the 1980s, Feynman's ideas of using the laws of quantum mechanics to perform computing were widely accepted in the scientific community, especially in physics.

In 1985, David Deutsch proposed the first ever computational model for quantum computing and demonstrated that such a quantum computing machine could theoretically outperform the conventional quantum computing machine.

For the remainder of the 1980s, quantum computation was just regarded as an area of academic curiosity, without any practical relevance. However, this view would be challenged and eventually defeated in 1994, when Peter Shor proposed a quantum algorithm that could potentially compromise the conventional cryptosystems that were widely used at the time.

Two years later, in 1996, Lov Grover proposed a quantum search algorithm that outperformed any known conventional search algorithm. These two key results renewed interest in quantum computing by the scientific community.

Even though there was a renewed interest in quantum computing, one major obstacle remained. Quantum systems are very fragile. They can easily interact with the environment, and hence easily lose their *quantumness* before completing the computation.

It is worth noting that this interaction with the environment by the quantum system can induce errors in the computation, rendering the possibility of physically implementing a quantum computer virtually impossible. However, this virtual impossibility changed with the independent development of **quantum error correction (QEC)** schemes by Peter Shor and Andrew Steane in 1995 and 1996, respectively.

At the turn of the 21st century, encouraged by the development of QEC schemes, the attention of the quantum information processing community switched to the physical technologies that can be used to implement quantum computing. Various technologies were explored.

These technologies include nuclear magnetic resonance, trapped atoms and ions, photonic devices, quantum dots, and superconducting circuits. To date, no single physical technology has proven to be the best among the rest. Therefore, different technologies are still pursued, with varying degrees of preference.

Building a full-scale quantum computer is still a challenge that is yet to be overcome. Currently, quantum computers with limited computational capabilities are already in use. These quantum computers are known as **noisy, intermediate-scale quantum (NISQ)** computers. They are called *noisy* because they do not implement error correction.

The lack of error correction in NISQ computers means that the circuits that can be implemented on them are very shallow. NISQ computers are currently being used to tackle various computational tasks, with applications in materials science, drug discovery, machine learning, and exploration of tasks where a NISQ computer offers a *quantum super-advantage*.

Having provided a brief history of quantum information processing, it is now imperative to provide information on the basic unit of quantum information. This, I will do in the next section.

Understanding the qubit

Just like in conventional information processing, where the **binary digit (bit)** is the basic unit of information, the **qubit (quantum bit)**, which is a quantum analogue of a bit, is also the basic unit of quantum information.

Unlike a bit, which exists in either of two states (*0 or 1*), a qubit can exist in a superposition of these two states. It is defined as a vector in a two-dimensional complex **Hilbert space**, with the Hilbert space being defined as a vector space equipped with an inner product.

As already stated in the previous chapter, a qubit |***ψ***> is mathematically written as:

$$\psi\rangle = \alpha 0\rangle + \beta 1\rangle,$$

with the complex coefficients ***α*** and ***β*** satisfying the relation:

$$|\alpha|^2 + |\beta|^2 = 1.$$

Any qubit |***ψ***> can be written as a linear combination of two basis vectors (states), |*0*> and |*1*>. This explains why we say that the qubit can exist in a superposition of states.

Superposition is one of the key properties of quantum theory that is harnessed in quantum information processing. Even though a qubit can exist as a linear combination of the basis states, as we will see later, the measurement forces the qubit to be either state *0* or state *1*.

The basis vector |*0*> is given as:

$$0\rangle = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

while on the other hand, the basis vector |*1*> is given as:

$$1\rangle = \begin{bmatrix} 0 \\ 1 \end{bmatrix}.$$

Now that we have a basic understanding of a qubit, I will cover some of the operations that can be performed on the qubit. This will be done in the next subsection.

Qubit operations

For an isolated quantum system (which does not interact with the environment), there are two classes of quantum operations that can be performed on a qubit. These classes of quantum operations are called the **quantum gates**. The first class of quantum operations is the **unitary class**, which is a class of **reversible quantum gates**.

The second class of quantum gates is the **measurement gates**, which is a class of **irreversible gates**. Since measurement is irreversible, the initial quantum state cannot be recovered after measurement. The measurement forces the qubit to collapse to either state *0* or state *1*.

As already stated, any *2X2* unitary matrix, which is also called a unitary operator in the language of quantum mechanics, is a single-qubit quantum gate. Some examples of the single-qubit quantum gates are the **Pauli gates**. These Pauli gates are *X, Y, and Z gates* (there are actually four Pauli gates. The other one is the *2x2 identity matrix I*, which leaves the qubit unchanged).

X gate

The *X gate*, which is also called the *NOT gate*, flips the state of the qubit. Thus, it turns state |*0*> into state |*1*>, and turns state |*1*> into state |*0*>. This means that:

$$X0\rangle = 1\rangle$$

and

$$X1\rangle = 0\rangle.$$

The *X gate* is defined by the following matrix:

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

It can then be observed that:

$$X0\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix} = 1\rangle$$

and

$$X1\rangle = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 0 \end{bmatrix} = 0\rangle.$$

The Python code for implementing the *X gate* in Python using **numpy** and **qutip** is shown here:

```
from qutip import *
import numpy as np
A = np.array([[1], [0]])
B = np.array([[0], [1]])
Rho_x = sigmax()
A = Qobj(A)
B = Qobj(B)
C = Rho_x * A
D = Rho_x * B
print(C)
print(D)
```

The preceding code uses **qutip** and **numpy**. Then, two vectors **A** and **B** are created using the **array()** function from **numpy**. Then, a new variable called **Rho_x** is assigned to the *X gate* using the **sigmax()** function from **qutip**. Finally, two variables, **C** and **D**, are assigned to the product of **Rho_x** and **A**, and the product of **Rho_x** and **B**. The variable **C** is equivalent to applying the *X gate* to **A**, while **D** is equivalent to applying the *X gate* to **B**.

Z gate

Another single-qubit Pauli gate is the *Z gate*. This gate leaves |*0*> unchanged, but changes the sign of |*1*>. The *Z gate* is defined by the following matrix:

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

It can be observed that:

$$Z0\rangle = 0\rangle$$

and

$$Z1\rangle = -1\rangle.$$

The Python code for implementing a *Z gate* is shown as follows:

```
from qutip import *
import numpy as np
```

```
A = np.array([[1], [0]])
B = np.array([[0], [1]])
Rho_z = sigmaz()
A = Qobj(A)
B = Qobj(B)
C = Rho_z * A
D = Rho_z * B
print(C)
print(D)
```

The preceding code snippet uses the `qutip` and `numpy` Python modules to demonstrate the application of the *Z gate* to two vectors, **A** and **B**, and the results of these operations are given as **C** and **D**, respectively.

Y gate

The *Y gate* is another single-qubit Pauli gate. It is defined by the following matrix:

$$Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}.$$

The assessment of the effect of the *Y gate* on a qubit, and the corresponding Python code to implement the *Y gate* using Python, are left as an exercise for you to do.

Another single-qubit quantum gate is the Hadamard gate, which is denoted by *H*. The Hadamard gate is defined by the following matrix:

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

It should be noted that the Hadamard gate can also be written in terms of the *X* and the *Z Pauli gates*. This is given as:

$$H = \frac{1}{\sqrt{2}} [X + Z].$$

Lastly, another group of the single-qubit gates is the **phase-shift group**, which is denoted by *R*, with the *R gate* being defined by the following matrix:

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

with

$$0 \leqslant \phi \leqslant \pi.$$

It is left as an exercise for you to show that the *Z gate* is a special type of the **R group** of quantum gates.

Two other examples of the R group of quantum gates are the *S gate* and the *T gate*. The *S gate* is defined by the following matrix:

$$S = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{2}} \end{bmatrix}$$

which can also be reformulated as:

$$S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}.$$

On the other hand, the *T gate* is defined by the following matrix:

$$T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\frac{\pi}{4}} \end{bmatrix}.$$

We have come to the end of our discussion on unitary (reversible) quantum computation. The next subsection explores quantum measurement. You will recall that unlike unitary operation, quantum measurement is not reversible.

Quantum measurement

Besides a single-qubit unitary quantum operation, another type of a single-qubit quantum operation is the quantum measurement. As already mentioned, quantum measurement collapses the quantum state of a qubit to a classical state. Thus, it is through the quantum measurement that we could be able to read the classical information from a quantum state.

A measurable physical property is called an observable, denoted by *O*, and is represented by a Hermitian operator. The eigenvalues of these Hermitian operators are real numbers. This is consistent with the fact that the output of a measuring device is a real number.

Consider a quantum system with *n* quantum states. In such a system, there exists *n* measurements, denoted by mn with *n = 1, 2, ..., n*. The measurement operator in such a system is denoted by Mm, and the probability *p(m)* of measuring *m* for state |*ψ*> is given as:

$$p(m) = \langle\psi|M_m^\dagger M_m|\psi\rangle$$

The probabilities *p(m)* are normalized. This means the following:

$$\sum_{m=1}^{n} p\,(m) = \langle\psi|\,\textstyle\sum_{m=1}^{n} M_n^\dagger\, M_M|\psi\rangle = 1.$$

This then implies that these measurement operators satisfy the completeness relation:

$$\sum_{m=1}^{n} M_m^\dagger\, M_M = I.$$

Types of measurement

There are two basic types of measurement. These are as follows:

- **Projective measurement (von Neumann) measurement**
- **Positive operator valued measurement (POVM)**

For projective measurements, the set of measurement operators $\{Mm\}$ is given by the projection operators, Pm. For a qubit, using the standard computational basis set, Pm is a set:

$$P_1 = 0\rangle\langle 0.$$

and

$$P_2 = 1\rangle\langle 1.$$

The projection operator $P1$ has the property that it does not affect the state |*0*>, but discards the state |*1*>. On the other hand, $P2$ has the property that it discards the state |*0*> and does not affect the state |*1*>.

As an example of the application of projective measurement, consider the quantum state:

$$\psi\rangle = \frac{1}{\sqrt{2}}[0\rangle + 1\rangle]$$

and the projection operators:

$$P_1 = 0\rangle\langle 0$$

and

$$P_2 = 1\rangle\langle 1.$$

It is worth noting that in this case:

$$P_1 = P_1 \dagger$$

and

$$P_2 = P_2 \dagger.$$

Then, the probabilities of measuring *0* (denoted by *p(0)*) and *1* (denoted by *p(1)*) are given by:

$$p(0) = \langle\psi|P_1 \dagger P_1|\psi\rangle = \frac{1}{2}$$

and

$$p(1) = \langle\psi|P_2 \dagger P_2|\psi\rangle = \frac{1}{2}.$$

Another basic type of quantum measurement is the **positive operator valued measurement** (**POVM**). It is more powerful and general than the projective measurement. Furthermore, POVM is the most widely used type of measurement in quantum information processing.

Let $\{Em\}$ be a set of measurement operators that do not necessarily have to be projectors, such that for any $\{Em\}$ and a normalized quantum state |***ψ***>, we have the following:

$$\langle\psi|E_m|\psi\rangle \geq 0.$$

These Hermitian operators also satisfy the completeness relation:

$$\sum_m E_m = I.$$

The operators *Em* are called the POVM elements, and the complete set { *Em* } is called the positive operator valued measurement.

In this section, I introduced single-qubit operations. I also introduced single-qubit quantum gates. In the next section, I will focus on multiple qubits, instead of the single qubit.

Dealing with multiple qubits

Recall that a qubit is a vector in a two-dimensional complex Hilbert space. Therefore, a quantum system with two or more qubits, say *n* qubits, can also be represented as a vector in a *2n*-dimensional complex Hilbert space.

Quantum states made up of two or more qubits are said to be composite quantum states. Additionally, a composite quantum state of two or more qubits is a tensor product of such states. Thus, if |***ψ***> is the composite state of *n* qubits, then we have the following:

$$\psi\rangle = \psi_1\rangle \otimes \psi_2\rangle \otimes \cdots \otimes \psi_n\rangle.$$

The preceding state |***ψ***> is also called the joint state of *n* qubits. If the joint state can be expressed as a tensor product of the qubits as previously, it is said to be **separable**.

On the other hand, if the composite quantum state is not separable, it is said to be **entangled**. This section will only focus on separable composite states, and entanglement will be covered in the next chapter (*Chapter 3*, *Entanglement and Quantum Teleportation*).

The dimension of the Hilbert space, $\mathscr{H}$, of a joint state of *n* qubits is mathematically expressed as:

$$dim(H) = dim(H1) \times \cdots \times dim(Hn)$$

Since a qubit is in a two-dimensional Hilbert space, it follows then that the joint state of *n* qubits will be in *2n*-dimensional Hilbert space.

Any quantum composite state can be written as a linear combination of the orthogonal basis states. Consider a two-qubit quantum system, and recall that the basis state for a single qubit consists of a set *{|0>, |1>}*. Then, the computational basis set of a two-qubit state composite quantum system has a basis set consisting of the following elements:

$$00\rangle = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix},$$

$$01\rangle = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix},$$

$$10\rangle = \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix},$$

and

$$11\rangle = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

The preceding four-dimensional basis set means that any two-qubit quantum state can be written as a linear combination of these four computational basis states. Thus, any two-qubit composite quantum state $|\boldsymbol{\psi}>$ can be written as:

$$\psi\rangle = \alpha_{00}00\rangle + \alpha_{01}01\rangle + \alpha_{10}10\rangle + \alpha_{11}11\rangle$$

such that:

$$|\alpha_{00}|^2 + |\alpha_{01}|^2 + |\alpha_{10}|^2 + |\alpha_{11}|^2 = 1.$$

Similarly, a three-qubit composite quantum state can be written as a linear combination of the $2^3 = 8$ computational basis states. This state is given as:

$$\psi\rangle = \alpha_0 000\rangle + \alpha_1 001\rangle + \alpha_2 010\rangle + \alpha_3 011\rangle + \alpha_4 100\rangle + \alpha_5 101\rangle + \alpha_6 110\rangle + \alpha_7 111$$

such that:

$$|\alpha_0|^2 + |\alpha_1|^2 + |\alpha_2|^2 + |\alpha_3|^2 + |\alpha_4|^2 + |\alpha_5|^2 + |\alpha_6|^2 + |\alpha_7|^2 = 1$$

The preceding examples can be generalized to any separable joint state. Thus, any n-qubit joint quantum state can be given as:

$$\sum_{i=o}^{2^n-1} \alpha_i \, i\rangle$$

with the condition that:

$$\sum_{i=0}^{2^n-1} |\alpha_i|^2 = 1.$$

Just like in the case of single-qubit quantum states, quantum operations can also be applied to a composite quantum state. As already stated, these quantum operations are also referred to as quantum gates, and they can either be unitary or non-unitary (measurement).

Quantum gates on multiple qubit systems

Quantum gates can be applied either on a single qubit or on more than one qubit of the composite. As an example, let's consider a three-qubit state, *|000>*. This state, as can be seen, is a tensor product of three qubits in the *|0>* state. Thus, we have the following:

$$000\rangle = 0\rangle \otimes 0\rangle \otimes 0\rangle.$$

On the other hand, the tensor products of three qubits in the *|1>* state is given as:

$$111\rangle = 1\rangle \otimes 1\rangle \otimes 1\rangle.$$

As an exercise for you, write the tensor products of the following three-qubit quantum systems:

- *|0>, |0>, and |1>*

- *|0>, |1>, and |0>*
- *|1>, |1>, and |0>*
- *|1>, |0>, and |0>*
- *|0>, |1>, and |1>*
- *|1>, |0>, and |1>*

Now, consider applying the *X gate* to the right-most qubit of one of the aforementioned joint states, say, state *|000>*. This will flip the right-most qubit, while leaving the other qubits unchanged. Since the two other qubits remain unchanged, it means that the two identity operators (matrices) are operating on them. This operation is given as:

$$I \otimes I \otimes X\rangle 000\rangle = 001\rangle.$$

It can then be observed that applying the *X gate* on the second qubit of *|000>* can be represented as:

$$I \otimes X \otimes I\rangle 000\rangle = 010\rangle$$

while applying it (*X gate*) to the left-most qubit of *|000>* would result in:

$$X \otimes I \otimes I\rangle 000\rangle = 100\rangle.$$

In general, any single-qubit operation (gate) can be applied to any qubit of the *n-qubit* composite quantum state by using a tensor product of *n-1* identity operators and that single-qubit operation. The identity operators will be in all the qubit positions except the position on which the single-qubit gate is being applied.

As an exercise, show the results of applying the *X gate* to each of the qubits of the composite quantum state, *|01010>*.

CNOT gate

Besides single-qubit quantum gates, there are also two-qubit and three-qubit quantum gates. The most prominent two-qubit gate is the **controlled NOT (CNOT) gate**. The CNOT gate uses the two-qubit computational bases states. It uses the first qubit as a control qubit and the second qubit as the target qubit. It operates as follows. If the control qubit is *|0>*, it leaves the target qubit unchanged. However, if the control qubit is *|1>*, it flips the target qubit. Thus, by applying the CNOT gate, we have the following:

$$CNOT 00\rangle \rightarrow 00\rangle,$$

$$CNOT 01\rangle \rightarrow 01\rangle,$$

$$CNOT10\rangle \rightarrow 11\rangle,$$

and

$$CNOT10\rangle \rightarrow 11\rangle.$$

Mathematically, a CNOT gate is represented as:

$$CNOT = 0\rangle\langle 0 \otimes I + 1\rangle\langle 1 \otimes X.$$

Finally, since a CNOT gate is an operator, its matrix is given as:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}.$$

The CNOT gate can be thought of as applying the *X gate* conditioned on the state of the control qubit. That being said, there are other two-qubit gates that apply the single-qubit operation on the target qubit conditioned on the state of a control qubit.

Examples of these two-qubit conditional gates include a *controlled Z gate* and *controlled S gate*. The operations and matrix representations of these two gates are left as an exercise to the reader.

Swap gate

Another two-qubit quantum gate is the **Swap gate**. As the name implies, the Swap gate swaps the positions of the qubits such that for a computational basis state, *|a,b>, we have the following*:

$$Swapa, b\rangle \rightarrow b, a\rangle.$$

The matrix representation of a Swap gate is as follows:

$$Swap = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Three-qubit gates

The two prominent three-qubit quantum gates are the **Toffoli gate** and the **Fredkin (controlled Swap) gate**. Both these gates are variants of the controlled gate operation.

Toffoli gate

The Toffoli gate uses two control qubits and one target qubit. The target qubit is flipped only when the two control qubits are in the state |*1*>. For a computational basis state |*a,b,c*>*, we have the following*:

$$Toffolia,b,c\rangle \rightarrow a,b,(ab) \oplus c\rangle$$

The matrix representation of a Toffoli gate is given as follows:

$$Toffoli = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}.$$

Fredkin gate

The Fredkin gate uses one qubit as a control qubit and the two qubits as the target qubit. For a three-qubit computational basis |*a,b,c*>, the Fredkin gate is given as:

$$Fredkin 0,a,b\rangle \rightarrow 0,a,b\rangle$$

and

$$Fredkin 1,a,b\rangle \rightarrow 1,b,a\rangle.$$

Finally, the matrix representation of a Fredkin gate is given as follows:

$$Fredkin = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}.$$

I have covered multi-qubit quantum operations in this section. The next section will focus on one of the most important properties of quantum mechanics, namely, the **quantum no-cloning theorem**.

The quantum no-cloning theorem

The quantum no-cloning theorem is one of the features of quantum mechanics that are harnessed for quantum information processing. In order to fully appreciate the significance of this theorem, it is imperative to contrast the quantum mechanical world with the macroscopic world that we live in.

In a macroscopic world, it is possible to make perfect copies of classical information such as images and text. However, this is not the case in the quantum world, thanks to the quantum no-cloning theorem.

As we are dealing with the quantum no-cloning theorem, it is important to clear up one misconception about the theorem that some people might have. This misconception asserts that it is impossible to clone a quantum state. However, this claim is misleading, since two orthogonal quantum states can be cloned.

On the other hand, the quantum no-cloning theorem is concerned with any unknown arbitrary quantum state. Additionally, what is prohibited is the *perfect* cloning of the unknown state; the imperfect cloning of a quantum state is possible.

In essence, the quantum no-cloning theorem asserts that an unknown quantum state cannot be perfectly copied. That is, it is impossible to create an identical quantum copy of an unknown arbitrary quantum state. This theorem is made possible thanks to the linearity of quantum mechanics and the unitarity of quantum evolutions.

The proof of the no-cloning theorem is as follows. Let $|\psi>$ be a pure state to be cloned, and $|e>$ be the initial state of the copy. Then, the initial state will be represented as follows:

$$\psi\rangle \otimes e\rangle.$$

Now, consider the possibility that the perfect quantum state copier exists. This would imply that for a unitary U:

$$U(\psi\rangle \otimes e\rangle)\psi\rangle \otimes \psi\rangle.$$

Therefore, for two quantum states, $|\psi>$ and $|\phi>$:

$$U(\psi\rangle \otimes e\rangle) = \psi\rangle \otimes \psi\rangle$$

and

$$U(\phi\rangle \otimes e\rangle) = \phi\rangle \otimes \phi\rangle.$$

Now, taking the inner product of $|\psi>$ and $|\phi>$, we end up with:

$$\langle\psi|\phi\rangle = ()\psi|\phi\rangle \quad ^{2}.$$

This implies that either:

$$\langle\psi|\phi\rangle = 1$$

or

$$\langle\psi|\phi\rangle = 0.$$

The preceding conditions are met only when $|\psi> = |\phi>$ or $|\psi>$ and $|\phi>$ are orthogonal, respectively, and not for any arbitrary quantum state. This completes the proof of the quantum no-cloning theorem.

The next section discusses quantum computing models beyond the gate model.

Quantum computing models – beyond the gate model

Although it is the most prominent in quantum computing, the quantum gate model discussed thus far is not the only model of quantum computing. There are other models of quantum computing that were proven to be computationally equivalent to the quantum model of quantum computing, up to a polynomial factor.

The quantum computing models to be discussed in this section include the **cluster-based model**, **adiabatic quantum computing model**, and **hybrid quantum computing model**.

Cluster-based quantum computing

Cluster-based quantum computing uses entangled states, which are also known as cluster states, as the physical resources for quantum computation. This quantum computing model is universal for quantum computation. Cluster-based quantum computing is also known as **one-way quantum computing**.

The operation of a cluster-based quantum computing model is as follows. First, a cluster state of a large number of entangled quantum states is prepared. This is then followed by the actual quantum computation. The quantum computation consists of a sequence of one-qubit quantum measurements on this cluster state. The sequence of measurements will be determined by the computing task that is being performed. Therefore, this model of quantum computing uses measurement as a tool to perform quantum computing.

Adiabatic quantum computing

Another model of quantum computing is the adiabatic model of quantum computing. This model is based on the adiabatic theorem of quantum mechanics. The adiabatic theorem states that gradually varying the conditions of the quantum system allows the system to adapt its functional form (configuration).

The adiabatic quantum computing model operates as follows. First, for the time interval *s = [0,1]*, two Hamiltonians, $H0$ and $H1$, are specified (a **Hamiltonian** is an operator corresponding to the total energy (kinetic + potential) in the quantum system). The computational task to be computed is then encoded on the ground state of $H0$. $H0$ is the Hamiltonian that is easy to prepare. After encoding the computational problem on $H0$, this Hamiltonian ($H0$) is then gradually evolved such that we have:

$$H(s) = (1 - s)H0 + sH1$$

where

$$0 \leqslant s \leqslant 1$$

After this gradual (quantum) evolution, the solution to the computational problem will be in the ground state of the final Hamiltonian ($H1$). A quantum evolution is a gradual change in the quantum system over time, mainly due to its interaction with the environment.

Hybrid quantum computing model

The key objective of a hybrid quantum computing model is to demonstrate the quantum super-advantage of the NISQ devices. In this model, computational load is shared between a conventional/classical computing unit and a quantum processing unit.

In a hybrid quantum computing model, the quantum processing unit first prepares a quantum state with a set of variational parameters. In this case, the quantum state that is being prepared through guessing is the Hamiltonian of the quantum system. This guessed Hamiltonian of a quantum system is known as an ansatz.

After creating an ansatz, the quantum processing unit then performs a measurement and sends these measured parameters to the classical processing unit for optimization of these variational parameters. The optimized parameters are then fed back to the quantum processing unit. This process iterates until the ground state of the computational problem's Hamiltonian is found.

The classical-quantum hybrid quantum computing model is normally deployed in finding solutions to optimization problems. There are two prominent classical-quantum hybrid variational algorithms. These are the **quantum approximate optimization (QAOA)** and the **variational quantum eigensolver (VQE)** algorithms.

The QAOA algorithm was developed by Farhi, Goldstone, and Gutmann in 2014. This variational algorithm produces approximate solutions for combinatorial optimization problems in polynomial time. It acts as a bridge between the adiabatic quantum computing model and the quantum gate model, since it uses two Hamiltonians in alternation, and initializes the state of one of the Hamiltonians using the quantum gate model.

The VQE was introduced by Peruzzo et al. in 2014. It is a classical-quantum hybrid variational algorithm that can be deployed to solve for the eigenvalues in a matrix and to find solutions to the optimization problems. The VQE algorithm uses both a classical processing unit and a quantum processing unit in order to find solutions to the optimization problems and eigenvalue problems.

Summary

This chapter delved into the basic mathematics that is required to understand the subsequent chapters. Python was used in this chapter as a tool for understanding the mathematical concepts covered. It provided a brief introduction to linear algebra. This is due to the fact that linear algebra forms the basis for quantum mechanics and, by extension, to quantum information processing.

Additionally, a qubit was introduced, together with the allowed quantum operations that can be performed on the qubit. The quantum no-cloning theorem, which plays a crucial role in quantum information processing, was also discussed. Finally, quantum computing models other than the quantum gate model were discussed.

Having introduced the necessary mathematics for quantum information in this chapter, it is now time to switch attention to the actual quantum information processing algorithms. The next chapter sheds some light on quantum entanglement and quantum teleportation.

Further reading

- Larson, R. (2016). *Elementary linear algebra*. Nelson Education.
- Nielsen, M. A and Chuang, I. L. (2011) *Quantum computation and quantum information: 10th Edition*. New York, NY, USA: Cambridge University Press, 1107002176, 9781107002173.
- Scherer, W. (2019). *Mathematics of Quantum Computing: An Introduction*. Springer Nature.

Section 2: Quantum Computers and Quantum Algorithms

This section covers the concept of quantum entanglement, which forms the core of the correlations that are possible between separated quantum systems. We will also define and show Python implementations of various quantum circuits and algorithms.

This section comprises the following chapters:

- *Chapter 3*, *Entanglement and Quantum Teleportation*
- *Chapter 4*, *Working with Quantum Circuits*
- *Chapter 5*, *Quantum Algorithms*

Chapter 3: Entanglement and Quantum Teleportation

In this chapter, we will cover the fundamentals of **entanglement**, and how it came to be referred to as *spooky action at a distance*. Entanglement plays a crucial role in **quantum information processing**, so it is imperative to have a chapter dedicated to entanglement.

After covering the fundamentals of entanglement, the chapter will discuss the **Bell theorem** and the tests of entanglement. Finally, the chapter will discuss one of the applications of entanglement, namely, **quantum teleportation**.

In this chapter, we will cover the following main topics:

- Exploring the history of quantum entanglement
- Understanding the Bell theorem and CHSH inequality
- Understanding composite systems and entanglement
- Understanding the CNOT gate – the entangling gate
- Understanding Bell states
- Understanding the entanglement of more than two quantum states
- Understanding entanglement as a resource – quantum teleportation

Technical requirements

The requirements for this chapter are the following:

- A basic understanding of the Python programming language
- Navigation of Google's Colab environment
- Elementary (post-secondary) mathematics knowledge

The GitHub link for this chapter can be found here: https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter03.

In the next section, we will cover the brief history of quantum entanglement.

Exploring the history of quantum entanglement

In our day-to-day lives, we are quite familiar with correlations of various affairs. At a sub-atomic level though, correlations of quantum particles go beyond our day-to-day experience. These quantum correlations are referred to as **quantum entanglement**. The term *entanglement* was coined by a German physicist, Erwin Schrödinger, using the German word *Verschränkung*, which he translated to mean *entanglement*.

In essence, quantum entanglement occurs when two quantum particles that have interacted remain a single, indivisible system, even if such quantum systems are separated by an arbitrarily long distance. This way, the quantum particles remain inter-dependent, and hence correlated. This means that for these entangled particles, if an action is performed on one particle, the second particle will also be affected by that action, and this influence occurs instantaneously.

Albert Einstein, who was one of the greatest scientists of the 20th century, was a firm critic of quantum entanglement. He referred to entanglement as *spooky action at a distance* because quantum entanglement suggests an instantaneous correlation between two quantum particles that are separated by an arbitrarily long distance.

In 1935, Einstein, together with his colleagues Boris Podolsky and Nathan Rosen, presented a very serious critique of quantum entanglement. This critique was presented in the form of a thought experiment, which would later be referred to as the **EPR paradox** (also referred to as the Einstein-Podolsky-Rosen paradox).

Through the formulation of the EPR paradox, Einstein, Podolsky, and Rosen used the concept of quantum entanglement to demonstrate that quantum mechanics cannot provide a complete description of reality and that it should be supplemented by additional parameters, the hidden variables.

The EPR paradox can be explained as follows. Consider two correlated quantum particles, Alice and Bob (thus, Alice and Bob are entangled). At first, Alice and Bob are allowed to interact. Then, Alice and Bob are separated such that they are so far apart that it would be impossible for them to communicate with one another.

Since Alice and Bob are entangled, the operations performed on one will also affect the other. Therefore, for instance, if measurement is performed on Alice, the influence of that correlation would be instantaneously reflected by Bob. The paradox in this (in the EPR paradox) is that this would imply that this influence would travel faster than the speed of light, and this is in direct conflict with the known fact that nothing travels faster than light.

The EPR paradox provided a serious challenge to quantum mechanics for nearly three decades. However, in 1964, the physicist John Bell proved that the interpretation given in the EPR paradox is inconsistent with quantum mechanics. This proof later came to be known as the Bell theorem, and will be discussed later in this chapter.

Like Einstein, Podolsky, and Rosen, Bell used a thought experiment in order to develop his theorem. However, since his theorem made testable predictions, these predictions were later tested experimentally. The first experiment was performed by John Clauser and Stuart Freedman in 1972.

Another experiment was conducted by Edward Fry and Randall Thompson in 1976. Furthermore, in the 1980s, Alain Aspect performed an entanglement-based experiment. Finally, in the late 1990s, Anton Zeilinger performed a set of experiments that firmly established the reality of quantum mechanics and hence provided a serious blow to the interpretation given in the EPR paradox.

Having provided a brief history of the concept of quantum entanglement, it is now imperative to further explore this concept. This exploration will be done in the next section, where the Bell theorem and CHSH inequality will be covered.

Understanding the Bell theorem and CHSH inequality

As already stated in the previous section, Bell's theorem renders the quantum mechanical interpretation used in the EPR paradox obsolete. This philosophical interpretation is known as **local realism**. The *realism* in this case posits that objects exist even when they are not observed/measured. For instance, the moon does exist regardless of whether you are looking at it. On the other hand, *local* in *local realism* posits that an event at one point cannot instantaneously have an effect at another point. This interpretation is also known as **the hidden variables** interpretation of quantum mechanics.

In order to address the hidden variables argument, Bell came up with a thought experiment that would test the validity of this interpretation. This thought experiment was later reformulated by the physicist David Bohm, who used the spin properties of quantum mechanical systems such as electrons.

Consider two quantum variables, each with a spin angular momentum. Now, consider the random variables $A^{1\alpha}$ $A^{2\alpha}$ to be the outcomes of the spin measurements made along the axes $\alpha = x, y$, and z, and taking the values *-1* or *1*. Then, if the correlations from these measurements are not quantum mechanical (there is no quantum entanglement), and if the aforementioned random variables are anti-correlated, then for probability *p* (as predicted by the hidden variables interpretation), we have the following:

$$p(A_x^1 \neq A_y^2) + p(A_y^1 \neq A_z^2) + p(A_z^1 \neq A_x^2) \geq 1$$

The inequality in the given equation is known as the **Bell theorem**, or **Bell's inequality**. Any quantum mechanical correlation, quantum entanglement, violates Bell's inequality.

Bell's inequality was later generalized by Clauser, Horne, Shimony, and Holt, in what would later be known as the **CHSH inequality**. The CHSH inequality is briefly summarized as follows.

Consider two quantum particles, Alice and Bob, and the detector measurement settings *a* and a^1 for Alice, and *b* and b^1 for Bob. Then, for expectation values $<A(\alpha)B(\beta)>$ corresponding to Alice measuring in either measurement setting *a* or a^1, and Bob measuring in measurement setting *b* or b^1, then we have the following inequality:

$$\langle A(a)B(b)\rangle + \langle A(a^1)B(b^1)\rangle + \langle A(a^1)B(b)\rangle - \langle A(a)B(b^1)\rangle \geq 2\sqrt{2}$$

The inequality shown in the equation is known as the CHSH inequality. Just like with Bell's inequality, quantum mechanics also violates CHSH inequality.

In this section, we have provided two key inequalities in quantum mechanics, namely, Bell's inequality and CHSH inequality. We have also stated in this section that quantum mechanics violates both these inequalities. The following section will cover composite quantum systems and the entangled quantum systems.

Understanding composite systems and entanglement

We have seen in *Chapter 2*, *Quantum States, Operations, and Measurements*, that composite quantum systems can either be separable or not separable. Let's consider two qubits:

$$u_1\rangle = \alpha_0 0\rangle + \alpha_1 1\rangle$$

and

$$u_1\rangle = \beta_0 0\rangle + \beta_1 1\rangle$$

As we have already seen in the previous section, if the two states are separable, then the composite system of these qubits will be given as follows:

$$u\rangle = \alpha_0\beta_0 00\rangle + \alpha_0\beta_1 01\rangle + \alpha_1\beta_0 10\rangle + \alpha_1\beta_1 11\rangle$$

Since the composite system |u> is separable, it is possible to factorize (decompose) it into its constituent quantum states, namely, $|u_1>$ and $|u_2>$. However, this decomposition is not always possible. For instance, consider the following composite system:

$$v\rangle = \frac{1}{\sqrt{2}}(00\rangle + 11\rangle)$$

Now, we want to see whether it is possible to decompose |v> into its constituent quantum states, say $|v_1>$ and $|v_2>$. Recall that states $|v_1>$ and $|v_2>$ can be written as follows:

$$v_1\rangle = \alpha_0 0\rangle + \alpha_1 1\rangle$$

and

$$v_2\rangle = \beta_0 0\rangle + \beta_1 1\rangle$$

Now, if |v> is a product state, it means the following:

$$v\rangle = \alpha_0\beta_0 00\rangle + \alpha_0\beta_1 01\rangle + \alpha_1\beta_0 10\rangle + \alpha_1\beta_1 11\rangle = \frac{1}{\sqrt{2}}(00\rangle + 11\rangle)$$

This can only be the case if the following apply:

$$\alpha_0\beta_1 = \alpha_1\beta_0 = 0$$

and

$$\alpha_0\beta_0 = \alpha_1\beta_1 = \frac{1}{\sqrt{2}}.$$

However, the two aforementioned equations cannot both be true. The implication of the first equation is that at least one of the probability amplitudes $\boldsymbol{\alpha}_0$, $\boldsymbol{\alpha}_1$, $\boldsymbol{\beta}_0$, or $\boldsymbol{\beta}_1$ is zero, while the second equation implies that neither of the probability amplitudes is zero. This leads to a contradiction. Therefore, the composite state |v> cannot be decomposed to its constituent quantum states, $|v_1>$ and $|v_2>$. Therefore, |v> is not separable and quantum states $|v_1>$ and $|v_2>$ are said to be entangled.

In this section, we have shown how quantum states can be entangled, instead of being separable states. The next section discusses how the **CNOT gate** is used as an entangling gate. The CNOT gate is used to entangle two quantum states.

Understanding the CNOT gate – the entangling gate

In *Chapter 2*, *Quantum States, Operations, and Measurements*, we gave a brief overview of a two-qubit gate called the **controlled-NOT (CNOT) gate**. The CNOT gate can be used for entangling quantum states. It is also referred to as the entangling qubit. As an

example, consider two qubits, $|w_1>$ and $|w_2>$, both in state |0>. Now, when we apply the Hadamard gate (the Hadamard gate was introduced in *Chapter 2*, *Quantum States, Operations, and Measurements*) to $|w_1>$, we get the following:

$$w_1\rangle \rightarrow H0\rangle = \frac{1}{\sqrt{2}}[0\rangle + 1\rangle]$$

Furthermore, by applying the CNOT gate to $|w_1>$ in the equation and $|w_2>$ (which is still in state |0>), we get the following:

$$w_2\rangle \rightarrow CNOT(w_1\rangle, 0\rangle) = \frac{1}{\sqrt{2}}(00\rangle + 11\rangle)$$

This, as we have already observed, is an entangled state. This state is one example of entangled quantum states known as **Bell states**, **Bell basis states**, or **EPR states**.

The Python code for generating this Bell state using **`qutip`** is shown as follows:

```
from qutip import *
w1 = bell_state(state="00")
print(" A matrix for Bell pair generation is:\n", w1)
```

The code uses the **`bell_state()`** function from **`qutip`** to create one of the four Bell states.

The following code snippet shows the output of this code:

```
A matrix for the Bell pair generated is:
Quantum object: dims = [ [2, 2], [1, 1] ],
shape = (4, 1), type = ket
Qobj data =
[ [0.70710678 ]
[0. ]
[0. ]
[0.70710678] ]
```

Let's now move on to the next section and learn about Bell states.

Understanding Bell states

As we have already mentioned, the $|w_2>$ state in the *Understanding the Bell theorem and CHSH inequality* section is an example of a Bell state. There are other examples of Bell states too (there are actually four Bell states). The Bell states are prepared by applying

the Hadamard gate to the first input qubit, and then applying the CNOT gate on the result to the first qubit and the second qubit input. This can be summarized as follows:

$$\psi_{00}\rangle = CNOT(H0\rangle, 0\rangle) = \frac{1}{\sqrt{2}}(00\rangle + 11\rangle)$$

$$\psi_{01}\rangle = CNOT(H0\rangle, 1\rangle) = \frac{1}{\sqrt{2}}(00\rangle - 11\rangle)$$

$$\psi_{10}\rangle = CNOT(H1\rangle, 0\rangle) = \frac{1}{\sqrt{2}}(01\rangle + 10\rangle)$$

$$\psi_{11}\rangle = CNOT(H1\rangle, 1\rangle) = \frac{1}{\sqrt{2}}(01\rangle - 10\rangle)$$

It is left as an exercise for you to prove that the equations provided are correct.

Let's discuss the four Bell states:

- As can be seen, the first Bell state, $|\psi_{00}>$, is created when the two input qubits are in state $|0>$.
- Furthermore, the Bell state $|\psi_{01}>$ is created when the first input qubit is in state $|0>$ and the second input qubit is in state $|1>$.
- Additionally, the Bell state $|\psi_{10}>$ is created when the first input qubit is in state $|1>$ and the second input qubit is in state $|1>$.
- Finally, the Bell state $|\psi_{11}>$ is created when both input qubits are in state $|1>$.

The Python code for generating these Bell states using **`qutip`** is shown as follows:

```
from qutip import *
v_00 = bell_state(state="00")
v_01 = bell_state(state="01")
v_10 = bell_state(state="10")
v_11 = bell_state(state="11")
print("v_00 is:", v_00)
print("v_01 is:", v_01)
print("v_10 is:", v_10)
print("v_11 is:", v_11)
```

The preceding code snippet demonstrates the generation of the four Bell states using the `bell_state()` function from `qutip`.

Finally, the simple quantum circuit for creating a Bell state is shown here (quantum circuits will be discussed in detail in *Chapter 4*, Working with *Quantum Circuits*):

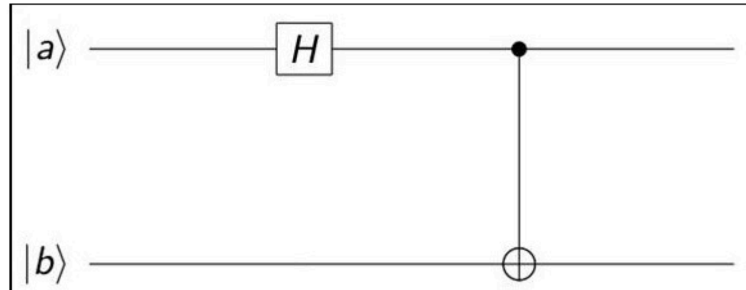


Figure 3.1 – Quantum circuit for creating a Bell state

In the preceding circuit, the Hadamard gate acts on the first qubit (|a>), and the CNOT gate acts on the output of H|a> and the second input qubit (|b>). As an exercise, show that the preceding circuit actually generates the Bell states when the first qubit |a> = {|0>, |1>} and the second qubit |b> = {|0>, |1>}.

In this section, we have demonstrated how Bell states are generated using the CNOT gate. In the next two sections, we will cover the entanglement of more than two quantum states. The next section will focus on the generation of the GHZ state. This will then be followed by a discussion of the generation of the W state.

Understanding the entanglement of more than two quantum states

In the previous section, we demonstrated how Bell states are generated using the CNOT gate. In this section, we will cover the entanglement of more than two quantum states. The next subsection will focus on the generation of the GHZ state. This will then be followed by a discussion of the generation of the W state.

Greenberger-Horne-Zeilinger state (GHZ state)

So far, we have focused on the entanglement of just two quantum states. However, in reality, many qubits can be entangled arbitrarily. One of the states that can be generated by entangling many qubits arbitrarily is the GHZ state. For *n* qubits, with all qubits initialized to |0>, the GHZ state is given as follows:

$$\psi_{GHZ}\rangle = \frac{1}{\sqrt{2}}(0\rangle^{\otimes n} + 1\rangle^{\otimes n})$$

For three qubits, the GHZ state is given as follows:

$$\psi_{GHZ}\rangle = \frac{1}{\sqrt{2}}(000\rangle + 111\rangle)$$

The Python code snippet of a three-qubit GHZ state using `qutip` is shown as follows:

```
from qutip import *
GHZ = ghz_state(N=3)
print(GHZ)
```

Finally, the circuit for the generation of a three-qubit GHZ state, with all the input qubits (|a>, |b>, and |c>) initialized to |0>, is shown as follows:

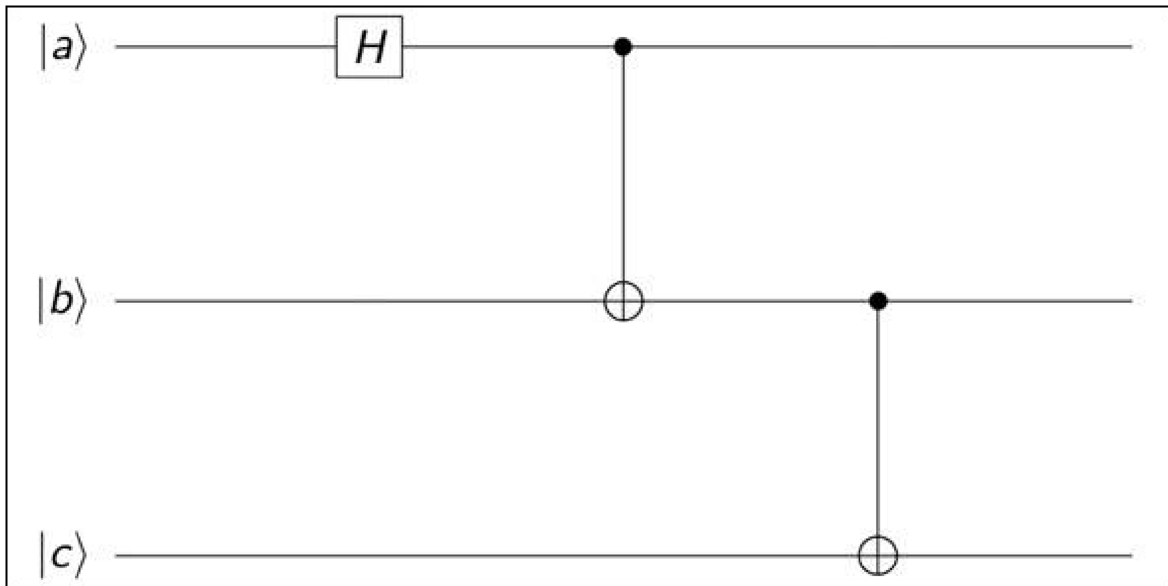


Figure 3.2 – Circuit for the generation of a three-qubit GHZ state

The W state

Besides the GHZ state, another composite quantum state that can be formed by the entanglement of three or more qubits is called the **W state**. A three-qubit W state is mathematically represented as follows:

$$\psi_W\rangle = \frac{1}{\sqrt{3}}(100\rangle + 010\rangle + 001\rangle)$$

In general, the W state generated from entangling *n* qubits is given as follows:

$$\psi_W\rangle = \frac{1}{\sqrt{n}}(10\cdots0\rangle + 01\cdots0\rangle + \cdots + 0\cdots01\rangle)$$

Finally, the Python code snippet for generating the three-qubit W state using **`qutip`** and **`numpy`** is shown here:

```
from qutip import *
import numpy as np
a = np.array([[1], [0]])
b = np.array([[0], [1]])
a = Qobj(a)
b = Qobj(b)
W = (1/np.sqrt(3))* (tensor(b, a, a) + \
tensor(a, b, a) + \
tensor(a, a, b))
print("The W state is:", W)
```

As can be observed, the preceding code uses two modules, namely, **`qutip`** and **`numpy`**. **`numpy`** is used to create two vectors, namely, **`a`** and **`b`**, using the **`array()`** function. These vectors are then converted into the **`qutip`** objects using the **`Qobj()`** function. Then, the W state is generated, and the **`W`** variable is assigned to such a state.

Now that we have discussed many-particle entanglement using either the GHZ states or the W states, the next section will discuss how quantum entanglement can be used as a resource in quantum information processing.

Understanding entanglement as a resource – quantum teleportation

In quantum information processing, entanglement is a crucial resource that is used to provide information processing advantages over conventional information processing. As such, quantum entanglement is typically used in various fields of quantum information processing, such as quantum cryptography, quantum computing, and quantum communication.

In quantum communication, quantum entanglement is used as a resource in applications such as **superdense coding** and **quantum teleportation**. Superdense coding will be covered in the next chapter (*Chapter 4*, Working with *Quantum Circuits*), while quantum

teleportation is covered in this section. It is imperative to note that the teleportation that will be covered here is that of a quantum state, not the one that is normally talked about in sci-fi (science fiction) movies.

Quantum teleportation

Quantum teleportation involves transmitting an arbitrary quantum state from one location to another, with the assistance of an entangled EPR pair. In this quantum communication protocol, quantum entanglement is used as a key resource. As already mentioned earlier, what is teleported in this protocol is a quantum state, and not a classical object such as a human body, as is normally depicted in popular culture.

In quantum teleportation, the objective is to transfer an unknown quantum state from one point to another. This transfer is made possible through the use of an EPR pair, which is shared by Alice and Bob (remote quantum particles) and a conventional communication circuit. Therefore, the objective of quantum teleportation is to enable Alice to transmit an arbitrary quantum state to Bob. In order to achieve this, both Alice and Bob use classical communication to communicate.

The procedure for a quantum teleportation circuit can be summarized by the following circuit. This circuit uses four unitary quantum gates, namely, the Hadamard gate, the *X* gate, the *Z* gate, and the *CNOT* gate, together with two measurement gates:

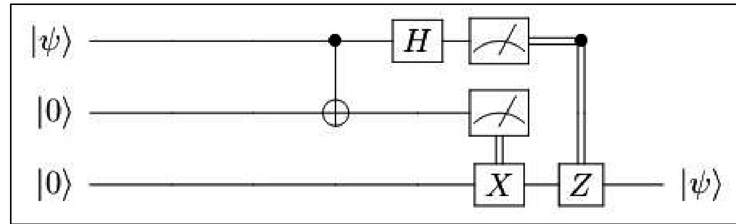


Figure 3.3 – Circuit for quantum teleportation

As can be observed from the diagram, Alice has the unknown state |*ψ*> and one half of the EPR pair that she shares with Bob. She then performs a measurement on the two qubits at her disposal, and sends the results of the measurement to Bob through a classical channel. Since she is measuring two qubit states, Alice's possible measurement outcomes m_1 (from the qubit to be transmitted) and m_2 (from one half of the EPR pair) are 00, 01, 10, or 11.

Based on the value of the measurement outcome received from Alice, Bob can perform any of the operations shown in the following table in order to reconstruct the state |*ψ*>:

m_1	m_2	$Z^{m1}X^{m2}$	Bob's Operation
0	0	I	I
0	1	X	X
1	0	Z	Z
1	1	ZX	ZX

Figure 3.4 – Table showing operations that can be performed by Bob based on the value of measurement outcome received from Alice

Using IBM's **Qiskit** platform and Google's **Colab** environment, the following Python code can be used to implement quantum teleportation:

```
#!pip install qiskit
from qiskit import *
```

```
from qiskit.visualization import plot_histogram
circuit = QuantumCircuit(3,3)
circuit.h(0)
circuit.h(1)
circuit.cx(1,2)
circuit.cx(0,1)
circuit.h(0)
circuit.measure([0, 1], [0, 1])
circuit.cx(1, 2)
circuit.cz(0, 2)
circuit.measure([2], [2])
circuit.draw(output='text')
simulator = Aer.get_backend('qasm_simulator')
result = execute(circuit, backend=simulator,
shots=1024).result()
plot_histogram(result.get_counts(circuit))
```

The code can be summarized as follows. The code uses the **`qiskit`** Python module. After importing the **`qiskit`** module into the workspace, the circuit is instantiated using the **`QuantumCircuit()`** function, and the **`circuit`** variable is assigned to this instantiated circuit. This circuit uses three quantum registers and three classical registers. After instantiating the circuit, the unitary circuits are then applied to the quantum states. Finally, a measurement is performed and the results stored in the classical registers. The final part of the code, starting from the **`circuit.draw(output='text')`** statement to the end, is just used for the visualization of the quantum teleportation circuit generated.

Summary

In this chapter, we have covered the basics of quantum entanglement. We have also seen the inequalities that can be used to determine the *quantumness* of the correlations. These inequalities are the Bell inequality and the CHSH inequality.

Additionally, we introduced the Bell basis states. Furthermore, we learned in this chapter that the CNOT gate is the quantum gate that is responsible for entanglement, hence it can also be referred to as the entangling gate.

In this chapter, we also covered the entangled states of more than two entangled qubits, and these states are the GHZ and W states. Additionally, we argued in this chapter that quantum entanglement is a crucial resource for quantum information processing applications. Finally, we provided a hands-on introduction to quantum teleportation.

The next chapter provides a hands-on exposition to various quantum circuits. It also discusses an implementation of a superdense coding quantum communication protocol.

Further reading

- Wilde, M. M. (2017). *Quantum Information Theory*. Cambridge University Press.

- Bell, J. S. (1987). *Speakable and Unspeakable in Quantum Mechanics*. Cambridge University.
- Bertlmann, R., and Zeilinger, A. (2017). *Quantum [Un] Speakables II*. Berlin: Springer.

Chapter 4: Working with Quantum Circuits

In this chapter, we will define and show implementations of various quantum circuits. We will begin with implementing examples of single-qubit games and also cover quantum circuits. We will also cover error correction techniques and superdense coding examples in this chapter. Finally, in this chapter, we will focus more on the use of IBM's qiskit (www.qiskit.org) and Quantum Experience (https://quantum-computing.ibm.com/).

At the end of this chapter, you should be able to understand what a quantum gate is, how a quantum gate is different from a conventional gate, and how quantum circuits can be formed from the quantum gates. Furthermore, this chapter will expose you to the Python code that can be used to represent quantum gates and quantum circuits.

Just like their classical/conventional circuit counterparts, which illustrate how a **classical** algorithm/program is executed, quantum circuits illustrate how **quantum** algorithms/programs are executed. Additionally, just like in classical computing, where circuits are made up of a collection of logic gates, quantum circuits are made up of a collection of quantum gates.

We will cover the following main topics in this chapter:

- Introducing classical logic gates
- Introducing single-qubit and multi-qubit gates
- Introducing quantum circuits
- Exploring quantum error correction
- Exploring superdense coding

Technical requirements

The requirements for this chapter are the following:

- A basic understanding of the Python programming language
- Navigation of Google's Colab environment

The GitHub link for this chapter can be found here:

https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter04

Introducing classical logic gates

Let's briefly detour from quantum computing and focus on classical computing. You will recall that the basic unit of information in classical computing is a **binary digit** (**bit**). Furthermore, you will recall that a bit can be in a state of either a zero (0) or one (1). Just as a bit is a basic unit of classical information, a logic gate is a basic unit of classical computation.

There are two single-bit logic gates. These gates are the `Identity` (`wire`) gate and the `NOT` gate. The former leaves the input unchanged, while the latter inverts/negates the input. The `Truth` table for the single-bit identity gate is given as follows:

Input	Output (Identity)
0	0
1	1

Table 4.1 – The Truth table for a single-bit identity gate

The Python code for implementing the identity gate is shown as follows:

```
#Identity gate
def IDTY(a):
if a == 0:
return 0
else:
return 1
if __name__ == '__main__':
print(IDTY(0))
print(IDTY(1))
```

As can be seen from the preceding code, after defining the function that can be used to represent the identity, the next step is to output the **Identity** gate when the inputs are **0** and **1**, respectively. This output is shown as follows:

```
The output of input 0 is:
0
The output of input 1 is:
1
```

On the other hand, the **Truth** table of the **NOT** gate is given as follows:

Input	Output (NOT)
0	1
1	0

Table 4.2 – A Truth table of a NOT gate

The Python code for implementing the **NOT** gate is shown as follows:

```
#NOT gate
def NOT(a):
if a == 0:
return 1
else:
return 0
```

```
if __name__ == '__main__':
print(NOT(0))
print(NOT(1))
```

The output of the **NOT** gate is shown as follows:

```
The output of NOT gate for input 0 is:
1
The output of NOT gate for input 1 is:
0
```

Besides the two one-bit gates, there are also various multi-bit logic gates. The two basic two-bit logic gates are the **OR** and **AND** gates. For two inputs **A** and **B**, the **Truth** table for the **OR** gate is given as follows:

A	B	Output (OR)
0	0	0
0	1	1
1	0	1
1	1	1

Table 4.3 – A Truth table of a two-bit OR gate

The Python code for the **OR** gate is shown as follows:

```
#OR gate
def OR(a,b):
if a == 0 and b == 0:
return 0
else:
return 1
if __name__ == '__main__':
print(OR(0,0))
print(OR(0,1))
print(OR(1,0))
print(OR(1,1))
```

The output for the **OR** gate is shown as follows:

```
The output of OR gate for inputs 0,0 is:
0
The output of OR gate for inputs 0,1 is:
1
```

```
The output of OR gate for inputs 1,0 is:
1
The output of OR gate for inputs 1,1 is:
1
```

On the other hand, the **Truth** table of the **AND** gate (given inputs **A** and **B**) is given as follows:

A	B	Output (AND)
0	0	0
0	1	0
1	0	0
1	1	1

Table 4.4 – A Truth table for a two-bit AND gate

The Python code for the **AND** gate is given as follows:

```
#AND gate
def AND(a,b):
if a == 1 and b == 1:
return 1
else:
return 0
if __name__ == '__main__':
print(AND(0,0))
print(AND(0,1))
print(AND(1,0))
print(AND(1,1))
```

The output of the **AND** gate is shown as follows:

```
The output of AND gate for inputs 0,0 is:
0
The output of AND gate for inputs 0,1 is:
0
The output of AND gate for inputs 1,0 is:
0
The output of AND gate for inputs 1,1 is:
1
```

Another multi-bit logic gate of interest is the exclusive-**OR** (**XOR**) gate. For two input bits **A** and **B**, the **Truth** table of the **XOR** gate is given as follows:

A	B	Output (XOR)
0	0	0
0	1	1
1	0	1
1	1	0

Table 4.5 – A Truth table of a two-bit XOR gate

The Python code for the **XOR** gate is given here:

```
#XOR gate
def XOR(a,b):
if a != b:
return 1
else:
return 0
if __name__ == '__main__:
    print(XOR(0,0)
    print(XOR(0,1)
    print(XOR(1,0)
    print(XOR(1,1))
```

The output of the **XOR** gate is shown as follows:

```
The output of XOR gate for inputs 0,0 is:
0
The output of XOR gate for inputs 0,1 is:
1
The output of XOR gate for inputs 1,0 is:
1
The output of XOR gate for inputs 1,1 is:
0
```

Finally, there are two other multi-bit logic gates that are worthy of our attention. These logic gates are the **NOR** gate and the **NAND** gate. In essence, the **NOR** gate is a *negation* of the **OR** gate, while the **NAND** gate is a *negation* of the **AND** gate. Furthermore, both the **NOR** and **NAND** gates are called **universal gates** because either of them can be used to construct any other logic gate. For two input bits **A** and **B**, the **Truth** table of a **NOR** gate is given as follows:

A	B	Output (NOR)
0	0	1
0	1	0
1	0	0
1	1	0

Table 4.6 – A Truth table of a two-bit NOR gate

The Python code for the **NOR** gate is given as follows:

```
#NOR gate
def NOR(a,b):
if a == 0 and b == 0:
return 1
else:
return 0
if __name__ == '__main_':
    print(NOR(0,0))
    print(NOR(0,1))
    print(NOR(1,0))
    print(NOR(1,1))
```

The output of the **NOR** gate is shown as follows:

```
The output of NOR gate for inputs 0,0 is:
1
The output of NOR gate for inputs 0,1 is:
0
The output of NOR gate for inputs 1,0 is:
0
The output of NOR gate for inputs 1,1 is:
0
```

On the other hand, the `Truth` table for the **NAND** gate (with input bits **A** and **B**) is given as follows:

A	B	Output (NAND)
0	0	1
0	1	1
1	0	1
1	1	0

Table 4.7 – A Truth table of a two-bit NAND gate

So far, in this section, we have provided an introduction to conventional logic gates. These gates include the **identity** gate, the **NOT** gate, the **OR** gate, the **AND** gate, the **XOR** gate, the **NOR** gate, and the **NAND** gate. Furthermore, the Python codes for implementing most of these gates were provided. In the next section, we will introduce quantum gates.

Introducing single-qubit and multi-qubit gates

Most of the multi-bit classical logic gates are not reversible. That is, given the output(s), it is not possible to determine what the inputs are. However, as we have already learned earlier in this chapter, all the quantum gates are reversible. This, as we have already seen earlier in *Chapter 2*, *Quantum States, Operations, and Measurements*, is due to the fact that all quantum gates are represented by the unitary matrices:

1. We have already seen earlier in this chapter that there are only two single-bit logic gates, namely, the identity gate and the **NOT** gate. However, in the quantum realm, there are an infinite number of single-qubit quantum gates. These single-qubit quantum gates, as already discussed in *Chapter 2*, *Quantum States, Operations, and Measurements*, correspond to any **2X2** complex unitary matrix. Additionally, there are an infinite number of multi-qubit quantum gates.

We discussed a few single-qubit quantum gates back in *Chapter 2*, *Quantum States, Operations, and Measurements*. One of the single-qubit quantum gates discussed was the Pauli **X** (**NOT**) gate. For the computational basis states |*0*> and |*1*>, the **Truth** table of the **X** gate is given as follows:

Input	Output (X Gate)
|0>	|1>
|1>	|0>

Table 4.8 – A Truth table of the X gate

On the other hand, for the same computational states |*0*> and |*1*>, and the input states *x* and *y*, the **Truth** table of a two-qubit **CNOT** gate (recall that **CNOT** is the entangling qubit) is given as follows:

Inputs		Outputs	
x	y	x	X XOR y
|0>	|0>	|0>	|0>
|0>	|1>	|0>	|1>
|1>	|0>	|1>	|1>
|1>	|1>	|1>	|0>

Table 4.9 – A Truth table of a two-qubit CNOT gate

It is important to note that since quantum computing is reversible, the number of inputs is the same as the number of outputs. This then ensures that there is no information that is lost during the computation. Additionally, it can be seen from the `CNOT` table in *Figure 4.9* that the operation of a `CNOT` gate is analogous to the operation of the `XOR` classical logic gate. The key difference though is that in the case of quantum computing, the input *x* also forms part of the output, so there is no information lost (due to the reversibility of computation).

We have discussed universal gates in the context of classical computing, whereby we established that both `NOR` and `NAND` logic gates are universal gates for classical computing. Analogous to classical computing, in quantum computing, the `CNOT` gate and any single-qubit gate form a set of universal quantum gates.

In this section, I have covered the quantum logic gate – both single-qubit and multi-qubit quantum gates. These gates can be connected together to form quantum circuits. Quantum circuits will be covered in the next section.

Introducing quantum circuits

We briefly came across a certain class of quantum circuits in the previous chapter, *Chapter 3*, *Entanglement and Quantum Teleportation*. These circuits are the entanglement-based circuits, and they include the **Bell** state preparation circuits and the **GHZ** circuits. We learned that the `CNOT` gate plays a crucial role in such a class of quantum circuits, namely, the entanglement-based circuits.

In essence, a quantum circuit is made up of a sequence of quantum gates. Typically, a quantum circuit consists of a set of unitary evolution operators followed by the measurement operators. That is, the measurement is typically performed at the end of the computation.

The first circuit to consider is the circuit that introduces superposition to the quantum state. This circuit uses the **Hadamard** gate in order to bring about the superposition of the quantum state. The Python code for implementing this circuit, using `qiskit`, is shown as follows:

1. The first step involves importing the packages that are required in this implementation. After importing the Python packages, the seed is set, so as to enable reproducibility of the results obtained:

```
#Import packages required
from qiskit import QuantumCircuit, QuantumRegister,\
ClassicalRegister, execute, Aer
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt
import numpy as np
np.random.seed(42)
```

2. The next step in this implementation of the quantum circuit is to actually define the circuit using **`QuantumCircuit()`**. This circuit has one classical register and one quantum register. After defining the circuit, the qubit, which is initially at state |*0*> (it is initialized to state |*0*>), is inverted using the Pauli **`X`** gate. After inverting the gate, the **`Hadamard`** gate is then applied. Finally, the state of the quantum system is measured. This is achieved by applying the measurement gate to the qubit. The application of the measurement gate completes the construction of this quantum circuit, which can later be implemented using either the actual quantum computer or the classical simulator:

```
# Define the Quantum and Classical Registers
qr = QuantumRegister(1)
cr = ClassicalRegister(1)
# Build the circuit
sup = QuantumCircuit(qr, cr)
sup.x(qr)
sup.h(qr)
sup.measure(qr, cr)
sup.draw(output='mpl')
plt.show()
```

3. Finally, the quantum circuit defined previously is simulated using the **`qasm_simulator`** from **`qiskit`**. The results obtained are then displayed:

```
# Execute the circuit (using the qasm simulator)
job = execute(sup, backend = \
              Aer.get_backend('qasm_simulator'),
              shots=1024)
result = job.result()
# Print the result
print(result.get_counts(sup))
#plot the results
counts = result.get_counts(sup)
plot_histogram(counts)
plt.show()
```

Furthermore, the block diagram of this circuit is shown as follows.

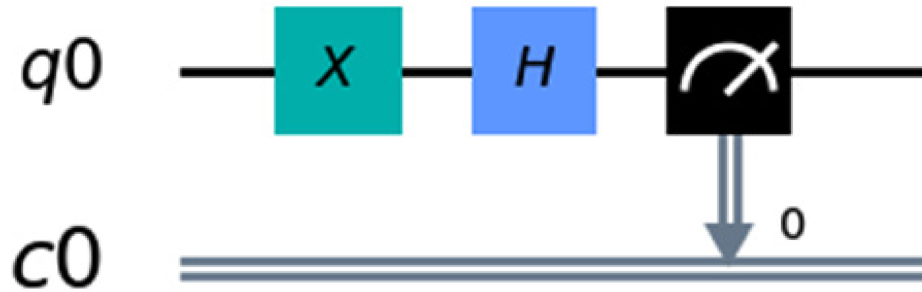


Figure 4.1 – Circuit introducing superposition to the quantum state

From *Figure 4.1*, note that the qubit is initialized to |*0*>. So, the Pauli **X** gate inverts the |*0*> qubit to |*1*>, and the **Hadamard** gate (**H**) introduces the superposition. Finally, the measurement operator is applied, and the results are stored in a classical register.

Finally, the following results prove that the given circuit is indeed the superposition circuit:

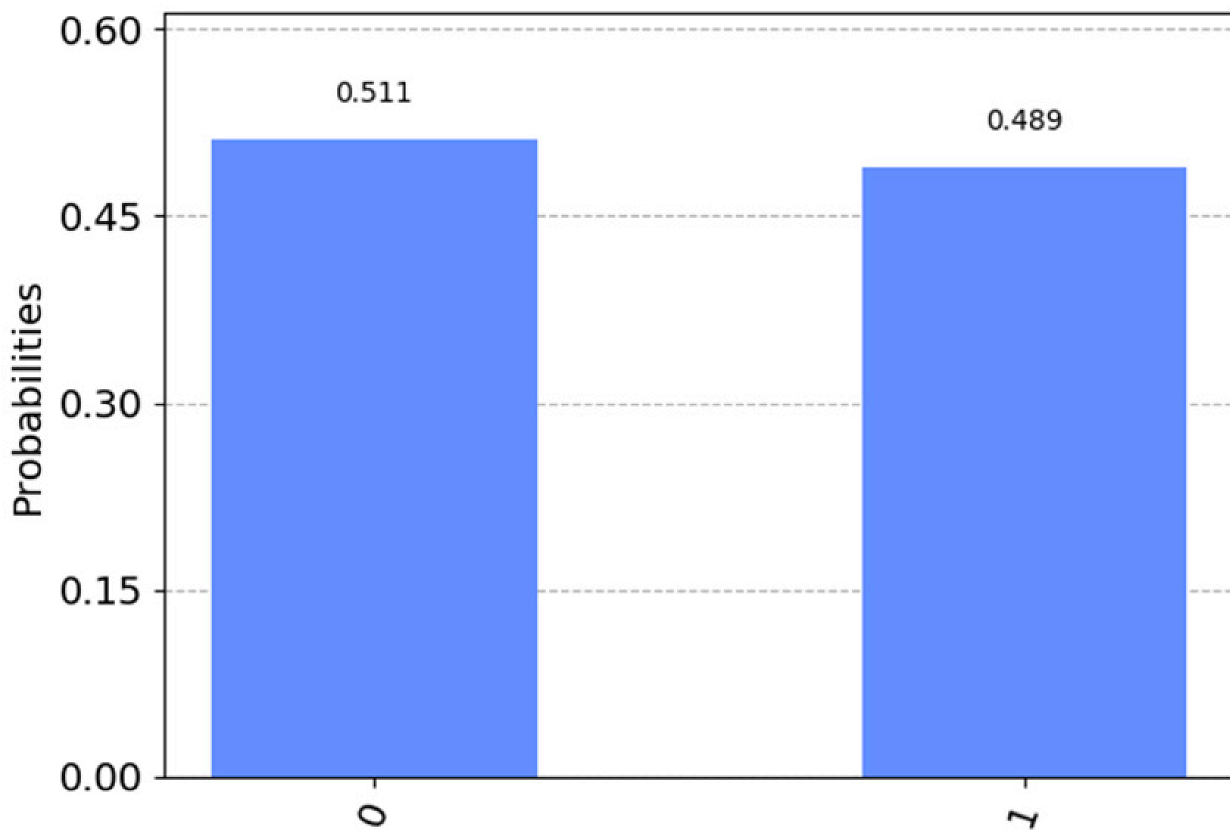


Figure 4.2 – The results of the simulation of the superposition circuit

Another example of a quantum circuit is shown here. This circuit is generated using IBM's Quantum Experience platform and Circuit Composer tool. The circuit uses the **Hadamard** gate, the Pauli **X** gate, and the measurement operator. Initially, the quantum state is initialized to |*0*>. Then, the **Hadamard** gate is used to introduce quantum superposition. This step is followed by the application of the **X** gate to the resultant state. Finally, measurement is applied to the final state. It can be observed that all the gates used in this circuit (**Hadamard**, **X** and measurement) are single-qubit gates:

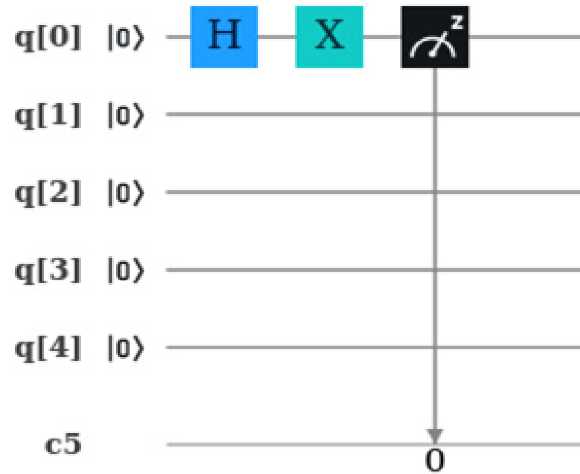


Figure 4.3 – The quantum circuit generated using IBM's Quantum Experience and Circuit Composer tool

The results after performing measurements, and running the experiment 1,024 times, are summarized as follows:

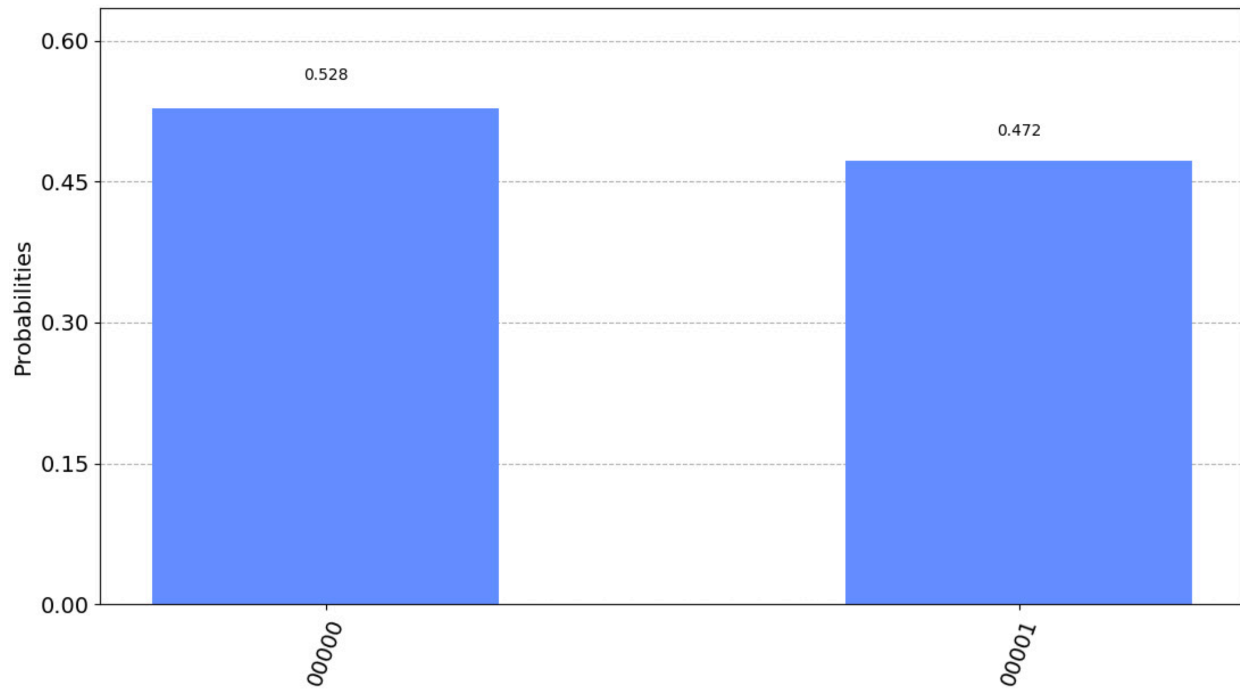


Figure 4.4 – The simulation results obtained from the implementation of the quantum circuit implemented using Quantum Experience and Circuit Composer (depicted previously in Figure 4.3)

From the results, it can be observed that there is roughly a 50% chance of obtaining either `0` or `1` after performing measurement. It is worth noting that this circuit was not run on an actual quantum computer, but rather, on IBM's classical simulator.

The following circuit shows the quantum circuit used for the Bell state preparation:

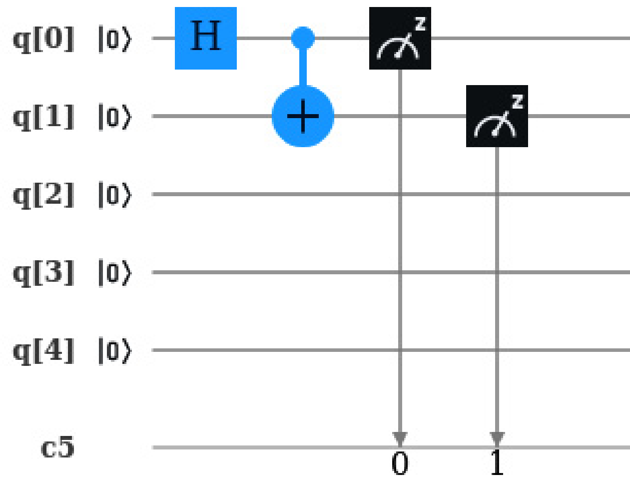


Figure 4.5 – The quantum circuit for implementing the Bell state preparation

Additionally, the results of running this experiment using IBM's classical simulator are summarized as follows:

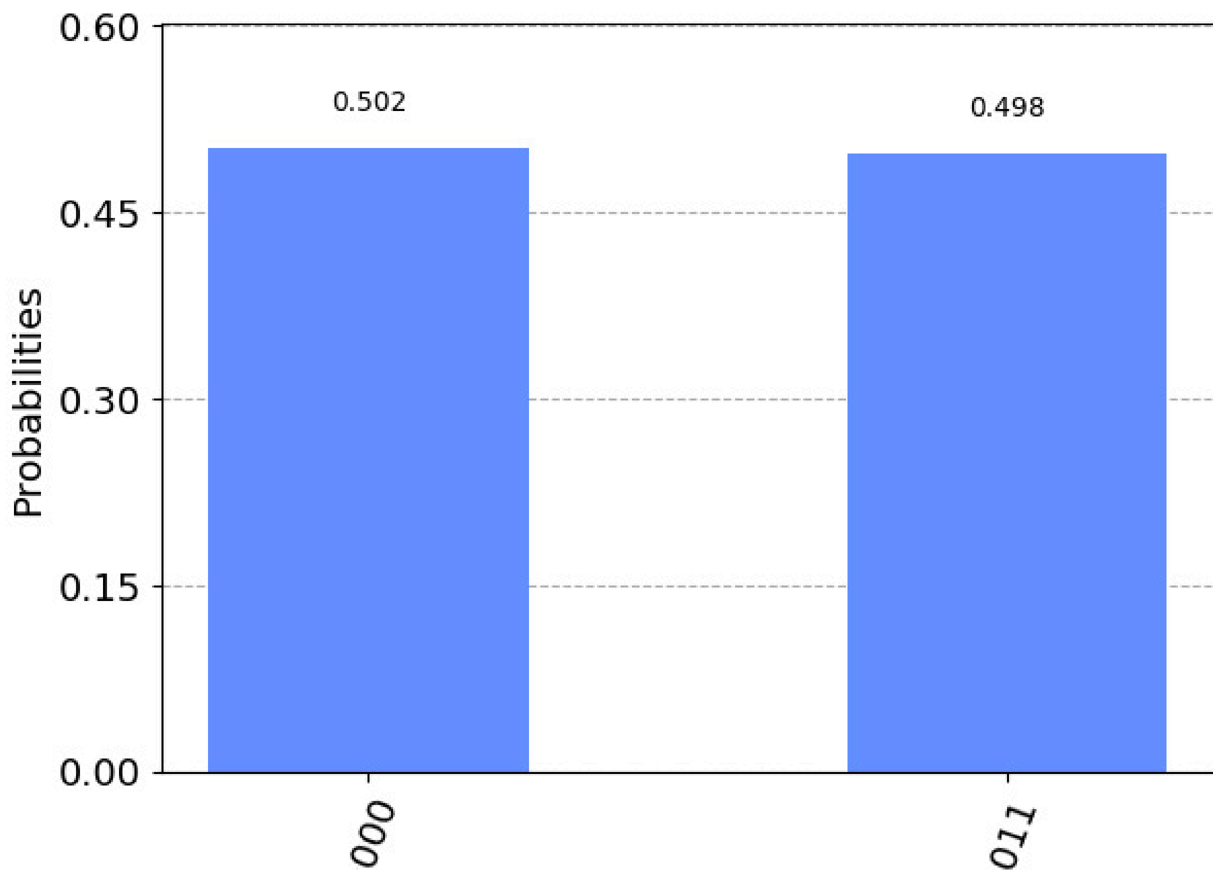


Figure 4.6 – The results obtained from simulating the Bell state preparation circuit using IBM's classical simulator

Finally, once again using Circuit Composer, the 5-qubit quantum circuit can be represented as follows:

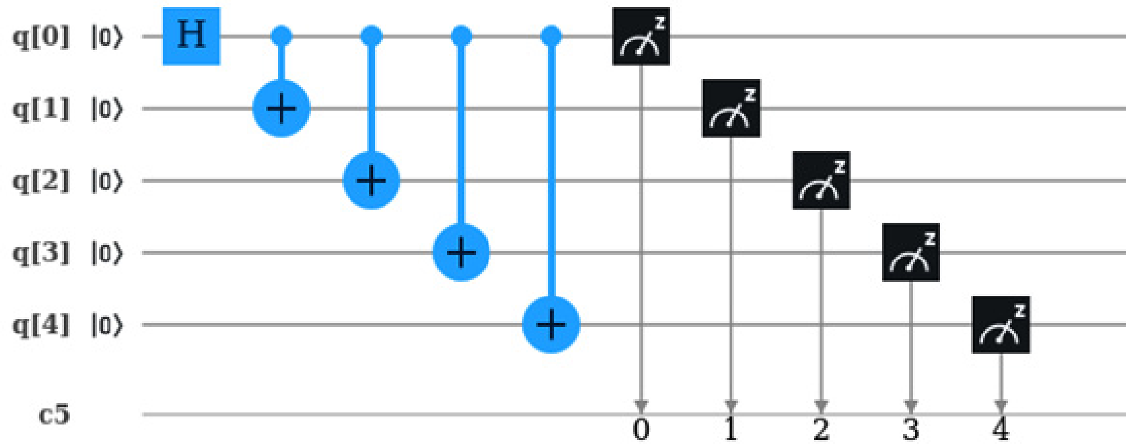


Figure 4.7 – The 5-qubit quantum circuit using Hadamard, CNOT, and measurement gates

As we have observed in *Chapter 3*, *Entanglement and Quantum Teleportation*, the preceding circuit should return all zeros half the time, and all ones half the time, too. The results obtained after running this circuit using the IBM's classical simulator are shown as follows:

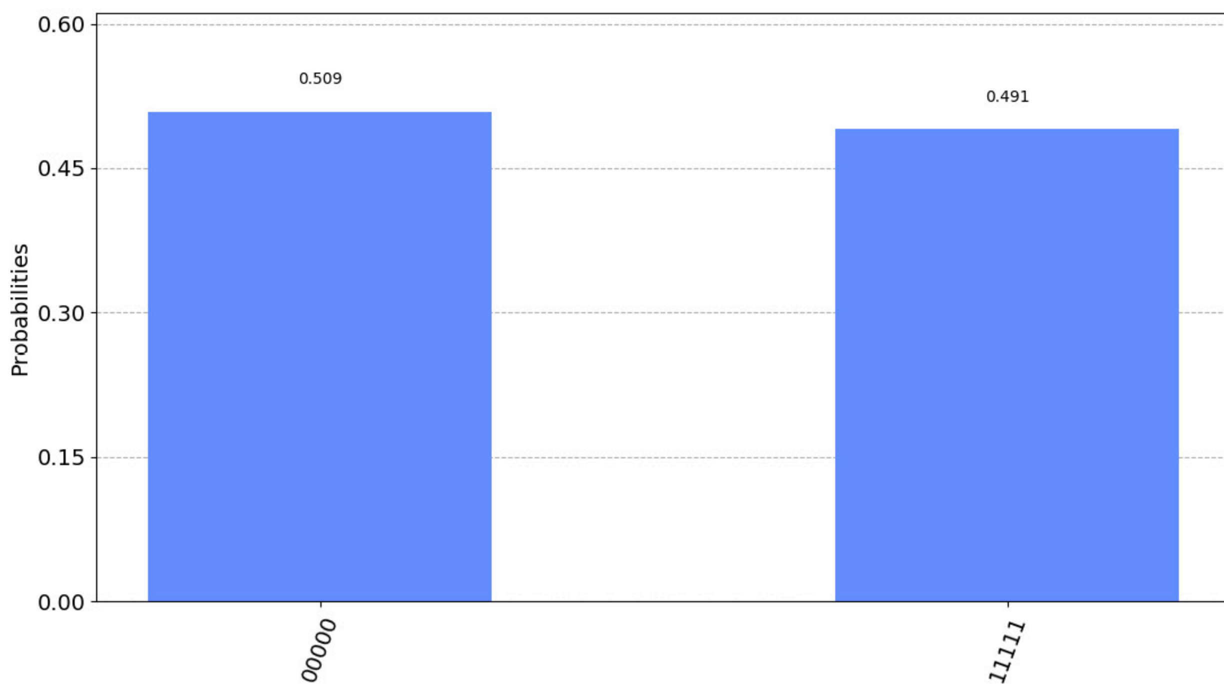


Figure 4.8 – The results obtained from simulating the 5-qubit quantum circuit depicted in Figure 4.7, using IBM's classical simulator

In this section, I have covered the quantum circuits that can be formed by connecting various quantum gates. The next section discusses another concept that is of the utmost importance in quantum information processing, namely, quantum error correction. Quantum error correction is used to correct some of the errors that may arise in quantum circuits.

Exploring quantum error correction

Before delving into quantum error correction, it is prudent to first consider error correction in the classical/conventional information and coding theory.

In classical information and coding theory, error-correction codes are used to mitigate the effects of noise in digital systems. The key idea of error-correction codes is to ensure that errors are detected and corrected from the received data. In essence, error correction is realized by encoding the message in such a way that the redundant information is added to the message. The addition of redundancy to the message is to ensure that even if such a message is affected by noise, there would be enough information that can be used in order to recover/decode the original message.

One of the basic classical error-correction codes is the **majority vote** code. This code is also referred to as the **repetition code**. The objective of majority vote error correction is to detect and correct a single-bit flip. As an example, using a three-bit majority vote error-correction code, bits `'0'` and `'1'` can be encoded by the computer as follows:

```
'0' (physical bit) -> '000' (logical '0')
'1' (physical bit) -> '111' (logical '1')
```

It is clear to see that the coding can correct a single-bit flip. For instance, the bit sequences `'001'`, `'010'`, and `'100'` can be error-corrected using the majority vote to `'000'`, which, in turn, can be decoded to `'0'`. On the other hand, using the majority vote error-correction code, the bit sequence `'110'`, `'101'`, and `'011'` can be error-corrected to `'111'`, which, in turn, can be decoded to `'1'`.

The task of the error-correction code is to consider the bits, and decide, based on the majority rule, whether the encoding message should be `'000'` or `'111'`. If there are more zeros than ones, then the coded message becomes `'000'`. On the other hand, if there are more ones than zeros, then the coded message becomes `'111'`.

Just like in classical information theory and coding, there is also a need to detect and correct errors in quantum information theory and coding. Quantum error correction codes generalize the classical repetition codes. However, since quantum states are also sensitive to the phase. The quantum repetition codes can correct either the qubit flip or the phase flip.

The circuit diagram of a qubit-flip quantum error-correction encoding circuit is shown in the following diagram:

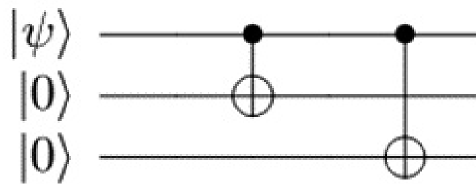


Figure 4.9 – The quantum circuit for encoding the qubit |ψ> using two redundant qubits, with both redundant qubits being initialized to |0>

Essentially, the preceding circuit encodes the physical qubit:

$$|\psi> = \alpha|0> + \beta|1>$$

with the logical qubit – $\alpha|000> + \beta|111>$. The two other qubits, initialized to $|0>$, are used to provide redundancy, in order to enable the recovery of the original quantum states. These qubits are called **auxiliary qubits**. Alternatively, the auxiliary qubits are referred to as **ancilla qubits**, an unfortunate expression that will not be used in this book.

On the other hand, the circuit diagram for a qubit phase-flip quantum error-correction encoding circuit is shown as follows:

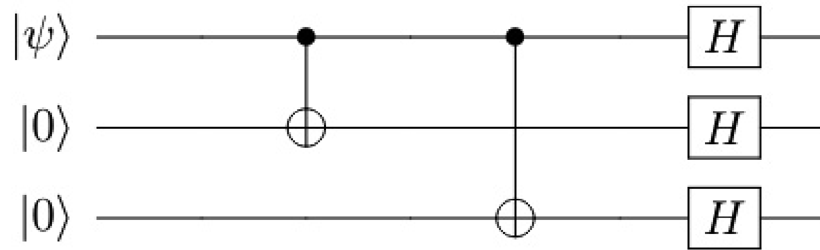


Figure 4.10 – The quantum circuit for implementing the qubit phase-flip quantum error-correction encoding

Finally, the Python code for implementing quantum repetition code error correction using **`qiskit`** is given as follows:

NOTE

The link for this code can be found here: https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter04.

1. The first step is to import all the packages that are required for the implementation of this quantum error correction scheme:

```
from qiskit import *
from qiskit.ignis.verification.topological_codes\
import RepetitionCode
from qiskit.ignis.verification.topological_codes\
import GraphDecoder
from qiskit.ignis.verification.topological_codes\
import lookuptable_decoding, postselection_decoding
from qiskit.compiler import transpile
from qiskit.transpiler import PassManager
from qiskit import QuantumCircuit, execute, Aer
from qiskit.providers.aer import noise
from qiskit.providers.aer.noise import NoiseModel
from qiskit.providers.aer.noise.errors import\
pauli_error, depolarizing_error
from qiskit import QuantumRegister, ClassicalRegister
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(42)
```

2. The following stage defines the noise model that will be used in this quantum circuit. The error model simulates the physical effects that may affect the performance of the qubits. This noise model uses two forms of error, namely, the quantum depolarizing noise (which replaces the qubit with a random state), and the measurement noise. This noise model is implemented with 5% probability of each type of noise (depolarizing noise and measurement noise):

```
def get_noise(p_meas,p_gate):
    error_meas = pauli_error([('X',p_meas),
```

```
                                ('I', 1 - p_meas)])
    error_gate1 = depolarizing_error(p_gate, 1)
    error_gate2 = error_gate1.tensor(error_gate1)
    noise_model = NoiseModel()
    noise_model.add_all_qubit_quantum_error(
        error_meas, "measure")
    noise_model.add_all_qubit_quantum_error(
        error_gate1, ["x"])
    noise_model.add_all_qubit_quantum_error(
        error_gate2, ["cx"])
    return noise_model
noise_model = get_noise(0.05,0.05)
```

3. The qubit phase-flip quantum error correction scheme is defined in this stage. The circuit uses three quantum registers and three classical registers:

```
qc0 = QuantumCircuit(3,3,name='0')
qc0.measure(qc0.qregs[0],qc0.cregs[0])
```

4. The circuit constructed is then simulated using the classical simulator and the noise model defined earlier. The results obtained (in terms of counts) are then displayed:

```
counts = execute(qc0, Aer.get_backend('qasm_simulator'),
                 noise_model=noise_model)\
                 .result().get_counts()
print(counts)
```

5. The next step involves the creation and simulation of another quantum circuit, and this time the qubit being measured is in state |*1*> instead of state |*0*>. Therefore, the **X** gate is used to invert all three qubits from state |*0*> to state |*1*>:

```
qc1 = QuantumCircuit(3,3,name='0')
qc1.x(qc1.qregs[0])
qc1.measure(qc1.qregs[0],qc1.cregs[0])
counts = execute(qc1,
                Aer.get_backend('qasm_simulator'),
                noise_model=noise_model)\
                .result().get_counts()
print(counts)
```

6. Now, let's repeat the simulation of the preceding circuit, but this time increasing the probability of measurement from 5% (0.05) to 50% (0.5):

```
noise_model = get_noise(0.5,0.0)
counts = execute(qc1,
                Aer.get_backend('qasm_simulator'),
                noise_model=noise_model)\
```

```
                 .result().get_counts()
print(counts)
```

7. We then introduce a means of tracking errors in our quantum error-correction circuit. We do so by introducing the complementing (or 'ancilla') qubit – which is always initialized to state |*0*>, the output of which (the complementing qubit) is collected as the syndrome bit. Ultimately, the circuit consists of two code qubits and one complementing qubit:

```
cq = QuantumRegister(2,'code\ qubit\ ')
lq = QuantumRegister(1,'auxiliary\ qubit\ ')
sb = ClassicalRegister(1,'syndrome\ bit\ ')
qc = QuantumCircuit(cq,lq,sb)
qc.cx(cq[0],lq[0])
qc.cx(cq[1],lq[0])
qc.measure(lq,sb)
qc.draw(output='mpl')
qc_init = QuantumCircuit(cq)
(qc_init+qc).draw(output='mpl')
```

8. The circuit (with the complementing qubit and the syndrome bit) is then simulated, and the output of the circuit displayed:

```
counts = execute(qc_init+qc,
                Aer.get_backend('qasm_simulator'))\
                .result().get_counts()
print('Results:',counts)
```

9. The error correction circuit is then simulated, first with code qubits in the state |*11*>, and then in a superposition of |*00*> and |*11*>:

```
qc_init = QuantumCircuit(cq)
qc_init.x(cq)
(qc_init+qc).draw(output='mpl')
counts = execute(qc_init+qc,
                Aer.get_backend('qasm_simulator'))\
                .result().get_counts()
print('Results:',counts)
qc_init = QuantumCircuit(cq)
qc_init.h(cq[0])
qc_init.cx(cq[0],cq[1])
(qc_init+qc).draw(output='mpl')
counts = execute(qc_init+qc,
                 Aer.get_backend('qasm_simulator'))\
                 .result().get_counts()
print('Results:',counts)
```

10. Then we define a circuit for repetition code, with three repetitions and one syndrome measurement. This is achieved by using **`RepetitionCode()`**:

```
code = RepetitionCode(3,1)
for reg in code.circuit['0']\
.qregs+code.circuit['1'].cregs:
    reg.name = reg.name.replace('_','\ ') + '\ '
code.circuit['0'].draw(output='mpl')
code.circuit['1'].draw(output='mpl')
```

11. The next step is to run the repetition code circuit on the **`'qasm_simulator'`**. This simulation is run without any addition of noise:

```
def get_raw_results(code,noise_model=None):
    circuits = code.get_circuit_list()
    raw_results = {}
    for log in range(2):
    job = execute(circuits[log],
                     Aer.get_backend(
                          'qasm_simulator'),
                     noise_model=noise_model)
    raw_results[str(log)] = \
        job.result().get_counts(str(log))
    return raw_results
raw_results = get_raw_results(code)
for log in raw_results:
    print('Logical',log,':',raw_results[log],'\n')
code = RepetitionCode(3,1)
```

12. The next step is to run the repetition code circuit on the **`'qasm_simulator'`**, but this time with the noise added (**`p_meas`** = 5% and **`p_gate`** = 5%):

```
noise_model = get_noise(0.05,0.05)
raw_results = get_raw_results(code,noise_model)
for log in raw_results:
    print('Logical',log,':',raw_results[log],'\n')
circuits = code.get_circuit_list()
table_results = {}
for log in range(2):
    job = execute(circuits[log],
                 Aer.get_backend('qasm_simulator'),
                 noise_model=noise_model, shots=10000 )
    table_results[str(log)] = \
    job.result().get_counts(str(log))
```

13. Finally, we calculate the probability that the qubit was flipped/garbled:

```
P = lookuptable_decoding(raw_results,
                          table_results)
print('P =',P)
plt.show()
```

This section provided a brief introduction to quantum error correction. In the next section, I will cover one of the most important protocols in quantum information processing, namely, superdense coding. Superdense coding is used to transmit more classical bits than is possible classically, using quantum systems.

Exploring superdense coding

In Chapter 3, *Entanglement and Quantum Teleportation*, we were exposed to the quantum teleportation protocol. Another protocol that is closely related to quantum teleportation is **superdense coding**. While quantum teleportation is intended to send an unknown quantum state from one communicating party (the sender, normally called Alice) to another (the receiver, normally called Bob), using two classical bits in the process, superdense coding is intended to send two classical bits from one communicating party to another, using a single quantum state.

The operation of superdense coding is as follows:

1. First, an EPR pair is created, and this pair is shared by both Alice and Bob.
2. Then, depending on which bits Alice intends to send to Bob, she applies the following operations to her qubit (part of the EPR pair):

Bit Pair to Send	Quantum Operation
00	I
01	Z
10	X
11	ZX

Table 4.10 – A set of quantum operations that Alice can perform based on the pair of bits she wants to communicate to Bob

3. On the receiving side, Bob first applies the `CNOT` gate, followed by the `Hadamard` gate. This is, in essence, the reverse of what Alice applied on her side.
4. Finally, Bob performs a measurement in order to recover the two classical bits that Alice sent.

As an example, the block diagram for sending the classical bits `'01'` using superdense coding is shown as follows:

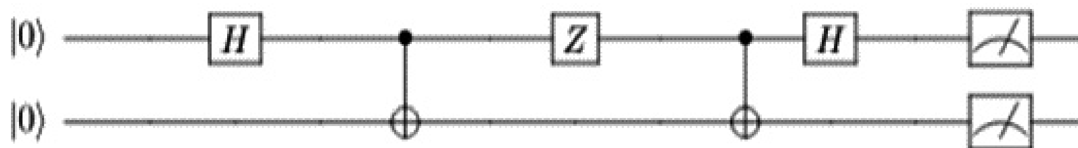


Figure 4.11 – The quantum circuit for sending the classical bits '01' using superdense coding

Similarly, the other pairs of classical bits can be sent by replacing the preceding Pauli `Z` gate with an appropriate gate operation, as provided in the preceding table.

Using the IBM's Circuit Composer, the superdense coding for transmitting classical bits `'01'` is given as follows:

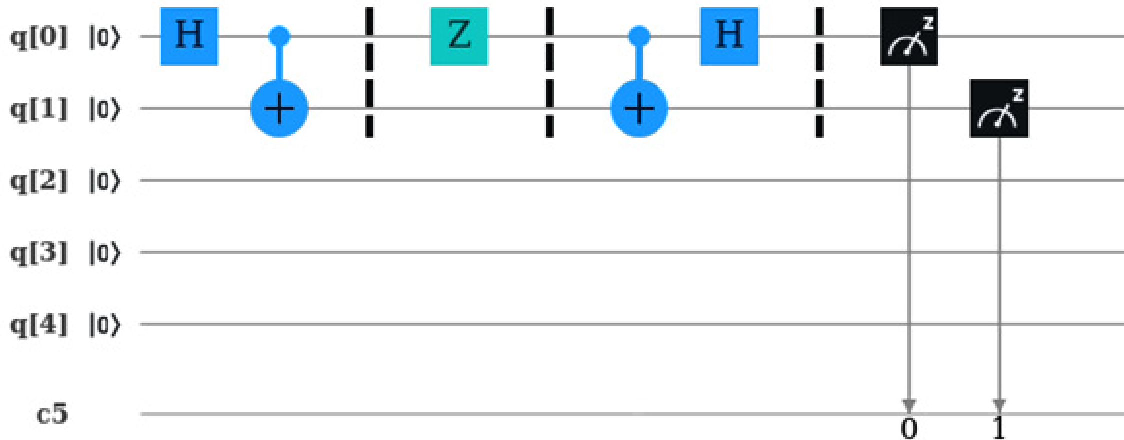


Figure 4.12 – The superdense coding circuit for sending classical bits '01'

NOTE

This circuit is generated using IBM's Circuit Composer.

The dotted vertical lines in the superdense coding circuit in the preceding diagram are the barriers. The barriers are used on the Circuit Composer to provide clarity and improve the visualization of the circuit. In this case, the gates to the left-most part of the circuit are used to generate an EPR pair. Then, the Pauli `Z` gate is the operation that Alice performs on her part of the EPR pair. Furthermore, the next set of gates is used for decoding. Then, finally, both Alice and Bob perform measurements on their qubits.

Another example of implementing superdense coding using IBM's Circuit Composer is shown in the following diagram. In this case, the classical bits being transmitted are `'11'`:

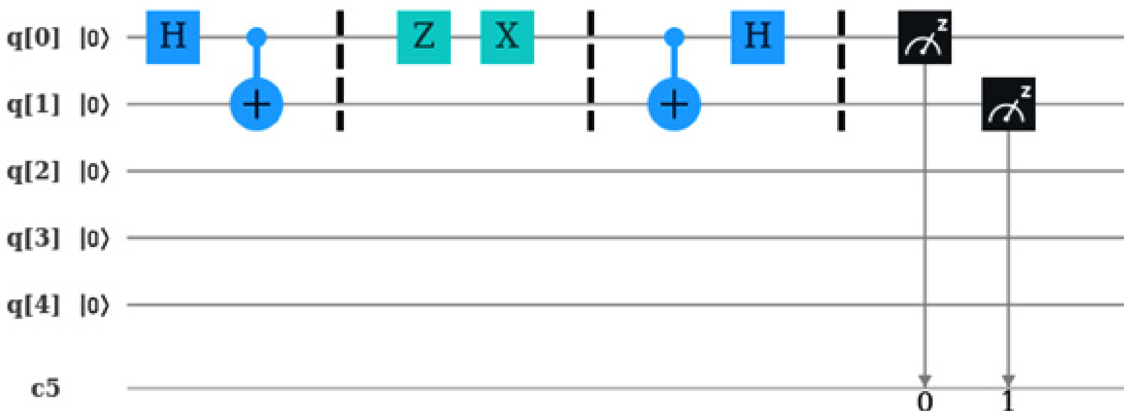


Figure 4.13 – The superdense coding circuit used to transmit the classical bits '11'

NOTE

This quantum circuit is generated using IBM's Circuit Composer.

The corresponding results for the preceding circuit (using IBM's classical simulator, with 1,024 runs) are summarized by the following histogram:

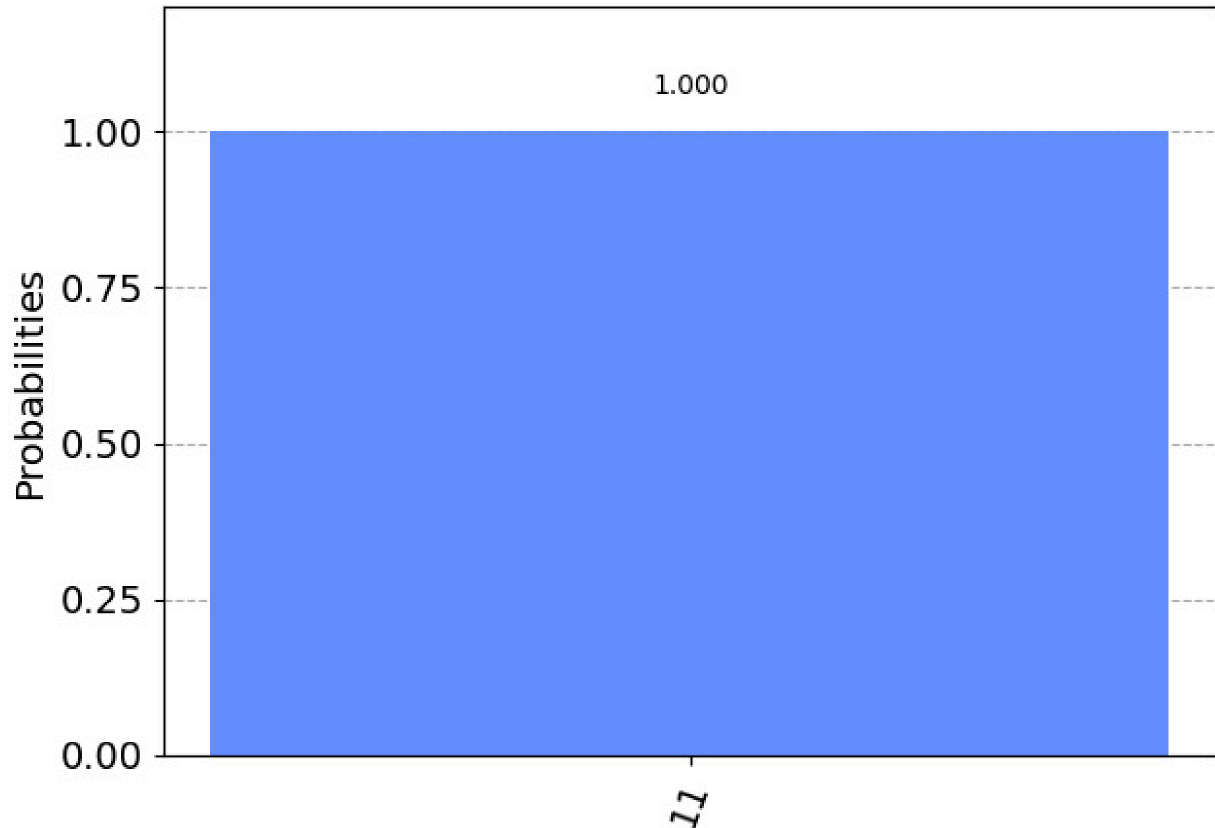


Figure 4.14 – The simulation results for transmitting classical bits '11' using superdense coding

Finally, using the `qiskit` platform, the Python code for implementing superdense coding is given as follows. In this code, the qubits being transmitted are `'11'`:

1. The first step entails importing the modules necessary for the implementation of superdense coding:

```
#Import modules
from qiskit import *
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt
import numpy as np
np.random.seed(42)
```

2. The next step is to create the Bell state, and this entangled state is shared by both Alice and Bob:

```
def create_bell_pair(qc, a, b):
    q.h(a)
```

```
    qc.cx(a,b)
```

3. This is followed by Alice's encoding of classical bits to be sent over to Bob, and Bob's decoding of such bits:

```
def encode_message(qc, qubit msg):
    if msg == "00":
     pass
    elif msg == "10":
     qc.xqubit)
    elif msg == "01":
     qc.zqubit)
    elif msg == "11":
     qc.z(qubit)
     qc.x(qubit)
def decode_message(qc,a, b):
    qc.x(a,b)
    qc.h(a)
```

4. The next step is to define the quantum circuit to be used for the implementation of superdense coding:

```
qc = QuantumCircuit(2)
create_bell_pair(qc, 0, 1)
qc.barrier()
message = "11"
encode_message(qc, 0, message)
qc.barrier()
decode_message(qc, 0, 1)
qc.measure_all()
qc.draw(output = "mpl")
#plt.show()
```

5. The superdense coding circuit is then simulated using IBM's **`'qasm_simulator'`**, and the results obtained are then displayed:

```
backend = Aer.get_backend('qasm_simulator')
job_sim = execute(qc, backend, shots=1024)
sim_result = job_sim.result()
measurement_result = sim_result.get_counts(qc)
print(measurement_result)
plot_histogram(measurement_result)
plt.show()
```

Now, let me briefly go through the code:

1. First, as is always the case, all the modules required are imported.

2. Then, the function for creating the *Bell states* is created.
3. This is followed by the function that defines the encoding of the classical message to be sent.
4. Furthermore, the circuit for decoding the classical message sent is also created.
5. After creating these functions, the superdense coding protocol is then implemented.
6. Finally, the circuit for implementing the superdense coding protocol is simulated using the `'qasm simulator'`.
7. The results from the simulation are then plotted.

In this section, we explored one of the most important protocols of quantum information processing, namely, superdense coding. We saw how we can use superdense coding to transmit more classical bits than is classically possible. Furthermore, we saw how we could use IBM's Circuit Composer in order to construct quantum circuits that could be used to implement superdense coding. Finally, we implemented superdense coding using Python and `qiskit`. Next, we will provide a brief summary of this chapter.

Summary

In this chapter, we have explored and provided Python codes for various implementations of quantum circuits. Additionally, we have discussed both quantum error-correction coding and the superdense coding protocol, together with the corresponding Python codes.

The next chapter covers quantum algorithms. Quantum algorithms are implemented using quantum circuits. Furthermore, these algorithms use the concepts from quantum mechanics in order to enable speed-ups compared to their classical counterparts.

Further reading

- Nielsen, M. A., and Chuang, I. L. (2011). *Quantum Computation and Quantum Information: 10th Edition*. New York, NY, USA: Cambridge University Press, 1107002176, 9781107002173.
- Sutor, R.S. (2019). *Dancing with Qubits: How Quantum Computing Works and How It Can Change the World*. Birmingham, UK: Packt Publishing.
- Abraham Asfaw, Luciano Bello, Yael Ben-Haim, Sergey Bravyi, Lauren Capelluto, Almudena Carrera Vazquez, Jack Ceroni, Richard Chen, Albert Frisch, Jay Gambetta, Shelly Garion, Leron Gil, Salvador De La Puente Gonzalez, Francis Harkins, Takashi Imamichi, David McKay, Antonio Mezzacapo, Zlatko Minev, Ramis Movassagh, Giacomo Nannicni, Paul Nation, Anna Phan, Marco Pistoia, Arthur Rattew, Joachim Schaefer, Javad Shabani, John Smolin, Kristan Temme, Madeleine Tod, Stephen Wood, and James Wootton. (2020). *Learn Quantum Computation Using Qiskit*. IBM. Available online at http://community.qiskit.org/textbook.

Chapter 5: Quantum Algorithms

In the previous chapter, we explored a variety of **quantum circuits**. In this chapter, I will discuss how such circuits can be used to implement **quantum algorithms**. From conventional computing, we know that an algorithm is essentially a recipe for solving problems. It is a sequence of instructions that are used to solve a problem (computational function). Thus, for a function, *f*, an algorithm acts on the input state, *x*, and produces an output, *f(x)*.

Just like a conventional algorithm, a quantum algorithm is also a sequence of instructions that is used for solving a problem. However, unlike a conventional algorithm, a quantum algorithm uses quantum mechanics in order to enable speed-ups, to offer a quantum super-advantage. Quantum algorithms are implemented using the quantum circuits covered in the previous chapter.

Typically, quantum algorithms involve the following steps:

1. Preparation and initialization of the qubits forming the quantum system
2. Transformation of the system into the superposition of many states
3. Unitary evolutions of the system
4. Measurements of the resulting qubits, in order to obtain the solution to the computational problem being solved

In this chapter, we will define and show Python implementations for the following various quantum algorithms:

- Introducing Deutsch's algorithm
- Exploring the Deutsch-Josza algorithm
- Exploring the Bernstein-Vazirani algorithm
- Introducing quantum Fourier transform and quantum phase estimation
- Introducing Simon's algorithm
- Exploring Shor's algorithm
- Exploring Grover's algorithm

In the next section, we will discuss the technical requirements for you to follow this chapter.

Technical requirements

The requirements for this chapter are the following:

- Basic understanding of the Python programming language
- Navigation of Google's Colab environment

The GitHub link for this chapter can be found here: https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter05.

Let's begin with the introduction of one of the first algorithms of quantum computing. This algorithm is known as Deutsch's algorithm.

Introducing Deutsch's algorithm

Deutsch's algorithm was one of the first quantum algorithms to be developed. It was developed by David Deutsch in 1985. Deutsch's algorithm was intended to show that a quantum computer can perform a task faster than a conventional computer. The task chosen for this algorithm is a query to an oracle.

Deutsch's algorithm is used to determine whether an unknown binary function, *f(x)*, is **constant** or **balanced**. Before stating what a constant or a balanced function is, let's take a slight step back, to discuss a binary function. A binary function is a function that takes a bit as an input and gives out a bit as an output. Thus, a univariate binary function, *f(x)*, is given as follows:

$$f(x): \{0,1\} \rightarrow \{0,1\}$$

There are four possible binary functions, which are summarized as shown in the following table:

Input	f_1	f_2	f_3	f_4
0	0	0	1	1
1	0	1	0	1

Figure 5.1 – Four possible binary functions for function f(x)

As you can see from Figure 5.1, the first function, function *f1*, gives an output of 0 regardless of the input. On the other hand, function *f2* gives the same output as the input. Furthermore, function *f3* gives the inverse of an input. Finally, the output of function *f4* is 1, regardless of the input.

Functions *f1* and *f4* from the preceding table are called **constant** binary functions while functions *f2* and *f3* are called **balanced** binary functions. In essence, constant binary functions give the same output (either 0 or 1) regardless of the input. Thus, for a constant binary function, the outputs do not depend on the input. On the other hand, a balanced binary function produces an equal number of 0s or 1s. That is, for a balanced binary function, the outputs depend on the inputs.

An alternative explanation of constant and balanced binary functions is given as follows. A binary function, *f*, is constant if the following applies:

$$f(0) \oplus f(1) = 0$$

On the other hand, a binary function, *f*, is balanced if the following applies:

$$f(0) \oplus f(1) = 1$$

Now that we have briefly explained the balanced and constant functions, let's probe Deutsch's algorithm even further. As stated earlier, the aim of this algorithm is to evaluate whether a given function is a constant or balanced function. This evaluation is done by querying a "blackbox" (an oracle).

Deutsch's problem can be framed as follows. Imagine that you have an oracle that gives a response to any query sent to it. In this case, imagine that the oracle gives either a 0 or a 1 as a response. Now imagine that you want to determine whether a given binary function is constant or balanced. Then, how many queries should you send to the oracle in order to convincingly determine this?

Having framed Deutsch's problem, now let's see how many function evaluations are required by a classical computer in order to achieve this task. To address this, we will use the table given earlier. Now consider this scenario. If you send 0 as an input, and you get

an output of 0, can you tell whether the function is constant or balanced? Clearly not, since both functions *f1* and *f2* have an output of 0 when the input is 0, and we have seen that function *f1* is constant while function *f2* is balanced. We will need a second function evaluation in order to convincingly make this determination.

We can see from the previous scenario that given a classical computer, a single function evaluation would not be enough to determine whether the function is constant or balanced. This determination would take at least two function evaluations. Now the question is whether this evaluation can be done better using a quantum computer. Fortunately for us, the answer to this question is affirmative and forms the basis of Deutsch's algorithms.

In order to realize the Deutsch algorithm, the key step is to realize the unitary operator *F* such that the application of this operator to the inputs |*x*> and |*y*> is given as follows:

$$F(x \otimes y\rangle) = x\rangle \otimes y \oplus f(x)\rangle$$

Pictorially, this unitary gate (*F*) can be shown as follows:

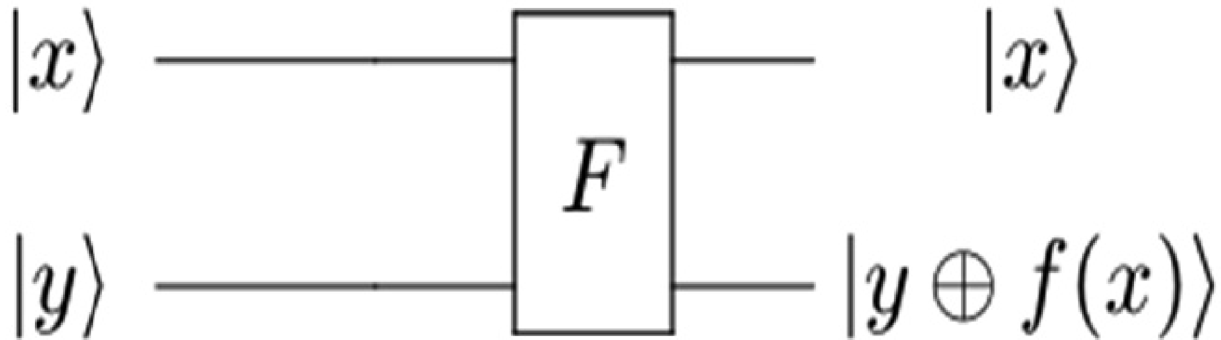


Figure 5.2 – Unitary gate (F)

Now that we have provided a unitary transformation that can be used for the Deutsch algorithm, it is important to explore the complete circuit that can be used to implement this algorithm. This circuit is diagrammatically given as follows:

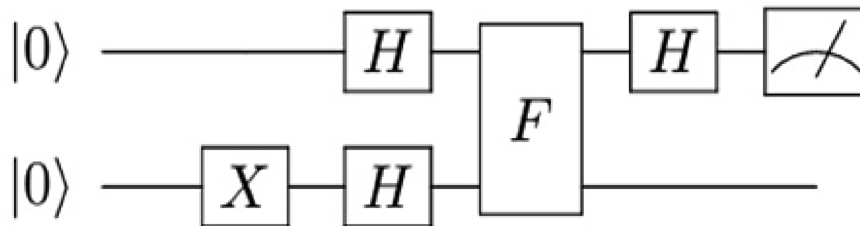


Figure 5.3 – A complete circuit to implement Deutsch's algorithm

As it can be seen from the previous figure, the qubits are first initialized to state |*0*>. Therefore, the first state of the preceding system is as follows:

$$\psi_0\rangle = 0\rangle \otimes 0\rangle$$

This state is then evolved by flipping the second qubit. This is achieved by applying the X gate to the second qubit. The new state of the system then becomes the following:

$$\psi_1\rangle = 0\rangle \otimes 1\rangle$$

This step is followed by the application of the Hadamard gate on each of the qubits, resulting in the following:

$$\psi_2\rangle = H0\rangle \otimes H1\rangle$$

This can also be given as follows:

$$\psi_2\rangle = \frac{1}{2}(00\rangle - 01\rangle + 10\rangle - 11\rangle)$$

Now, the next step is to apply the unitary transformation *F* on the preceding state. This results in the following state:

$$\psi_3\rangle = \frac{1}{2}(0\rangle \otimes f(0)\rangle - 0\rangle \otimes 1 \oplus f(0)\rangle + 1\rangle \otimes f(1)\rangle - 1\rangle \otimes 1 \oplus f(1)\rangle$$

Now we have to consider two scenarios. The first scenario is when the function being evaluated is a constant function. This, as we have already seen, means that $f(0) = f(1)$. Therefore, the preceding state would be simplified to the following:

$$\psi_3\rangle = \frac{1}{2}[(0\rangle + 1\rangle) \otimes (f(0)\rangle - 1 \oplus f(0)\rangle)$$

The next step of Deutsch's algorithm involves applying the Hadamard gate to the first qubit of the preceding state. This results in the following:

$$\psi_4\rangle = \frac{1}{\sqrt{2}}[0\rangle \otimes (f(0)\rangle - 1 \oplus f(0)\rangle)]$$

Finally, measurement is performed on the first qubit, resulting in bit 0. This is consistent with the fact that the function is constant, since we now know that for a constant function, the following applies:

$$f(0) \oplus f(1) = 0$$

Now, let's see what happens when the function being evaluated is the balanced binary function. Recall that for a balanced function, the following applies:

$$f(0) \neq f(1)$$

Furthermore, for a balanced binary function, we have the following:

$$1 \oplus f(0)\rangle = f(1)\rangle$$

The following also applies:

$$1 \oplus f(1)\rangle = f(0)\rangle$$

So, assuming that the function being evaluated is a balanced function instead of a constant function, we then have the following:

$$\psi_3\rangle = \frac{1}{2}[(0\rangle - 1\rangle) \otimes (f(0)\rangle - f(1)\rangle)]$$

Now, applying the Hadamard gate to the first qubit, as we did in the first case, results in the following:

$$\psi_4\rangle = \frac{1}{\sqrt{2}}[1\rangle \otimes (f(0) - f(1)\rangle)]$$

It can then be observed that measuring the first qubit of the preceding state would result in state 1, which is consistent with the fact that the function is the balanced function.

The preceding technique is quite tedious. Fortunately for us, there is a more concise and intuitive way of computing Deutsch's algorithm. This concise technique is called the **phase kickback** technique. It can be summarized as follows. We have already seen the output of the system after the inputs go through the unitary transformation (*F*). That output can be simplified to the following:

$$\psi_3\rangle = \frac{1}{2}[0\rangle \otimes (f(0)\rangle - 1 \oplus f(0)\rangle) + 1\rangle \otimes (f(1)\rangle - 1 \oplus f(1)\rangle)]$$

From this, it is clear that depending on whether $f(0)$ is 0 or 1, we can have the following when $f(0) = 0$:

$$f(0)\rangle - 1 \oplus f(0)\rangle = 0\rangle - 1\rangle$$

We have the following when $f(0) = 1$:

$$f(0)\rangle - 1 \oplus f(0)\rangle = 1\rangle - 0\rangle$$

The following can then be observed:

$$f(0)\rangle - 1 \oplus f(0)\rangle = (-1)^{f(0)}(0\rangle - 1\rangle)$$

The same analysis can also be applied to the function $f(1)$. Refer to the quiz at the end of this chapter for this exercise.

Now, armed with this intuition, we can represent the state after applying the unitary transformation F as follows:

$$\psi_3\rangle = \frac{1}{2}\left[0\rangle \otimes (-1)^{f(0)}(0\rangle - 1\rangle) + 1\rangle \otimes (-1)^{f(1)}(0\rangle - 1\rangle)\right]$$

This can then be simplified to the following:

$$\psi_3\rangle = \frac{1}{2}\left[(-1)^{f(0)}0\rangle + (-1)^{f(1)}1\rangle\right] \otimes \left[0\rangle - 1\rangle\right]$$

The next step is the application of the Hadamard gate to the first qubit. This results in the following:

$$\psi_4\rangle = \frac{1}{2}\left[\left((-1)^{f(0)} + (-1)^{f(1)}\right)0\rangle + \left((-1)^{f(0)} - (-1)^{f(1)}\right)1\rangle\right] \otimes (0\rangle - 1\rangle)$$

Finally, the measurement is performed on the first qubit. It should be clear by now that if the function is constant, then the measured state would be 0 whereas if the function is balanced, the measured state would be 1.

Having provided an overview of Deutsch's algorithm, it is now time to explore how this algorithm can be implemented.

NOTE

This code is adopted from https://github.com/quantumlib/Cirq/blob/master/examples/deutsch.py.

The link for this code can be found here: https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter05.

The Python code for Deutsch's algorithm using **`cirq`** is given as follows:

1. The first step is to introduce the necessary modules:

```
import random
import numpy as np
import cirq
```

```
from cirq import H, X, CNOT, measure
```

2. After importing the necessary modules, the next step is to choose the qubits to use for this algorithm:

```
q0, q1 = cirq.LineQubit.range(2)
secret_function = [random.randint(0,1) for _ in range(2)]
```

3. Define the functions for realizing the oracle circuit and the complete Deutsch's algorithm:

```
def make_oracle(a, b, c):
    if c[0]:
     yield [CNOT(a,b), X(b)]
    if c[1]:
     yield CNOT(a,b)
def make_deutsch_circuit(d,e,f):
    c = cirq.Circuit()
    c.append([H(e), H(e), H(d)])
    c.append(f)
    c.append([H(d), measure(d, key=›'result'›)])
    return c
oracle = make_oracle(q0, q1, secret_function)
circuit = make_deutsch_circuit(q0, q1, oracle)
print(circuit)
```

4. Finally, simulate the quantum circuit that is used to implement Deutsch's algorithm:

```
simulator = cirq.Simulator()
result = simulator.run(circuit)
print(result)
```

The output of the circuit is shown in the following figure:

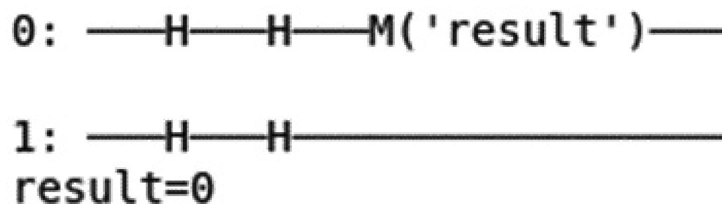


Figure 5.4 – An output of the circuit used to implement Deutsch's algorithm

The preceding code can be summarized as follows. After importing the necessary modules, the two qubits are then instantiated using **`cirq`**'s **`LineQubit.range()`** function. This is then followed by defining two functions that will be used by the algorithm. The first function defines the function to be used to realize the oracle. This oracle uses the **`CNOT`** gate for unitary transformation *F* discussed earlier.

The next function defines the necessary steps used in realizing the Deutsch algorithm. After defining these two functions, the next step is to simulate Deutsch's algorithm using **`cirq`**'s **`Simulator()`** function.

So far, we have covered Deutsch's algorithm and how it can be implemented using Python. In the next section, we will cover an algorithm that generalizes Deutsch's algorithm to multiple qubits. This algorithm is called the Deutsch-Josza algorithm.

Exploring the Deutsch-Josza algorithm

As already stated, the Deutsch-Josza algorithm generalizes the *Deutsch algorithm* discussed previously. Instead of being a univariate function evaluation like the Deutsch algorithm, the Deutsch-Josza algorithm evaluates a multi-variate binary function. The goal of this algorithm is also to determine whether the said function is **constant** or **balanced**.

In essence, a Deutsch-Josza algorithm uses an oracle ("blackbox") that implements the function *f(x)*:

$$f(x): \{0,1\}^n \rightarrow \{0,1\}$$

Here, *n* represents the number of variables. As can be observed from the preceding equation, the function takes as input *n* bits and outputs either a 0 or a 1, corresponding to whether the function being evaluated is constant or balanced. Using a conventional computer, the exercise of evaluating this multi-variate function would require *2n* steps. However, with a quantum computer, this can be done in a single step.

The circuit for implementing a Deutsch-Josza algorithm is shown as follows:

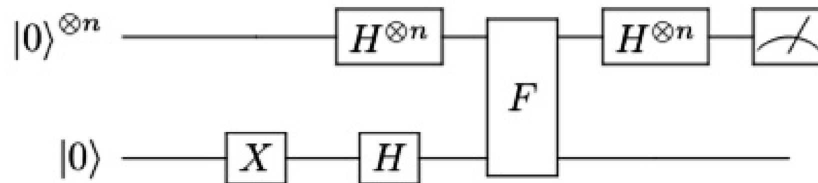


Figure 5.5 – A circuit for implementing a Deutsch-Josza algorithm

The upper part of the circuit shows a register with *n* qubits, while the lower part of the circuit shows a register with a single qubit.

Just like the Deutsch-Josza algorithm, the qubits are initialized to the |*0*> state. Thus, the state of the system is given as follows:

$$\psi_0\rangle = 0\rangle \otimes \cdots \otimes 0\rangle$$

Then, the next step involves flipping the last qubit, which is the qubit in the bottom register, with the resulting state of the system being the following:

$$\psi_1\rangle = 0\rangle \otimes \cdots \otimes 1\rangle$$

Now, apply the Hadamard gate on each of the *n* qubits in state |*0*>, and another Hadamard gate to the last qubit, which now has state |*1*>. The resulting state of the system is given as follows:

$$\psi_2\rangle = H \otimes \cdots \otimes H(0 \otimes \cdots \otimes 0\rangle) \otimes H(1\rangle$$

The following applies:

$$H \otimes \cdots \otimes H 0 \otimes \cdots \otimes 0\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=1}^{2^n} x\rangle$$

It is not difficult to see that upon simplification, the preceding state can be given as follows:

$$\psi_2\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=1}^{2^n} x\rangle \left[\frac{1}{\sqrt{2}} [0\rangle - 1\rangle]\right]$$

Just like in the case of the Deutsch algorithm, the next step in the Deutsch-Josza algorithm is to apply the unitary transformation to the system. As a generalization, this unitary transformation is given as follows:

$$F(x \otimes y\rangle) = x \otimes [y \oplus f(x)]\rangle$$

Therefore, applying *F* to the system's state results in the following:

$$\psi_3\rangle = \frac{1}{\sqrt{2^n}} \left[\sum_{x=1}^{n} (-1)^{f(x)} x\rangle\right] \otimes \left[\frac{1}{\sqrt{2}} [0\rangle - 1\rangle]\right]$$

Following this step, the Hadamard gate is then applied to all qubits except the last qubit. This results in the following state:

$$\psi_4\rangle = \frac{1}{2^n} \sum_{i=1}^{n} \sum_{x} (-1)^{x.i+f(x)} i\rangle \otimes \left[\frac{1}{\sqrt{2}} [0\rangle - 1\rangle]\right]$$

The following applies:

$$x.i = x_1 i_1 \oplus \cdots \oplus x_n i_n$$

Finally, all the qubits except the last qubit are measured. It should be noted from the state of the preceding system that the amplitude of the upper register is as follows:

$$\left| \frac{1}{2^n} \sum_{x=0}^{n} (-1)^{f(x)} \right|$$

Therefore, the probability of measuring qubits in the upper register is the square of these probability amplitudes:

$$p(0 \otimes \cdots \otimes 0\rangle) = \left| \frac{1}{2^n} \sum_{x=0}^{n} (-1)^{f(x)} \right|^2$$

If all the measured qubits yield a value of 0, then the function being evaluated is a constant function. If this is not the case, then the function being evaluated is balanced.

Having demonstrated the steps involved in the implementation of the Deutsch-Josza algorithm, it is now time to focus on the code being used to implement such an algorithm. The following Python code, which makes use of Qiskit, shows an implementation of the Deutsch-Josza algorithm:

NOTE

The code is adopted from https://qiskit.org/textbook/ch-algorithms/deutsch-jozsa.html#4.-Qiskit-Implementation-.

The link for this code can be found here: https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter05

1. The first step is the import of the modules necessary for the implementation of the Deutsch-Josza algorithm:

```
import numpy as np
from qiskit import BasicAer
from qiskit import QuantumCircuit, execute
from qiskit.visualization import plot_histogram
```

```
np.random.seed(42)
```

2. Following the importing of the modules, the next step is to configure the number of qubits to use in order to implement the Deutsch-Josza algorithm and the construction of the oracle that will be used to implement the algorithm:

```
n = 3
```

3. The next step then is to define and draw the constant oracle:

```
const_oracle = QuantumCircuit(n+1)
output = np.random.randint(2)
if output == 1:
const_oracle.x(n)
const_oracle.draw()
```

4. This is then followed by defining and drawing the balanced oracle:

```
balanced_oracle = QuantumCircuit(n+1)
b_str = "101"
for qubit in range(len(b_str)):
if b_str[qubit] == '1':
balanced_oracle.x(qubit)
balanced_oracle.barrier()
for qubit in range(n):
balanced_oracle.cx(qubit, n)
balanced_oracle.barrier()
for qubit in range(len(b_str)):
if b_str[qubit] == '1':
balanced_oracle.x(qubit)
balanced_oracle.draw()
```

5. After constructing the oracle, the next step is to construct the complete quantum circuit that can be used to implement the Deutsch-Josza algorithm:

```
dj_circuit = QuantumCircuit(n+1, n)
for qubit in range(n):
dj_circuit.h(qubit)
dj_circuit.x(n)
dj_circuit.h(n)
dj_circuit += balanced_oracle
for qubit in range(n):
dj_circuit.h(qubit)
dj_circuit.barrier()
for i in range(n):
dj_circuit.measure(i, i)
dj_circuit.draw()
```

6. Finally, simulate the quantum circuit for the Deutsch-Josza algorithm, using **qasm_simulator**:

```
backend = BasicAer.get_backend('qasm_simulator')
shots = 1024
results = execute(dj_circuit,
                  backend=backend,
                  shots=shots).result()
answer = results.get_counts()
plot_histogram(answer)
```

The output of the simulated quantum circuit is shown as follows:

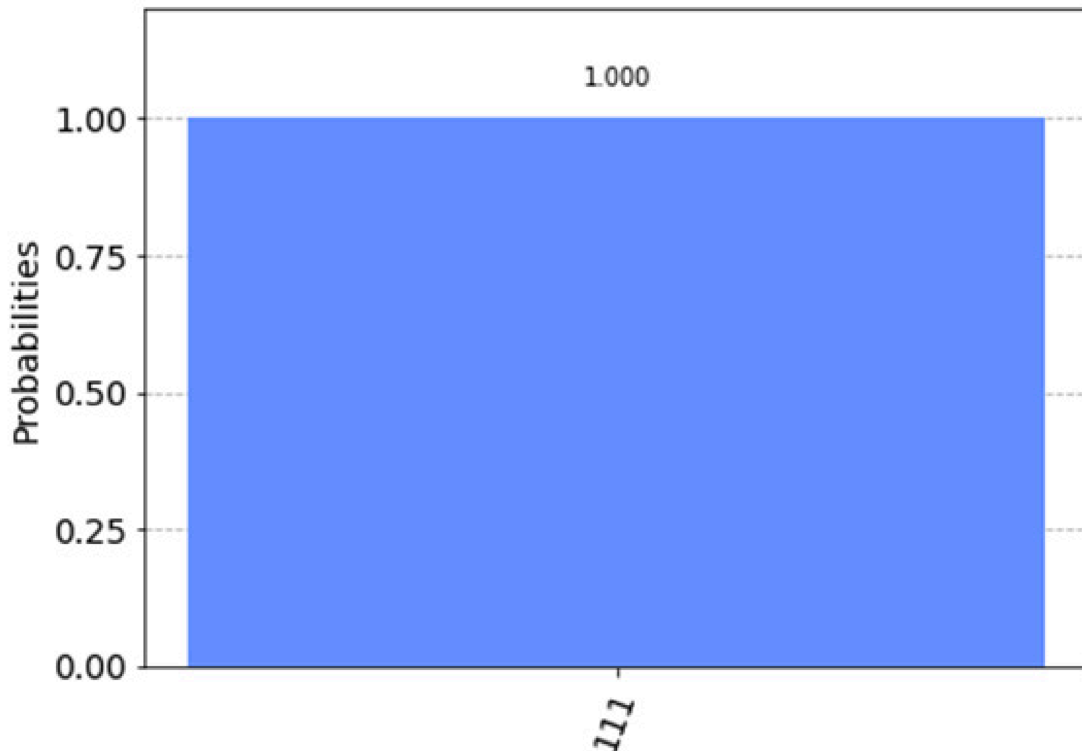


Figure 5.6 – The output of the Deutsch-Josza three-qubit algorithm

The preceding code can be summarized as follows. As is always the case, the first step is the importing of modules that are to be used for the implementation of the Deutsch-Josza algorithm. The code implements the Deutsch-Josza algorithm for a three-qubit function. The next step in the code is to construct the constant oracle, followed by the construction of the balanced oracle. This is followed by the actual implementation of the Deutsch-Josza algorithm. Finally, this algorithm is simulated using **qasm_simulator**. It is worth noting that **qasm_simulator**, which is a local simulator, is used to simulate quantum circuits, but on a classical computer. This simulation is done instead of running such circuits on an actual quantum computer.

Now that we have covered the Deutsch-Josza algorithm, the next step is to explore another algorithm that can in essence be viewed as an extension of the Deutsch-Josza algorithm. This algorithm is called the **Bernstein-Vazirani algorithm**.

Exploring the Bernstein-Vazirani algorithm

The Bernstein-Vazirani algorithm is in a sense similar to the *Deutsch-Josza algorithm* discussed previously. Thus, the oracle still uses the following function:

$$f(x): \{0,1\}^n \rightarrow \{0,1\}$$

On the other hand, unlike the Deutsch-Josza algorithm, where the objective is to determine whether the given function is **constant** or **balanced**, the objective of the Bernstein-Vazirani algorithm is to find the secret string **s**:

$$s: s \in \{0,1\}^n$$

Given the promise that:

$$f(x) = \left(\sum_{i=1}^{n} s_i\, x_i\right) mod2$$

The Bernstein-Vazirani algorithm works as follows. First, two registers, one the input register with *n* qubits and the other the output register with a single qubit, are initialized to state |*0*>. Thus, the state of the system is given as follows:

$$\psi_0\rangle = 0 \otimes \cdots \otimes 0\rangle \otimes 0\rangle$$

Then, the qubit in the second register is flipped, with the new state of the system being the following:

$$\psi_1\rangle = 0 \otimes \cdots \otimes 0\rangle \otimes 1\rangle$$

The next step in the algorithm is to apply the Hadamard gate to each of the qubits in both registers. Then, we end up with the following state:

$$\psi_2\rangle = \frac{1}{\sqrt{2^n}} \sum_{x=1}^{n} x\rangle \otimes \left(\frac{1}{\sqrt{2}}[0\rangle - 0\rangle]\right)$$

This step is then followed by the querying of the oracle, and then applying the Hadamard gate to each of the qubits on the input register. Finally, the measurement is applied to the input register in order to retrieve the secret string.

NOTE

This code is adopted from https://github.com/qiskit-community/qiskit-community-tutorials/blob/master/Coding_With_Qiskit/ep6_Bernstein-Vazirani_Algorithm.ipynb.

The link for this code can be found here: https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter05.

Using **Qiskit**, the Python code for implementing the Bernstein-Vazirani algorithm is given as follows:

1. The first step is to introduce the necessary modules for the implementation of the Bernstein-Vazirani algorithm:

```
import numpy as np
from qiskit import *
from qiskit.visualization import plot_histogram
np.random.seed(42)
s = input("Enter the secret bit string:\n")
n = len(s)
```

2. This step is followed by the construction of the quantum circuit that will be used to construct the Bernstein-Vazirani algorithm:

```
circuit = QuantumCircuit(n+1,n)
```

   Since all the qubits are initialized to state |*0*>, step 0 involves inverting these qubits by applying the X gates:

```
circuit.x(n)
circuit.barrier()
```

3. The next step is to apply the Hadamard gates on all the qubits:

```
circuit.h(range(n+1))
circuit.barrier()
```

4. The next step is to apply the controlled gates on the qubits:

```
for ii, yesno in enumerate(reversed(s)):
if yesno == '1':
circuit.cx(ii, n)
circuit.barrier()
```

5. The next step involves applying the Hadamard gates on the qubits, followed by the measurements of such qubits:

```
circuit.h(range(n+1))
circuit.barrier()
circuit.measure(range(n), range(n))
%matplotlib inline
circuit.draw(output='mpl')
```

6. Finally, simulate the operation of the Bernstein-Vazirani algorithm using **`qasm_simulator`**:

```
simulator = Aer.get_backend('qasm_simulator')
result = execute(circuit, backend=simulator,
                 shots=1024).result()
plot_histogram(result.get_counts(circuit))
```

The preceding code consists of four key stages. The first one initializes the qubits in both registers. This is then followed by applying the Hadamard gate to all the qubits in both registers. Furthermore, stage three involves the implementation of the oracle, which is made up of the **`CNOT`** gate. Finally, the last stage measures all the qubits in the input register.

The output of the Bernstein-Vazirani algorithm (using the input bit string **`100`**) is shown as follows:

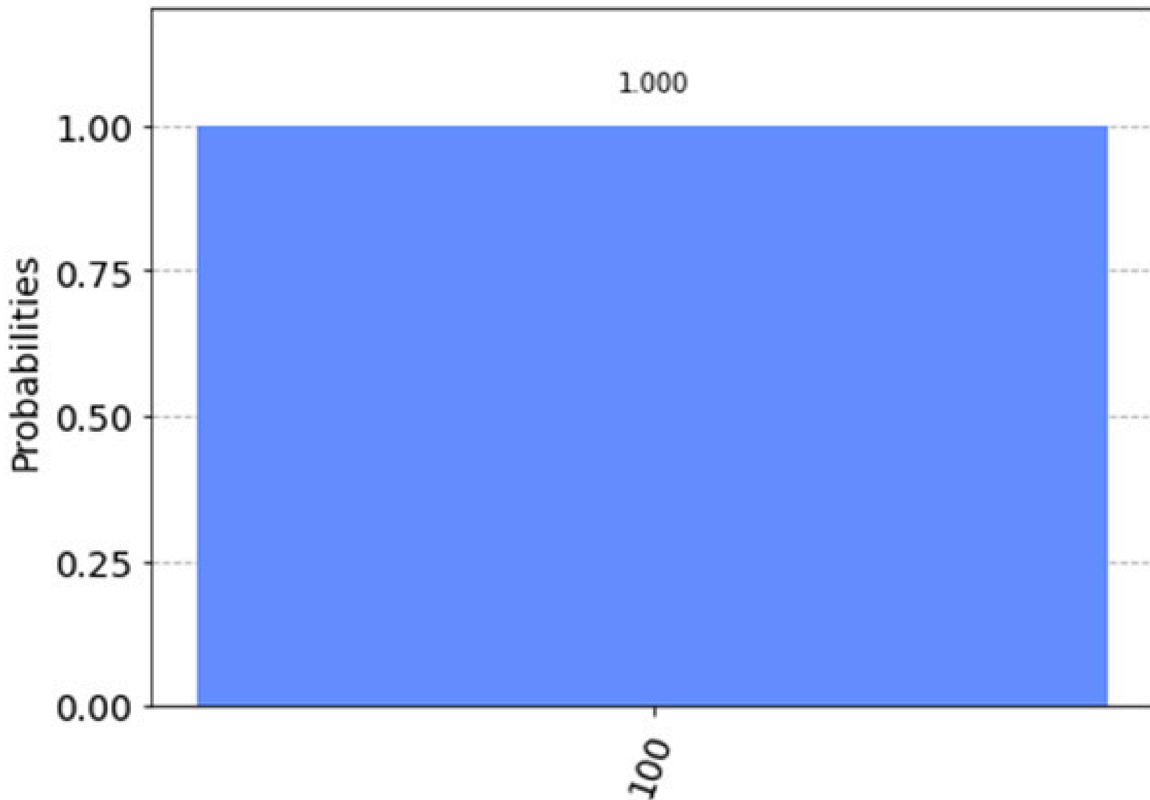


Figure 5.7 – The output of the Bernstein-Vazirani algorithm for an input bit string 100

So far, we have covered three quantum algorithms in this chapter. In this section, we have learned about the Bernstein-Vazirani algorithm, and have seen how such an algorithm can be used to find a secret bit string. In the next section, we will cover two other techniques that play a crucial role in the quantum algorithms to be discussed later in this chapter. These techniques are **quantum Fourier transform** (**QFT**) and quantum phase estimation.

Introducing quantum Fourier transform and quantum phase estimation

As already stated previously, the quantum techniques to be discussed in this section play a crucial role in the quantum algorithms to be discussed later in this chapter. Therefore, it is imperative to cover the following techniques in this section:

- QFT
- Quantum phase estimation

First, let's introduce and explore QFT.

Quantum Fourier transform (QFT)

It has been discovered that QFT can be computed faster on a quantum computer than on a classical computer. QFT is the quantum analog of the discrete Fourier transform. We shall recall a discrete Fourier transform that maps the input vector

$$x = \sum_{i=0}^{N-1} x_i$$

into the output vector

$$y_k = \frac{1}{\sqrt{N}} \sum_{j=o}^{N-1} x_j \, w_N^{jk};$$

where

$$w_N^{jk} = e^{2\pi i \frac{jk}{N}}$$

As already said, QFT is the quantum analog of the discrete Fourier transform discussed previously. Therefore, QFT transforms the input quantum system $|x>$ into the output quantum system $|y>$ such that the following applies:

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

Using Python and Qiskit, an *n*-qubit QFT can be implemented using the following code snippet:

NOTE

The link for this code can be found here: https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter05.

```
from qiskit.circuit.library import QFT
n = input(" Enter the number of qubits: \n")
qft_circuit = QFT(num_qubits = n)
qft_circuit.draw()
```

The preceding code uses Qiskit's circuit library to implement the QFT. The code first prompts the number of qubits to be transformed, then constructs the QFT circuit based on the supplied number of qubits:

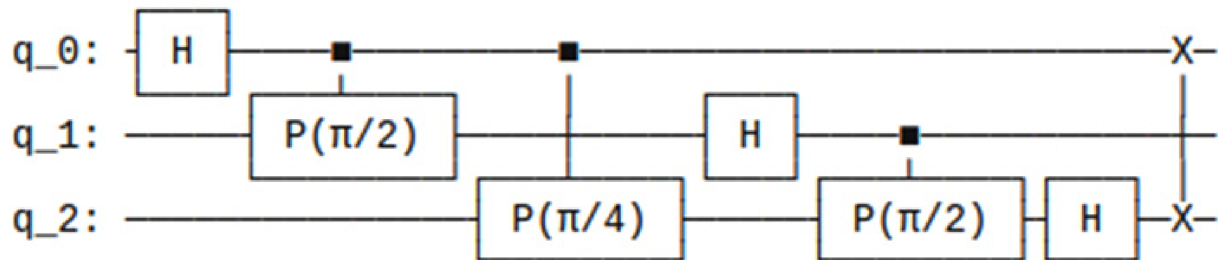


Figure 5.8 – An output of a three-qubit QFT circuit

The output of the implementation of the three-qubit QFT technique is shown in the previous figure.

Quantum phase estimation

The quantum phase estimation technique is used to estimate the eigenvalue (phase) of a unitary operator. In order to achieve this task, the quantum phase estimation technique makes use of the QFT technique discussed earlier.

Suppose that a unitary operator, *U*, has an eigenvector, |*u*>, with an eigenvalue as follows:

$$e^{2\pi i\theta}$$

Also, the following applies:

$$U u\rangle = e^{2\pi i\theta} u\rangle$$

Then, the objective of the quantum phase estimation algorithm is to estimate the value of θ.

The procedure for implementing the quantum phase estimation can be summarized as follows:

1. The first step entails the creation of two registers. The first register contains *n* qubits initialized to state |*0*>. The second register is initialized to state |*u*>.
2. This stage is then followed by the application of the Hadamard gate on each of the *n* qubits in the first register.
3. Furthermore, for the unitary operator *U*, the controlled-*U* is applied on the quantum state |*u*> in the second register. The controlled-*U* gate is somehow similar to the controlled-NOT gate discussed in previous chapters. The only difference is that instead of applying the NOT gate to the target qubit when the control qubit is in state |*1*>, the controlled-*U* applies the *U* unitary to the target qubit (in the second register) when the corresponding control qubit (in the first register) is in state |*1*>.
4. After applying the controlled-*U* gate, the inverse QFT is then applied to all the *n* qubits in the first register.
5. Finally, measurement is performed on all the qubits on the first register.

NOTE

This code is adopted from https://github.com/quantumlib/Cirq/blob/master/examples/phase_estimator.py.

The link for this code can be found here: https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter05.

The Python code used to implement the quantum phase estimation using **`cirq`** is given as follows:

1. As is always the case, the first step entails the importing of the modules necessary for the implementation of the quantum phase estimation technique:

```
import numpy as np
import cirq
```

2. Now, we construct the quantum circuit for implementing the quantum phase estimation technique. This quantum circuit will then be used to execute the quantum phase estimation for two, four, and eight qubits:

```
def run_estimate(unknown_gate, qnum, repetitions):
    ancilla = cirq.LineQu-1)
    qubits = cirq.LineQubit.rangem)
    oracle_raised_to_power= [
     unknown_gate.on(ancilla).controlled_by\
     (qubits[i])**i)
     for i in range(qnum)
    ]
    circuit = cirq.Circuit(cirq.H.on_each(*qubits),
                           oracle_raised_to_power,
```

```
                            cirq.QFT(*qubits,
                                without_reverse=True)**-1,
                            cirq.measure(*qubits,
                                         key='phase'))
    return cirq.sample(circuit, repetitions=repetitions)
def experiment(qnum, repetitions):
    def example_gate(phi):
        gate = cirq.MatrixGate(
           matrix=np.array([
                [np.exp(2 * np.pi * 1.0j * phi), 0],
                [0, 1]]))
        return gate
```

3. The next step is to test the accuracy of the quantum phase estimation using the **root mean square** (**RMS**) error as the metric. The lower the RMS, the better the estimation:

```
    print(f'Testing with {qnum} qubits.')
    errors = []
    for target in np.arange(0, 1, 0.1):
       result = run_estimate(example_gate(target),
                             qnum, repetitions)
        mode = result.data['phase'].mode()[0]
        guess = mode / 2**qnum
        print(f'target={target:0.4f}, '
        f'estimate={guess:0.4f}={mode}/{2**qnum}')
        errors.append((target - guess)**2)
        rms = np.sqrt(sum(errors) / len(errors))
        print(f'RMS Error: {rms:0.4f}\n')
```

4. Execute the quantum phase estimation for two, four, and eight qubits:

```
def main(qnums = (2, 4, 8), repetitions=100):
    for qnum nums:
        experiment(qnum, repetitions=repetitions)
if __name__ == '_n__':
    main()
```

The preceding code implements the quantum phase estimation for two, four, and eight qubits. Furthermore, in order to get the results, the phase estimation circuit (experiment) is run for a total of 100 repetitions.

In this section, we have covered two quantum techniques that play a crucial role in some of the quantum algorithms. These techniques are QFT and quantum phase estimation. In the next section, we will cover yet another quantum algorithm that has been shown to be more powerful than its classical counterpart. This algorithm is Simon's algorithm.

Introducing Simon's algorithm

Simon's algorithm builds on the ideas of the *Bernstein-Vazirani algorithm* discussed earlier in this chapter. For Simon's algorithm, suppose that for *n* qubits, we have the following function:

$$f: \{0,1\}^n \rightarrow \{0,1\}^n$$

This comes with the promise that for some non-zero bitstring, *s*, such that:

$$s \in \{0,1\}^n$$

And for all values of *x, y* such that:

$$x, y \in \{0,1\}^n$$

Then:

$$f(x) = f(y)$$

If and only if:

$$x \oplus y \in \{0^n, s\}$$

Simon's algorithm can be summarized as follows. First, the two registers are used, each with *n* qubits. Therefore, the initial state of the quantum system is given as follows:

$$\psi_0\rangle = 0 \otimes \cdots \otimes 0\rangle \otimes 0 \otimes \cdots \otimes 0\rangle$$

This step is followed by the application of the Hadamard gate on each of the qubits in the first register. This results in the following quantum state:

$$\psi_1\rangle = H|0 \otimes \cdots \otimes 0\rangle \otimes (0 \otimes \cdots \otimes 0\rangle$$

The next step in the implementation of Simon's algorithm is to apply the unitary transformation *F* (the oracle function) such that the new state of the system becomes the following:

$$|\psi_2\rangle = \frac{1}{\sqrt{2^n}} \sum_{x\in\{0,1\}^n} |x\rangle \otimes |f(x)\rangle$$

Furthermore, we measure the second register. This step is followed by the application of the Hadamard gate on all the qubits on the first register, followed by the measurement of such qubits.

NOTE

This code is taken from https://qiskit.org/textbook/ch-algorithms/simon.html.

The link for this code can be found here: https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter05.

The Python code for implementing Simon's algorithm using Qiskit is given as follows:

1. The first step involves importing the necessary modules for the implementation of Simon's algorithm:

```
from qiskit import BasicAer
from qiskit.providers.ibmq import least_busy
from qiskit import QuantumCircuit, execute
from qiskit.visualization import plot_histogram
from qiskit_textbook.tools import simon_oracle
import matplotlib.pyplot as plt
```

2. The next step involves the construction of the quantum circuit used for the implementation of Simon's algorithm. The circuit takes a three-bit bit string, **`'110'`**, as an input:

```
b = '110'
n = len(b)
simon_circuit = QuantumCircuit(n*2, n)
simon_circuit.h(range(n))
simon_circuit.barrier()
simon_circuit += simon_oracle(b)
simon_circuit.barrier()
simon_circuit.h(range(n))
simon_circuit.measure(range(n), range(n))
simon_circuit.draw('mpl')
plt.show()
```

3. Finally, simulate the quantum circuit using **`qasm_simulator`**. You should note that should you wish to run this code on an actual quantum computer, you will have to change the backend from **`qasm_simulator`** to the **`ibmq_vigo`**

backend:

```
backend = BasicAer.get_backend('qasm_simulator')
shots = 1024
results = execute(simon_circuit, backend=backend,
                  shots=shots).result()
counts = results.get_counts()
plot_histogram(counts)
def bdotz(b, z):
    accum = 0
    for i in range(len(b)):
        accum += int(b[i]) * int(z[i])
   return (accum % 2)
print('b = ' + b)
for z in counts:
    print('{}.{} = {} (mod 2) ({:.1f}%)'\
        .format(b, z, bdotz(b,z), counts[z]*100/shots))
```

The code can be summarized as follows:

1. First, as is usually the case, the modules that are going to be used are imported.

2. Then, the quantum circuit for implementing the **`b='110'`** string is constructed.

3. After constructing the circuit, the constructed circuit is simulated using **`qasm_simulator`**.

Having covered Simon's algorithm in this section, the next section discusses Shor's algorithm.

Exploring Shor's algorithm

Shor's algorithm builds on Simon's algorithm, which was discussed in the previous section. This algorithm is used for the factorization of composite (prime) numbers, by first finding the period of such numbers. From a historical standpoint, Shor's algorithm played a crucial role in bringing quantum computing in to the limelight.

In essence, Shor's algorithm solves the problem of factoring integers in polynomial time! This is a significant improvement over the known classical algorithm. Shor's algorithm was mainly inspired by Simon's algorithm, which was discussed in the previous section. Furthermore, this algorithm makes use of quantum parallelism and QFT, which were discussed earlier in this chapter.

As already stated, Shor's algorithm efficiently finds the period of a periodic function. A periodic function, *f(x)*, is given as follows:

$$f(x) = a^x mod N$$

Here, *mod* is a modulo operator, *a* and *N* are positive integers that have no common factors (thus *a* and *N* are co-prime), and the following applies:

$$a < N$$

Additionally, a period, *r*, is the smallest non-zero integer such that the following applies:

$$a^r modN = 1$$

The implementation of Shor's algorithm can be summarized as follows:

1. First, two registers are used.

   The first register consists of *m* qubits, while the second register consists of *n* qubits, where the following applies:

$$m = 2n$$

   The circuit for implementing Shor's algorithm is shown in the following figure:

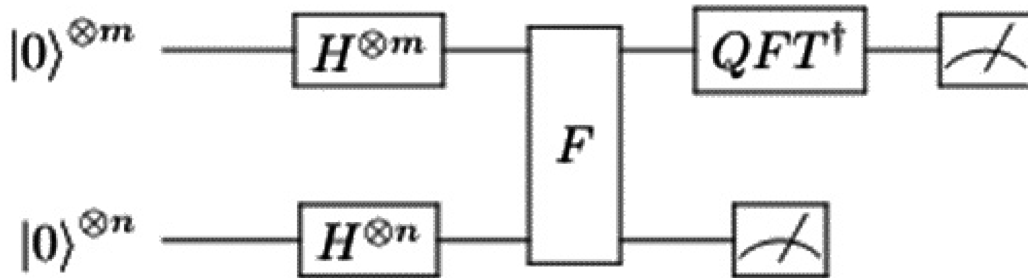


Figure 5.9 – A quantum circuit for implementing an n-qubit Shor's algorithm

   Both registers of Shor's algorithm are initialized to the |*0*> state for all the qubits. Therefore, the initial state is as follows:

$$\psi_0\rangle = 0_m\rangle \otimes 0_n\rangle$$

2. This step is then followed by the application of the Hadamard gate to each of the *m* qubits in the first register. The state of the quantum system then changes to the following:

$$\psi_1\rangle = H0 \otimes \cdots \otimes 0\rangle \otimes 0_n\rangle$$

3. After applying the Hadamard gate to the qubits in the first register, the next step in Shor's algorithm is to apply the unitary transformation *F*, which transforms the state of the system to the following:

$$|\psi_2\rangle = \frac{1}{\sqrt{2^m}} \sum_{x \in \{0,1\}^m} |x\rangle \otimes |a^x mod N\rangle$$

4. This step is followed by the application of the inverse QFT on the top register. This is followed by the measurement of the qubits in the second register. Finally, the qubits in the first register are also measured.

NOTE

The link for this code can be found here: https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter05.

The Python code for implementing Shor's algorithm using Qiskit is shown as follows:

1. The first step involves importing the modules necessary for the implementation of Shor's algorithm:

```
from qiskit import BasicAer
from qiskit.aqua import QuantumInstance
from qiskit.aqua.algorithms import Shor
```

2. The next step involves the actual implementation of Shor's algorithm and simulating it using **`qasm_simulator`**:

```
number = Shor(N=15, a=7)
simulator = BasicAer.get_backend('qasm_simulator')
results_dictionary = number.run(QuantumInstance(
                                backend=simulator,
                                shots=5))
#result = results_dictionary['number']
print(results_dictionary)
```

The preceding code can be summarized as follows.

After importing the necessary modules and classes, the circuit for computing the Shor algorithm is constructed, using the **`Shor`** class from Qiskit's Aqua element.

The output of implementing Shor's algorithm to factor the number 15 is shown in the following figure:

```
'factors': [[3, 5]],
```

Figure 5.10 – An output of factoring 15 using Shor's algorithm

We have covered Shor's algorithm in this section. The next section discusses Grover's search algorithm.

Exploring Grover's algorithm

Grover's search algorithm is used to search for an element in an unstructured database. It offers quadratic speed-ups compared to the conventional search algorithms. This quadratic speed-up is optimal! This means that in principle, this search algorithm cannot be improved further.

Grover's algorithm is the search algorithm that seeks to identify a distinct element, *w*, from an unstructured list of *n* elements. Typically, this list is encoded in terms of a function *f* such that the following applies for the distinct element (normally called a winner):

$$f(w) = 1$$

The following, meanwhile, applies for other elements:

$$f(x) = 0$$

To implement the Grover's algorithm, use the following list of steps:

1. The first step in the implementation of Grover's algorithm is to encode the following string:

$$x, w \in \{0,1\}^n$$

   The following should apply:

$$N = 2^n$$

2. The next step in the algorithm is to define the unitary transform (oracle), *F1*, which acts on the state |*x*> as follows:

$$F_1 x\rangle = (-1)^{f(x)} x\rangle$$

3. With the encodings of the elements and the oracle done, the next step in the implementation of Grover's algorithm is to initialize the *n* qubits. Therefore, the state of the system |*s*> at this stage will be as follows:

$$\psi_0\rangle = 0 \otimes \cdots \otimes 0\rangle$$

4. The next step is to apply the Hadamard gate to all the *n* qubits such that |*s*> becomes the following:

$$\psi_1\rangle = H0 \otimes \cdots \otimes 0\rangle$$

5. This step is followed by the application of *F1* to the state. The next step is to apply another transformation, *F2*. For state |*s*>, this transformation is given as follows:

$$F_2(s) = 2s\rangle\langle s - I$$

Here, *I* is an identity matrix. Therefore, the new state of the quantum system will be given as follows:

$$\psi_2\rangle = F_2 F_1 \psi_1\rangle$$

6. The previous step is repeated *t* times such that eventually, the state of the quantum system becomes the following:

$$\psi_t\rangle = (F_2 F_1)^t \psi_1\rangle$$

7. Finally, the measurement is performed on the *n* qubits.

NOTE

The link for this code can be found here: https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter05.

The Python code for implementing Grover's algorithm using Qiskit is shown as follows:

1. The first step involves importing the modules necessary for the implementation of Grover's algorithm:

```
from qiskit import QuantumRegister, ClassicalRegister,\
QuantumCircuit, execute, BasicAer
from qiskit.visualization import plot_histogram
import matplotlib.pyplot as plt
```

2. The next step is to define the quantum circuit to implement Grover's algorithms. In this case, the circuit to be used includes a two-qubit quantum register and a two-bit classical register:

```
c = ClassicalRegister(2, 'c')
q = QuantumRegister(2,'q')
qc = QuantumCircuit(q,c)
```

3. Then, the quantum gates are implemented in order to realize this circuit:

```
qc.h([q[0]])
qc.h([q[1]])
qc.x([q[0]])
qc.x([q[1]])
qc.cz(0,1)
qc.x([q[0]])
qc.x([q[1]])
```

```
qc.h([q[0]])
qc.h([q[1]])
qc.z(q[0])
qc.z(q[1])
qc.cz(0,1)
qc.h([q[0]])
qc.h([q[1]])
```

4. After applying the Hadamard, X, Z, and controlled-NOT gates as shown in the previous code block, the next step is to perform measurement on the two qubits and store the results in the two-bit classical register:

```
qc.measure(q[0], c[0])
qc.measure(q[1], c[1])
qc.draw('mpl')
plt.show()
```

5. Finally, use `qasm_simulator` to simulate this quantum circuit on the local simulator:

```
simulator = BasicAer.get_backend('qasm_simulator')
job = execute(qc, simulator, shots=1024)
result = job.result()
count = result.get_counts(qc)
plot_histogram(count)
plt.show()
```

The preceding code can be summarized as follows.

First, the necessary modules are imported. This is followed by the construction of the circuit to implement Grover's algorithm. This circuit can be summarized by the following figure:

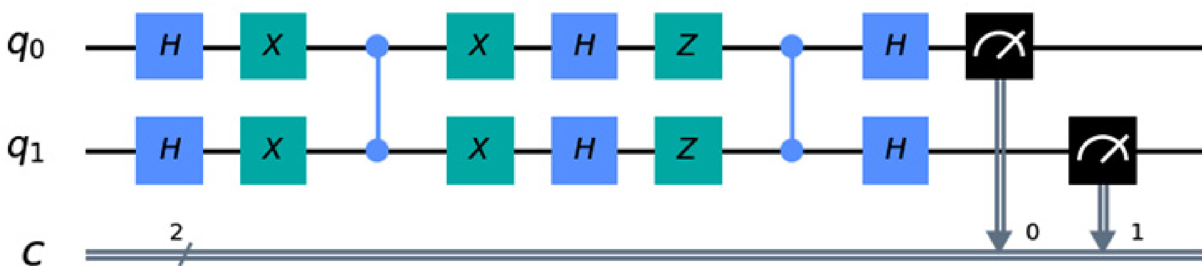


Figure 5.11 – A circuit for implementing Grover's algorithm

After constructing the Grover circuit, the execution of this circuit is simulated using `qasm_simulator`.

The output for implementing the two-qubit Grover's algorithm is shown in the following figure. This circuit outputs the bit string `'00'`:

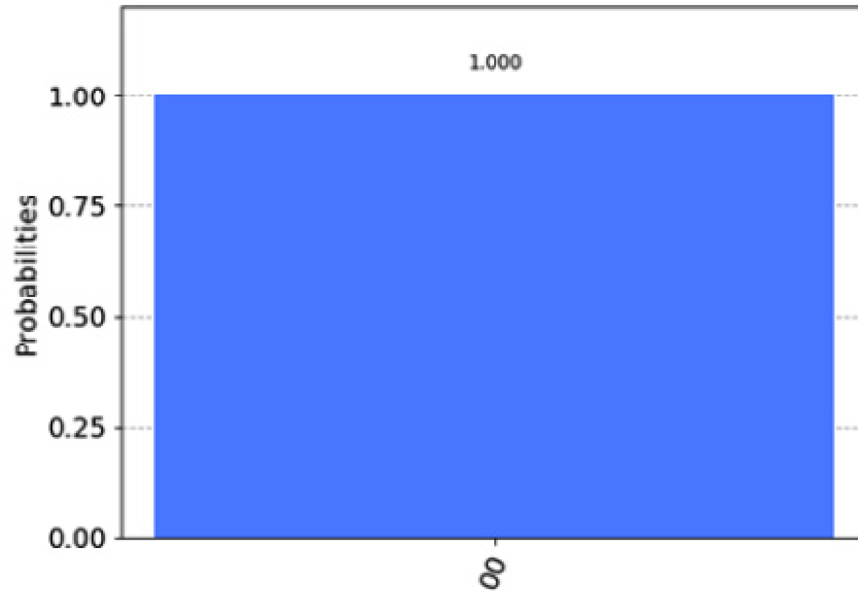


Figure 5.12 – An output for implementing a two-qubit Grover's algorithm

In this section, we have covered Grover's algorithm. We have provided the background information of the algorithm. Furthermore, we have discussed the implementation of Grover's algorithm using Python and Qiskit. The next section provides a summary of what was covered in this chapter.

Summary

In this chapter, we have explored a variety of quantum algorithms and techniques. We have also learned how these quantum algorithms and techniques offer advantages over their classical counterparts. The quantum algorithms and techniques covered in this chapter include Deutsch's algorithm, the Deutsch-Josza algorithm, the Bernstein-Vazirani algorithm, the QFT and quantum phase estimation techniques, Simon's algorithm, Shor's algorithm, and Grover's algorithm.

In the next chapter, we will cover another aspect of quantum information processing, namely quantum non-local games.

Further reading

- Nielsen, M. A and Chuang, I. L. (2011) *Quantum computation and quantum information: 10th Edition*. New York, NY, USA: Cambridge University Press, 1107002176, 9781107002173.
- Bernhardt, C. (2019) *Quantum computing for everyone*. MIT Press.
- Abraham Asfaw, Luciano Bello, Yael Ben-Haim, Sergey Bravyi, Lauren Capelluto, Almudena Carrera Vazquez, Jack Ceroni, Richard Chen, Albert Frisch, Jay Gambetta, Shelly Garion, Leron Gil, Salvador De La Puente Gonzalez, Francis Harkins, Takashi Imamichi, David McKay, Antonio Mezzacapo, Zlatko Minev, Ramis Movassagh, Giacomo Nannicni, Paul Nation, Anna Phan, Marco Pistoia, Arthur Rattew, Joachim Schaefer, Javad Shabani, John Smolin, Kristan Temme, Madeleine Tod, Stephen Wood, and James Wootton. (2020). *Learn Quantum Computation Using Qiskit*. IBM. Available online: http://community.qiskit.org/textbook.

Section 3: Deep Diving into Quantum Information

This section discusses the strategies that players can use in a non-local game and then proceeds to implement examples of CHSH and GHZ games. We will also showcase applications of quantum cryptography and Python implementations of certain quantum key distribution schemes. Furthermore, we will cover the key concepts and ideas in continuous-variable **quantum information processing** (**QIP**). Finally, we will talk about current research being undertaken, developments by various vendors, and what the future holds for the QIP domain.

This section comprises the following chapters:

- *Chapter 6*, *Non-Local Quantum Games*
- *Chapter 7*, *Quantum Cryptography*
- *Chapter 8*, *Quantum Machine Learning*
- *Chapter 9*, *Continuous-Variable Quantum Information Processing*
- *Chapter 10*, *Current Trends in Quantum Information Processing*

Chapter 6: Non-Local Quantum Games

This chapter covers another aspect of quantum information processing, namely, **quantum game theory**. Furthermore, in this chapter, we will showcase tasks in which quantum mechanics can allow participants (players) to perform more optimally than if they made no use of quantum resources.

In this chapter, we will first provide an introduction to classical game theory and then proceed to quantum game theory. Furthermore, we will discuss the strategies that players can use in non-local quantum games and proceed to implement examples of *CHSH* and *GHZ* games.

In this chapter, we will cover the following main topics:

- Understanding classical game theory
- Understanding quantum game theory
- Understanding non-local quantum games
- Understanding the CHSH game
- Understanding the GHZ game
- Understanding the XOR game

Technical requirements

In the next section, we will discuss the technical requirements needed for you in order to follow this chapter.

The requirements for this chapter are the following:

- A basic understanding of the Python programming language
- Navigation of Google's Colab environment

The GitHub link for this chapter can be found here:

https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter06.

The next section will introduce basic information regarding classical game theory. This information will form the basis for the quantum non-local games that will be covered later in this chapter.

Understanding classical game theory

In order to delve into non-local quantum games, in this section, we will first discuss non-quantum (classical) game theory, together with its quantum counterpart. The former (non-quantum/classical game theory) will just be referred to as game theory in this chapter.

Game theory tends to arise in virtually every facet of human (even non-human) interaction. Its applications can be found in diverse areas such as economics, business, project management, political science, finance, military science, psychology, biology, mathematics, computer science, philosophy, law, and international relations.

In essence, game theory is a field of mathematics that studies interactive decisions made by rational agents (players). The key assumption in game theory is that the interacting agents are acting rationally. This assumption implies that game theory cannot be satisfactorily applied in cases where agents act irrationally.

The main objective of game theory is to use mathematical models in order to analyze situations of interactive decision making. This analysis is intended to provide predictions of the behavior of the interacting rational agents and sometimes also to provide suggestions to the decision makers.

In game theory, the game consists of at least three essential elements, namely:

- The players (the agents)
- The strategies
- The payoffs

Based on the information provided previously, it should further be stated that the game in game theory can be thought of as involving the following:

- A number of players (at least two) who participate in the game.
- The strategies, which are the plans that are used by each player. These plans describe the action to be taken by each player in any situation.
- The payoffs, which are the rewards for each player for all the combinations of the strategies.

Now that we have provided the basic exposition to game theory, the next step is to delve further, focusing more on the history of game theory.

A brief history of game theory

The roots of game theory can be traced back to the eighteenth century. It was first developed by mathematicians, who started investigating parlor games, in order to formulate optimal strategies in such games.

In 1713, a mathematician named Charles Waldegrave made one of the early key contributions to the field of game theory. He found a solution to one of the parlor games of the time (the problem for which he found the solution has since been referred to as the Waldegrave problem). Waldegrave then communicated his solution to his contemporaries. These contemporaries included the likes of Pierre-Remond de Montmort and one of the famous members of the Bernoulli family, namely, Nicolas Bernoulli.

Another early contribution to game theory came in the nineteenth century. In 1838, another mathematician, Augustin Cournot, explored the game of oligopoly. He also presented a solution to this game, and demonstrated that such a solution is an equilibrium of the game.

In 1913, a mathematician and a logician by the name of Ernst Zermelo developed what could be described as the first theorem of game theory. This theorem, which is known as the Zermelo theorem, results from the game of chess. The Zermelo theorem asserts that in a game of chess, either a white (player) can force a win, or a black (player) can force a win, or both can force a draw.

Another contribution to game theory came from Emile Borel in the 1920s (1921–1927). Borel's key contribution to game theory involved the introduction of strategies to this field (of game theory).

Up to this point, game theory did not exist as true, rigorous, and general theory until the contributions of Jon von Neumann and Oskar Morgenstein in the late 1920s and the early 1930s. In 1928, von Neumann published a seminal paper that focused on the development of a theory for strategies that are used in game theory.

The von Neumann paper of 1928 was then followed by the publication of a seminal book by von Neumann and Morgenstein in 1944. This book discussed many areas of game theory, some of which are to be covered later in this chapter. These areas include the following:

- Two-player zero-sum games
- Cooperative games
- Utility theory, which later found applications in the field of economics

Following the seminal work of von Neumann and Morgenstein, John Nash contributed further to the field of game theory in the 1950s. Nash's contribution to game theory involved the following areas:

- Non-cooperative games
- Cooperative games
- Bargain games

In the ensuing decades, game theory has been greatly explored and enriched. Furthermore, it has found applications in different fields, as stated earlier in this chapter. If you are interested in knowing more about the history of game theory, you are advised to go through some of the references provided in the *Further reading* section.

Now that we have provided a brief history of game theory, it is time to cover other aspects of game theory.

Having provided a brief history of game theory, the next subsection covers the strategies used in game theory.

Leaning about strategies in game theory

As already stated in this chapter, a strategy is a set of actions that each player takes in the game. Strategies are a significant component of game theory. A strategy gives a complete specification of a player's behavior. This (complete) specification describes actions that a player would take at each of their decision points (information sets).

In essence, for an n-player game, a strategy is a function that assigns each of the players' information sets (places where a player makes decisions) to an action available to said player at that information set. Therefore, it can be seen that a strategy can be thought of as a prescription for how to play a game in all possible scenarios.

Consider an n-player game. In such a setting, each player, i, has their own set of possible strategies, denoted as S_i. In order to play a game, each player, i, then selects the strategy:

$$s_i \in S_i$$

In such a setting, a vector of strategies (strategy set) selected by a player i is given by:

$$s = (s_1, \cdots, s_n)$$

It is worth noting that a vector of strategies selected by the players in an n-player game ultimately determines the outcome for each player in the game.

A strategy that results in the player having the best outcome regardless of the strategies from other players is called the **dominant strategy**. On the other hand, a strategy that results in the worst outcome regardless of the strategies of other players is known as the **dominated strategy**.

Another comparison of strategies in game theory involves whether the strategy is deterministic. If the strategy of a player involves a deterministic set of actions, it is said to be a **pure strategy**. On the other hand, if the strategy is stochastic (in which case, the player chooses randomly among a set of possible actions), then the strategy is said to be a **mixed strategy**.

In this subsection, we have covered the strategies used in game theory. Furthermore, we have briefly discussed the dominant strategy. Finally, we have contrasted the pure and mixed states. The following section will explore the differences between cooperative games and the non-cooperative games.

Exploring cooperative and non-cooperative games

In game theory, games can either be *cooperative* or *non-cooperative*. A non-cooperative game is also referred to as a **competition** (**strategic**) **game**, while a cooperative game is also known as a **coalition** (**coalitional**) **game**.

In a non-cooperative game, there is a competition among players. Furthermore, the aim of a player in non-cooperative games is to win such games. Thus, in non-cooperative games, players take individual actions, and the objective of each player is to win the game.

In non-cooperative game theory, the most important solution concept is **the Nash equilibrium**, which is a concept named after John Nash discussed earlier in this chapter. In essence, a Nash equilibrium is a stable set of strategies, one for each player (in a non-cooperative game) such that no player has an incentive to change their strategy given what other players are doing.

For an *n*-player non-cooperative game, a Nash equilibrium has the property that each player's choice of action is the best response to the choices of the *i* other competing players.

On the other hand, the objective and the mechanism of cooperative games are in stark contrast to those of non-cooperative games. In cooperative games, the actions of the players are coordinated.

Furthermore, unlike in non-cooperative games, where the objective of the player is to win the game, the objective of a player in cooperative games is to achieve agreed-upon principles, such as justice, efficiency, non-discrimination, and fairness.

Finally, the most important solution concept in cooperative games is the **Shapley value**. This is characterized by the fair distribution of both the costs and the gains to several players working in coalition.

Having explored the differences between cooperative and non-cooperative games in this subsection, the next subsection will focus more on non-cooperative games. It will introduce and then explore the differences between the zero-sum games and non-zero-sum games.

Exploring zero-sum and non-zero-sum games

In non-cooperative game theory, games can be categorized using different criteria. One of the criteria of categorizing the games is whether such games are zero-sum or non-zero-sum games. For our analysis here, we will only focus on a two-player game. However, the analysis can be generalized to any *n*-player game, where $n > 2$.

A zero-sum two-player game is a game in which one player wins what the other player loses. In this game, the players have opposite evaluations of the outcome, thereby resulting in the payoff of zero.

From the information provided previously, it is clear to see that in two-player zero-sum games, the players are antagonistic, since in order for one player to win, the other player must lose.

On the other hand, in non-zero-sum competition games, one player's gain (loss) does not necessarily result in the other player's loss (gain). Thus, in non-zero-sum games, there is a possibility of a non-zero net gain (loss). This implies that the payoff is not zero, unlike in the case of the zero-sum games.

In this subsection, we have explored the differences between zero-sum and non-zero-sum non-cooperative games. The following subsections will cover some of the examples of these non-cooperative games.

Understanding the prisoner's dilemma

One example of a non-zero-sum non-cooperative game is the prisoner's dilemma. For simplicity, we will only focus on the two-player version of the prisoner's dilemma non-cooperative game.

Now, let's briefly explore the prisoner's dilemma. Consider two criminals—Alice and Bob. These criminals are arrested and ultimately imprisoned, with the condition that Alice and Bob are placed in different prison cells in such a way that they cannot communicate. However, the prosecutors realize that they do not have enough evidence to guarantee conviction of Alice and Bob. So, they (the prosecutors) come up with a shrewd plan! They offer a bargain to either Alice or Bob, on condition that either player comes clean and confesses.

Based on the preceding information, there are four possible scenarios:

- **Neither Alice nor Bob confess**: If neither Alice nor Bob confesses, each is sentenced to 3 years' imprisonment.
- **Alice confesses while Bob does not**: If Alice confesses and Bob does not, Alice is sentenced to 1 year in prison, while Bob is sentenced to 4 years in prison.
- **Bob confesses while Alice does not**: If Bob confesses and Alice does not, Bob is sentenced to 1 year in prison, while Alice is sentenced to 4 years in prison.
- **Both Alice and Bob confess**: If both Alice and Bob confess, they are each sentenced to 2 years in prison.

Using the pay-off matrix, the two-player prisoner's dilemma game between Alice and Bob can be given as follows:

		Bob	
		Confesses	**Does not confess**
Alice	**Confesses**	(3, 3)	(1, 4)
	Does not confess	(4, 1)	(2, 2)

Table 6.1 – A pay-off matrix for a two-player prisoner's dilemma game

Using the preceding pay-off matrix, the Nash equilibrium state is reached if both Alice and Bob confess.

Using the `python`, `numpy`, and the `nashpy` libraries, the code snippet for implementing a two-player prisoner's dilemma is given as follows:

```
import nashpy as nash
import numpy as np
Alice = np.array([[3, 1], [4, 2]])
```

```
Bob = np.array([[3, 4], [1, 2]])
prisoner_dilemma = nash.Game(Alice, Bob)
print(prisoner_dilemma)
Alice_sigma = np.array([1, 0])
Bob_sigma = np.array([1, 0])
print(prisoner_dilemma[Alice_sigma, Bob_sigma])
for eq in prisoner_dilemma.support_enumeration():
    print(eq)
```

As you can see from the preceding code snippet, the first step is to import the two Python libraries, namely, **`numpy`** and **`nashpy`**, with the former being given an alias of **`np`**, and the latter being given an alias of **`nash`**.

After importing the necessary libraries, the next step is to construct the pay-off matrix for both Alice and Bob. This is followed by the creation of the prisoner's dilemma game using the **`nash.Game`** class. After creating the game, the utilities (payoffs) are then calculated, followed by a calculation of the Nash equilibrium.

In this subsection, we have explored one of the examples of non-cooperative games, namely, the prisoner's dilemma game. In the next subsection, we will explore another example of non-cooperative games, namely, the battle of the sexes game.

Understanding the battle of the sexes game

The game of the battle of the sexes is another example of non-zero-sum non-cooperative games. In order to explore this game, we will consider the game being played by two players, namely, Alice and Bob. The battle of the sexes game is relatively easier than the prisoner's dilemma.

Unlike in the case of the prisoner's dilemma (where Alice and Bob were criminals), in this case, Alice and Bob are wife and husband, respectively. Alice and Bob both want to go out for an evening. However, they have conflicting interests. Alice wants to go to the ballet, while Bob wants to go to a soccer match.

However, they both would prefer to be together, either at the soccer match or at the ballet, rather than going alone to separate events. That is, even though Bob would prefer to go to the soccer match, he would rather go to the ballet with Alice than to go to the soccer alone. This is also the case with Alice. Even though she prefers going to ballet, she would rather go to the soccer match with Bob than go to the ballet alone.

The pay-off matrix for the battle of the sexes game is given as follows:

Alice		**Bob**	
		Soccer	**Ballet**
	Soccer	(2,3)	(1,1)
	Ballet	(1,1)	(3,2)

Table 6.2 – A pay-off matrix for a battle of the sexes game

The Python code snippet for implementing the battle of the sexes game using **`python`**, **`numpy`**, and **`nashpy`** is given as follows:

```
import nashpy as nash
```

```
import numpy as np
Alice = np.array([[2, 1], [1, 2]])
Bob = np.array([[3, 1], [1, 3]])
battle_sexes= nash.Game(Alice, Bob)
print(battle_sexes)
Alice_sigma = np.array([1, 0])
Bob_sigma = np.array([1, 0])
print(battle_sexes[Alice_sigma, Bob_sigma])
for eq in battle_sexes.support_enumeration():
    print(eq)
```

The preceding code snippet is structurally similar to that of the prisoner's dilemma covered previously. It involves the importing of Python modules that are going to be used in the implementation of the battle of the sexes game. This is then followed by the construction of the pay-off matrix. Furthermore, the payoffs and Nash equilibrium are then calculated and displayed.

In this subsection and the one preceding it, we have explored two non-zero-sum non-cooperative games. It is now time to focus our attention on zero-sum non-cooperative games. This will be our focus in the following two subsections.

Understanding the matching pennies game

This game is one of the simplest examples of zero-sum non-cooperative games. This game involves two players. Let's continue with our tradition of calling these players Alice and Bob, as we did in the previous two-player games.

In the matching pennies game, both Alice and Bob have a coin. The idea is for each player to secretly flip their coin and simultaneously reveal the state of the coin to the opponent (Alice to Bob, and Bob to Alice). If both coins are in the same state (both heads, or both tails), then Alice wins the game. Otherwise, Bob wins the game.

The pay-off matrix for the matching pennies zero-sum non-cooperative game can be given as follows:

Alice		Bob	
		Heads	**Tails**
	Heads	(+1, -1)	(-1, +1)
	Tails	(-1, +1)	(+1, 1)

Table 6.3 – A pay-off matrix for a matching pennies game

From the preceding pay-off matrix, we can see that for each player, the sum of the outcome is always zero. This is to be expected, as this is a feature of zero-sum games. That is, in a two-player zero-sum game, the sum of one player's gain is equal to the other player's loss.

The Python code for implementing the matching pennies game using **`python`**, **`nashpy`**, and **`numpy`** is given as follows:

```
import nashpy as nash
import numpy as np
```

```
Alice = np.array([[1, -1], [-1, 1]])
Bob = np.array([[-1, 1], [1, -1]])
missing_pennies= nash.Game(Alice, Bob)
print(missing_pennies)
Alice_sigma = np.array([1, 0])
Bob_sigma = np.array([1, 0])
print(missing_pennies[Alice_sigma, Bob_sigma])
for eq in missing_pennies.support_enumeration():
    print(eq)
```

The preceding code can be summarized as follows. First, the Python modules necessary for the implementation of the matching pennies game are imported. This is then followed by the construction of the pay-off matrix, which, in turn, is followed by the calculation of both the payoffs and the Nash equilibrium.

In this subsection, we have covered one of the examples of zero-sum non-cooperative games, namely, the matching pennies game. In the next subsection, we will cover another example of the zero-sum non-cooperative game, namely, the rock-paper-scissors game.

Exploring the rock-paper-scissors game

Another example of the zero-sum non-cooperative game is the famous rock-paper-scissors game. In this game, there are two players choosing the moves from three options, namely, the rock, the paper, and the scissors. In this zero-sum game, we can have the following outcomes:

- The rock beats the scissors.
- The scissors beat the paper.
- The paper beats the rock.

The preceding outcomes are applicable for the players Alice and Bob, such that, if Alice chooses the scissors and Bob chooses the rock, Bob wins the game. On the other hand, if Alice chooses the paper and Bob chooses the rock, then Alice wins.

The pay-off matrix for the rock-paper-scissors game is shown as follows:

		Bob		
		Rock	**Paper**	**Scissors**
Alice	**Rock**	(0,0)	(-1,1)	(1,-1)
	Paper	(1,-1)	(0,0)	(-1,1)
	Scissors	(-1,1)	(1,-1)	(0,0)

Table 6.4 – A pay-off matrix for a rock-paper-scissors game

The Python code for implementing a rock-paper-scissors game using `python`, `numpy`, and `nashpy` is given as follows:

```
import nashpy as nash
```

```
import numpy as np
Alice = np.array([[0, -1, 1], [1, 0, -1], [-1, 1, 0]])
Bob = np.array([[0, 1, -1], [-1, 0, 1], [1, -1, 0]])
rock_paper_scissors = nash.Game(Alice, Bob)
print(rock_paper_scissors)
Alice_sigma = np.array([0, 0, 1])
Bob_sigma = np.array([0, 1, 0])
for eq in rock_paper_scissors.support_enumeration():
    print(eq)
```

The preceding code is similar in structure to those discussed earlier. It involves the importing of the necessary Python modules, which, in turn, is followed by the construction of the pay-off matrix. This is then followed by calculations of the utilities and the Nash equilibrium.

In this section, we learned about the history of game theory and the various game strategies. We also learned about cooperative and non-cooperative games, together with zero-sum and non-zero-sum games. This concludes the section on classical game theory. The next section will cover quantum game theory.

Understanding quantum game theory

Basically, a quantum game can be conceived of as any quantum system that can be manipulated by two or more parties, and where the payoff of the moves from these parties can be reasonably quantified.

The quantum game consists of the following elements:

- The number of players, N
- The set of strategies, S
- The utility/payoff functionals, P
- The initial quantum state, ρ
- The Hilbert space, H (refer to *Chapter 2*, *Quantum States, Operations, and Measurements*, for further details on the Hilbert space)

In essence, quantum game theory generalizes and quantizes classical game theory, discussed in the previous section, to the quantum domain. Simply put, quantum game theory is the game-theoretic framework that makes use of ideas from quantum mechanics.

In quantum game theory, quantum mechanical concepts, such as the quantum superposition of states and entanglement, can be used in the following ways:

- Quantum superposition of initial states
- Quantum entanglement of initial states
- Quantum superposition of strategies

As already stated previously, quantum game theory quantizes classical games. There are several reasons for quantizing classical games. These are as follows:

- To tap into the already established classical games in order to develop a new framework of game theory. This is due to the fact that both classical games and quantum mechanics are based on probability theory.
- To investigate whether the game-theoretic effects can be observed at a sub-atomic level, where quantum mechanics dictates the rules.
- To investigate the connection between game theory and the theory of quantum communication.
- To explore whether quantum mechanics can offer an advantage in order to win some of the specially designed non-cooperative games.

The first quantum game was proposed by Meyer in 1999. In this work, Meyer proposed a quantum game called the penny flip coin. This game is closely related to the matching penny game discussed earlier. Furthermore, Meyer demonstrated in this work that through the use of a quantum strategy, a player can increase their expected payoff.

The work of Meyer was then shortly followed by the work of *Eisert et. al.*, who further introduced quantum games and quantum strategies. Since these early contributions, there has been a significant progress in the development of a quantum game-theoretic framework.

So far, we have provided a brief introduction to quantum game theory. In the next subsection, we will focus more on quantum game theory by exploring non-local quantum games.

Understanding non-local quantum games

A non-local quantum game is a game where the quantum players are in different locations when they are playing the game. Non-local games are typically used to test non-locality in quantum systems. A two-player non-local game involves two players (Alice and Bob) and the referee.

The role of the referee, in this case, is to ensure that neither player (Alice or Bob) communicates with the other while playing the game. However, Alice and Bob are allowed to entangle their qubits (to share correlations). Typically, in non-local quantum games, the players are allowed to determine a joint strategy from the complete knowledge of an input distribution.

In this subsection, we have covered quantum non-local games. In the next subsection, we will briefly explore quantum strategies in non-local quantum games.

Exploring quantum strategies in non-local quantum games

In classical game theory, strategies can either be pure (deterministic) or mixed (probabilistic). On the other hand, in quantum game theory, the strategies can only be probabilistic.

The quantum strategy for each player, denoted by s_i, is a player's procedure for deciding which action to take, depending on the player's information. The quantum strategy space, $S = \{s_i\}$, is a set of strategies available to the player.

In order to probe the quantum strategies further, consider a two-player quantum game, τ, characterized by:

$$\tau = (H, \rho, S_A, S_B, P_A, P_B)$$

where H is the Hilbert space, ρ is the initial quantum state, S_A and S_B are the strategies of the two players, and P_A and P_B are the pay-off functionals of these players. In such a case, quantum strategies s_A and s_B, which can be given as:

$$s_A \in S_A$$

and

$$s_B \in S_B$$

are the quantum operations.

A two-player quantum game is referred to as a zero-sum quantum game if the expected payoffs total zero for all pairs of quantum strategies. Otherwise (if this is not the case), it is called a **non-zero-sum quantum game**.

Furthermore, consider the strategies s_A, s'_A, s_B, and s'_B such that, $s_A, s'_A \in S_A$ and $s_B, s'_B \in S_B$.

Furthermore, the quantum strategies s_A and s'_A are equivalent if $P_A(s_A, s_B) = P_A(s'_A, s_B)$ and $P_B(s_A, s_B) = P_A(s'_A, s_B)$ for all s_B.

Similarly, quantum strategies s_B and s'_B are equivalent if $P_B(s_A, s_B) = P_B(s_A, s'_B)$ and $P_A(s_A, s_B) = P_B(s_A, s'_B)$ for all s_A.

Additionally, a quantum strategy s_A is called a dominant strategy if:

$$P_A(s_A, s'_B) \geqslant P_A(s'_A, s'_B)$$

In a similar manner, a quantum strategy s_B is called a dominant strategy if:

$$P_B(s'_A, s_B) \geqslant P_B(s'_A, s'_B)$$

At this juncture, it is worth noting that a pair of quantum strategies (s_A, s_B) is said to be an equilibrium in strategies if s_A and s_B are the dominant strategies for the two players in this two-player quantum game.

Finally, a pair of quantum strategies (s_A, s_B) is called a Nash equilibrium if $P_A(s_A, s'_B) \geqslant P_A(s'_A, s'_B)$ and $P_B(s'_A, s_B) \geqslant P_B(s'_A, s'_B)$.

In this subsection, we have provided a brief exposition to non-local quantum strategies. In the next subsection, we will discuss one of the examples of non-local quantum games, namely, the **Clauser-Horne-Shimony-Holt** (**CHSH**) quantum game.

Understanding the CHSH game

The **CHSH** quantum game is an example of a non-local quantum game. It uses the CHSH inequality that was discussed earlier in this book, in *Chapter 3*, *Entanglement and Teleportation*. It just turns inequality into a quantum game that is played by Alice and Bob, in the presence of the referee.

The CHSH quantum game can have an application in the witnessing of bipartite entanglement in quantum systems.

The CHSH quantum game can be summarized as follows. Consider two players, Alice and Bob. They respectively receive bits x and y from the referee, and they have to respond with bits a and b, respectively. Additionally, Alice and Bob win if:

$$a \oplus b = x \cdot y$$

with both Alice and Bob communicating on the joint strategy before the start of the game, and not being permitted to communicate during the game.

The following steps summarize the procedure used for implementing the CHSH game:

- Before the start of the game, Alice and Bob share a Bell state (Bell pair).
- If Alice receives x = 0, she measures her qubit and outputs the results.
- If she receives x = 1, she applies the rotation gate of $\pi/_2$ along the y axis, Ry($\pi/_2$), and measures her qubit.
- If Bob receives y = 0, he applies the rotation gate Ry(π/4) and then measures his qubit.
- If Bob receives y = 1, he applies the rotation gate Ry(-π/4) and then measures his qubit.

The following steps demonstrate implementation of the CHSH game using Python:

1. Import the necessary modules:

```
from qiskit import *
import numpy as np
np.random.seed(42)
```

2. Then, define the quantum circuit for implementing the CHSH game:

```
def CHSH_circuit(x,y):
    a0=0
    a1=np.pi/2
    b0=np.pi/4
    b1=-np.pi/4
    circ = QuantumCircuit(2,2)
    circ.h(0)
    circ.cx(0,1)
    if(x==0):
        circ.ry(a0,0)
    else:
```

```
        circ.ry(a1,0)
    if(y==0):
        circ.ry(b0,1)
    else:
        circ.ry(b1,1)
    circ.measure([0,1], [0,1])
    return circ
```

3. Furthermore, define the probability of Alice and Bob winning this game:

```
def winning_probability(backend, shots):
    a0=0
    a1=np.pi/2
    b0=np.pi/4
    b1=-np.pi/4
    total = 0
    circuits = [CHSH_circuit(0,0), CHSH_circuit(0,1),
                CHSH_circuit(1,0), CHSH_circuit(1,1)]
    job = execute(circuits, backend=backend,
                  shots = shots)
    for qc in circuits[0:3]:
        counts = job.result().get_counts(qc)
    if('00' in counts):
        total += counts['00']
    if('11' in counts):
        total += counts['11']
    counts = job.result().get_counts(circuits[3])
    if('01' in counts):
        total += counts['01']
    if('10' in counts):
        total += counts['10']
    return total/(4*shots)
```

4. Finally, simulate the CHSH circuit using **`'qasm_simulator'`** and display the probability of winning:

```
backend = Aer.get_backend('qasm_simulator')
print(winning_probability(backend, shots=1024))
```

The preceding code can be summarized as follows. First, the necessary Python modules are imported. This is then followed by construction of the CHSH quantum circuit, using two qubits (for Alice and Bob). This quantum circuit starts with both qubits being initialized to the quantum state, |0>. This is then followed by the creation of the Bell pair. Furthermore, depending on the classical

bits x and y for Alice and Bob, respectively, appropriate rotation gates are applied by both Alice and Bob. Eventually, both Alice and Bob measure their respective qubits.

Following construction of the quantum circuit for implementing the CHSH game, the next step in the preceding Python code is to simulate the circuit using `'qasm_simulator'`. The results obtained are then displayed.

In this subsection, we have covered the CHSH quantum game. We have also provided a Python code for implementing the CHSH game. In the next subsection, we will explore the GHZ game.

Understanding the GHZ game

Just like the CHSH quantum game discussed previously, the **Greenberger-Horne-Zeilinger** (**GHZ**) quantum game is also an example of a non-local quantum game. It is used to witness the tripartite entanglement in quantum systems. However, unlike the CHSH game, the GHZ game is a three-player game (Alice, Bob, and Charlie are the players).

In essence, the setup for the GHZ game is similar to that of the CHSH game, except that in the former, there are three players (Alice, Bob, and Charlie) who initially share the entanglement (tripartite entanglement), while in the latter, there are only two players (Alice and Bob) who share the entanglement (bipartite entanglement).

The three-player GHZ quantum game be summarized as follows. Consider three players, namely, Alice, Bob, and Charlie. The referee sends the bits x, y, and z to Alice, Bob, and Charlie, respectively. In response, the players Alice, Bob, and Charlie output the bits a, b, and c, respectively. The players win if:

$$a \bigoplus b \bigoplus c = x \vee y \vee z$$

The following procedure summarizes the implementation of the GHZ quantum game:

- Before the start of the game, ensure that all three players share the tripartite entanglement.
- Each player applies the Hadamard gate to their qubit when they receive 1.
- Each player measures their qubit state.

The following steps show the implementation of the GHZ quantum game using Python:

1. The first step entails the importing of the required modules:

```
from qiskit import *
from qiskit.visualization import plot_histogram
import numpy as np
np.random.seed(42)
```

2. The next step involves defining the quantum circuit that is used to implement the three-qubit GHZ game:

```
circ = QuantumCircuit(3,3)
circ.h(0)
circ.cx(0,1)
circ.cx(0,2)
circ.h(0)
```

```
circ.h(1)
circ.h(2)
circ.measure([0,1,2], [0,1,2])
```

3. The next step involves simulating the GHZ game using `'qasm_simulator'`:

```
backend = Aer.get_backend('qasm_simulator')
shots= 1024
job = execute(circ, backend=backend, shots = shots)
result = job.result()
count = result.get_counts(circ)
```

4. Eventually, the probability of winning the GHZ game is then calculated and displayed:

```
total = 0
if('000' in count):
    total += count['000']
if('011' in count):
    total += count['011']
if('101' in count):
    total += count['101']
if('110' in count):
    total += count['110']
probability_winning = total/shots
print("Probability of winning is:", probability_winning)
```

The preceding code can be summarized as follows. After importing the necessary modules for the implementation of the GHZ quantum game, the circuit for the game is then constructed. This is followed by the simulation of the circuit using `'qasm_simulator'`, and a display of the results obtained.

So far, we have covered the GHZ quantum game in this subsection. In the next subsection, we will cover the XOR quantum game.

Understanding the XOR game

Just like the CHSH and GHZ games, the XOR game is also an example of non-local quantum games. Additionally, the XOR game is the most studied and the simplest non-local quantum game.

The global task of the XOR quantum game is to apply the XOR (exclusive OR) function to the outcomes of all the players participating in the game. That is, an XOR quantum game is a binary game (a game whose outputs are bits) whose outcome depends solely on the XOR of the players' responses.

The CHSH game discussed earlier is an example of a two-player XOR quantum game. Another example of a multi-player XOR game is the GHZ quantum game, which was also discussed earlier.

In this section, we have covered the XOR quantum game. Furthermore, we have stated that the CHSH quantum game is an example of a two-player XOR quantum game. The next section provides a summary of what was covered in this chapter.

Summary

In this chapter, we have delved into one of the branches of quantum information processing, namely, quantum game theory. We have discovered that by using some of the concepts of quantum mechanics, the players in the game can perform more optimally than if they did not deploy quantum mechanics.

Furthermore, in this chapter, we provided the background information on classical (non-quantum) game theory. Additionally, we covered some of the examples of quantum game theory, and these examples include the CHSH quantum game, the GHZ game, and the XOR game.

By the end of this chapter, you should be familiar with the concepts associated with classical game theory. Furthermore, you should be able to implement some of the classical games using Python. Additionally, you should be familiar with concepts in quantum game theory. Furthermore, you should be able to use Python to implement non-local quantum games, such as the CHSH game and the GHZ game.

The next chapter explores quantum cryptography. Quantum cryptography is arguably the most successful branch of quantum information processing.

Further reading

- Barron, Emmanuel N. *Game Theory: an Introduction*. Vol. 2. John Wiley & Sons, 2013.
- Guo, Hong, Juheng Zhang, and Gary J. Koehler. *A Survey of Quantum Games*. Decision Support Systems 46, no. 1 (2008): 318-332.
- Kolokoltsov, Vassili. *Quantum Games: a Survey for Mathematicians*. arXiv preprint arXiv:1909.04466 (2019).

Chapter 7: Quantum Cryptography

This chapter covers arguably the most successful sub-field of quantum information processing, namely, **quantum cryptography**. Additionally, in this chapter, we will use Python as a tool for the hands-on implementation of both conventional cryptography and quantum cryptography.

In this chapter, we will showcase applications of quantum cryptography and Python implementations of certain quantum key distribution schemes. We will begin by implementing classical cryptographic primitives in Python. Furthermore, we will also learn how to implement quantum cryptographic primitives in Python.

In this chapter, we will cover the following main topics:

- Introducing classical cryptography
- Quantum cryptography
- Post-quantum cryptography

By the end of this chapter, you should be able to do the following:

- Implement classical cryptographic protocols such as Caesar's cipher and one-time pads in Python.
- Implement quantum cryptographic protocols such as the BB84, the B92, and the E91 in Python.
- Understand how post-quantum cryptography operates.

In the next section, we will cover the technical requirements for you to understand this chapter.

Technical requirements

The requirements for this chapter are the following:

- A basic understanding of the Python programming language
- Navigation of Google's Colab environment
- Elementary (post-secondary) mathematics

The GitHub link for this chapter can be found here:

https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter07

The next section will provide an introduction to classical cryptography.

Introducing classical cryptography

In this section, we will provide a basic introduction to classical cryptography. We will briefly go through the history of cryptography. Furthermore, we will explore cryptographic schemes such as the Diffie-Hellmann scheme. Finally, we will cover classical cryptographic primitives, such as random number generators.

Cryptography is the science and art of securing information in such a way that only the intended (legitimate) parties have access to such a communication. That is, cryptography makes it possible to ensure that illegitimate parties do not have access to the message

that is being protected by the legitimate, communicating entities.

Cryptography is carried out by first converting an ordinary piece of information (referred to as plaintext) into an unintelligible string of characters (referred to as ciphertext). This conversion process is done so that the information transmitted from the transmitter (conventionally called Alice) is not intercepted by an eavesdropper (conventionally called Eve) on its way to the receiver (conventionally called Bob).

Once the plaintext has been converted to ciphertext, it can then be transmitted over the channel to Bob. In the channel, Eve can have access to such a ciphertext, but cannot make sense of such information (ciphertext), since it is unintelligible. On the receiver's side (Bob's side), the ciphertext is then converted back to plaintext.

From the details discussed previously, it can be observed that cryptography can be divided into two processes. These processes are encryption and decryption. The former (encryption) consists of the conversion of plaintext to ciphertext, using a cryptographic key. On the other hand, the latter (decryption) consists of the conversion of ciphertext to plaintext, with the help of a cryptographic key.

A schematic diagram of the cryptographic system (cryptosystem) is shown here:

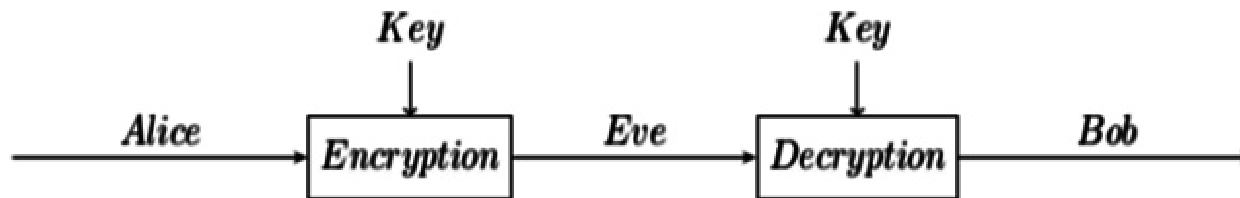


Figure 7.1 – A schematic diagram of a cryptographic system

The main goals of cryptography can be summarized as follows:

- **Confidentiality** (**secrecy**): To ensure that information is rendered inaccessible to the illegitimate communicating parties. Thus, secrecy ensures that there is no unauthorized access to the information.
- **Data integrity**: To ensure that information is not modified by illegitimate communicating parties during transmission.
- **Authentication**: To guarantee that the sender of the information is who they claim to be. This, in turn, ensures that the only parties that can legitimately communicate are those that are verified as genuine.
- **Non-repudiation**: To provide protection against one of the entities involved in a communication from denying having participated in such a communication.

Based on the nature of the cryptographic key being used, cryptography can be broadly divided into two branches, namely:

- Symmetric cryptography (secret-key cryptography)
- Asymmetric cryptography (public-key cryptography)

Symmetric cryptography uses a single cryptographic key that is shared by both Alice and Bob (recall that Alice and Bob are the legitimate communicating parties). In order to guarantee the protection of the information being shared between Alice and Bob, the cryptographic key used must be kept secret for as long as the information is kept secret.

Now that we have provided a brief description of symmetric cryptography, it is time to turn our attention to asymmetric cryptography. In this branch of cryptography, two separate but related cryptographic keys are used. One key, known as the private key, is kept secret, while the other key, known as the public key, is made public (and hence can be transmitted over an insecure but authenticated communication channel).

Having provided some background information on cryptography, it is now time to briefly explore the history of cryptography. Therefore, the next subsection will cover the history of cryptography.

A history of classical cryptography

Throughout history, cryptography has been used for various purposes, such as the following:

- Diplomatic communications
- Military operations
- Commercial activities
- Private communications
- Religious applications

Earlier implementations of cryptography, which is known as classical cryptography, were primarily concerned with the substitution of characters in a message. One of the most prominent schemes in classical cryptography is Caesar's substitution cipher.

Caesar's cipher

The Caesar's cipher cryptographic scheme uses a cyclic alphabet and substitutes a character with another three places later in the alphabet. Thus, a character *a* will be substituted with a character *d*.

As an example, suppose you wanted to communicate the message *YES* using Caesar's cipher. So, the encrypted message would be:

- Y = B
- E = H
- S = V

This is *BHV*.

The Python code for implementing Caesar's cipher is shown as follows:

```
message = "YES"
shift = 3
encryption = ""
for c in message:
    if c.isupper():
        c_unicode = ord(c)
        c_index = ord(c) - ord("A")
        new_index = (c_index + shift) % 26
        new_unicode = new_index + ord("A")
        new_character = chr(new_unicode)
        encryption = encryption + new_character
    else:
        encryption += c
```

```
print("Plain text:", message)
print("Encrypted text:", encryption)
```

The preceding Python code can be summarized as follows. First, the message that is to be encrypted using Caesar's cipher is provided. In this case, the message is *YES*. Then the shift of three steps (three letters, in a cyclic manner) is provided. Thereafter, each of the characters of the message are shifted using a shift of three. Finally, both the original message and the encrypted message are displayed.

Having covered Caesar's cipher, we will now turn our attention to the one-time pad. This will be covered next.

The one-time pad

Another key development in the history of classical cryptography was the development of the **one-time pad** (**OTP**). The OTP ensures that the encrypted message cannot be compromised, as long as the cryptographic key used to encrypt the message is used only once (hence, the one-time pad). Furthermore, the cryptographic key should have the same length (number of characters) as the message being encrypted.

The Python code for implementing the one-time pad scheme is shown as follows:

```
key = 5
message = "HELLO"
encrypt = ""
for i in range(len(message)):
    letter = ord(message[i])-65
    letter = (letter + key)%25
    letter +=65
    encrypt = encrypt + chr(letter)
print("Original message is:", message)
print("Encrypted message is:", encrypt)
```

The preceding Python code can be summarized as follows. First, the length of the key for the OTP scheme is set (in this case, it is set to 5). Then the message to be encrypted using OTP is provided. Furthermore, OTP is implemented using the given parameters, namely, the key length and the message. Finally, both the original message and the encrypted message are displayed.

So far, we have focused on the history of what is known as classical cryptography. Next, we will cover the history of what is now known as modern cryptography.

A history of modern cryptography

In the late 1970s, a new field of cryptography emerged. This would later be known as **modern cryptography**. Unlike classical cryptography, modern cryptography is more systematic, and takes a rigorous mathematical approach to data protection.

Modern cryptography is said to provide computational security. By this, we mean that the security of modern cryptography takes into account the computational capabilities of an eavesdropper, in order to guarantee the security of the cryptographic scheme.

In essence, modern cryptography provides a means of securing information by using mathematical techniques that are easier to compute for the legitimate communicating parties (due to the presence of a shared cryptographic key), but are computationally difficult to compute for illegitimate communicating parties (who do not have access to the cryptographic key).

One of the key contributions of modern cryptography was in the development of the cryptographic key exchange schemes. So far, we have just stated that these keys are required for both encryption and decryption processes. The natural question that arises then is how these keys are shared between Alice and Bob. Surely, merely transmitting them through the insecure channel would expose them to Eve! This challenge of exploring ways to securely exchange the cryptographic key between Alice and Bob through an insecure problem is known as the key distribution problem.

The advent of modern cryptography brought into sharp focus the key distribution problem discussed previously. Furthermore, in the late 1970s, Whitfield Diffie and Martin Hellman developed a cryptographic key exchange scheme that would later be known as the Diffie-Hellman key exchange protocol.

Diffie-Hellmann key exchange protocol

The Diffie-Hellman key exchange protocol, which is an example of asymmetric cryptography, can be summarized as follows. Alice and Bob agree on the key pair (*p*,*g*), where *p* is a large prime and *g* is a generator (base) such that:

$$0 < g < p$$

Alice then chooses her secret integer *a* (which is her private key) and then computes:

$$g^a mod p$$,

which is her public key, and the mod function is the modulo function. Alice's public key is also known by Bob and Eve.

On the other end of the communication, Bob chooses his integer *b* (which is his private key). By now, Bob knows his private key (*b*) and Alice's public key (g_a *mod p*). Furthermore, Bob can then perform the following computation:

$$(g^a)^{(b)} mod p$$

It is not difficult to observe that.

$$(g^a)^{(b)} mod p = (g)^{ab} mod p$$

Since Bob's public key is also known by Alice (and Eve for that matter), Alice can then compute.

$$(g^b)^{(a)} mod p = (g^a)^{(b)} mod p$$

Therefore, Alice and Bob have eventually generated a shared key, $g^{ab} mod p$.

This is the same for both Alice and Bob. However, since Eve has no access to either *a* or *b*, she cannot compute the shared key given above. That way, Alice and Bob have managed to securely exchange a cryptographic key that can be used for the encryption of information.

A Python code for implementing the Diffie-Hellman scheme is shown as follows:

```
import math
p = input("Enter the shared prime number: \n")
p = int(p)
g = input("Enter the shared base: \n")
g = int(g)
a = input("Enter Alice's secret key: \n")
a = int(a)
b = input("Enter Bob's secret key: \n")
b = int(b)
AlicePublicKey = (g ** a ) % p
BobPublicKey = (g ** b) % p
AliceSecret = (BobPublicKey ** a) % p
BobSecret = (AlicePublicKey ** b) % p
print(AliceSecret)
print(BobSecret)
```

The preceding code snippet can be summarized as follows. First, the required parameters are entered. These parameters include *p*, *g*, *a*, and *b* discussed earlier. Since, in Python, the input is, by default, a string datatype, these inputs are then converted into integer datatypes using the `int()` function.

The next step in the previous code snippet is for both Alice and Bob to compute their public keys. Once this is done, Alice uses Bob's public key and her private key (*a*) to compute the shared secret key. On the other hand, Bob uses Alice's public key and his private key (*b*) to compute the shared secret key.

So far, we have provided a brief history of cryptography. We have introduced classical cryptography and modern cryptography. Furthermore, we have briefly discussed how the Diffie-Hellman key exchange scheme works. In the next subsection, we will cover the cryptographic primitives.

Cryptographic primitives

Cryptographic primitives are algorithms that can be used to build cryptographic protocols (schemes). That is, cryptographic primitives are the basic building blocks of cryptographic protocols, such as the Diffie-Hellman key exchange discussed earlier.

Examples of cryptographic primitives include the following:

- **Pseudorandom number generators (PRNGs)**
- **Digital signatures**
- **Hash functions**
- **Secret sharing**

Pseudorandom number generators

Randomness can be generated either from a true source of randomness or from a source that may seem random but, in reality, is not. The former option results in true randomness, while the other results in pseudorandomness.

The numbers generated from a pseudorandom generator are very close to the random number, even though they come from a source that is known to be deterministic as opposed to being truly random.

The Python code snippet for generating pseudorandomness using the **`random`** module is given as follows:

```
import random
random.seed(42)
x = []
y = []
for i in range(10):
    a = random.random()
    x.append(a)
print(x)
for i in range(5):
    b = random.randint(0,4)
    y.append(b)
print(y)
```

The preceding code snippet can be summarized as follows. First, the **`random`** module is imported into the workspace. Next, the seed is used in order to ensure the reproducibility of results. Then, two empty arrays, **`x`** and **`y`**, are created. The first array (**`x`**) is then used to store 10 pseudorandom floating-point numbers, while the second array (**`y`**) is used to store five pseudorandom integers.

We have provided a brief introduction to one of the primitives of cryptography, namely, PRNGs. We have also seen how PRNGs can be implemented in Python. Next, we discuss another primitive of cryptography, namely, the hash function.

Hash functions

Informally, hash functions are also referred to as one-way hash functions or message digests. A hash function is a computationally efficient mapping that maps data of arbitrary length to a fixed length.

The output of a hash function, which is of fixed length, is referred to as the hash value. Typically, the input to the hash function is a bit string of arbitrary length, while the output (hash value) is a bit string of fixed length.

Hash functions are normally used for cryptographic tasks such as ensuring data integrity. In such tasks, they are used in conjunction with another cryptographic primitive, namely, the digital signature.

Examples of hash functions include the following:

- **Message digest 5** (**MD5**)
- **Secure hashing algorithm** (**SHA**)
- **BLAKE**

A Python code snippet for implementing the MD5 hash function using the `hashlib` module is as follows:

```
import hashlib
AliceMessage = hashlib.md5()
Alice = "This is Alice"
Alice = Alice.encode(encoding='utf-8')
AliceMessage.update(Alice)
print("Alice's MD5 digest is: \n", AliceMessage.hexdigest())
print("Alice's digest size is: \n",
     AliceMessage.digest_size)
print("Alice's block size is: \n", AliceMessage.block_size)
BobMessage = hashlib.md5()
Bob = "This is Bob"
Bob = Bob.encode(encoding='utf-8')
BobMessage.update(Bob)
print("Bob's MD5 digest is: \n", BobMessage.hexdigest())
print("Bob's digest size is: \n", BobMessage.digest_size)
print("Bob's block size is: \n", BobMessage.block_size)
```

The preceding code snippet first imports the `hashlib` module from Python's standard library. The `hashlib` module contains a variety of hash functions. In the preceding code snippet, we focused on the MD5 hash function, and this is represented by the `hashlib.md5()` function. The next step was to provide the message that would be hashed, and then properly encode such a message.

The MD5 is not the only hash function that can be implemented in Python. Several others, including SHA and BLAKE, can also be implemented. The following code shows the Python implementation of the SHA scheme using the `hashlib` module:

```
import hashlib
AliceMessage = hashlib.sha3_512()
Alice = "This is Alice"
Alice = Alice.encode(encoding='utf-8')
AliceMessage.update(Alice)
print("Alice's SHA digest is: \n", AliceMessage.hexdigest())
```

```
print("Alice's digest size is: \n",
      AliceMessage.digest_size)
print("Alice's block size is: \n", AliceMessage.block_size)
BobMessage = hashlib.sha3_512()
Bob = "This is Bob"
Bob = Bob.encode(encoding='utf-8')
BobMessage.update(Bob)
print("Bob's SHA digest is: \n", BobMessage.hexdigest())
print("Bob's digest size is: \n", BobMessage.digest_size)
print("Bob's block size is: \n", BobMessage.block_size)
```

Just like the code snippet of the previous example (the MD5 code snippet), the preceding code snippet computes the hash function for both Alice and Bob's messages. The only difference is that instead of using the MD5 algorithm, the Python code provided above uses the **Secure Hash Algorithm** (**SHA**). Therefore, the function used in this case is the `hashlib.sha3_512()` function, which uses SHA-3 and has a 512-bit hash value.

So far, we have considered two cryptographic primitives, namely, PRNGs and hash functions. The final cryptographic primitive to consider in this chapter is secret sharing. This primitive is explored next.

Secret sharing

So far, we have considered cryptography from an angle where two communicating parties want to share a secret message. However, this communication can involve more than two communicating parties. In such cases, a suitable cryptographic primitive is secret sharing.

Secret sharing is a cryptographic primitive that is used to distribute a secret value among multiple communicating parties. This distribution is done by splitting/fragmenting the original message into various small fragments, and sending each fragment to each communicating party.

In order to recover the original message that was shared, the communicating parties would have to collaborate so that they can combine their fragments into the original message. The security of this cryptographic primitive lies in the fact that in order to recover the original message, the communicating parties would have to collaborate with other parties and recombine the various fragments of the message. This way, no individual can have access to the entire message without first collaborating with other communicating parties.

The secret sharing cryptographic primitive was independently proposed by Adi Shamir and George Blakley in 1979. Here, we focus only on the Shamir version of the secret sharing primitive.

The Shamir version of the secret sharing cryptographic primitive can be summarized as follows. Consider a secret message denoted by S, which is then divided into n parts, such that:

$$S = s_1, s_2, \cdots, s_n$$

Then, a number k is chosen such that if

$$s_i \geq k$$,

the message *S* could be successfully recovered. However, if the parts of the messages combined together are

$$s_i \leq (k-1)$$

then the original message *S* could not be successfully recovered.

We have covered the cryptographic primitives. Furthermore, we have provided a brief introduction to classical cryptography and modern cryptography. In the next section, we will cover quantum cryptography.

Quantum cryptography

We have learned in the previous section that modern cryptography makes assumptions about the computational capabilities of an eavesdropper. Although this cryptographic approach is still widely used, it faces a serious challenge in that its security cannot be proven theoretically.

As an alternative to modern cryptography, there is another approach to data protection. This approach provides an information-theoretic security guarantee instead of the computational security guarantee provided by modern cryptography. This approach is known as **quantum cryptography**.

In essence, quantum cryptography is a cryptographic approach that uses quantum mechanical concepts such as *quantum entanglement*, *superposition*, and the *quantum no-cloning theorem* (these concepts were discussed earlier in *Chapter 2*, *Quantum States, Operations, and Measurements*) in order to protect data.

A history of quantum cryptography

The history of quantum cryptography can be traced back to the 1970s, when Stephen Wiesner proposed a scheme that can be used to provide secure quantum money. The ideas from Wiesner's work were then expanded by Charles Bennett and Gilles Brassard in order to propose the first ever **quantum key distribution** (**QKD**) protocol in 1984. This QKD protocol would later be referred to as the BB84 protocol.

Another major breakthrough in quantum cryptography came in 1991, when Artur Ekert proposed another QKD protocol based on quantum entanglement. This QKD protocol proposed by Ekert would later be referred to as the E91 protocol.

Since its inception some decades back, quantum cryptography has enjoyed a massive growth over the years. It is now arguably the most successful field of quantum information processing.

In this subsection, we have just covered a brief history of quantum cryptography. In the next subsection, we will cover quantum cryptography primitives.

Quantum cryptography primitives

Earlier in this chapter, we explored the cryptographic primitives for modern cryptography. Here, we explore some of quantum cryptography primitives. The cryptographic primitives to be explored include the following:

- **Quantum random number generator** (**QRNG**)

- **Quantum key distribution** (**QKD**)

Quantum random number generator

Unlike the PRNG, the QRNG is truly random. Its randomness stems from an intrinsic randomness of performing measurements of quantum systems.

The Python code snippet for implementing a QRNG is as follows:

1. First, import the necessary modules:

```
import random
from qiskit import *
from qiskit.visualization import plot_histogram
random.seed(42)
```

2. Then, define the circuit for implementing the QRNG:

```
circuit = QuantumCircuit(5,5)
circuit.h(0)
circuit.x(1)
circuit.h(1)
circuit.h(2)
circuit.x(3)
circuit.h(3)
circuit.h(4)
circuit.measure([0,1,2,3,4], [0,1,2,3,4])
circuit.draw(output='text')
```

3. Finally, simulate the circuit using the `'qasm_simulator'`:

```
simulator = Aer.get_backend('qasm_simulator')
result = execute(circuit, backend=simulator,
                 shots=1024).result()
plot_histogram(result.get_counts(circuit))
```

The preceding code snippet shows how a five-qubit QRNG can be implemented using Python and `qiskit`. The five qubits are, by default, initialized to the quantum state, |0>. So, what we did was to apply the `Hadamard` gate in odd-numbered qubits while applying the `X` gate followed by the `Hadamard` gate for all the even-numbered qubits. The next step is then to apply the measurement to each of the five qubits. Finally, `'qasm_simulator'` was used to simulate the results.

Now that we have covered the first quantum cryptography primitive, it is time to shift our focus to another primitive, namely, the key exchange primitive. This key exchange primitive in quantum settings is known as quantum key distribution.

Quantum key distribution

Quantum Key Distribution (**QKD**) is the key distribution scheme that employs quantum mechanical concepts in order to enable sharing of the cryptographic key between the legitimate communicating parties, Alice and Bob. The use of quantum mechanics ensures that the presence of an eavesdropper, Eve, could be revealed to Alice and Bob.

QKD uses two communication channels, namely, the quantum channel and the classical channel. The quantum channel is used for quantum communication, while the classical channel is used for classical post-processing. The following diagram shows the schematic diagram of a typical QKD scheme:

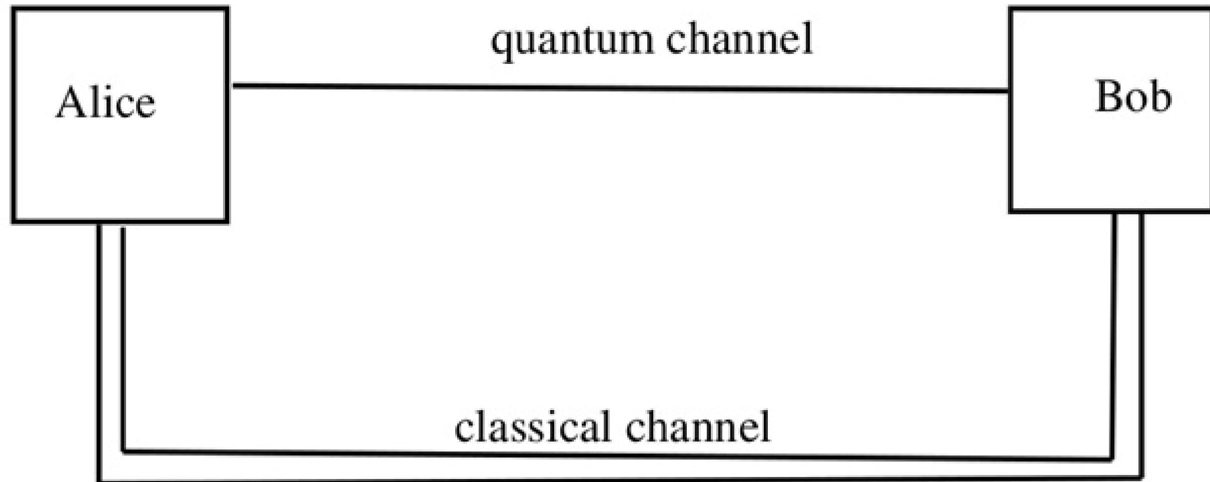


Figure 7.2 – A schematic diagram of a typical QKD scheme

Typically, a QKD protocol should address some or all of the following issues: security, feasibility, and communication distance.

QKD protocols, which will be discussed later in this chapter, can be broadly divided into two categories, namely:

- *Prepare-and-measure QKD protocols*. The BB84 discussed earlier is an example of this category.
- *Entanglement-based QKD protocols*. The E91 protocol is an example of this category.

A typical QKD protocol is implemented through the phases summarized in the following diagram:

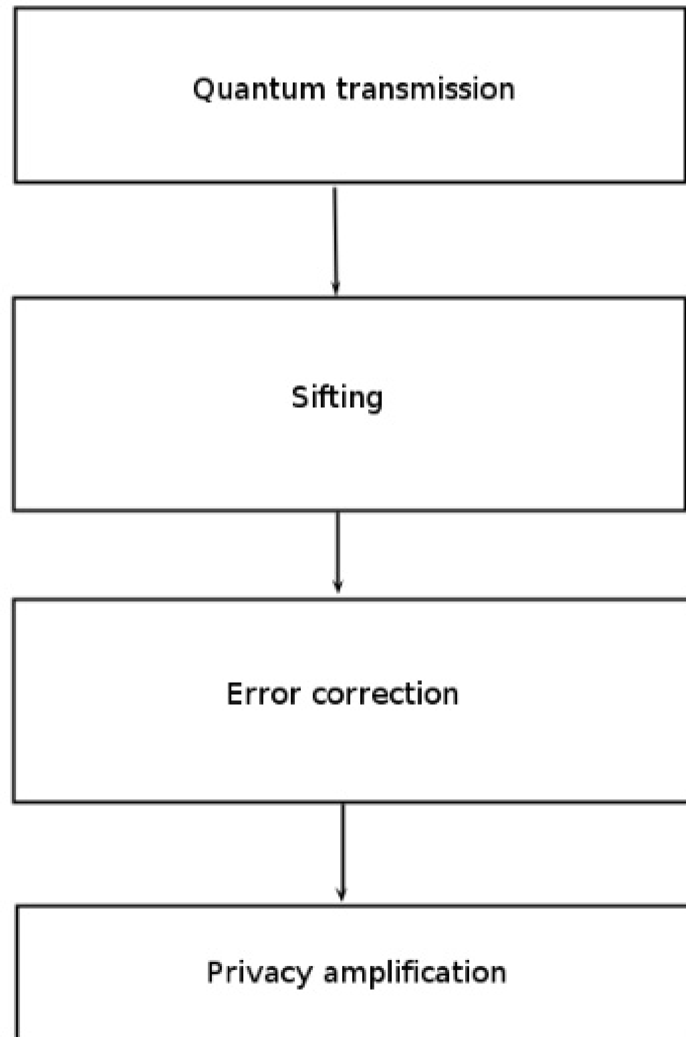


Figure 7.3 – A diagram of a typical QKD protocol implementation

We have so far covered another quantum cryptographic primitive, and this primitive is referred to as the QKD. Having provided the background information on QKD, it is now time to focus on some of the QKD protocols. This will be discussed in the next subsection.

Quantum key distribution protocols

In this subsection, we will cover some of the QKD protocols. The protocols to be covered in this subsection include the following:

- The BB84 QKD protocol
- The B92 QKD protocol
- The SARG04 QKD protocol
- The E91 QKD protocol

The BB84 protocol

The BB84 protocol was the first QKD protocol to be invented by Bennett and Brassard. It uses four quantum states, which comprise any pair of conjugate states. The BB84 protocol functions as follows. Alice randomly chooses a quantum state and sends it over to Bob. On the receiving side, Bob then randomly chooses a measurement basis, and performs the measurement.

Once the quantum communication is done, Alice and Bob both use a classical channel to compare their values. Where they used similar bases for state preparation (Alice) and measurement (Bob), they keep bit values corresponding to such cases. Otherwise, they discard the values.

The Python code snippet for implementing the BB84 protocol is shown as follows:

1. The first step involves importing the necessary modules:

```
from qiskit import QuantumCircuit, execute, Aer
from qiskit.visualization import *
import matplotlib.pyplot as plt
import numpy as np
np.random.seed(42)
```

2. The next step is to define the circuit for generating a random sequence of bit strings:

```
circ = QuantumCircuit(1,1)
circ.x(0)
circ.barrier()
circ.h(0)
circ.barrier()
circ.measure(0,0)
circ.barrier()
backend = Aer.get_backend('qasm_simulator')
```

3. Alice then uses this defined random circuit to generate 128 random bits:

```
result = execute(circ, backend, shots=128,
                 memory = True).result()
bits_alice = [int(q) for q in result.get_memory()]
print(bits_alice)
```

4. Furthermore, Alice uses the random circuit to randomly choose the basis she is going to use in order to implement the BB84 protocol:

```
result = execute(circ, backend, shots=128,
                 memory = True).result()
basis_alice = [int(q) for q in result.get_memory()]
print(basis_alice)
```

5. The next step involves Bob randomly choosing his basis using the random circuit defined earlier:

```
result = execute(circ, backend, shots=128,
                 memory = True).result()
basis_bob = [int(q) for q in result.get_memory()]
```

```
print(basis_bob)
```

6. Now, Alice encodes the random bits she has generated into qubits and sends them over to Bob:

```
bits_bob = []
for i in range(128):
    circ_send = QuantumCircuit(1,1)
    if bits_alice[i]:
        circ_send.x(0)
    if basis_alice[i]:
        circ_send.h(0)
    if basis_bob[i]:
        circ_send.h(0)
    circ_send.measure(0,0)
    result = execute(circ_send, backend, shots = 1,
                     memory = True).result()
    bits_bob.append(int(result.get_memory()[0]))
print(bits_bob)
```

7. Bob then performs measurements and communicates to Alice regarding the bases he used to perform measurements. Eventually, if Bob's bases match with Alice's, then they generate and agree on a secret key:

```
key = []
for i in range(128):
    if basis_alice[i] == basis_bob[i]:
        key.append(bits_bob[i])
print("Key length", len(key))
print(key)
```

The preceding Python code for the implementation of the BB84 QKD protocol can be summarized as follows. First, the modules necessary for the implementation of the BB84 QKD protocol are imported. The next step involves the implementation of the BB84 protocol. Finally, Alice and Bob compare their bit strings, and retain those strings where their use of bases corresponds and discard the rest. The retained bit strings are then used as a secret key generated using the BB84 protocol.

The B92 protocol

The B92 QKD protocol is the simpler version of the BB84 protocol. It was proposed by Charles Bennett in 1992.

Instead of using four quantum states, like the BB84 protocol, it only uses two non-orthogonal states. The B92 QKD protocol is based on the principle that two quantum states that are not orthogonal are sufficient to detect the presence of an eavesdropper.

In order to realize the B92 protocol, Alice randomly prepares a quantum state in either of the two non-orthogonal states and sends it over to Bob. On the receiving end, Bob then performs the measurement and records the time slots where he received inconclusive results and where he measured the results with certainty. He then communicates this information over the classical channel to Alice.

They then retain bit values for time slots where the measurements were done with certainty, and discard bit values for cases where the measurements were inconclusive.

The following steps show the implementation of the B92 protocol using Python:

1. The first step entails importing the necessary modules:

```
from qiskit import QuantumCircuit, execute, Aer
from qiskit.visualization import *
import matplotlib.pyplot as plt
import numpy as np
np.random.seed(42)
```

2. Then, the circuit for generating a random string of bits (128 bits) is generated by Alice:

```
circ = QuantumCircuit(1,1)
circ.x(0)
circ.barrier()
circ.h(0)
circ.barrier()
circ.measure(0,0)
circ.barrier()
circ.draw(output='text')
n = 128
backend = Aer.get_backend('qasm_simulator')
result = execute(circ, backend, shots=n,
                 memory = True).result()
bits_alice = [int(q) for q in result.get_memory()]
print(bits_alice)
```

3. The next step involves Bob choosing at random the bases he will use to perform measurements:

```
result = execute(circ, backend, shots=n,
                 memory = True).result()
basis_bob = [int(q) for q in result.get_memory()]
print(basis_bob)
bits_bob = []
for i in range(n):
    circ_send = QuantumCircuit(1,1)
    if bits_alice[i] == 0:
        circ_send.id(0)
    if bits_alice[i] == 1:
        circ_send.h(0)
    else:
```

```
        circ_send.id(0)
    circ_send.measure(0,0)
    result = execute(circ_send, backend, shots = 1,
                     memory = True).result()
    bits_bob.append(int(result.get_memory()[0]))
print(bits_bob)
```

4. Finally, both Alice and Bob communicate to generate and agree on the secret key generated:

```
key = []
for i in range(n):
    if bits_alice[i] == bits_bob[i]:
        key.append(bits_bob[i])
print("Key length is:", len(key))
print("The secret Key is:", key)
```

The preceding code snippet can be summarized as follows. First, as is always the case, the necessary modules are imported. The next step involves Alice's random generation of her bits to transfer to Bob. Based on the value of the bits generated, Alice can perform two tasks. If the generated bit value is '0', Alice does not change the basis (applies the identity gate). On the other hand, if the generated bit is '1', Alice changes the basis (applies the `Hadamard` gate).

On Bob's side, Bob randomly chooses the basis to use for measurement. Then, both Alice and Bob compare the bases they used. If the bases correspond, they keep the bit value corresponding to that. Otherwise, they discard the bit value.

The SARG04 QKD protocol

The SARG04 QKD protocol was invented by Scarani, Acin, Ribordy, and Gisin in 2004. This protocol is an improvement on the BB84 protocol discussed earlier. This protocol is intended to be robust even when Alice uses a coherent light source (instead of a single-photon light source) for the preparation of quantum states.

Just like the BB84 protocol, the SARG04 protocol is also the four-state quantum key distribution protocol. The key difference between the two protocols lies in the classical post-processing.

So far, we have covered three QKD protocols, namely, the BB84, the B92, and the SARG04. All these QKD protocols are examples of the prepare-and-measure category of QKD schemes. Next, we will discuss another protocol, namely, the E91 protocol. Unlike the previous three QKD protocols, the E91 protocol is an example of the entanglement-based category of the QKD schemes.

The E91 QKD protocol

As already stated, the E91 protocol uses quantum entanglement as a quantum resource in order to secure the data. Additionally, this protocol was developed by Ekert, building on the ideas proposed by David Deutsch (Deutsch was mentioned earlier in this book, in *Chapter 5*, *Quantum Algorithms*).

The E91 QKD protocol operates as follows. First, both Alice and Bob share EPR pairs. Then they perform measurements on the parts of the pairs that are on their side. Since entanglement states are correlated, any measurement of the EPR pairs by either Alice or Bob would result in a correlated state on the side of the other communicating party (Bob or Alice).

The following steps show the implementation of the E91 protocol using Python:

1. First, import the modules necessary for the implementation of the E91 protocol:

```
from qiskit import *
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(42)
```

2. Then, both Alice and Bob choose their bases to use:

```
A = [0, np.pi/8, np.pi/4]
B = [0, np.pi/8, -1*np.pi/8]
basesA = []
basesB = []
output = []
```

3. The next step involves defining the circuit to be used by Alice and Bob in order to generate a 100-bit random key:

```
for i in range(100):
    circ = QuantumCircuit(2, 2)
    circ.h(0)
    circ.cx(0,1)
    Ta = np.random.choice(A)
    Tb = np.random.choice(B)
    circ.rz(Ta, 0)
    circ.rz(Tb, 1)
    circ.measure([0, 1], [0, 1])
```

4. Then, simulate the circuit using the **`'qasm_simulator'`**:

```
    backend = Aer.get_backend('qasm_simulator')
    result = execute(circ, backend, shots=1,
                     memory=True).result()
```

Finally, generate and display the random key.

```
    value = result.get_memory()
    output.append(value)
print("The output is:", output)
```

The preceding Python code for the implementation of the E91 QKD protocol can be summarized as follows. The first step involves importing the necessary modules. This is then followed by the construction of the circuit for entanglement generation. Now that both Alice and Bob share an entangled state, the next step is for Alice to randomly and uniformly choose a measurement axis for each or her qubits. She does so by randomly and uniformly choosing to rotate her state by any of the angles 0, $\pi/8$, or $\pi/4$. She then performs a measurement on her qubit.

Bob, on the other hand, chooses his measurement axis by randomly and uniformly choosing to rotate his state by any of the angles 0, $\pi/8$, or $-\pi/8$. Just like Alice, Bob also performs a measurement on his qubit.

The final step in the implementation of the E91 QKD protocol from the preceding code involves simulating the circuit used for this implementation, using the `'qasm_simulator'`. The results obtained are then displayed as a list of pairs of Alice's and Bob's measurement results.

We have provided a brief discussion of the QKD protocol. In the next section, we move beyond quantum cryptography. We explore the field of study that aims to develop non-quantum cryptographic schemes that are resistant to quantum computers. This field of study is known as post-quantum cryptography.

Post-quantum cryptography

Post-quantum cryptography is concerned with the development of cryptographic algorithms that are believed to be resistant to code breaking by quantum computers. Currently, most modern cryptography algorithms are known to be vulnerable to attacks using quantum computers.

Post-quantum cryptography makes use of the some of the mathematical concepts in order to design cryptographic systems that appear to be difficult to break, even for a cryptanalyst with access to a powerful quantum computer. Like modern cryptography systems discussed earlier, post-quantum cryptography systems provide computational security instead of the information-theoretic security provided by quantum cryptography.

Post-quantum cryptography can be implemented using any of the following approaches/classes:

- Lattice-based cryptography, which can be used to develop digital signatures and key exchange cryptographic schemes
- Multivariate quadratic equations cryptography, which is typically used to develop digital signature cryptographic schemes
- Hash-based cryptography, which is typically used to develop digital signatures
- Code-based cryptography, which is typically used to develop key exchange cryptographic schemes
- Supersingular elliptic curve isogeny cryptography, which can be used to develop encryption schemes

Now that we have provided a brief introduction to post-quantum cryptography, the next subsections will cover some of the post-quantum cryptographic technics. In the next subsection, we will explore a lattice-based key exchange technique called the **NewHope key exchange scheme**.

The NewHope key exchange scheme

One of the best examples of post-quantum cryptographic tools is the NewHope key exchange scheme. This scheme uses the lattice-based approach. As a lattice-based approach, the NewHope cryptographic technique offers resistance to all known quantum algorithms. Furthermore, the NewHope key exchange technique is one of the fastest and, hence, most efficient lattice-based techniques.

The Python implementation of the NewHope key exchange scheme is given in the following code snippet:

```
from pynewhope import newhope
import numpy as np
np.random.seed(42)
ka, ma = newhope.keygen()
skb, mb = newhope.sharedB(ma)
ska = newhope.sharedA(mb, ka)
```

```
if ska == skb:
    print("\nSuccessful key exchange! Keys match.")
else:
    print("\nError! Keys do not match.")
print("The shared key is:", ska)
```

In the preceding code snippet, the modules necessary to implement the NewHope key exchange scheme are imported. This is followed by Alice generating a random key and the message she intends to send to Bob. Bob then receives the message from Alice and responds to her with his own message. The two communicating parties then use the information communicated to generate the secret key. Finally, they verify that indeed their keys match. Finally, the shared key is displayed.

Having explored the NewHope key exchange technique in this subsection, the next subsection will cover the hash-based, post-quantum cryptographic scheme known as **SPHINCS+**.

The SPHINCS+ digital signature scheme

The SPHINCS+ cryptographic technique is an example of the hash-based approach to post-quantum cryptography. Additionally, it offers long-term security even against attackers equipped with quantum computers.

SPHINCS+ can only be used for developing quantum-resistant digital signatures. That is, it cannot be used for developing key exchange schemes.

The SPHINCS+ digital signature scheme can be implemented using either of the three hash functions, namely, the SHA-256 hash function, the SHA-2 hash function, or the Haraka short-input hash function.

The Python code snippet for implementing the SPHINCS+ digital signature scheme is given as follows:

1. The first step involves importing the necessary modules:

```
import pyspx.shake256_128s as sphincs
import os, binascii
import numpy as np
np.random.seed(42)
```

2. This step is then followed by the generation of both the private and public keys:

```
# Key generation: private + public key
seed = os.urandom(sphincs.crypto_sign_SEEDBYTES)
public_key, secret_key = sphincs.generate_keypair(seed)
```

3. Next, the message is provided, and its corresponding digital signature generated and verified:

```
message = b'Hello World'
signature = sphincs.sign(message, secret_key)
valid = sphincs.verify(message, signature, public_key)
message = b'Hello World'
valid = sphincs.verify(message, signature, public_key)
print("Tampered message:", message)
```

```
print("Tampered signature valid?", valid)
message = b'Bye World'
valid = sphincs.verify(message, signature, public_key)
print("Tampered message:", message)
print("Tampered signature valid?", valid)
```

The preceding Python code can be summarized as follows. First, the necessary modules are imported. This is then followed by the provision and digital signature of the message. Next, verification is performed to confirm that the message was not tampered with.

We have now completed our coverage of post-quantum cryptography. The following section will summarize this chapter.

Summary

In this chapter, we have explored the field of quantum information processing known as quantum cryptography. Furthermore, we have discussed the use of Python for implementing cryptographic schemes, both for non-quantum and quantum cryptography. Additionally, we have covered the field of cryptography that is believed to be resistant to hacking by quantum computers. This field of cryptography is known as post-quantum cryptography.

Furthermore, in this chapter, you are expected to have learned more about some of the concepts in classical, modern, and quantum cryptography. Additionally, you are expected to have acquired hands-on skills in terms of how to implement cryptography using Python.

The skills acquired in this chapter can be applied in cybersecurity, where such skills can be used for the real-life design and engineering of cryptographic systems.

In the next chapter, we will explore another field of quantum information processing – quantum machine learning.

Further reading

- Schneier, Bruce. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley and sons, 2007.
- Mafu, Mhlambululi, and Senekane, Makhamisa. *Security of Quantum Key Distribution Protocols*. In Advanced Technologies of Quantum Key Distribution, pp. 3-15, IntechOpen, 2018.
- Bernstein, Daniel J. *Introduction to Post-Quantum Cryptography*. In Post-quantum cryptography, pp. 1-14. Springer, Berlin, Heidelberg, 2009.

Chapter 8: Quantum Machine Learning

Another field of QIP that is worth exploring is quantum machine learning. **Quantum machine learning** makes use of quantum mechanics to enable computers to learn from data. Recently, quantum machine learning has enjoyed both theoretical and practical implementation success. Examples of successful implementation of quantum machine learning include implementations of quantum machine learning in sectors such as finance, materials science, and drug discovery.

Essentially, quantum machine learning fuses together ideas from quantum physics and ideas from computer science. The key objective in this case is to use quantum mechanical concepts such as superposition and quantum entanglement in order to improve the performance of machine learning techniques.

In this chapter, we will begin with an introduction to conventional machine learning and then move to quantum machine learning, while also showcasing the applications of quantum machine learning.

In this chapter, we will cover the following main topics:

- Understanding conventional (classical) machine learning
- Understanding quantum machine learning
- Quantum machine learning algorithms

By the end of the chapter, you should be able to do the following:

- Understand key ideas and concepts in conventional machine learning.
- Understand key ideas and concepts in quantum machine learning.
- Summarize key concepts in quantum machine learning.

In the next section, we will discuss the technical requirements for you to follow along with this chapter.

Technical requirements

The requirements for this chapter are the following:

- Basic understanding of Python programming language
- Navigation of Google's Colab environment

The GitHub link for this chapter can be found here:

https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter08

The next section provides a basic introduction to conventional machine learning.

Understanding conventional (classical) machine learning

Machine learning is arguably the most successful branch of artificial intelligence. It has applications in various fields:

- Digital signal processing
- Image processing
- Drug discovery
- Financial analysis
- Cybersecurity
- Financial analysis
- Marketing

In essence, machine learning is a set of techniques that can be used by the computer to identify the patterns in data, build models that explain the world, and/or make predictions, without being explicitly programmed.

The overall goal of machine learning is to enable computers to learn on their own, without having explicitly pre-programmed rules and models. It enables computers to modify or adapt their actions (such as making predictions) in order to increase the accuracy of such actions.

The three main categories of machine learning

Machine learning can be classified into three main categories:

- Supervised learning
- Unsupervised learning
- Reinforcement learning

Let's have a quick overview of each of these machine learning categories.

Supervised learning

In a supervised learning algorithm, both the training data (input data) and the target data (labels, output data, or responses to training data) are provided. The task of such an algorithm then is to make predictions about future output points based on the future input points. Supervised learning algorithms can be divided into the following classes:

- **Classification learning**: The target data has a fixed number of classes.
- **Regression learning**: The target data has a continuous number of classes.

Examples of supervised learning algorithms include the following:

- Linear regression
- Logistic regression
- Artificial neural networks
- Support Vector Machines (SVMs)

Unsupervised learning

In unsupervised learning, only the training data is provided, while the target data is not provided. The objective, in this case, is not to use the training data to make the predictions. Rather, unsupervised learning learns and/or discovers the distribution of the input (training) data.

Examples of unsupervised learning algorithms include the following:

- Clustering
- Recommendation
- Dimensionality reduction
- Principal component analysis
- Anomaly detection

Reinforcement learning

Finally, in reinforcement learning algorithms, the computer is induced to learn from a series of reinforcements, namely punishments or rewards. If an algorithm succeeds in taking a correct action, it is rewarded. However, if an algorithm fails in taking a correct action, it is punished.

The notion of reward and punishment in reinforcement learning can be summarized as follows. A computer takes a series of steps, and these steps can either lead to the desired outcomes or not. If the steps lead to the desired outcome, then the computer is rewarded so that it knows which steps to take to arrive at the desired outcome. If the steps do not lead to the desired outcome, then the computer is punished so that it does not repeat those steps in the future.

Examples of reinforcement learning algorithms include the following:

- Q-learning
- Sarsa

As already stated, machine learning algorithms are used to enable computers to accurately make predictions. In order to measure the performance of such algorithms, various performance metrics can be used. These include the following:

- Accuracy
- Precision and recall
- Confusion matrix
- **Receiver operator characteristic** (**ROC**) curve

So far, we have provided a brief introduction to classical machine learning. In the next subsections, we are going to discuss two of the most prominent supervised learning algorithms, namely, artificial neural networks and SVMs.

Exploring artificial neural networks

Artificial neural networks (**ANNs**) are inspired by the operation of the biological neurons in the brain. A typical ANN structure is shown in the following schematic diagram:

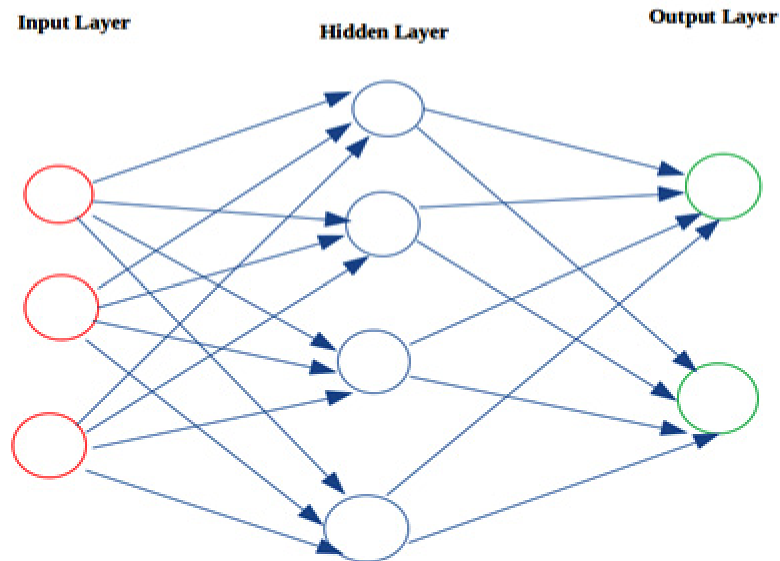


Figure 8.1 – A typical structure of an ANN

As can be seen from the diagram, a typical ANN consists of an input layer, one or more hidden layers, and an output layer. In such a setting, each layer consists of a number of nodes, where these nodes represent artificial neurons. These nodes are connected by the weights.

The input nodes encode the input data to the ANN algorithm. On the other hand, each of the other nodes (besides the input nodes) consists of two functions:

- Transfer function
- Activation function

For inputs x_i and weights w_i, the transfer function $f(x)$ is given as $f(x)=b_i+\sum x_i w_i$, where b_i is a bias value.

An activation function (φ) on the other hand is a non-linear, differentiable function such that for a transfer function $f(x)$, the output of the node, denoted y_i, is given as follows:

$$y_i = \phi(f(x))$$

Depending on the number of hidden layers in the ANN architecture, an ANN can be either a shallow neural network or a deep neural network. The former has few hidden layers, while the latter has several hidden layers.

ANNs have several architectures. These include the following:

- **Multi-layer perceptron** (**MLP**)
- **Recurrent neural networks** (**RNNs**)
- **Conventional neural networks** (**CNNs**)
- **Autoencoders** (**AEs**)
- **Generative adversarial networks** (**GANs**)

A Python code snippet for implementing an MLP ANN using **`sklearn`** (**`sklearn`** can be downloaded from https://scikit-learn.org) is shown next:

NOTE

The GitHub link for this chapter can be found here:

https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter08

1. The first step is to import the necessary modules for the implementation of an MLP ANN:

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score,
     classification_report, confusion_matrix
np.random.seed(42)
```

2. The data to be used for the implementation of the ANN algorithm is then loaded. This is followed by splitting the dataset into the training and the test data. 20% of the data is used to test the accuracy of the algorithm, while 80% of the data is used to train the model:

```
iris = load_digits()
X = iris.data
Y = iris.target
x_train, x_test, y_train, y_test = \
train_test_split(X,Y, test_size=0.2, random_state=42)
```

3. The next step is to actually implement the MLP ANN algorithm:

```
clf = MLPClassifier(alpha=1, max_iter=1000)
clf.fit(x_train, y_train)
y_pred = clf.predict(x_test)
```

4. Finally, the accuracy of the MLP ANN algorithm is explored, as shown in the following code snippet:

```
print('Accuracy is:', accuracy_score(y_test, y_pred))
print('\nClassification Report is:\n',
classification_report(y_test, y_pred))
print('\nConfusion Matrix is:\n',
confusion_matrix(y_test, y_pred))
```

The preceding code can be summarized as follows:

1. The first step is to import all the packages that will be used. This is followed by importing the dataset that will be used. In this case, the dataset to be used is the MNIST digits dataset, which is a popular dataset in machine learning.
2. After importing the dataset, then pre-processing is done, followed by classifying the digits with an MLP classifier.

3. Finally, different performance metrics are calculated. These metrics include accuracy and a confusion matrix.

Now that we have provided a basic introduction to ANNs, the next step is to cover another machine learning algorithm, namely **SVMs**.

Exploring SVMs

SVMs are suitable for complex but small or medium-sized datasets. They are used to linearly separate data points using the property of maximum margin separator. If the linear separation is not possible, then the data points are projected to higher dimensions where linear separability is possible. This projection of data points to higher dimensions is done using the kernel function.

SVMs can be used in the following tasks:

- Linear and non-linear classification
- Regression
- Anomaly detection

The Python code snippet for implementing an SVM using **`sklearn`** is as follows:

NOTE

The GitHub link for this chapter can be found here:

https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter08

1. The first step in this code snippet is to import the necessary modules:

```
import numpy as np
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score,
     classification_report, confusion_matrix
np.random.seed(42)
```

2. This is followed by loading the data that will be used for the SVM classification algorithm and splitting the data into training and test sets. In this case, 20% of the data will be used to test the algorithm, while 80% will be used to train the SVM algorithm:

```
iris = load_digits()
X = iris.data
Y = iris.target
x_train, x_test, y_train, y_test = \
train_test_split(X,Y, test_size=0.2, random_state=42)
```

3. After splitting the data, the next step is to actually implement the SVM classifier:

```
clf = SVC(kernel='rbf', gamma = 0.0001, C=1e1)
```

```
clf.fit(x_train, y_train)
y_pred = clf.predict(x_test)
```

4. Finally, the last step is to explore the accuracy of the algorithm:

```
print('Accuracy is:', accuracy_score(y_test, y_pred))
print('\nClassification Report is:\n',
classification_report(y_test, y_pred))
print('\nConfusion Matrix is:\n',
confusion_matrix(y_test, y_pred))
```

The preceding code snippet can be summarized as follows:

1. First, all the necessary modules are imported to the workspace, together with the MNIST dataset that is used.
2. The next step is to perform pre-processing.
3. Pre-processing is followed by classifying the MNIST digits with the SVM classifier.
4. Finally, accuracy and confusion matrix metrics are used to assess the performance of the SVM algorithm.

So far, we have covered conventional machine learning. In the next section, we will focus our attention on quantum machine learning.

Understanding quantum machine learning

As stated earlier in this chapter, quantum machine learning fuses together techniques from conventional machine learning with concepts from quantum information theory. This fusion of conventional machine learning and quantum information theory aims to improve the performance of machine learning techniques.

Generally, depending on the kind of dataset (classical or quantum) being used and the computing platform/algorithm (classical or quantum) used, machine learning can be summarized as shown in the following figure:

		Type of Algorithm	
Type of Data		Classical	Quantum
	Classical	CC	CQ
	Quantum	QC	QQ

Figure 8.2 – A summary of machine learning approaches

In the preceding diagram, the CC approach denotes the use of a classical dataset on classical hardware, while the QC approach denotes the use of a quantum dataset on a classical computer. On the other hand, the CQ approach denotes the use of a classical dataset on a quantum computer, while the QQ approach denotes the use of a quantum dataset on a quantum computer.

Both the CQ and QQ approaches can be considered as quantum machine learning. In this chapter, our focus will be on the CQ approach – the use of classical datasets on quantum computers.

So far, a variety of quantum machine learning algorithms have been developed. These include the following:

- Quantum neural networks
- Quantum principal component analysis
- Quantum SVMs
- Variational quantum machine learning algorithms such as the **variational quantum eigensolver** (**VQE**) and **quantum approximate optimization algorithm** (**QAOA**)

So far, we have provided a brief introduction to quantum machine learning. We have also seen that in the CQ approach, a classical dataset can be used on a quantum computer. Next, we will discuss how classical data can be encoded so that it can be processed by a quantum computer.

Data encoding

There are various ways of encoding classical data so that it can be processed on a quantum computer. These data encoding techniques are discussed in brief in this section. They are basis encoding, amplitude encoding, and Hamiltonian encoding.

Basic encoding

In basic encoding, data is encoded in the basic (computational) states of a quantum computer. That is, each data point is represented as the superposition of the basic states. This form of encoding is limited to binary data.

Amplitude encoding

On the other hand, in amplitude encoding, data is encoded on the amplitudes of a quantum state. Unlike basis encoding, amplitude encoding is not limited to binary data. However, amplitude encoding is vulnerable to noise, as the information is encoded in the amplitude, which might vary with noise.

Hamiltonian encoding

Finally, in Hamiltonian encoding, data is encoded using the Hamiltonian of the system. That is, in this data encoding scheme, the Hamiltonian of the system is associated with a Hermitian matrix that represents the data.

We have briefly gone through some of the data encoding techniques. The next subsection will cover quantum SVMs.

Quantum SVMs

Quantum SVMs form a class of prominent quantum machine learning algorithms. In essence, quantum SVMs are the quantum analogs of the conventional SVMs introduced in the previous section. Quantum computers can speed up learning in SVMs.

The `python` code snippet presented next shows the implementation of the quantum SVM using `qiskit`:

NOTE

The GitHub link for this chapter can be found here:

https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter08

1. The first step involves importing the modules required for the implementation of the quantum SVM algorithm:

```
from qiskit import BasicAer
from qiskit.aqua import QuantumInstance, aqua_globals
from qiskit.aqua.algorithms import VQC, QSVM
from qiskit.aqua.components.multiclass_extensions \
import *
from qiskit.aqua.components.optimizers import COBYLA
from qiskit.aqua.components.feature_maps import
     RawFeatureVector
from qiskit.circuit.library import ZZFeatureMap,
     ZFeatureMap, PauliFeatureMap
from qiskit.ml.datasets import breast_cancer
from qiskit.circuit.library import TwoLocal
seed = 42
aqua_globals.random_seed = seed
```

2. This is followed by the loading of the dataset and the preparation of data (pre-processing) for the implementation of the quantum SVM:

```
feature_dim = 4
_, training_input, test_input, _ = breast_cancer(
training_size=12,
test_size=4,
n=feature_dim)
feature_map = ZZFeatureMap(feature_dimension=feature_dim,
reps=2, entanglement='linear')
#feature_map = \
RawFeatureVector(feature_dimension=feature_dim)
qsvm = QSVM(feature_map, training_input, test_input)
```

3. The next step is to implement the quantum support vector algorithm and simulate this algorithm using **'qasm_simulator'**:

```
backend = BasicAer.get_backend('qasm_simulator')
quantum_instance = QuantumInstance(backend, shots=1024,
seed_simulator=seed,
seed_transpiler=seed)
```

```
result = qsvm.run(quantum_instance)
print('Testing accuracy: {:0.2f}'\
.format(result['testing_accuracy']))
```

The preceding code snippet can be summarized as follows:

1. First, the required modules are imported.
2. Then a classical dataset is also imported.
3. After importing the classical dataset, the basic pre-processing measures are undertaken.
4. This is followed by the implementation of the **quantum support vector machine** (**QSVM**) algorithm, and the execution of the algorithm using `'qasm_simulator'`.
5. Finally, the performance of the QSVM algorithm is assessed using the accuracy metric.

The output of this quantum SVM is shown in the following figure:

```
Testing accuracy: 0.75
```

Figure 8.3 – The accuracy of the QSVM algorithm

So far, we have explored the QSVM algorithm. We have also seen how it can be implemented using `qiskit`. The next quantum machine learning algorithm to explore is the quantum variational classifier.

Quantum variational classifier

The other prominent class of quantum machine learning algorithms is the quantum variational algorithms. Essentially, quantum variational algorithms are classical-quantum hybrid learning algorithms.

The `python` code snippet for implementing a quantum variational classifier using `qiskit` is shown as follows:

NOTE

The GitHub link for this chapter can be found here:

https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter08

1. The first step in the implementation of the quantum variational classifier is to import the necessary modules:

```
from qiskit import BasicAer
from qiskit.aqua import QuantumInstance, aqua_globals
from qiskit.aqua.algorithms import VQC
from qiskit.aqua.components.optimizers import COBYLA
from qiskit.circuit.library import ZZFeatureMap,\
ZFeatureMap, PauliFeatureMap
from qiskit.aqua.components.feature_maps import \
RawFeatureVector
```

```
from qiskit.ml.datasets import breast_cancer
from qiskit.circuit.library import TwoLocal
seed = 42
aqua_globals.random_seed = seed
```

2. The next step is the preparation of data for the implementation of the quantum variational classifier algorithm:

```
feature_dim = 4 # dimension of each data point
_, training_input, test_input, _ = breast_cancer(
training_size=12,
test_size=4,
n=feature_dim)
feature_map = \
RawFeatureVector(feature_dimension=feature_dim)
vqc = VQC(COBYLA(maxiter=1000), feature_map,
TwoLocal(feature_map.num_qubits, ['ry', 'rz'],
'cz', reps=3),
training_input,
test_input)
```

3. The quantum variational classifier is then simulated using the `'qasm_simulator'`:

```
backend = BasicAer.get_backend('qasm_simulator')
quantum_instance = QuantumInstance(backend, shots=1024,
seed_simulator=seed,
seed_transpiler=seed)
result = vqc.run(quantum_instance)
print('Testing accuracy: {:0.2f}'\
.format(result['testing_accuracy']))
```

The previous code is similar to the QSVM code discussed earlier. However, there are two key differences. The first difference is that in the case of the quantum SVM, the `python` class that is being used is the `QSVM()` class, while in the case of the quantum variational classifier, the `python` class that is being used is the `VQC()` class.

Another key difference is that in the case of the quantum variational classifier, several optimization techniques are available. These optimization techniques are as follows:

- **The Constrained Optimization by Linear Approximation** (**COBYLA**) optimizer
- **The Sequential Least Squares Quadratic Programming** (**SLSQP**) optimizer
- **The Simultaneous Perturbation Stochastic Approximation** (**SPSA**) optimizer

The technique used in the previous code is the COBYLA optimizer. On the other hand, the QSVM technique only uses the SPSA optimizer.

The following figure shows the output obtained from implementing the quantum variational classifier:

```
Testing accuracy: 0.88
```

Figure 8.4 – The accuracy of the quantum variational classifier algorithm

We have now come to the end of this chapter. The next section summarizes what was covered in this chapter.

Summary

In this chapter, we have covered the basics of both classical machine learning and quantum machine learning. We have learned that quantum information theory can be fused with computer science in order to improve the performance of machine learning techniques. Furthermore, we have provided and discussed Python implementations of two quantum machine learning techniques. These quantum machine learning techniques are quantum SVMs and quantum variational classifiers.

In the next chapter, we will cover another aspect of QIP, but this time we will be using continuous-variable quantum systems instead of the qubit-based quantum systems discussed up to this point.

Further reading

- Schuld, M., & Petruccione, F. (2018). *Supervised learning with quantum computers* (Vol. 17). Springer.
- Senekane, Makhamisa, Motobatsi Maseli, and Molibeli Benedict Taele. *Noisy, Intermediate-Scale Quantum Computing and Industrial Revolution 4.0*. In The Disruptive Fourth Industrial Revolution, pp. 205-225. Springer, Cham, 2020.
- Bishop, Christopher M. *Pattern recognition and machine learning*. Springer, 2006.

Chapter 9: Continuous-Variable Quantum Information Processing

Quantum systems can have either a discrete-variable or continuous-variable spectrum.

In this chapter, we begin by introducing the continuous-variable **quantum information processing** (**QIP**) field. Furthermore, we will also cover the key concepts and ideas in continuous-variable QIP.

We will cover the following topics in this chapter:

- Introducing continuous-variable quantum information processing
- Understanding the theory of continuous-variable quantum systems
- Exploring continuous-variable quantum teleportation
- Continuous-variable quantum key distribution
- Understanding continuous-variable quantum machine learning

At the end of this chapter, you should be able to do the following:

- Differentiate between discrete-variable QIP and continuous-variable QIP.
- Understand key ideas and concepts in continuous-variable QIP.

In the next section, we will cover the necessary technical requirements that will make it possible for you to follow this chapter.

Technical requirements

The requirements for this chapter are the following:

- A basic understanding of the Python programming language
- Navigation of Google's Colab environment

The GitHub link for this chapter can be found here:

https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter09

The next section provides an introduction to continuous-variable QIP.

Introducing continuous-variable quantum information processing

So far, we have only focused on QIP using qubits. As we stated earlier in this book, a qubit can exist in a two-dimensional (finite) Hilbert space (refer to *Chapter 2*, *Quantum States, Operations, and Measurements*, for further details on qubits). This implies that QIP using qubits is an example of a discrete-variable QIP.

Another approach to QIP involves the use of quantum systems that exist in infinite-dimensional Hilbert space. That is, unlike the discrete-variable QIP – where information is in discrete quantum systems such as qubits, in continuous-variable QIP Hilbert space

(and hence quantum states with a continuous basis).

Let's recall that for quantum systems in two-dimensional Hilbert space, the unit of quantum information is the qubit. On the other hand, the unit of information for a continuous-variable quantum system is the *qumode*.

We have just briefly contrasted continuous-variable quantum computing with discrete-variable quantum computing. It is now important to further explore the former (continuous-variable quantum computing). This exploration of the theory of continuous-variable QIP is done next.

Understanding the theory of continuous-variable quantum systems

In essence, discrete-variable QIP can be thought of as *digital* QIP. On the other hand, continuous-variable QIP can be thought of as "analog" QIP. This owes to the fact that the former uses quantum states in finite-dimensional Hilbert space (hence *digital*) while the latter uses quantum states in infinite-dimensional Hilbert space (hence *analog*).

Its compact nature and versatility are two of the key benefits of using the continuous-variable QIP framework. This owes to the fact that in continuous-variable QIP, information is encoded in the systems with continuous degrees of freedom (such as the quantum states/amplitudes of the electromagnetic field), and the physical systems realize the QIP framework is readily available in nature.

Another key advantage of this framework is that it leverages the wave-like properties of nature. This in turn makes continuous-variable QIP suited to the photonic implementation of the QIP.

Just like in discrete-variable quantum systems such as qubits, where the computational basis states are given as |0> and |1>, in continuous-variable quantum systems, we can also construct the qumodes $|x>_x$ such that:

$$x \in R$$

In essence, the qumode states $|x>_x$ are the eigenstates of an operator $\hat{x}$. The operator $\hat{x}$ has a continuous spectrum. Examples of this operator include position, momentum, and quadrature operators (position and momentum quadratures) of an electromagnetic field.

In reality, the eigenstates of the position and momentum operators are not physical. Therefore, in practice, the position and momentum coherent quadratures of an electromagnetic field are used. Finally, the output of the result of a continuous-variable quantum operation is obtained by measuring the observables of the operator $\hat{x}$.

Now that we have provided a basic introduction to the theory of continuous-variable QIP, the next section will cover one of the fields of application of continuous-variable quantum information, namely continuous-variable quantum teleportation.

Exploring continuous-variable quantum teleportation

We shall recall that quantum teleportation is used to transfer/transport an arbitrary quantum state (either a qubit or qumode) from Alice to Bob, who are separated by an arbitrary distance. Earlier in this book (Chapter 3, *Entanglement and Quantum Teleportation*), we discussed quantum teleportation in the context of discrete-variable QIP.

In this section, we will discuss quantum teleportation using the continuous-variable QIP framework. Let's recall that quantum teleportation requires the use of entangled quantum states and the classical channel. The entanglement required is possible in the discrete-variable QIP framework, but not practical in continuous-variable QIP.

In the continuous-variable QIP setting, quantum teleportation is realized through the following procedure, depicted by the circuit:

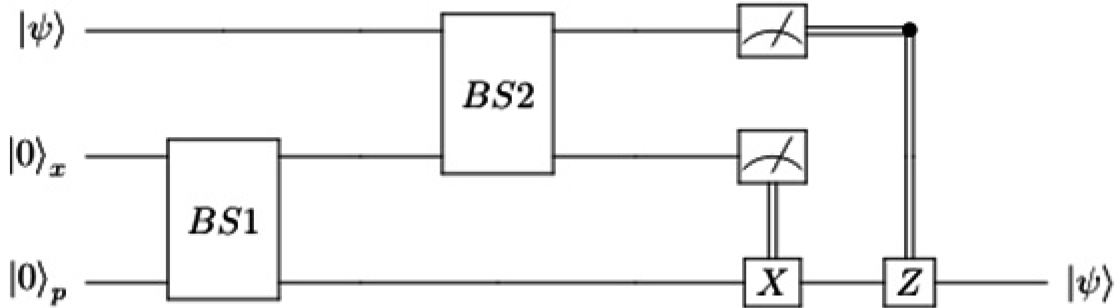


Figure 9.1 – A circuit diagram for the implementation of continuous-variable quantum teleportation

The preceding circuit can be summarized by the series of steps, and these steps are stated as follows:

1. First, the objective is to transmit the arbitrary qumode |ψ> using two qumodes $|\psi>_p$ and $|\psi>_x$ for Alice and Bob respectively.
2. After initializing Alice's and Bob's qumodes, these qumodes are then maximally entangled using a beam-splitter. This is labeled as *BS1* in *Figure 9.1*.
3. The next step after entangling the two qumodes to the maximum degree from Alice to Bob is to separate Alice and Bob by an arbitrary distance. However, just as in the case of discrete-variable quantum teleportation, Alice and Bob are still able to communicate using the classical channel. This step is then followed by the actual teleportation of an arbitrary qumode state from Alice to Bob. In order to do this, Alice entangles the part of an entangled pair to the maximum degree on her side with the arbitrary state to be teleported. This is also done through the use of the beam-splitter (labeled *BS2* in *Figure 9.1*). Once this is done, Alice then performs a measurement on her side.
4. Finally, Alice sends the results of her measurement over a classical channel to Bob. Bob then uses the information received from Alice to perform a series of operations on his entangled qumode in order to recover the teleported qumode state.

The Python code snippet for implementing continuous-variable quantum teleportation using Strawberry Fields is shown next:

NOTE

This code is taken from https://strawberryfields.ai/photonics/demos/run_teleportation.html.

The GitHub link for this chapter can be found here:

https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter09

1. The first step in this code is to import the modules that are necessary for the implementation of the continuous-variable quantum teleportation:

```
import strawberryfields as sf
from strawberryfields.ops import *
import numpy as np
```

```
from numpy import pi, sqrt
from matplotlib import pyplot as plt
np.random.seed(42)
```

2. The next step involves the initialization phase of the circuit used for the implementation of continuous-variable quantum teleportation. In this case, the circuit uses three quantum registers:

```
prog = sf.Program(3)
alpha = 1+0.5j
r = np.abs(alpha)
phi = np.angle(alpha)
with prog.context as q:
Coherent(r, phi) | q[0]
Squeezed(-2) | q[1]
Squeezed(2) | q[2]
```

3. After the initialization stage, the next step is to construct the quantum circuit to be used for the implementation of quantum teleportation, as summarized in *Figure 9.1*:

```
BS = BSgate(pi/4, pi)
BS | (q[1], q[2])
BS | (q[0], q[1])
MeasureX | q[0]
MeasureP | q[1]
Xgate(sqrt(2) * q[0].par) | q[2]
Zgate(sqrt(2) * q[1].par) | q[2]
```

4. Finally, continuous-variable quantum teleportation is simulated using Xanadu's **Strawberry Fields** framework. The results obtained are then displayed:

```
engine = sf.Engine('fock',
backend_options={"cutoff_dim": 15})
result = engine.run(prog, shots=1, modes=None,
compile_options={})
print(result.samples)
print(result.state)
state = result.state
print(state.dm().shape)
rho2 = np.einsum('kkllij->ij', state.dm())
print(rho2.shape)
probs = np.real_if_close(np.diagonal(rho2))
print(probs)
plt.bar(range(7), probs[:7])
```

```
plt.xlabel('Fock state')
plt.ylabel('Marginal probability')
plt.title('Mode 2')
plt.show()
fock_probs = state.all_fock_probs()
fock_probs.shape
np.sum(fock_probs, axis=(0,1))
```

We have now discussed the implementation of continuous-variable quantum teleportation. Furthermore, we have explored how continuous-variable quantum teleportation can be implemented using the Strawberry Fields framework. The next section will cover continuous-variable quantum game theory.

Understanding continuous-variable quantum game theory

Earlier in this book (Chapter 6, *Non-Local Quantum Games*), we covered the quantization of conventional games into discrete-variable quantum games. Now it is time to introduce yet another way of implementing quantum games. However, this time around, our focus will be on the implementation of quantum games using the continuous-variable approach. Therefore, in this section, we will explore the quantization of conventional games, but this time around, quantizing them into continuous-variable quantum games.

The key difference between the discrete-variable approach to quantum games and the continuous-variable approach to quantum games lies in the strategy used in each approach (access to strategic space for the players). In the former approach, the number of strategies that the players can use is finite. On the other hand, in the latter approach, there is a continuum of strategies that the players can choose from. Therefore, since the players in the continuous-variable quantum games have access to a continuum of strategies, such games can use infinite-dimensional quantum systems (qumodes) instead of finite-dimensional systems such as qubits.

The continuous-variable quantum games approach provides a more realistic quantization of the conventional games than its discrete-variable counterpart. This is due to the fact that in real-life settings, several cases are represented by the games in which players have access to a continuum of strategies.

In this section, we have provided a brief introduction to continuous-variable quantum game theory. Having introduced the theory on continuous-variable quantum games here, you should now be in a position to fully appreciate the difference between discrete-variable quantum game theory (covered in Chapter 6, *Non-local Quantum Games*) and the continuous-variable quantum game theory covered in this chapter. Furthermore, you should now be in a position to appreciate the application of continuous-variable QIP to the field of quantum game theory.

In the next section, we will cover the use of infinite-dimensional quantum systems for application in quantum cryptography. Specifically, we will cover continuous-variable quantum key distribution.

Continuous-variable quantum key distribution

We have already learned about **Quantum Key Distribution** (**QKD**) using qubits, in *Chapter 7*, *Quantum Cryptography*. Now it is time to explore the use of qumodes for the implementation of QKD. Therefore, in this section, we will turn our attention to continuous-variable QKD.

Continuous-variable QKD relies on continuous-variable quantum systems (coherent quantum states) and either homodyne or heterodyne detection/measurement. Succinctly, both homodyne and heterodyne detection use the input signal and the frequency mixer (this frequency mixer is typically called a *local oscillator*) in order to detect/measure a quantum signal. Homodyne detection uses the same frequency for both the input signal and the local oscillator. On the other hand, heterodyne detection uses different frequencies for the input signal and the local oscillator.

Continuous-variable QKD offers some advantages over discrete-variable QKD. These benefits are highlighted here:

- The first advantage of continuous-variable QKD over discrete-variable QKD is that the former uses a relatively simpler hardware setup compared to the latter.
- Additionally, continuous-variable **QKD** has the capability to generate higher secret key rates than discrete-variable QKD.
- Another advantage of continuous-variable QKD over discrete-variable QKD concerns the choice of photon sources that can be used. Unlike in the latter, where the use of single-photon sources is necessary, this is not the case with continuous-variable QKD.

The operation of the continuous-variable QKD exchange between Alice and Bob can be summarized as follows:

1. Alice first randomly selects the position and momentum variables. She then uses these random variables to prepare coherent states.
2. Alice then sends the prepared coherent states over to Bob over a noisy channel.
3. On the other side, Bob performs either homodyne or heterodyne measurement, choosing position and momentum bases at random.
4. Finally, Alice and Bob communicate over a classical channel, to perform classical post-processing.

In this section, we have provided background information on continuous-variable QKD. In the next section, we will explore another field of continuous-variable QIP, namely continuous-variable quantum machine learning.

Understanding continuous-variable quantum machine learning

In this section, we discuss the continuous-variable version of quantum machine learning. As we have already learned, this approach to machine learning uses continuous variable quantum systems instead of discrete variable quantum systems such as a qubit.

One of the software toolkits that can be used to implement continuous-variable quantum machine learning is the **PennyLane** framework. This framework supports both discrete-variable quantum machine learning and continuous-variable quantum machine learning. Furthermore, the PennyLane software toolkit, which is based on the Python programming language, also supports classical-quantum hybrid algorithms such as variational algorithms.

The following code snippet shows the use of Python and the PennyLane framework in the design of the quantum variational algorithm that is used to find the ground state of the hydrogen molecule:

NOTE

The following code is taken from www.pennylane.ai.

The GitHub link for this chapter can be found here:

https://github.com/PacktPublishing/Hands-On-Quantum-Information-Processing-with-Python/tree/master/Chapter09

1. The first step is to import the necessary modules:

```
#hydrogen variational quantum eigensolver
import pennylane as qml
from pennylane import numpy as np
```

2. The next step involves the description/specification of the hydrogen molecule to be used in this algorithm:

```
geometry = 'h2.xyz'
name ='h2'
charge = 0
multiplicity=1
basis= 'sto-3g'
h, nr_qubits = qml.qchem.generate_hamiltonian(
name,
geometry,
charge,
multiplicity,
basis,
mapping='jordan_wigner',
n_active_orbitals=2,
n_active_electrons=2,
)
print("Hamiltonian is: \n", h)
print("Number of qubits is: \n", nr_qubits)
```

3. After specifying the hydrogen molecule, the next step is to formulate an ansatz. An ansatz is a rough guess of the state of the hydrogen molecule. It is used as a starting point of the quantum variational algorithm, and then the hydrogen system is optimized from there until the ground state (the lowest-energy state) is found:

```
dev = qml.device('default.qubit', wires=nr_qubits)
def ansatz(params, wires):
qml.BasisState(np.array([1, 1, 0, 0]), wires=wires)
for i in wires:
qml.Rot(*params[i], wires=i)
qml.CNOT(wires=[2, 3])
```

```
qml.CNOT(wires=[2, 0])
qml.CNOT(wires=[3, 1])
cost_fn = qml.VQECost(ansatz, h, dev)
```

4. This is then followed by the implementation of the quantum variational algorithm:

```
opt = qml.GradientDescentOptimizer(stepsize=0.4)
np.random.seed(42)
params = np.random.normal(0, np.pi, (nr_qubits, 3))
print(params)
max_iterations = 250
step_size = 0.05
conv_tol = 1e-06
prev_energy = cost_fn(params)
for n in range(max_iterations):
params = opt.step(cost_fn, params)
energy = cost_fn(params)
conv = np.abs(energy - prev_energy)
if n % 20 == 0:
print('Iteration = {:},
Ground-state energy = {:.8f} Ha,
Convergence parameter = {'':.8f} Ha'\
.format(n, energy, conv))
if conv <= conv_tol:
break
prev_energy = energy
```

5. Finally, the results obtained are displayed:

```
print()
print('Final convergence parameter = {:.8f} Ha'\
.format(conv))
print('Final value of the ground-state energy = \
{:.8f} Ha'.format(energy))
print('Accuracy with respect to the FCI energy: \
{:.8f} Ha ({:.8f} kcal/mol)'.format(
np.abs(energy - (-1.136189454088)), \
np.abs(energy - (-1.136189454088))*627.503))
print()
print('Final circuit parameters = \n', params)
```

The implementation of this VQE algorithm starts with the initialization of an ansatz, which is then trained, with the objective of finding the ground state of the hydrogen molecule. Finally, the VQE circuit is optimized, with the optimization being run for 250 steps. The results obtained are then given.

The results obtained from executing this VQE algorithm are shown in the following figure:

```
Final convergence parameter = 0.00000564 Ha
Final value of the ground-state energy = -1.13600250 Ha
Accuracy with respect to the FCI energy: 0.00018695 Ha (0.11731347 kcal/mol)

Final circuit parameters =
 [[ 1.56047353e+00 -1.04676705e-14  2.03477355e+00]
 [ 4.78473941e+00 -3.14789937e-14 -7.35562944e-01]
 [ 4.96124338e+00 -2.08614783e-01 -1.00326059e-01]
 [ 1.70450265e+00 -9.33663429e-17 -1.46313317e+00]]
```

Figure 9.2 – Results obtained from implementing the VQE algorithm

In this section, we have explored continuous-variable quantum machine learning. Furthermore, we have discussed the Python implementation of the continuous-variable quantum eigensolver using Xanadu's PennyLane. The next section summarizes this chapter.

Summary

In this chapter, we covered continuous-variable QIP. We first drew on the difference between continuous-variable QIP and the discrete-variable QIP covered in the early chapters of this book. Furthermore, we discussed various fields of continuous-variable QIP, such as continuous-variable quantum teleportation, continuous-variable quantum game theory, continuous-variable QKD, and continuous-variable quantum machine learning.

In the next chapter, we will conclude this book. We will cover the current trends in QIP and try to explore the future prospects of QIP.

Further reading

- Schuld, Maria and Petruccione, Francesco. *Supervised learning with quantum computers*. Springer, 2018.
- Killoran, Nathan, Josh Izaac, Nicolás Quesada, Ville Bergholm, Matthew Amy, and Christian Weedbrook. *Strawberry Fields: A software platform for photonic quantum computing*. Quantum 3 (2019): 129.
- Bergholm, Ville, Josh Izaac, Maria Schuld, Christian Gogolin, Carsten Blank, Keri McKiernan, and Nathan Killoran. *Pennylane: Automatic differentiation of hybrid quantum-classical computations*. arXiv preprint arXiv:1811.04968 (2018).

Chapter 10: Current Trends in Quantum Information Processing

In this chapter, which concludes this book, we will focus on the current trends in **quantum information processing** (**QIP**). This will cover trends in various fields of QIP. Furthermore, we will explore the future prospects of various fields of QIP.

At the end of this chapter, you will have a rough picture of the direction that QIP is taking. Furthermore, you should be in a position to fully appreciate the current trends in various fields of QIP.

In this chapter, we will cover the following topics:

- Exploring current trends in quantum cryptography
- Exploring current trends in quantum communication
- Understanding current trends in quantum algorithm design
- Exploring current trends in quantum machine learning
- Understanding current trends in quantum computing hardware technologies
- Future prospects of QIP

Exploring current trends in quantum cryptography

The key progress in quantum cryptography is mainly in **quantum key distribution** (**QKD**). This progress is both on the practical implementation side and in the theoretical security analysis of QKD. In this section, we will highlight some of these trends.

One of the serious advances of QKD is in its application to real-world problems. In 2007, QKD was used to secure elections in Geneva, Switzerland. This QKD implementation project was led by the University of Geneva. Three years later, QKD was used to secure communications in Durban, South Africa, during the soccer World Cup. This project was spearheaded by the University of KwaZulu-Natal.

These two implementations of QKD (the 2007 Geneva implementation of QKD and the 2010 Durban implementation of QKD) propelled the cybersecurity community to seriously consider the significance and viability of quantum cryptography. However, the research in QKD was mainly limited to university campuses.

With the successful demonstration of QKD networks across various university campuses, the need arose for spin-off companies to commercialize QKD. One of the leading QKD spin-off companies is ID Quantique from Switzerland, which is a spin-off company from the University of Geneva.

Another current key area of interest in quantum cryptography is the use of QKD for communication between parties separated by long distances. One project of note in this respect is one pursued by China. The objective of this project is to enable QKD communication between the ground and satellites in space. Ultimately, the project would make it possible to have secure inter-continental communication that implements QKD.

QKD can also be implemented using higher-dimensional discrete quantum systems. These systems are generally called **qudits**. It should be easy to observe that a qubit is an example of a qudit in two-dimensional Hilbert space.

In essence, a qudit is a generalization of a qubit. Recall that in *Chapter 1*, *Getting Started with Quantum Information Processing*, we stated that a qubit is a unit of quantum information that exists in two-dimensional Hilbert space. On the other hand, a qudit exists in any finite-dimensional Hilbert space. This means that a qubit is a special case of a qudit, with a dimension of two (since it exists in two-dimensional Hilbert space).

Having stated that QKD can also be implemented using qudit systems instead of just the normal qubit systems, let's now briefly explore why and how this can be done. The use of higher-dimensional quantum systems (as opposed to qubits) in order to implement QKD has a few benefits. These benefits include improved information capacity per quantum system, a high key generation rate, improved security, and improved robustness against noise. Typically, a photon's orbital angular momentum is used for the implementation of higher-dimensional QKD.

The quantum cryptography trends covered thus far are concerned with the implementation aspect of QKD. However, as already stated, the trends in QKD are not only limited to the implementation part of QKD but also involve the theoretical advances in QKD. Therefore, it is important to also discuss the current trends in the theoretical aspects of QKD.

The security aspect of QKD is concerned with the analysis of the security of the QKD protocols under various assumptions. The current approach is to make assumptions about the devices to be used in QKD to be as realistic as possible, and then assess the security of the QKD protocols based on such assumptions.

In this section, we have briefly explored the current trends in quantum cryptography. The discussion was limited to QKD since it is the most successful and advanced primitive of quantum cryptography. In the next section, we will explore the trends in quantum communication.

Exploring current trends in quantum communication

Another aspect of QIP that is attracting a lot of attention is quantum communication. In this section, our focus will be on quantum networking, which is also referred to as the quantum internet.

The key objective of the quantum internet is to enable the transfer of the quantum state from the source to the destination, through the use of various intermediary nodes, which are entangled. These intermediary nodes are called **quantum repeaters**. Quantum repeaters are central to the proper operation of the quantum internet.

At a very basic level, the quantum internet can be thought of as a network that connects various potentially heterogeneous quantum networks into one quantum network, to enable communication among various quantum networks. It is worth noting that currently, there is no practical implementation of any quantum internet. Thus, the quantum internet is currently approached from a theoretical perspective, not from an implementation perspective.

Now that we have provided a brief exposition of the quantum internet in this section, the next section will explore the current trends in quantum algorithm design.

Understanding current trends in quantum algorithm design

One of the key objectives of exploring quantum algorithms is to design algorithms that make use of quantum mechanics to outperform their non-quantum counterparts. Currently, quantum algorithm design is an active area of research. In this section, we will explore the current trends in quantum algorithm design.

Quantum computing algorithms can be implemented in the following fields:

- **Cryptography**: For developing cryptographic schemes that provide better security than their conventional counterparts
- **Search and optimization**: For developing quantum optimization systems that are more efficient than their conventional counterparts
- **Simulating quantum systems**: For better understanding the behavior of quantum systems
- **Solving large systems of linear equations**: For developing mathematical algorithms for systems that outweigh their conventional counterparts in terms of performance

In the next section, we will discuss the current trends in quantum machine learning.

Exploring current trends in quantum machine learning

Like other fields of QIP, quantum machine learning is also gaining traction, both in the theoretical and the practical aspects of it. In this section, we will first explore the theoretical aspects of quantum machine learning and then move on to its practical aspects.

The theoretical aspect of quantum machine learning concerns the design of quantum machine learning algorithms such as quantum support vector machines and quantum neural networks. Furthermore, this aspect involves the assessment of the designed quantum machine learning algorithms in order to ascertain whether they offer a quantum advantage over their classical counterparts. Currently, a variety of quantum machine learning algorithms have been proposed in the literature.

On the other hand, the practical aspect of quantum machine learning is concerned with the application of quantum machine learning problems to solve real-world problems. Currently, quantum machine learning has been used to solve, among others, some of the following problems:

- Quantum chemistry and materials science to better understand the behavior of atoms and molecules
- Drug discovery in medicine in order to come up with new drugs for the treatment of diseases
- Optimization for more efficient search and optimization algorithms
- Portfolio optimization in finance

In this section, we briefly discussed the current trends in quantum machine learning. In the next section, we will shift our focus toward the current trends in quantum computing hardware technologies.

Understanding current trends in quantum computing hardware technologies

In this section, we offer a brief overview of the quantum systems that can be used as basic units of information in quantum computing. We should recall that quantum computing can be realized using either discrete-variable quantum systems (such as qubits) or continuous-variable quantum systems (such as qumodes).

For qubit-based, discrete-variable quantum computing, among others, the following quantum systems can be used to realize the qubit:

- Nuclear magnetic resonance

- Trapped atoms or ions
- Superconducting circuits
- Photon polarization
- Non-Abelian anyons

On the other hand, the quantum computing hardware technology used to implement continuous-variable quantum computing is different from the ones discussed here. Currently, continuous-variable quantum computing uses photons as the physical quantum systems for implementing qumodes (refer to the previous chapter for further details on continuous-variable quantum computing).

In this section, we have briefly discussed the current trends in quantum computing hardware technologies. In the next section, we will cover the future prospects of QIP.

Exploring the future prospects of QIP

Various fields of QIP have gained traction lately. The future of these fields offers several opportunities. For instance, with quantum cryptography, there is a possibility that the technology can have applications in areas where security is an absolute necessity. These areas include lotteries and banking.

Furthermore, in the future, more quantum cryptography primitives beyond QKD are likely to be the subject of investigation by QIP researchers. Additionally, more emphasis will be put on the implementation of long-distance quantum cryptography networks. This will likely be followed by attempts to physically implement the quantum internet, which will be used to connect various potentially heterogeneous quantum networks.

Concerning quantum computing, we should expect more strides into the design of more powerful quantum computing algorithms. We should also expect more progress to be made with the design of fault-tolerant quantum computing that will replace the currently used **noisy intermediate-scale quantum** (**NISQ**) computers. Furthermore, we should see the further mainstreaming of quantum computing as a key driving technology of the **fourth industrial revolution** (**4IR**).

Quantum machine learning is likely to receive the attention of researchers in QIP. We are likely to see more quantum machine learning algorithms to address various real-world problems. Since the datasets that would be used in quantum machine learning might be sensitive, I anticipate that there will be more emphasis on the development of privacy-preserving quantum machine learning algorithms.

Finally, there is a possibility that advances in QIP might lead to quantum-inspired algorithms that can be implemented on conventional computers.

Summary

Congratulations on making it to the end of this chapter and this book! I hope this book was worth your while. Now that we are nearing the end of this book, let me provide a brief summary of this chapter, and what is expected of you now that you have gone through the various chapters of this book.

In this chapter, we explored and discussed the different research paths pursued by various researchers and the various technological advancements related to the field of QIP.

Additionally, in this chapter, we discussed the future prospects of QIP. We discussed the likely prospects of quantum cryptography, quantum computing, and quantum machine learning.

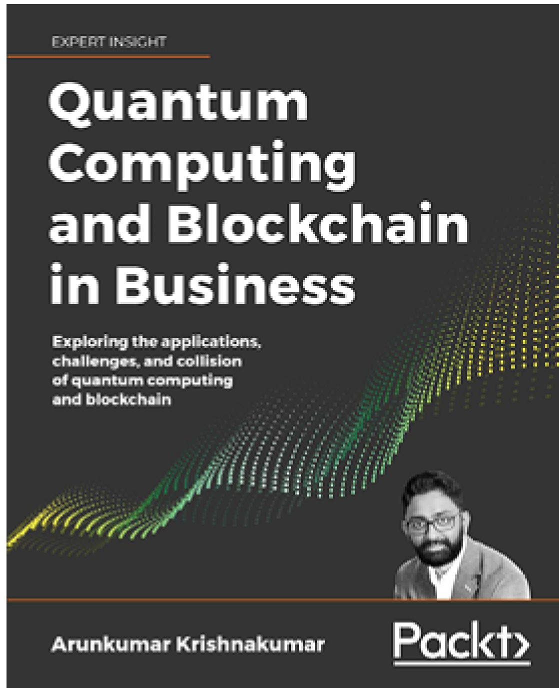
This chapter concludes this book. It is my hope that you enjoyed reading this book! I strongly believe that you will use the knowledge provided in this book to make contributions to the various fields of QIP covered in this book.

Further reading

- Van Meter, Rodney. *Quantum networking*. John Wiley & Sons, 2014.
- Senekane, Makhamisa, Mhlambululi Mafu, and Benedict Molibeli Taele. *Privacy-preserving quantum machine learning using differential privacy*. In 2017 IEEE AFRICON, pp. 1432-1435. IEEE, 2017.
- Sutor, R.S. (2019). *Dancing with Qubits: How Quantum Computing Works and How It Can Change the World*. Birmingham, UK: Packt Publishing.
- Moran, C.C. (2019). *Mastering Quantum computing with IBM QX*. Birmingham, UK: Packt Publishing.

Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

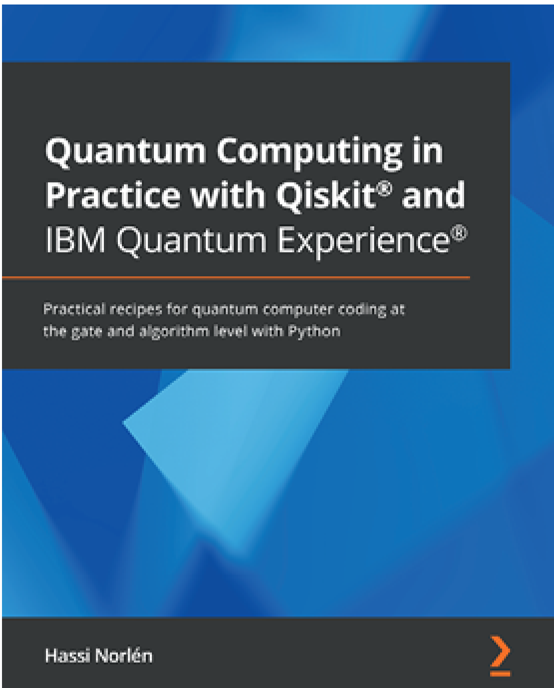


Quantum Computing and Blockchain in Business

Arunkumar Krishnakumar

ISBN: 978-1-83864-776-6

- Understand the fundamentals of quantum computing and Blockchain
- Gain insights from the experts who are using quantum computing and Blockchain
- Discover the implications of these technologies for governance and healthcare
- Learn how Blockchain and quantum computing may influence logistics and finance
- Understand how these technologies are impacting research in areas such as chemistry
- Find out how these technologies may help the environment and influence smart city development
- Understand the implications for cybersecurity as these technologies evolve



Quantum Computing in Practice with Qiskit® and IBM Quantum Experience®

Hassi Norlen

ISBN: 978-1-83882-844-8

- Visualize a qubit in Python and understand the concept of superposition
- Install a local Qiskit® simulator and connect to actual quantum hardware
- Compose quantum programs at the level of circuits using Qiskit® Terra
- Compare and contrast Noisy Intermediate-Scale Quantum computing (NISQ) and Universal
- Fault-Tolerant quantum computing using simulators and IBM Quantum® hardware
- Mitigate noise in quantum circuits and systems using Qiskit® Ignis
- Understand the difference between classical and quantum algorithms by implementing Grover's algorithm in Qiskit®

Leave a review - let other readers know what you think

Please share your thoughts on this book with others by leaving a review on the site that you bought it from. If you purchased the book from Amazon, please leave us an honest review on this book's Amazon page. This is vital so that other potential readers can see and use your unbiased opinion to make purchasing decisions, we can understand what our customers think about our products, and our authors can see your feedback on the title that they have worked with Packt to create. It will only take a few minutes of your time, but is valuable to other potential customers, our authors, and Packt. Thank you!