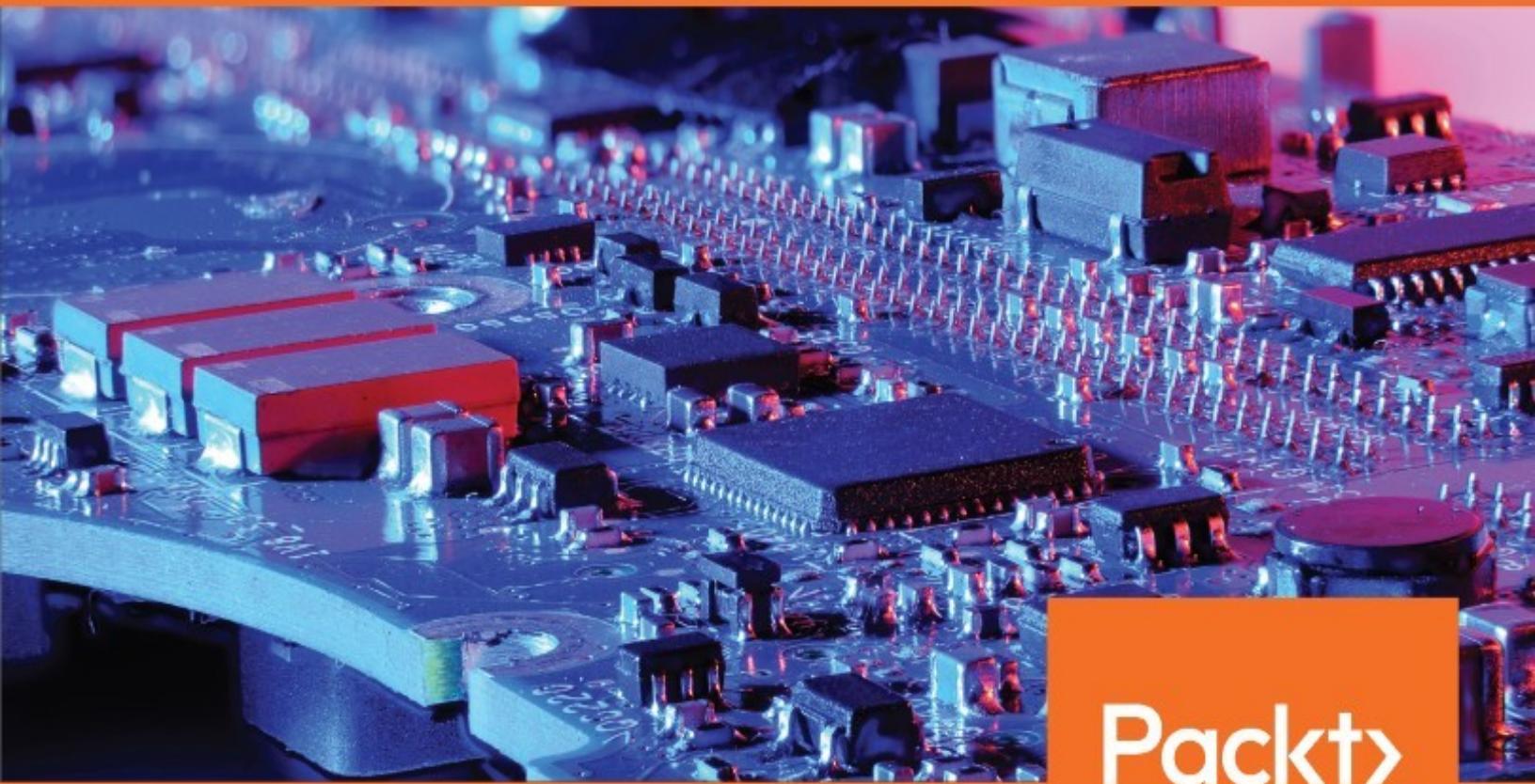


# Mastering Quantum Computing with IBM QX

Explore the world of quantum computing using the Quantum Composer and Qiskit



Dr. Christine Corbett Moran

Packt

[www.packt.com](http://www.packt.com)

# **Mastering Quantum Computing with IBM QX**

Explore the world of quantum computing using the Quantum Composer and Qiskit

Dr. Christine Corbett Moran



**BIRMINGHAM - MUMBAI**



# Mastering Quantum Computing with IBM QX

Copyright © 2019 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(s), 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.

**Commissioning Editor:** Richa Tripathi

**Acquisition Editor:** Sandeep Mishra

**Content Development Editor:** Zeeyan Pinheiro

**Technical Editor:** Romy Dias

**Copy Editor:** Safis Editing

**Project Coordinator:** Vaidehi Sawant

**Proofreader:** Safis Editing

**Indexer:** Pratik Shirodkar

**Graphics:** Alishon Mendonsa

**Production Coordinator:** Jisha Chirayil

First published: January 2019

Production reference: 1310119

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham

B3 2PB, UK.

ISBN 978-1-78913-643-2

[www.packtpub.com](http://www.packtpub.com)



[mapt.io](https://mapt.io)

Mapt is an online digital library that gives you full access to over 5,000 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
- Mapt is fully searchable
- Copy and paste, print, and bookmark content

# Packt.com

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 [www.packt.com](http://www.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](mailto:customercare@packtpub.com) for more details.

At [www.packt.com](http://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.



# About the author

**Dr. Christine Corbett Moran** is a researcher and engineer at NASA JPL, specializing in cybersecurity. She also serves as a guest scientist at Caltech, specializing in astrophysics. Her research spans fundamental physics and computer science, and she has published peer-reviewed papers on astrophysics, astronomy, artificial intelligence, and quantum computing, garnering thousands of citations. Her software products range from iOS applications to quantum computing simulators and have received millions of downloads. She has a PhD and master's in Astrophysics from the University of Zurich, and a B.S. in Computer Science and Engineering, and a B.S. in Physics from MIT. She can be found on Twitter at [@corbett](https://twitter.com/corbett).

*I'd like to thank my husband, Dr. Casey Handmer, for his support, and the editors at Packt for their encouragement. I'd also like to thank the South Pole Telescope collaboration for having me as a winterover scientist in Antarctica for 10.5 months in 2016; it was during the long winter months at the South Pole that I first found the time to dig into IBM QX.*

# About the reviewer

**Edward L. Platt** is a researcher at the University of Michigan School of Information and the Center for the Study of Complex Systems. He has published research on large-scale collective action, social networks, and online communities. He has also worked as a staff researcher and software developer for the MIT Center for Civic Media. He contributes to many free/open source software projects, including tools for media analysis, network science, and cooperative organizations. He has also done research on quantum computing and fault tolerance. He has an M.Math in applied mathematics from the University of Waterloo, as well as B.S. degrees in both computer science and physics from MIT. He is the author of *Network Science in Python with NetworkX*.

*I would like to thank Seth Lloyd and Paul Penfield Jr. for introducing me to information and entropy. Thanks to Edward Farhi, Peter Shor, and Frank Wilhelm-Mauch for their mentorship in quantum computing. Thanks to my PhD mentor, Daniel M. Romero. Thanks to Andy Brosius and Persephone Hernandez-Vogt for bringing me joy, chocolate, and support. And thanks to Dr. Christine Corbett Moran for the opportunity to review this exciting book.*

# Packt is searching for authors like you

If you're interested in becoming an author for Packt, please visit [authors.packtpub.com](https://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

Title Page

Copyright and Credits

    Mastering Quantum Computing with IBM QX

About Packt

    Why subscribe?

    Packt.com

Contributors

    About the author

    About the reviewer

    Packt is searching for authors like you

Preface

    Who this book is for

    What this book covers

    To get the most out of this book

        Download the example code files

        Download the color images

        Conventions used

    Get in touch

        Reviews

## 1. What is Quantum Computing?

    Technical requirements

    What are quantum computers?

        Uses of quantum computers

        Experts weigh in &#x2013; why is quantum computing important?

    History, present, and future of quantum computing

        History of quantum computing

        Present/current state of quantum computing&#xA0;

        Future of quantum computing

            Experts weigh in &#x2013; what is the outlook for quantum computing in the future?

    Setting up and running Python code examples

- Get the book code
- Set up Jupyter Notebook
- Setting up and running IBM QX examples
  - Hello quantum world
  - API key
- Setting up and running Qiskit examples
- Summary
- Exercises and questions

## 2. Qubits

- Technical requirements
- Qubits
  - Storing a qubit
  - Simulating a qubit
    - $|0\rangle$  and  $|1\rangle$
    - Combinations of  $|0\rangle$  and  $|1\rangle$
  - Three different representations of qubits
    - Zero and one basis
    - The plus and minus basis
    - The clockwise and counterclockwise basis
  - The Bloch sphere
    - $|0\rangle$  and  $|1\rangle$  and other basis on the Bloch sphere
    - Bloch coordinates from qubit
    - Plotting Bloch coordinates on the Bloch sphere
  - Superposition and measurement of qubits
    - Quantum superposition for qubits
    - Quantum measurement for qubits
    - Measurement of a single qubit on the Bloch sphere
  - Summary
  - Questions

## 3. Quantum States, Quantum Registers, and Measurement

- Technical requirements
- Quantum states and registers
- Separable states
- Entanglement
- Quantum measurement and entanglement

An algorithm for simulating quantum measurement in Python

Decoherence, T1, and T2

Decoherence

T1 and T2

Summary

Questions

## 4. Evolving Quantum States with Quantum Gates

Technical requirements

Gates

Classical gates

Quantum gates

Gates acting on states

Single-qubit gates

Hadamard gate (H)

Pauli gates (X, Y, Z)

The X&#xA0;gate

The Y&#xA0;gate

The Z gate

Phase gate (S) and&#xA0;/&#x3C0;/8&#xA0;gate (T)

Phase gate (S)

&#x3C0;/8 gate (T)

The "dagger" gates S&#x2020;&#xA0;and T&#x2020;

Multi-qubit gates

CNOT gate

Python code for the CNOT gate

CNOT with control qubit of choice, target qubit of choice

Summary

Questions

## 5. Quantum Circuits

Technical requirements

Quantum circuits and quantum circuit diagrams

Using Qiskit to generate quantum circuits

Single-qubit circuits in Qiskit

Qiskit's QuantumCircuit class and universal gate methods

Multiqubit gates in Qiskit

Classical registers in Qiskit circuit

Measurement in a Qiskit circuit

Reversible computation

Useful quantum circuits

Using the X gate to prepare any binary input

Swapping two qubits

Summary

Questions

## 6. The Quantum Composer

Technical requirements

The Quantum Composer

Hardware

Gates, operations, and barriers

Translating quantum circuits into the Quantum Composer

Executing quantum circuits in simulation or hardware from the Quantum Composer

Executing a quantum circuit in simulation

Executing a quantum circuit on quantum computing hardware

Summary

Questions

## 7. Working with OpenQASM

Technical requirements

OpenQASM

Translating OpenQASM programs into quantum scores

OpenQASM to negate one qubit

OpenQASM to apply gates to two qubits, and measure the first qubit

Representing quantum scores in OpenQASM 2.0 programs

Using OpenQASM to interface with IBM QX

Advanced OpenQASM usage

Resetting a qubit

if statements

User-defined gates and primitive gates

Primitive gates; CX; and U

2-qubit gate CX

1-qubit gate U

Opaque gates

Summary

Questions

## 8. Qiskit and Quantum Computer Simulation

Technical requirements

Qiskit installation and usage

Testing Qiskit installation

Using OpenQASM with Qiskit

Load OpenQASM from a file

Working with OpenQASM loaded from a string

Qiskit Aqua introduction and installation

Qiskit Terra &#x2013; Capstone project

A brief introduction to the MIDI specification

Quantum computing with MIDI

Synthesizing notes

Playing notes and chords to represent quantum measurement

Playing a note for each quantum measurement

Playing a chord (group of notes at the same time) to represent the results of many quantum measurements

Creating a superposition of notes

Two note chord

Four note chord

Summary

Questions

## 9. Quantum AND (Toffoli) Gates and Quantum OR Gates

Technical requirements

Boolean satisfiability problem (SAT)

3SAT classical implementation

3SAT &#x2013; why is it so interesting?

Quantum AND and OR

Toffoli gate &#x2013; quantum AND gate

Quantum OR gate

Quantum AND and Quantum OR over multiple qubits

3SAT quantum circuit implementation

Summary

Questions

## 10. Grover's Algorithm

Technical requirements

An overview and example use case of Grover's algorithm

Grover's algorithm steps

- Setup step
- Checker and mover steps
  - Naming conventions
  - Measurement step

3SAT as a Grover's algorithm checker

- Two- and three-qubit quantum AND (Toffoli) in Qiskit
  - Quantum AND reverse
- Two- and three-qubit quantum OR in Qiskit
  - Quantum OR reverse

Testing gates and their reversibility

- General framework

Solving a 3SAT problem with Grover's algorithm

- Oracle implementation in Qiskit

- Test classic logic
- Quantum 3sat\_mystery implementation logic
  - Setup or teardown logic function
  - Compute clauses one by one
  - Combine clauses
- Reverse logic so that we are back to the original

Testing the \_3sat\_mystery\_3 function

- Testing the \_3sat\_mystery\_3 function without reverse
- Testing the \_3sat\_mystery\_3 function's reversibility
- Testing \_3sat\_mystery\_3 function reversibility (except for final result)

Mover step implementation

- Full implementation of the mover function
- Full algorithm setup
- Running the algorithm on Qiskit

Summary

Questions

## 11. Quantum Fourier Transform

Classical Fourier Transform

Sine waves

The Fourier transform in action

The Quantum Fourier Transform&#xA0;

Implementing the Quantum Fourier Transform

Implementing a controlled rotation gate,  $R_k$ , in Python

Reverse gate &#x2013; REV

QFT circuit

QFT circuit implementation in IBM QX

Implementing the REV&#xA0;gate in IBM QX

Implementing the  $R_k$ &#xA0;gate in IBM QX

1-qubit QFT in IBM QX

2-qubit QFT in IBM QX

3-qubit QFT in IBM QX

Generalizations

Summary

Questions

## 12. Shor&#x27;s Algorithm

Shor's algorithm

Factoring large integers efficiently disrupts modern cryptography

Shor's algorithm overview

Shor's algorithm described

Shor's algorithm described in symbols/mathematics

Shor's algorithm examples

Example when  $N$  is prime,  $N = 7$

Example when  $N$  is the product of two prime numbers,  $N$  is small,  $N = 15$

Example when  $N$  is the product of two prime numbers,  $N$  is larger,  $N = 2257$

Example when  $N$  is the product of one prime number and one non-prime number,  $N = 837$

Implementing Shor's algorithm in Python

Shor's algorithm &#x2013; classical implementation

Shor's algorithm &#x2013; quantum implementation

Example implementation on a quantum computer,  $N = 15$ ,  $a = 2$

Finding the period of  $f(x) = ax \pmod{N} = 2x \pmod{15}$

5) using a quantum computer

Subroutine to find  $g(x) = 2x \pmod{15}$

Full algorithm implementation

- Reading out the results
    - Running Shor's algorithm on a quantum computer
    - Tracing $\&#xA0;$ ; after the period is found on the quantum computer
    - Example implementation on a quantum computer,  $N = 35$ ,  $a = 8$
    - Finding the period of  $f(x) = ax\&#xA0;(\text{mod } N) = 8x\&#xA0;(\text{mod } 3)$
    - 5) using a quantum computer
    - After the period is found

Summary

Questions

## 13. Quantum Error Correction

Quantum errors

- Errors on hardware, demonstrating a bit flip error

- Modeling errors in a simulator

Quantum error correction

- Single bit flip error correction

- Classic error correction bit flip

- Quantum error correction single qubit flip

- Quantum error correction single phase flip

- The Shor code $\&#xA0;\&#x2013;$  single bit and/or phase flip

Summary

Questions

## 14. Conclusion - The Future of Quantum Computing

Key concepts of quantum computing

Fields where quantum computing will be useful

Pessimism about the quantum computing field

Optimism about the quantum computing field

Concluding thoughts on quantum computing

Appendix

Useful mathematics

- Summation

- Complex numbers

- Complex conjugation

- Linear algebra

- Matrix

- Matrix multiplication

- A constant multiplied by a matrix

Two matrices multiplied together

Conjugate matrix

Matrix transpose

Matrix conjugate transpose

Matrix Kronecker product&#xA0;

Bra-ket notation

Qubits, states, and gates in terms of matrices

Qubits

Gates

Gate definitions in terms of matrices

Additional gates

Quantum measurement

Other Books You May Enjoy

Leave a review - let other readers know what you think

# Preface

Quantum computers offer a potential sea change in computing power. IBM Research has made quantum computing available to the public for the first time, providing access to IBM QX from any desktop or mobile device via the cloud. Complete with cutting-edge practical scenarios, this book will help you understand the power of quantum computing by programming for the real-world.

*Mastering Quantum Computing with IBM QX* begins with an introduction to the concept and principles of quantum computing and the areas in which they can be applied. You'll see how quantum computing can be used to improve the classical methods of computer processing and convert a classical algorithm into a quantum algorithm. You'll then explore the IBM ecosystem, which facilitates quantum development with the help of the Quantum Composer and Qiskit. As you progress through the chapters, you'll learn how to implement algorithms on the quantum processor and how quantum computations are actually performed.

By the end of the book, you'll have learned how to create quantum programs of your own, explored the impact of quantum computing on your business and career, and future-proofed your programming career.

# **Who this book is for**

If you're a developer or data scientist interested in learning quantum computing, this book is for you. You're expected to have basic understanding of the Python language, but knowledge of physics, quantum mechanics, or advanced mathematics is not required.

# What this book covers

[Chapter 1](#), *What is Quantum Computing?*, gives an overview of what quantum computing is in contrast to classical computing, focusing on its potential advantages over classical computers. We then go over the history of classical and quantum computing and get an overview of the state-of-the-art in computation.

[Chapter 2](#), *Qubits*, gives us an overview of why qubits are the cornerstones of quantum computation along with code to simulate it in Python. It discusses superposition and uses Python code to explore three different representations of a single qubit. The chapter then introduces the Bloch sphere and describes the superposition and measurement of single qubits.

[Chapter 3](#), *Quantum States, Quantum Registers, and Measurement*, is about the quantum version of classic registers, quantum registers, which hold quantum states. We will discuss separable states and entanglement, which leads to a focus on quantum measurement and a Python implementation of quantum measurement for multiple, possibly entangled, qubits.

[Chapter 4](#), *Evolving Quantum States with Quantum Gates*, gives an overview of the most commonly-used gates in quantum computing: I, X, Y, Z, H, S,  $S^\dagger$ , T,  $T^\dagger$ , and CNOT, which form a universal gate set that can be combined to perform any quantum computation. It provides a Python implementation of these commonly used quantum gates, and goes over examples within Python of these gates being applied to the states we have examined so far.

[Chapter 5](#), *Quantum Circuits*, expands on the idea of quantum gates to introduce quantum circuits, the quantum analogue of classical circuits. It goes over how classical gates can be reproduced by quantum circuits and proceeds to introduce a visual representation of quantum circuits that can be used to easily define a quantum circuit without reference to mathematics or use of a programming language.

[Chapter 6](#), *The Quantum Composer*, is all about the interactive interface in IBM QX used to create quantum circuits via quantum scores, the graphical manner of representing circuits introduced earlier. Via the Quantum Composer, you can define your own circuits for implementation on the IBM QX hardware or software simulator. The chapter proceeds to translate the examples previously coded in Python to the Quantum Composer representation and gives you the opportunity to run them on IBM QX hardware.

[Chapter 7](#), *Working with OpenQASM*, describes the **Open Quantum Assembly Language** (pronounced *open kazm*). This language can be used within IBM QX. The chapter revisits some of the quantum circuits defined in previous chapters and redefines them within the OpenQASM language. It then provides you with the opportunity to rerun these circuits on the IBM QX, using OpenQASM.

[Chapter 8](#), *Qiskit and Quantum Computer Simulation*, introduces **Qiskit (Quantum Information Software Kit)**, focusing on how it can be used to run programs in IBM QX via the cloud, as well as its capabilities for quantum simulation. The chapter then moves on to a capstone project using Qiskit to illustrate, in one project, the concepts of quantum circuits, measurement, and Qiskit usage, while producing a useful demo of using a quantum computer to represent musical chords.

[Chapter 9](#), *Quantum AND (Toffoli) Gates and Quantum OR Gates*, explores some of the quantum equivalents of classic Boolean logical gates in order to specify logic problems to be solved by a quantum computer for use with Grover's algorithm and other algorithms.

[Chapter 10](#), *Grover's Algorithm*, goes over the classical implementation compared to the quantum implementation as given by Grover's algorithm. Finally, the chapter provides an implementation of Grover's algorithm in OpenQASM, Qiskit, and quantum score.

[Chapter 11](#), *Quantum Fourier Transform*, describes the Quantum Fourier Transform, which is a sub-routine of many important quantum algorithms, including Shor's algorithm. The chapter next shows the use a classical algorithm to compute the discrete Fourier transform, that is a Fourier transform of a signal represented on a classical computer. Finally, a Quantum Fourier Transform algorithm is given in OpenQASM, Qiskit, and quantum score.

[Chapter 12](#), *Shor's Algorithm*, explores Shor's algorithm. It goes over what prime factorization is, the classical implementation of an algorithm to perform prime factorization as compared to the quantum implementation as given by Shor's algorithm. Finally, the chapter provides an implementation of Shor's algorithm in Qiskit.

[Chapter 13](#), *Quantum Error Correction*, describes the problem of quantum error propagation and illustrates the need for quantum error correction. Next, an overview of **quantum error correction (QEC)** is given, and finally, the chapter provides an implementation of a simple QEC algorithm.

[Chapter 14](#), *Conclusion – The Future of Quantum Computing*, reviews what we have learned and places it in the context of

areas where quantum computing is poised to disrupt, and those understanding quantum computation and quantum programming are likely to be in high demand. In addition, the book discusses why an executive, a programmer, or a technically minded person who doesn't happen to be a quantum physicist might care about quantum computing.

The solutions to practice questions at the end of each chapter are available at <https://www.packtpub.com/sites/default/files/downloads/Assessments.pdf>.

# **To get the most out of this book**

This book assumes you are familiar with the Python programming language. Python 3.4+ is used. You won't need knowledge of quantum mechanics, physics, or advanced mathematics, but should have an understanding of algebra at a high-school level. To get the most out of the book, go through the exercises in the Jupyter Notebooks provided and answer the questions posed at the end of each chapter.

# Download the example code files

You can download the example code files for this book from your account at [www.packt.com](http://www.packt.com). If you purchased this book elsewhere, you can visit [www.packt.com/support](http://www.packt.com/support) and register to have the files emailed directly to you.

You can download the code files by following these steps:

1. Log in or register at [www.packt.com](http://www.packt.com).
2. Select the SUPPORT tab.
3. Click on Code Downloads & Errata.
4. Enter the name of the book in the Search box and follow the onscreen instructions.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR/7-Zip for Windows
- Zipeg/iZip/UnRarX for Mac
- 7-Zip/PeaZip for Linux

The code bundle for the book is also hosted on GitHub at [http://github.com/PacktPublishing/Mastering-Quantum-Computing-with-IBM-QX](https://github.com/PacktPublishing/Mastering-Quantum-Computing-with-IBM-QX). 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://www.packtpub.com/sites/default/files/downloads/9781789136432\\_ColorImages.pdf](https://www.packtpub.com/sites/default/files/downloads/9781789136432_ColorImages.pdf).

# Conventions used

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

`CodeInText`: 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: "Now open the `Hello Quantum World.ipynb` notebook by navigating to it from the Jupyter Notebook GUI."

A block of code is set as follows:

```
| from qiskit.tools.visualization import plot_histogram  
| plot_histogram(job_exp.result().get_counts(qc))
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
| import time  
| q = qiskit.QuantumRegister(5)  
| c = qiskit.ClassicalRegister(5)  
| qc = qiskit.QuantumCircuit(q, c)
```

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

```
|     cd Mastering-Quantum-Computing-with-IBM-QX
```

**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. Here is an example: "Now, click Simulate, then click OK, and within seconds you will get the output."



*Warnings or important notes appear like this.*



*Tips and tricks 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](mailto: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.packt.com/submit-errata](http://www.packt.com/submit-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](mailto: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](http://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](http://packt.com).

# What is Quantum Computing?

In this chapter, we will give an overview of what quantum computing is in contrast to classic computing, focusing on potential advantages quantum computers offer, compared to classic computers. We then go over the history of classic and quantum computing and provide an overview of the present day state of the art in computation. Finally, we provide an overview and an introduction to the approach taken in the book to teaching programming on quantum computers, including an outline of the chapter structure as well as instructions to download and set up necessary software tools. By the end of this chapter, you will be able to describe what a quantum computer is, why they are useful, and their present state of the art. We will also set up the environment to start developing your own quantum computing simulator in Python, to interact with existing quantum computer simulators and hardware via the Python library Qiskit, and to run directly on real quantum computing hardware, IBM QX.

The following topics will be covered in this chapter:

- What are quantum computers?
- The history, present, and future of quantum computing
- Setting up and running Python code examples
- Setting up and running IBM QX examples
- Setting up and running Qiskit examples

# Technical requirements

You will need a Python 3.4+ installation and Git installed. You should ensure you have access to a Unix-like command line, whatever your operating system of choice. This could be the `Terminal.app` on OS X, a Terminal on any Linux machine, or Cygwin or the Bash shell on Windows. Code for this chapter is available in a Jupyter Notebook under <https://github.com/PacktPublishing/Mastering-Quantum-Computing-with-IBM-QX>. The preferred way of interfacing with code is within a Python virtual environment in a Jupyter Notebook, but feel free to use whatever setup works best for you.

# What are quantum computers?

In this book, I call the central processing units that power our laptops, desktops, servers, and mobile phones today, **classical** computers, to contrast them from quantum computers. These classical computers, made of transistors, function by manipulating bits, pieces of data which are either 0 or 1, to perform computation. Quantum computers on the other hand are able to exploit the properties of quantum physics to perform some computations in a different manner, the quantum properties of superposition and entanglement, to perform computations much more efficiently than on a classical computer. It is important to note that any quantum computation can be simulated on a classical computer, but, for specific types of computations, the simulation would take so long or use so much memory as to be practically impossible. It is these problems that quantum computing professionals seek out because they show the benefits of quantum computation over classical computation.

Quantum computing can be performed physically in a variety of fashions. Some quantum computers use the properties of quantum mechanics, including entanglement and superposition, to solve one particular type of optimization problem, that of finding a global minimum of a specific type of function. These computers are called quantum annealing computers, and while there are many optimization problems which can be reformulated in this fashion there are a variety of algorithms they are too

specialized to run. Examples of quantum computers which use quantum annealing are D-Wave machines.

This book focuses on **universal gate quantum computing**. It is the most general form of quantum computation, capable of performing any quantum computation desired, and has analogs to concepts in classic computing such as bits, gates, and circuits. Examples of universal gate quantum computers include those of IBM and Rigetti.

General purpose quantum computers are more difficult to design robustly, but special purpose quantum computers such as quantum annealing machines may ultimately have limited use.

# Uses of quantum computers

**Quantum supremacy** is when a quantum computing algorithm offers significant speedup compared to the best possible algorithm on a classical computer, and this speedup is demonstrated on a quantum computer. When quantum supremacy is achieved over a given classical algorithm, it then makes sense to run the algorithm on a quantum computer instead of a classical computer as the quantum computer will be more efficient. At the time of writing this book, there are a variety of areas which offer the promise of quantum supremacy, but no areas which have convincingly demonstrated it yet. These areas include machine learning and artificial intelligence, cryptography and computer security, searching, sampling and optimization, quantum chemistry and quantum dynamics, and more. But, because no area has demonstrated quantum supremacy as of the writing of this book, at present, it makes more sense to run algorithms on a classical computer instead of a quantum computer.

Quantum supremacy can provide the building blocks to power innovations and disruptions in medicine, finance and business processes, and more. As of the writing of this book, quantum supremacy has not been clearly demonstrated in any area on any machine, although engineers and scientists have a variety algorithms to offer the potential, quantum hardware still needs to improve before this potential is realized. These hardware improvements necessary to enable quantum supremacy may include, for example, making more qubits available, making the qubits more

stable, and increasing the error correction of those qubits. Google, IBM, and others expect to demonstrate quantum supremacy in the near future because they expect to be able to make these improvements.

Why is quantum computing important? There's no one reason that outshines the others, but here are some comments from experts in the field highlighting why they work on it:

- Harnessing quantum computation is a worthy challenge that pushes humanity to be its best
- Quantum computing is needed because digital (classic) computing is running into the limits of its capability and quantum computing is the next frontier
- The universe is a quantum world, so to understand it we need quantum computation
- Quantum computers will help us tackle problems too difficult for a classic computer
- Developing new hardware based on new physics has historically helped progress in technology quantum computation should be no different
- Quantum computers will let us do things we can't do today

# Experts weigh in - why is quantum computing important?

*Quantum computing is important because the universe is a fundamentally quantum place. We have had great success over the past century building computing systems that allow us to model, simulate, analyze and interpret the world around us. However, as those computing systems have been based on classic logic, they are limited that they can only efficiently model a universe that approximates a classic one. We know, with astounding precision that the universe in which we exist is fundamentally quantum. And so in order to understand the universe as it actually exists, we must use a quantum representation. We do not know for sure if our own universe is itself a simulation. But we can say that if it is, it's running on a quantum computer.*

- Aaron Van Devender, PhD; Chief Scientist, Founders Fund

Because quantum computing will play a major role in the future of computing, studying the field isn't just for physicists and researchers. Software engineers, executives, investors, and more need to develop fluency as well. Here, an expert states why:

*In many cases, executives care because they have to—because of hype or pressure or whatever. Today, though, what they need to understand is quite limited. Commercially applicable quantum algorithms are still many years from being realized. I think in any company with a potential stake in new technology, the most important thing is due diligence and the skills to spot charlatans. Quantum computing should be on the radar and you should know how to evaluate the claims of quantum salespeople.*

- Dr. Chris Ferrie, Centre for Quantum Software and Information, University of Technology Sydney

# **History, present, and future of quantum computing**

In this section we will focus on the history, state of the art, and future of universal gate quantum computing, as it is the model used for IBM QX and the focus of this book.

# History of quantum computing

Quantum computing has made great strides in recent years but it is not a new idea. Quantum physics was developed in the 1920s and 1930s. In the early 1980s, physicist Richard Feynman encouraged computer scientists to develop new models of computation based on quantum physics, which showed the promise of performing computation in a different and more efficient manner. In the 1990s, the field of quantum computing progressed when the first algorithms were developed that could run more efficiently on quantum computers than on classic computers. In the 2000s, practical quantum computers began to be able to implement some of the theoretical algorithms from the 1990s for very small inputs.

Important historical developments in quantum computing include:

- Feynman's introduction of quantum computing (1981-1982)
- The demonstration that quantum computing can perform better than classic computing (1985)
- Shor's algorithm (1994)
- Quantum error correction (1995)
- Grover's algorithm (1996)
- The quantum threshold theorem (1999)
- Cross pollination from different quantum computing systems in recent years

# **Present/current state of quantum computing**

At the time of writing this book, IBM offers a 5-qubit and 16-qubit computer available in the cloud via IBM QX and has a 50-qubit quantum computer available to research partners and a 100-qubit quantum computer under way. Rigetti Computing offers an 8-qubit device. While these computers currently demonstrate interesting principles of quantum computation and allow for prototyping and testing algorithms, they are not yet large enough in terms of qubit size and robust enough to exhibit quantum supremacy.

# Future of quantum computing

At the time of this book's writing, IBM estimates that exhibiting quantum supremacy for a specialized application should be possible with 50-100-qubits. This should happen in the near future. Demonstrating quantum supremacy for one specialized application is different to being able to realize the full potential of quantum computing. To have truly useful quantum computers across a variety of fields with universal gate quantum computers, the quantum computing hardware must advance to incorporate more qubits that work together robustly for a longer period of time. Many of these qubits will not be used in the core computation, but rather for error correction to deal with errors introduced by the environment during the course of computation.

The outlook for quantum computing going forward is a topic of debate, and some experts argue it's difficult to predict. Some ideas about where quantum computing is heading in the near future include:

- Hybrid quantum systems that are made of different quantum computing techniques
- Continued steady progress in hardware and big leaps in quantum algorithms
- Real world practical applications

In the far future, we may see these advances enabling large quantum computers, and encouraging their wide-scale use.

# **Experts weigh in - what is the outlook for quantum computing in the future?**

*The big near term milestones will be a demonstration of quantum computational supremacy (performing a computation on a quantum processor that takes much longer to compute using conventional high-performance computing), a demonstration of quantum advantage (demonstrating that a quantum processor can perform a -useful- computation faster than conventional computing resources) and a demonstration of fault-tolerance with active error correction. I would expect each of these to occur within the next ten years, and probably within the next five. The demonstration of quantum computational supremacy is likely to be the first to occur, and is likely within the next two years. Within fifty years, I would certainly expect quantum computers to be much more similar to todays computer architectures in terms of having multiple components dedicated to different functions (processing, storage, communication) and in terms of being used as general purpose computing devices rather than single purpose co-processors.*

*- Dr. Joe Fitzsimons, Principal Investigator, Centre for Quantum Technologies; Founder, Horizon Quantum Computing*

# **Setting up and running Python code examples**

In this section I will show you how to set up a Python virtual environment for the Python code examples in the book, set up a checkout, and run simple examples. We will be working with Python 3.4+, so if you have Python 2 installed on your system, for some of the commands, you may need to type `python3` to call the correct version of Python in the commands to be followed.

# Get the book code

You can clone the code for this book using the following steps:

1. Clone the repository containing all the code examples with the following command:

```
| git clone https://github.com/PacktPublishing/Mastering-Quantum-  
| Computing-with-IBM-QX
```

2. Navigate to the directory using the following command:

```
| cd Mastering-Quantum-Computing-with-IBM-QX
```

3. Create a virtual environment in which to run all code examples using the following command:

```
| python -m venv MQC
```

4. You must always make sure you are running in the correct virtual environment before trying to run chapter examples or attempting chapter projects as follows:

```
| source MQC/bin/activate
```

5. Next install the `requirements.txt` file present in the GitHub link, for running chapter examples and attempting chapter projects by using the following command:

```
| pip install -r requirements.txt
```

6. Check your installation by running the following command:

```
| python chapter01/test_installation.py
```

7. Debug, if there are any failures.

That's it. You are now ready to take advantage of the Python examples and learn by doing chapter projects. If you'd rather not run in a virtual environment, make sure you have installed all the modules listed in `requirements.txt` via `pip`, your preferred package manager, or the source.

# Set up Jupyter Notebook

Jupyter Notebooks are convenient workbooks to organize Python code, text and the results of running code all in one place. This book illustrates all the Python focused examples within a Jupyter Notebook although, in principle, you can execute the Python code in any Python interpreter you choose. If you want to run a Jupyter Notebook within a Python virtual environment, run within that environment: `pip install ipykernel` and `ipython kernel install --user --name=bookkernel`. Then after launching Jupyter Notebook, navigate to Kernel | bookkernel in the notebook's UI. To test out your Jupyter Notebook setup, run the following command:

```
| jupyter notebook
```

Now open the notebook `Hello Quantum World.ipynb` by navigating to it from the Jupyter Notebook GUI. Run the example line. If you have trouble at any step, consider an online tutorial or consulting Jupyter's documentation at <https://jupyter-notebook.readthedocs.io/en/stable/>.

# Setting up and running IBM QX examples

In this section you will setup an account with IBM QX and run a simple program:

1. Setup a new account at <https://quantumexperience.ng.bluemix.net/qx/signup>.
2. After you sign in with your new account, click on the Composer tab under the Learn drop-down menu.

Here you will see something that looks like five wires, or if you are familiar with sheet music, like a music score, without any music notes. This is an interface to the IBM QX quantum computers called the **quantum composer** and we will use it to program and then run our first quantum program in simulation. In a later chapter, this interface will be explored in detail and we will run our quantum programs on the IBM hardware itself.

# Hello quantum world

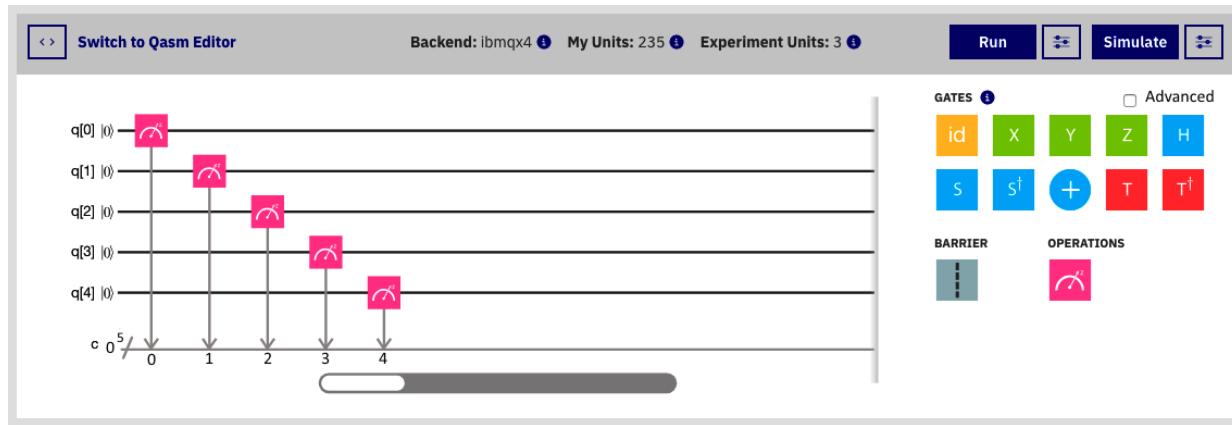
Every classic bit either 0 or 1 can be written as either a "0" or a "1" qubit, which to show that it is a qubit, is written as a name surrounded by / and >. So, for example, the qubit named "0" is written as /"0">> and the qubit named "1" is written as /"1">>. Throughout this book, the qubits will always be written as names surrounded by quotation marks and / and > to indicate that they are, indeed, qubits.

Each wire in the quantum composer starts at a quantum equivalent of the 0 bit, the /"0">> qubit. To get the output of a quantum computation, we measure the qubits we are interested in, and the measurement will always output either a 0 or a 1 classic bit. So we have a classic input in bits, that we translate to quantum qubits, after which we can perform quantum computations. Then, to get the output, we perform a quantum measurement which gives us classic bits out.

For our first quantum program, we will input five 0 bits, then measure the output. We should get 00000 out. The way we program measurement in the IBM QX is with the pink box



. Drag one to each wire, for a total of five. It doesn't matter which order they are in in this case:



Now, click Simulate, name your experiment Hello Quantum World, click OK, and within seconds you will get the output. Not surprisingly it is `00000` 100% of the time:



Congratulations, you just wrote and executed your first quantum program.

Any quantum program can be simulated on a classic computer. The question is, how efficient is the simulation? For small quantum programs, for example ones dealing with five qubits (quantum bits), the results can be easily simulated on a classic computer, like our laptops or desktops at home, or remotely on IBM Cloud compute services. As the

programs get larger in terms of the number of quantum bits they manipulate, simulating gets more computationally intense, meaning they won't run efficiently on our own machines, and it is quicker to use IBM's simulation infrastructure. At a certain size, simulating a quantum program in a classic computer is so computationally difficult, it isn't worth bothering. Luckily, IBM and other companies are working on quantum computing that can run such programs efficiently by executing them not in simulation, but in physical hardware.

# API key

The IBM QX allows us to interface with IBM's remote quantum hardware and quantum simulation devices. We can execute code to simulate a quantum computer on our local machines, but it is good to have the option to run on IBM's simulation architecture or IBM's quantum computing hardware as we so choose. To do this, we will need an API key. Find the API key within the IBM QX menu under your account name called My Account (It should be in the upper right of the user interface). Click on Advanced, then Regenerate, and then on Copy API Token, to copy your API token. Insert this token into the [Chapter 1 Jupyter Notebook](#) and make note of it for future use, or come back to this section when you need it.

# Setting up and running Qiskit examples

Qiskit is an open source quantum computing framework for writing quantum experiments, programs, and applications. It provides tools for simulating quantum computation, as well as providing a Python interface to the IBM QX via an API, among many other features. We won't go into Qiskit in detail in this chapter, but we will make sure you can install Qiskit and run examples.

If you followed the instructions to set up a virtual environment for the exercises in this book, Qiskit should already be installed. If not, make sure you have installed Qiskit via `pip`, your preferred package manager, or the source. Qiskit documentation and source is available at [http://github.com/QISKit](https://github.com/QISKit).

1. Check your installation of Qiskit within the Jupyter Notebook of this chapter by executing the following command:

```
|     from qiskit import IBMQ
```

2. Now that that's working, insert your API token in the following code from the previous subsection:

```
|     IBMQ.save_account("INSERT_YOUR_API_TOKEN_HERE")
```

A variety of backends are available, some physical quantum computing hardware, and other quantum simulators, either locally or in the cloud.

4. You can get a list of available remote backends with the following code:

```
| print(IBMQ.backends())
```

Here's an example list:

```
[<IBMQBackend('ibmqx4') from IBMQ(),>
 <IBMQBackend('ibmq_16_melbourne') from IBMQ(),>
 <IBMQBackend('ibmq_qasm_simulator') from IBMQ(),>]
```

As IBM adds and modifies its capabilities, and as you get increased access (some hardware configurations are only available via application or industry partnership), this list may change, so make sure to execute the code yourself and choose a backend off of the list that you have access to. Pick an item off the list that ends in `_qasm_simulator` as your backend. For example, I might choose the following based on the preceding list:

```
| backend = IBMQ.get_backend('ibmq_qasm_simulator')
```

Next, we will execute our first quantum program in simulation.

This will be the same program we executed in the simulator on the IBM QX website by programming it in the quantum composer. This time we will program it in Qiskit. Recall, this program starts with an input of the quantum equivalent of five "0" bits, five quantum "0" bits. It then performs no action before it outputs the classic equivalent of the quantum bits. Since they haven't changed, the output should be `00000`.

We will learn more about this code, line by line, in future chapters. For now, know that for each line of code we can do the same thing in the quantum composer.

5. Execute your code on the IBM simulator by running the following:

```
| q = qiskit.QuantumRegister(5)
| c = qiskit.ClassicalRegister(5)
| qc = qiskit.QuantumCircuit(q, c)
| qc.measure(q, c)
| job_exp = qiskit.execute(qc, backend=backend)
```

The code may take some time to complete. Now we want to visualize our results. Just as before in the GUI, we can see the results graphically with the following code:

```
| from qiskit.tools.visualization import plot_histogram
| plot_histogram(job_exp.result().get_counts(qc))
```

Indeed, we do get 100% of the time.

# Summary

In this chapter, we learned that quantum computers exploit the properties of quantum physics, superposition and entanglement, to perform some computations much more efficiently than on a classic computer. We learned that while quantum computing has been shown to theoretically offer significant speedup over classic computers in a variety of areas, quantum computers are under development and aren't yet robust enough to exhibit this speedup. We introduced areas which will be impacted by quantum supremacy which include machine learning and artificial intelligence, cryptography and computer security, searching, sampling and optimization, quantum chemistry and quantum dynamics and more.

In the next section, we will be introducing qubits in more depth, and cover how to simulate, represent, and measure qubits.

# Exercises and questions

1. Create a new quantum score called `Playing Around` and drag some boxes from the user interface to the wires. Then add in measurement boxes for each of the wires and click Simulate. What was your result?
2. Try changing the number of qubits and classic bits in your first quantum program from five to two. When you run it, what are your results?
3. Let's run your first quantum program on an actual quantum computer instead of a simulator! You can choose to do this either from the quantum composer user interface, or from Qiskit. If you choose to do from the quantum composer user interface, do as you did in the *Hello quantum world* section, only, this time, instead of clicking Simulate, click Run. If you choose to do this from Qiskit, then change the backend to an option beginning with `ibmq_5`, and then Run. Note that, in either case, the run will take a lot longer. What is your result? Why do you think it is different from the simulation?
4. If you wanted a qubit halfway between 0 and 1, where would it lie on the sphere we have used to visualize a qubit?

# Qubits

This chapter goes over the quantum generalization of the classic bit, the qubit. It gives an overview of why qubits are the cornerstones of quantum computation. It provides code that simulates a qubit in Python. It then discusses superposition and uses Python code to explore three different representations of a single qubit, which this book calls zero/one, plus/minus, and clockwise/counterclockwise. The chapter then introduces the Bloch sphere, a way to visualize a single qubit on a sphere, and provides Python code to explore this visualization. It provides a section describing the superposition and measurement of single qubits.

The following topics will be covered in this chapter:

- Qubits
- Simulating a qubit
- Three different representations of qubits
- The Bloch sphere
- Superposition and measurement of qubits

# Technical requirements

Code for this chapter is available in a Jupyter Notebook at [http://github.com/PacktPublishing/Mastering-Quantum-Computing-with-IBM-QX](https://github.com/PacktPublishing/Mastering-Quantum-Computing-with-IBM-QX) under chapter02. See [Chapter 1](#), *What is Quantum Computing?* for more details as to how to set up the Python prerequisites.

# Qubits

A qubit is the quantum equivalent of a classic bit.

A classic bit is a binary piece of information used in classic computing that can take one of two values, for example 0 or 1, on or off, black or white, up or down. Manipulating bits of information is what powers our modern digital computers, which, to contrast them with quantum computers, I call classical computers in this book.

A qubit is a quantum piece of information used in quantum computing that represents one possible combination of two values. Manipulating qubits of information is what powers many modern quantum computers, including IBM QX.

By analogy, imagine if you had two paints, black paint and white paint. A classic bit could represent either all black paint, or all white paint, whereas in quantum computing, a qubit could represent any shade of gray resulting from a mixture of the two. This is just an analogy because a qubit is far more powerful than just representing shades of gray, as the possible combinations of the two values that they represent are themselves more complicated than black or white.

# **Storing a qubit**

Storing the equivalent of a qubit of information in a classic computing scenario would require more than one classic bit. We couldn't just say 0 or 1, white or black, because we are dealing with a potential mixture. But in principle we can represent a qubit using classic bits, it would just take up significant storage space on a classic computer. Likewise, we can simulate manipulation of qubits using classic computation. It would just take up significant storage space to represent the computation and its intermediate steps and a significant amount of time to perform the computation.

Quantum physics allows us to store a qubit of information in just one physical entity, such as a two-level quantum mechanical system. Just like a bit, which has many possible representations, 0 or 1, on or off, black or white, up or down, and so on, a qubit has many possible representations. If we work directly with qubits instead of trying to simulate them with classic bits, we can do some sorts of computations in less space and less time than they would take on a classic computer.

# Simulating a qubit

In this section, you will learn to use Python code to simulate one representation of a qubit. Just like a classic bit, a qubit can have many ways of expressing itself physically and conceptually.



*To distinguish a qubit (or collection of qubits, called a quantum state) from a classic bit or any other variable, when I write about it in text, I will always enclose name in quotations and that itself in `|>`, for example, `|"qubit_name">`, or `|"state_name">`. To read this out loud, just read the part between the quotation marks, in this case `qubit_name`. When I provide code examples, the code will always be in fixed width font and a qubit variable name will always end in `_qubit`, for example `zero_qubit` or `one_qubit`.*

In this section, I will go over the most common representation, combinations of `|"0">` and `|"1">`, and use Python to explore the combinations.

One thing to note is that because of the connection between mathematics and quantum physics, anything in quantum physics can be written down as mathematical operations and vice versa. Any mathematical operation can be written out as classic computer code. Thus, any quantum computing physics we want to talk about can just as well be written by math or by code. Here, we'll choose to write it out with code. If you are interested in the details of the math, see the Appendix of this book.

Because most textbooks and discussions in quantum computing choose to write things out with mathematics, I will briefly review how to translate your code into mathematical notation at the end of this section. That way, you'll be able to get started coding without having to understand the math, but still be able to follow other descriptions of quantum computing, which assume you know the math inside and out.

# $|"0">$ and $|"1">$

We've learned that a qubit represents a combination between two things on opposite ends of some spectrum. So for each type of qubit, we need two things to combine. Here, we choose two items, which we'll call  $|"0">$  and  $|"1">$ . We could name them  $|"Alice">$  and  $|"Bob">$  or anything we like, but in the quantum computing community, for these items,  $|"0">$  and  $|"1">$  are the convention.

If you say  $|"0">$  and  $|"1">$  referencing qubits, quantum computing experts will know what you mean. Here is the Python code to implement these qubits:

```
import numpy as np  
zero_qubit=np.matrix('1; 0')  
one_qubit=np.matrix('0; 1')
```



*Note: the quantum computing community drops the quotation marks surrounding the names of qubits. Here I use them to emphasize that even when the names of qubits happen to be numbers, they are still just names. `zero_qubit` or  $|"0">$  for example, does not equal 0. The purpose of the quotation marks is to make extra clear that "0" is just the name of the qubit and doesn't have anything to do with the integer 0.*

# Combinations of $|0\rangle$ and $|1\rangle$

Now, we need to know how to combine  $|0\rangle$  and  $|1\rangle$  as the most general qubit is a combination between the two. Physicists, mathematicians, and those involved in quantum computing call these combinations **superpositions**. Our choices of what to combine are between  $|0\rangle$  and  $|1\rangle$ ; the options we are choosing between are called the **basis**. There are many different options for a *basis* in quantum computing, and we will go over a few sets in this chapter.  $|0\rangle$  and  $|1\rangle$  are the most common *basis*. If we choose  $|0\rangle$  and  $|1\rangle$  as our basis, to specify our superposition we can specify the percentage of *0* we want and the percentage of *1* that we want.

Since,  $|0\rangle$  and  $|1\rangle$  are on opposite ends of the spectrum, and we can't have more than 100% of *1* or less than 0% of *0*, these two percentages should add up to 100% and both should be more than 0%. The combination of the *1* and *0*, which can represent any sort of qubit, is going to be  $|0\rangle$  or, in code, `a*zero_qubit+b*one_qubit`. Here, `a` and `b` are variables that help us describe the percentage of `zero_qubit` and percentage of `one_qubit`.

Unfortunately, `a` and `b` aren't exactly equal to the percentage, for reasons we'll go over later in the *Superposition and measurement of qubits* section of this chapter, but the percentages can be calculated from them. The way the math works out is that the fraction of `zero_qubit` is equal to whatever is number before it, `a`, squared. So, if we want to convert from percentage to a fraction, we'll need to divide by 100, then convert from a fraction to the number to put before

`zero_qubit` in our function, and we'll need to take the square root, or `sqrt`. It is the same with whatever is before `one_qubit`, `b`.



A square root result can be either of two values, positive or negative. For simplicity, in the following sections we will consider only one possibility: the square roots are all positive. In one of the exercises at the end of the chapter, you will modify these functions to consider any possibility.

We will define a Python function called `zero_to_one_qubit` to help us out:

```
def zero_to_one_qubit(percentage_zero,percentage_one):
    if not percentage_zero+percentage_one==100 or percentage_zero<0
        or percentage_one<0: raise Exception(
            "percentages must add up to 100% and both be positive ")
    return
    sqrt(percentage_zero/100.)*zero_qubit+sqrt(percentage_one/100.)*one_qubit
```

We can give a name to the new qubit we create. Using our preceding function, let's create a few new qubits:

```
fifty_fifty_qubit=zero_to_one_qubit(50,50)
ten_ninety_qubit=zero_to_one_qubit(10,90)
```

If we wrote this in words/algebra instead of code, we would write `fifty_fifty_qubit` as follows:

$$\begin{aligned} |“fifty\_fifty”\rangle &= \\ \sqrt{0.5}|“0”\rangle + \sqrt{0.5}|“1”\rangle &= \\ 0.70710678|“0”\rangle + 0.70710678|“1”\rangle \end{aligned}$$

For `ten_ninety_qubit`, we would write it in algebra as follows:

$$\begin{aligned} |“fifty\_fifty”\rangle &= \\ \sqrt{0.5}|“0”\rangle + \sqrt{0.5}|“1”\rangle &= \\ 0.70710678|“0”\rangle + 0.70710678|“1”\rangle \end{aligned}$$

Here, we can see that the qubit "*fifty\_fifty*" and the qubit "*ten\_ninety*" is just composed of parts of the "0" qubit and parts of the "1" qubit. The code lets us skip doing any square roots on our calculators and read off what to put before we can print this out to see what it represents:

The `print(fifty_fifty_qubit)` statement prints the following:

```
| [[ 0.70710678]
| [ 0.70710678]]
```

And the `print(ten_ninety_qubit)` statement prints the following:

```
| [[ 0.31622777]
| [ 0.9486833 ]]
```

Compare this to what we wrote out in words/algebra, and you can see that the first number printed is the number that goes before the first basis qubit, and the second number printed is the number that goes before the second basis qubit. With this Python code, we can skip the algebra.

# Three different representations of qubits

In this section, you will learn to use Python code to simulate other representations of a qubit and their superposition.  $|0\rangle$  and  $|1\rangle$  will remain our standard basis, but this chapter will show you that a variety of possibilities exist and help you to read and follow writing on quantum computation.

Primarily, we'll be dealing with the  $|0\rangle$  and  $|1\rangle$  basis, but this section should help illustrate that just like a classic bit, which we can represent by many different physical and conceptual things, qubits can have a variety of different representations. This section will illustrate the most common representations: the  $|0\rangle$  and  $|1\rangle$ , the  $|+\rangle$  and  $|-\rangle$ , and the  $|+\rangle$  and  $|-\rangle$  basis. Which basis you use is dependent on the algorithm; in some choices, writing an algorithm is easier in a particular basis. In addition, the physics of each basis is different, so it may be more convenient to describe a physical system in a particular basis. As a final note, with some code or math, it is possible to switch from one basis to another, which might be useful in a calculation the same way that looking at the same object from a different angle might help tell what shape it is, metaphorically, for example. We won't cover that in this section.

The important part when choosing a basis is making sure its parts are "opposite" in some sense. This is defined mathematically (for details, see the Appendix of the book), and for each of the examples here the two parts of the basis

are opposite. This makes sense in the paint analogy. If we had only white paint and one shade of gray paint, we wouldn't be able to make any shade of gray. For that, we'd need the two opposites: white and black paint. The same goes with the basis. If the choices weren't completely "opposite," then we'd be missing out on the types of things we could represent.

To prepare for different representations, we can write some code that will return a qubit for any basis:

```
def qubit(percentage_first,percentage_second,basis_first,basis_second):
    if not percentage_first+percentage_second==100
        or percentage_first<0 or percentage_second<0:
        raise Exception(
            "percentages must add up to 100% and both be positive ")
    return sqrt(percentage_first/100.)
        *basis_first+sqrt(percentage_second/100.)*basis_second
```

This means, in terms of our previous function, `qubit(50,50,zero_qubit,one_qubit)` is the same as `zero_to_one_qubit(50,50)` and `qubit(10,90,zero_qubit,one_qubit)` is the same as `zero_to_one_qubit(10,90)`.

# Zero and one basis - additional notes

We already went over  $|0\rangle$  and  $|1\rangle$  in the last section, but here I will make three additional comments that will help you read information about quantum computing from other sources:

- This basis is also described as the **computational basis**, the ***z* basis**, or the **standard basis**, all of which are synonyms for the  $|0\rangle$  and  $|1\rangle$  basis. In the next section, we will go over visualizing these states on a sphere and see that  $|0\rangle$  and  $|1\rangle$  lie on the *z* axis. This is where the *z* basis name comes from. Since  $|0\rangle$  and  $|1\rangle$  are the most standard basis choice in quantum computation, this is where the names computational basis and standard basis come from.
- Throughout this book, I keep the qubit name in quotations to make it clear that it's a name, but in other books you'll see people drop the quotations and just write  $|0\rangle$  and  $|1\rangle$ . For this book, you'll continue to see me always put the name of the qubit in quotations and use the  $|\text{"qubit\_name"}\rangle$  notation.
- You might also see the  $|$  and  $>$  parts reversed and written as  $\langle "0" |$  and  $\langle "1" |$ . Under the hood, this means the math the symbols represent changes slightly, but we are not going to need to go into that for now. If you are curious about the mathematics already, see the Appendix for more details.

# The plus and minus basis

$|+>$  and  $|->$  are, just as  $|0>$  and  $|1>$  are, in some sense opposites. Even their names,  $|+>$  and  $|->$ , help us remember that they are opposites and that they can be grouped together as a basis. They defined in words/algebra as follows:

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}} \quad \text{and} \quad |-\rangle = \frac{|0\rangle - |1\rangle}{\sqrt{2}}$$

We'll define these in code:

```
plus_qubit=1/sqrt(2)*np.matrix('1; 1')
minus_qubit=1/sqrt(2)*np.matrix('1; -1')
```

We can use them in our new qubit function to obtain qubits between  $|+>$  and  $|->$ . For example, `qubit(50,50,plus_qubit,minus_qubit)` or `qubit(10,90,plus_qubit,minus_qubit)` are used to get qubits between these two opposites.

This basis is also described as the  $x$  basis or the **diagonal basis**. In the next section, we'll go over visualizing these states on a sphere, and see that  $|+>$  and  $|->$  lie on the  $x$  axis of the sphere. This is where the name  $x$  basis comes from.

# The clockwise and counterclockwise basis

$|“\circlearrowleft\rangle$  and  $|“\circlearrowright\rangle$  are defined in words/algebra as follows:

$$|\text{“}\circlearrowleft\text{”}\rangle = \frac{|\text{“}0\text{”}\rangle + i |\text{“}1\text{”}\rangle}{\sqrt{2}} \quad \text{and} \quad |\text{“}\circlearrowright\text{”}\rangle = \frac{|\text{“}0\text{”}\rangle - i |\text{“}1\text{”}\rangle}{\sqrt{2}}$$

Here,  $i$  is  $\sqrt{-1}$ .

We'll define these in code as follows:

```
|clockwisearrow_qubit=1/sqrt(2)*np.matrix([[1],[np.complex(0,1)]])  
|counterclockwisearrow_qubit=1/sqrt(2)*np.matrix([[1],[-np.complex(0,1)]])
```

Then, we can use them in our new qubit function to obtain qubits between  $|+>$  and  $|->$ , for example,

`qubit(50,50,clockwisearrow_qubit,counterclockwisearrow_qubit)` or  
`qubit(10,90,,clockwisearrow_qubit,counterclockwisearrow_qubit)` to get qubits between these two opposites.

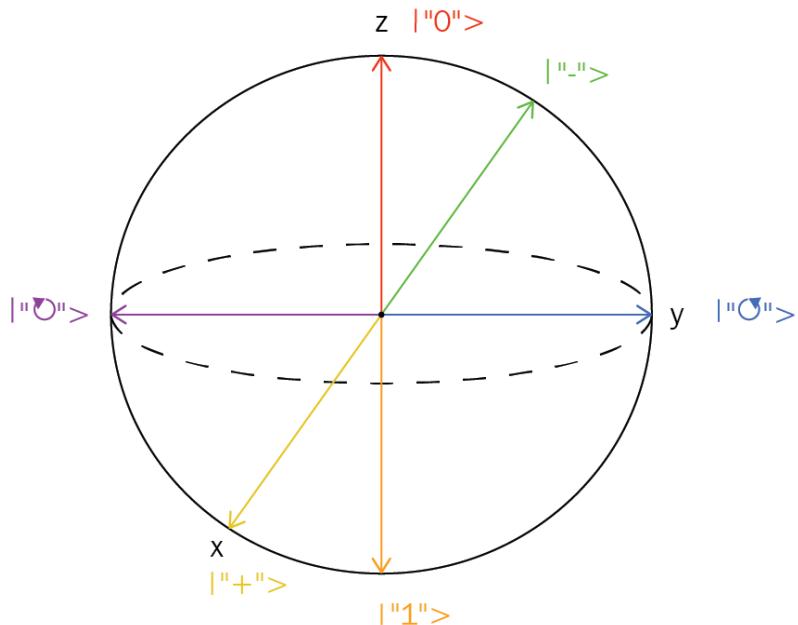
This basis is also described as the  $y$  basis or the **circular basis**. In the next section, we will go over visualizing these states on the Bloch sphere and see that  $|\text{“}\circlearrowleft\text{”}\rangle$  and  $|\text{“}\circlearrowright\text{”}\rangle$  lie on the  $y$  axis of this sphere. This is where the name  $y$  basis comes from.

# The Bloch sphere

In this chapter, you will learn to visualize a single qubit on the Bloch sphere with Python code. Unless your qubit is on one extreme or the other of the continuum, it might be hard to get a sense from looking at the code of how it compares to other qubits. Visualizing a qubit on something called a **Bloch sphere** is one technique to picture the qubit. This technique will be especially useful as we start talking about manipulating qubits. We will see that for a single qubit, any operation we perform on it is the same thing as just moving the qubit around on the Bloch sphere in a certain predefined way. Unfortunately, the Bloch sphere will not work for visualizing more than one qubit, but it is a great tool to visualize a single qubit.

# $|"0">$ and $|"1">$ and other basis' on the Bloch sphere

The Bloch sphere is a three-dimensional sphere that has unit radius. Any qubit can be represented as a line from the center of the sphere to a point on the sphere with an arrow at the end (a vector), as shown in the following diagram:



On the Bloch sphere, we visualize all the basis qubits we have learned so far,  $|"0">$  and  $|"1">$ ,  $|"+>$  and  $|"->$ ,  $|"\circlearrowleft">$  and  $|"\circlearrowright">$ .

We can see on the Bloch sphere  $|"0">$  and  $|"1">$  are opposites,  $|"+>$  and  $|"->$  are opposites, and  $|"\circlearrowleft">$  and  $|"\circlearrowright">$  are opposites. This makes sense, because as we saw in the previous section, the important part when choosing a basis is making sure its parts are "opposite" in some

sense. Here, each of these basis choices are good choices, as each basis is formed of states that are opposites. We can also see that  $|"+">\rangle$  and  $|"-">\rangle$ ,  $|“\circlearrowleft”\rangle$  and  $|“\circlearrowright”\rangle$  are all 50%  $|“0”\rangle$  and 50%  $|“1”\rangle$ , each in different directions.

# Bloch coordinates from qubit

By now, if I give you any of  $|0\rangle$  and  $|1\rangle$ ,  $|+\rangle$  and  $|-\rangle$ ,  $|(\circlearrowleft)\rangle$  or  $|(\circlearrowright)\rangle$ , you should be able to plot it on the Bloch sphere, but a qubit could be anywhere on the sphere. This subsection will go over how to figure out the coordinates of the qubit. There is a fair bit of mathematics involved to compute the coordinates, so I've gone ahead and provided you with a function, `get_bloch_coordinates`, which returns the  $x$ ,  $y$ ,  $z$  coordinates on the Bloch sphere of any qubit. If you are curious about the mathematics, check the Appendix. For now, feel free to use this function to help visualize any qubit you are working with:

```
def get_bloch_coordinates(qubit):
    def get_x_bloch(qubit):
        qubit_x_basis=1./np.sqrt(2)*np.matrix('1 1; 1 -1')*qubit
        prob_zero_qubit=
            (qubit_x_basis.item(0)*qubit_x_basis.item(0).conjugate()).real
        prob_one_qubit=
            (qubit_x_basis.item(1)*qubit_x_basis.item(1).conjugate()).real
        return prob_zero_qubit-prob_one_qubit

    def get_y_bloch(qubit):
        qubit_y_basis=1./np.sqrt(2)*np.matrix('1 1; 1 -1')*np.matrix(
            [[1,0],[0,-np.complex(0,1)]])*qubit
        prob_zero_qubit=
            (qubit_y_basis.item(0)*qubit_y_basis.item(0).conjugate()).real
        prob_one_qubit=
            (qubit_y_basis.item(1)*qubit_y_basis.item(1).conjugate()).real
        return prob_zero_qubit-prob_one_qubit

    def get_z_bloch(qubit):
        qubit_z_basis=qubit
        prob_zero_qubit=
            (qubit_z_basis.item(0)*qubit_z_basis.item(0).conjugate()).real
        prob_one_qubit=
            (qubit_z_basis.item(1)*qubit_z_basis.item(1).conjugate()).real
        return prob_zero_qubit-prob_one_qubit
    return (get_x_bloch(qubit),get_y_bloch(qubit),get_z_bloch(qubit))
```

# Plotting Bloch coordinates on the Bloch sphere

Now for the fun part; let's visualize the qubits in a 3D plot on the Bloch sphere. You can use this code whenever you want to visualize a qubit:

```
def plot_bloch(qubit,color='b',ax=None):
    import matplotlib.pyplot as plt
    from mpl_toolkits.mplot3d import Axes3D
    if not ax:
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')

    # draw sphere
    u, v = np.mgrid[0:2*np.pi:20j, 0:np.pi:10j]
    x = np.cos(u)*np.sin(v)
    y = np.sin(u)*np.sin(v)
    z = np.cos(v)
    ax.plot_wireframe(x, y, z, color="k",alpha=.1)
    ax.grid(False)

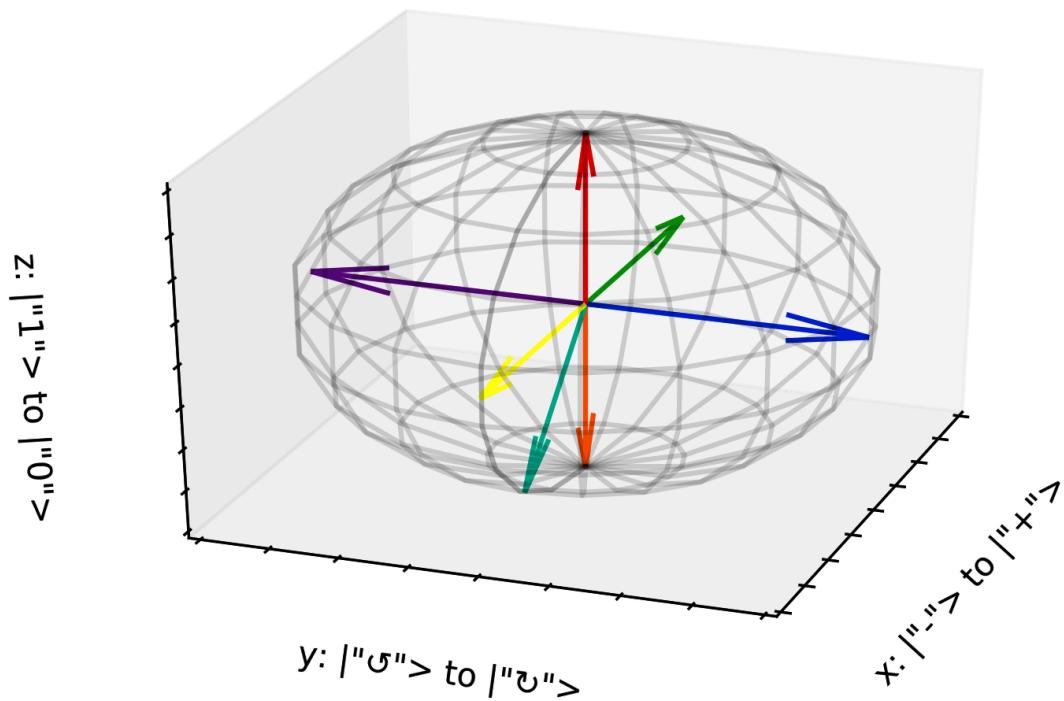
    coordinates=get_bloch_coordinates(qubit)
    ax.quiver([0],[0],[0],[coordinates[0]],[coordinates[1]],
              [coordinates[2]],length=1,color=color,arrow_length_ratio=0.3)
    ax.set_xlim([-1,1])
    ax.set_ylim([-1,1])
    ax.set_zlim([-1,1])
    ax.set_xlabel('x: | "-"> to | "+">')
    ax.set_ylabel('y: | "σ"> to | "υ">')
    ax.set_zlabel('z: | "1"> to | "0">')
    ax.view_init(azim=20)
    return ax
```

Now, we can visualize all the basis qubits, plus a bonus qubit, on the same sphere with the following code:

```
# Plotting all of our basis qubits, colors and orientation match the
# textbook figure
ax=plot_bloch(zero_qubit,color='xkcd:red')
plot_bloch(one_qubit,color='xkcd:orange',ax=ax)
plot_bloch(plus_qubit,color='xkcd:yellow',ax=ax)
plot_bloch(minus_qubit,color='xkcd:green',ax=ax)
plot_bloch(clockwisearrow_qubit,color='xkcd:blue',ax=ax)
plot_bloch(counterclockwisearrow_qubit,color='xkcd:purple',ax=ax)
```

```
# Now plotting a qubit that is 10% |"0"> and 90% |"1"> in turquoise
plot_bloch(zero_to_one_qubit(10,90),color="xkcd:turquoise",ax=ax)
```

That will give us the following:



# **Superposition and measurement of qubits**

In this section, you will learn the physics behind superposition and how to explain it. You will learn about quantum measurement of a single qubit and how to represent it on the Bloch sphere.

# Quantum superposition for qubits

In this chapter, we've already dealt with the concept of **superposition** of single qubits: two or more qubits combined to produce another qubit.

Some of these examples of superposition are when we wrote  $|"+\rangle$ ,  $|"- \rangle$ ,  $|“\circlearrowleft\rangle$  or  $|“\circlearrowright\rangle$  in terms of  $|"0"\rangle$  and  $|"1"\rangle$ . None of  $|"+\rangle$ ,  $|"- \rangle$ ,  $|“\circlearrowleft\rangle$  or  $|“\circlearrowright\rangle$  are 100%  $|"0"\rangle$  or 100%  $|"1"\rangle$ ; they are all a mixture of each.  $|"0"\rangle$  and  $|"1"\rangle$  can also be written as a combination of other qubits. We can write both as a combination of  $|"+\rangle$ ,  $|"- \rangle$  or a combination of  $|“\circlearrowleft\rangle$  and  $|“\circlearrowright\rangle$ , for example:

$$|“0”\rangle = \frac{|“+\rangle + |“-\rangle}{\sqrt{2}} = \frac{|“\circlearrowleft\rangle + |“\circlearrowright\rangle}{\sqrt{2}}$$

And for  $|"1"\rangle$  we can write these combinations as follows:

$$|“0”\rangle = \frac{|“+\rangle + |“-\rangle}{\sqrt{2}} = \frac{|“\circlearrowleft\rangle + |“\circlearrowright\rangle}{\sqrt{2}}$$



*Two or more qubits can always be added together to produce a valid qubit. Any qubit can be represented as the sum of two or more other qubits. This is the principle of quantum superposition for qubits.*

In the next chapter, we'll see examples of superposition acting on multiple qubits, known as quantum states, at once.

# Quantum measurement for qubits

Quantum measurement is a two-step process. First, we choose what basis we are performing the measurement in, and finally we choose what we are measuring and perform the measurement in that basis. Measurement of a qubit will always give us a state in the basis.

If we measure any qubit in the  $|0\rangle$  and  $|1\rangle$  basis, we will always get either  $|0\rangle$  or  $|1\rangle$ . If we measure any qubit in the  $|+\rangle$  and  $|-\rangle$  basis, we will always get either  $|+\rangle$  or  $|-\rangle$ . And if we measure any qubit in the  $|(\circ)\rangle$  and  $|(\circ)\rangle$  basis, we will always get, you guessed it, either  $|(\circ)\rangle$  or  $|(\circ)\rangle$ . There are countless more bases to choose from (any diameter on the Bloch sphere could be a basis, for example), but these are the only ones we'll go through.

If we choose to measure a qubit in a basis that happens to be one of the basis qubits, we'd always get the same thing. For example, if we measure a qubit  $|0\rangle$  in the  $|0\rangle$  and  $|1\rangle$  basis, we will get  $|0\rangle$  100% of the time. Likewise, if we measure a qubit  $|1\rangle$  in the  $|0\rangle$  and  $|1\rangle$  basis, we will get  $|1\rangle$  100% of the time.

But when we choose to measure a qubit that isn't in the basis, some of the fun of quantum mechanics comes in. For example, let's try measuring  $|+\rangle$  in the  $|0\rangle$  and  $|1\rangle$  basis. Remember that  $|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$ , which happens to be using our Python code from before, the same

as `zero_to_one_qubit(50,50)`, that is,  $|+\rangle$  is 50%  $|0\rangle$  and 50%  $|1\rangle$ . What does that mean? Well, if we have 100  $|+\rangle$  qubits we have prepared, we'd expect if we did 100 measurements on them in the  $|0\rangle$  and  $|1\rangle$  basis, we would get  $|0\rangle$  about 50 times (50%) and  $|1\rangle$  about 50 times (50%). It isn't always going to be exactly 50% of each, but the more qubits we measure, the closer it should get to 50% in the final statistics.

 *Quantum physics means we can't copy a qubit, but you can copy the process that generated the qubit. This is called **identical preparation**. In this, we can generate a bunch of different qubits with the same properties, for example, hundreds of  $|+\rangle$  qubits. Because of their identical preparation, we can then perform the same measurement in our chosen basis on each of those qubits separately, to get an idea as to what  $|+\rangle$  looks like in that basis.*

To perform the measurement of a qubit relative to a basis, we determine what percentage of each basis qubit the qubit we want to measure in is. We can do this in words/algebra, in code, or simply by performing the measurement on many identically prepared qubits and seeing what percentages we get out.

One final note about measurement: once you have chosen to measure a qubit in a basis once, that measurement will give you a qubit in the basis set of qubits. If you repeat the measurement over and over, you will always get the same qubit. So, if we measured a single  $|+\rangle$  qubit, and got a  $|1\rangle$  as our measurement result, we'd always get a  $|1\rangle$  when remeasuring, no matter how many times we perform the measurement. If we knew nothing about  $|+\rangle$  qubits and we only had the chance to perform that one measurement, we wouldn't have any idea that a  $|+\rangle$  qubit is 50%  $|0\rangle$  and 50%  $|1\rangle$ . To get any idea, we'd have to have a bunch of measurements on different  $|+\rangle$  qubits identically prepared.

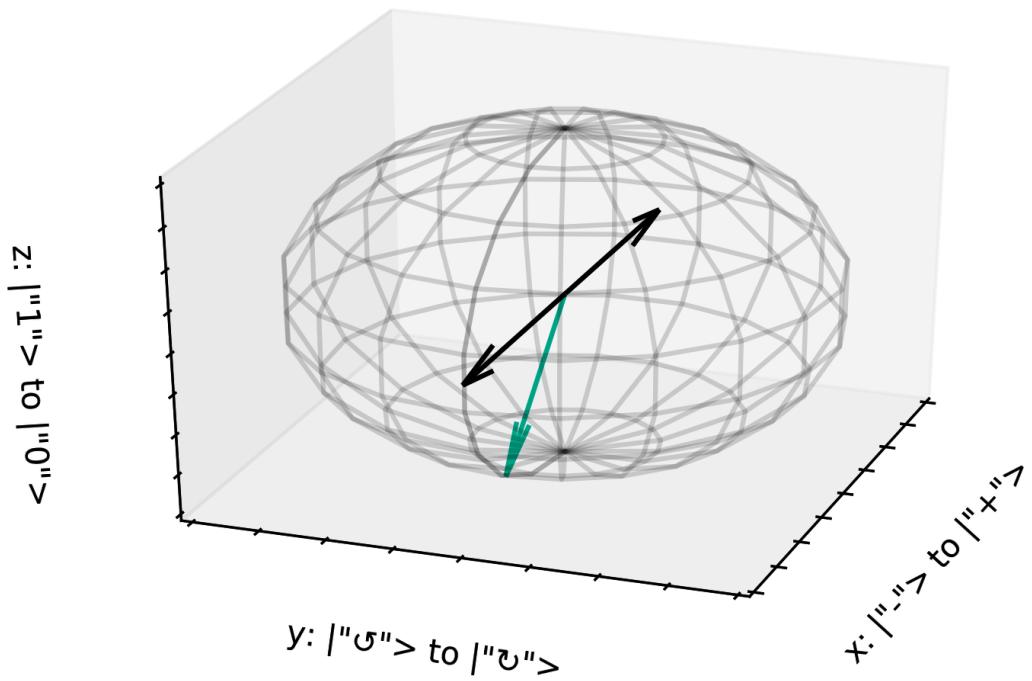
# Measurement of a single qubit on the Bloch sphere

We can visualize the measurement process on the Bloch sphere. First, we choose our basis. Visualize this by drawing a black line through the basis qubits. Here, I've chosen the  $|+\rangle$  and  $|-\rangle$  to be the basis, so I've made them black. Next, draw the qubit you want to measure in another color. Here, I've drawn the qubit that was 10%  $|0\rangle$  and 90%  $|1\rangle$  from earlier in the chapter, also called `ten_ninety_qubit` in code:

$$\begin{aligned} |“fifty\_fifty”\rangle &= \\ \sqrt{0.5}|“0”\rangle + \sqrt{0.5}|“1”\rangle &= \\ 0.70710678|“0”\rangle + 0.70710678|“1”\rangle \end{aligned}$$

Since we are now measuring in the  $|+\rangle$  and  $|-\rangle$  basis, we are asking what percentage  $|+\rangle$  and what percentage  $|-\rangle$  it is.

Let's look at our figure to have a guess:



Here, we can see the turquoise is almost at the  $|+\rangle$  qubit and is pretty far from the  $|-\rangle$  qubit. So we know, just from looking, that measuring identically prepared *"ten\_ninety"* qubits in the  $|+\rangle$  and  $|-\rangle$  basis will return the  $|+\rangle$  qubit much more often than the  $|-\rangle$  qubit.

Algebra or code can give us the exact percentage; you can compute this as part of the chapter questions.

# Summary

A qubit is a quantum piece of information used in quantum computing that represents one possible combination of two values. Manipulating qubits of information is what powers many modern quantum computers, including IBM QX. Any quantum computing physics we want to talk about can just as well be written by math or by code on a classic computer. In this chapter, we simulated a single qubit in Python code to aid our understanding of qubits. Quantum physics allows us to store a qubit of information in just one physical entity. The Bloch sphere is a three-dimensional sphere that has unit radius. Any qubit can be represented as a line from the center of the sphere to a point on the sphere, with an arrow at the end. Superposition of single qubits is two or more qubits added together to produce another qubit. Quantum measurement is a two-step process. First, we choose what basis we are performing the measurement in, and then we choose what we are measuring and perform the measurement in that basis.

This chapter gave an overview of a single qubit. In the next chapter, we will move on to multiple qubits.

# Questions

1. Try the `zero_to_one_qubit` function out with different percentages. What do you get for 100% `zero_qubit`? For 100% `one_qubit`?
2. Write a function to go from the qubit back to your original percentages, and call it `qubit_to_percentages`.
3. Using the definition of  $|“\circlearrowleft\rangle$  in words/algebra given in the chapter, rewrite `clockwisearrow_qubit` to be a sum of `zero_qubit` and `one_qubit`.
4. Use `get_bloch_coordinates` to check the coordinates of all the basis qubits we have learned in this chapter. Verify that they match the diagram.
5. Use `plot_bloch` and `zero_to_one_qubit` to plot a qubit that is 50%  $|“0\rangle$  and 50%  $|“1\rangle$ . Notice that you only see one of the four possible options with the code in this chapter. This is because whenever you take a square root, it could be either a positive or a negative value (for example, 4 is both  $2^2$  and  $-2^2$  so  $\sqrt{4} = \pm 2$ ). Our existing functions, `zero_to_one_qubit` and `qubit`, only consider the case where the square roots are both positive values, when in fact there are four possibilities: positive/positive, positive/negative, negative/positive and negative/negative. Change the code to take in an argument indicating which of the possibilities you would like. Now, plot all possible qubits that are 50%  $|“0\rangle$  and 50%  $|“1\rangle$  using your new code.
6. Write  $|“\circlearrowleft\rangle$  and  $|“\circlearrowright\rangle$  as a superposition between  $|“+\rangle$ ,  $|“-\rangle$  either in words/algebra or in code.
7. Measuring 1,000 identically prepared  $|“ten\_ninety\rangle$  qubits in the  $|“+\rangle$  and  $|“-\rangle$  basis will return  $|“+\rangle$  approximately what

percentage of the time, and /"-> what percentage of the time?

# **Quantum States, Quantum Registers, and Measurement**

We have learned about the quantum version of classic bits, qubits. In this chapter, we will learn about the quantum version of classic registers, quantum registers, which hold quantum states. The chapter will provide code to simulate quantum states and registers in Python. We discuss separable states and entanglement, which leads to a focus on quantum measurement and a Python implementation of quantum measurement for multiple, possibly entangled, qubits. The chapter ends with a description of decoherence and parameters that quantify decoherence, a key figure of merit for quantum computing.

The following topics will be covered in this chapter:

- Quantum states and registers
- Separable states
- Entanglement
- Quantum measurement and entanglement
- Decoherence,  $T_1$ , and  $T_2$

# Technical requirements

Code for this chapter is available in a Jupyter Notebook under <https://github.com/PacktPublishing/Mastering-Quantum-Computing-with-IBM-QX> in the Chapter03 folder.

# Quantum states and registers

Quantum states are groupings of one or more qubits that are physically related. In this section, you will learn to implement a simulation of quantum states in Python and learn about the concept of a quantum register.

Each of the qubits we have worked with so far is a quantum state. However, a quantum state can also be composed of more than one qubit. In a single quantum state, each of the qubits that compose it can be entangled, that is related in a way particular to quantum mechanics. To make use of the power of quantum computation, we will in general want more than one qubit.

A classic register holds any arbitrary number,  $n$ , bits. A quantum register, the analog of the classic register, can hold any superposition of  $n$ -qubit states. So, while an  $n$ -bit classic register can store any one of  $2^n$  possible numbers, it can store just one at a time. An  $n$ -qubit register, on the other hand, can store any combination of  $2^n$  numbers.

In a classic  $n$ -bit register, we can initialize each bit to 0 or 1. For example, to represent the base 10 number 19 in a classic 5-bit register, we can set its elements to 10011. For  $n$  qubits, to create an analogous state, we prepare the state  $|"10011"> = |"1"> \otimes |"0"> \otimes |"0"> \otimes |"1"> \otimes |"1">$ . Here,  $\otimes$  is a mathematical function, which is given in detail in the appendix, and called `kron` in `numpy`. For now, know that it serves the function of binding the two states on either side of it together, so that they can be discussed together.

Given a list of qubits, we can bind them all together by using this function. Here is an example of creating a quantum state from a list of qubits:

```
| from functools import reduce
| def create_quantum_state(qubits):
|     return reduce(lambda x,y:np.kron(x,y),qubits)
```

In this function, if there's just one qubit, the function returns. If there is more than one qubit, the function starts with the first qubit, and applies the  $\otimes$  operator `np.kron` together with the second qubit. If there is a third qubit, the  $\otimes$  operator is again applied to the result so far. This process continues until all the qubits have been  $\otimes$  together. The `reduce` operation makes sure that this is handled effectively. If you aren't familiar with the `reduce` operation in Python, it is a functional style of programming and serves to produce a rolling computation on pairs of elements of a list. For example, if we have a list `a`, then `sum(a)` would produce a sum of all the elements of the list. Using the functional style of programming, `reduce(lambda x,y:x+y, range(a))` would produce the same result.

Now, for our example of `|"10011">`, we can create this state with the following:

```
| zero_qubit=np.matrix('1; 0') # this qubit is |"0">
| one_qubit=np.matrix('0; 1') # this qubit is |"1">
| five_qubit_register=create_quantum_state([one_qubit,zero_qubit,
| ,one_qubit,one_qubit])
```

An  $n$ -qubit quantum register in simulation is any abstraction, say the Python variable `five_qubit_register` in our example, which has the ability to store an  $n$ -qubit state. Physically, on a real quantum computer, an  $n$ -qubit register is a physical system capable of storing an  $n$ -qubit state.

If we try out the `create_quantum_state` function on a variety of different numbers of qubits, we can see the result is much bigger than the original number of qubits. For example, `five_qubit_register.size` is equal to `32`. One with four qubits would be equal to `16` in size, one with three to `8`, one with two to `4`, and one with one to `2`. Essentially, the size of the  $n$ -qubit register will be  $2^n$ . So, given the size, we can discover the number of qubits it must be able to hold by taking the logarithm base 2 of the size. A function to do this is as follows:

```
| def get_nqubits_quantum_state(state):  
|     return int(log(state.size,2))
```

# Separable states

In this section, you will implement an algorithm to try to separate quantum states into the qubits they were constructed from, see that the algorithm is incomplete, discuss ways to improve the algorithm, and introduce the notion that some quantum states cannot be separated in this manner.

With the `create_quantum_state` function, any time we put in a list of qubits, we get a single state out, representing them together. We could go in the other direction, and write a function called `get_qubits_from_state`, which can, given a state, return the original qubits from which it was generated.

There are some states that are impossible to separate, no matter how good we make our function that determines whether it is separable. The mathematics works out that for some states, no matter what, we can't describe the parts of the state individually. One example of such a state is the following state, which is composed of two qubits:

```
| non_separable_state_00_plus_11=1/np.sqrt(2)*  
| (create_quantum_state([zero_qubit,zero_qubit])+create_quantum_state([one_qubit,one_qubit]))
```

The mathematics work out that there's no way to divide the two qubits so that we can talk about them separately. Let's call the two qubits  $a$  and  $b$ ; there is no way to separate  $a$  and  $b$  into states so that `non_separable_state_00_plus_11` is equal to  $a \otimes b$ . This is called a non-separable state, and these non-separable states are crucial to the power of quantum computing, as we will learn in the next section of this chapter. A non-separable state is called an **entangled** state.

To make a stab at separating a state, we will assume that the state was created from qubits that are one of the basis states we went over in [Chapter 2](#), *Qubits*,  $|0\rangle$ ,  $|1\rangle$ ,  $|+\rangle$ ,  $|-\rangle$ ,  $|(\circlearrowleft)\rangle$ , or  $|(\circlearrowright)\rangle$ .

Now, we can define a function that will return which possible combination of basis states that the state we are interested in is composed of, and, if that doesn't apply, return `None`:

```
def get_qubits_from_state(state):
    basis_states=[zero_qubit,one_qubit,plus_qubit,minus_qubit,
                  clockwisearrow_qubit,counterclockwisearrow_qubit]
    for separated_state in itertools.product(basis_states,
                                              repeat=get_nqubits_quantum_state(state)):
        candidate_state=create_quantum_state(separated_state)
        if np.allclose(candidate_state,state):
            return separated_state
```

This function computes all possible permutations of the basis states that are of our desired length. It then computes what the entangled quantum state would be for each of those permutations. Finally, it checks whether that candidate is equal to the state in question. If it is, it returns the permutation. If no such state is found, it returns `None`. Our simplified code won't catch all possible separations that could ever exist, though.

One example that our `get_qubits_from_state` would fail to separate, but is in fact possible to separate, would be something as simple as a qubit from the previous chapter, that is, 10%  $|0\rangle$  and 90%  $|1\rangle$ , defined as

`ten_ninety_qubit=np.sqrt(0.1)*zero_qubit+np.sqrt(0.9)*one_qubit.` Not only would the `get_qubits_from_state(ten_ninety_qubit)` function return `None`, but also

`get_qubits_from_state(create_quantum_state([ten_ninety_qubit,ten_ninety_qubit]))` would return `None`. It's clear that, since `get_qubits_from_state` is simply supposed to tell us what list went into the `create_quantum_state` function, it isn't a complete function. It didn't work in this case. We could switch the `get_qubits_from_state`

function to consider the possibility that one of the original states is `ten_ninety_qubit`. I'll call the function `get_qubits_from_state_with_additional_guess` as follows:

```
def get_qubits_from_state_with_additional_guess(state):
    basis_states=[zero_qubit,one_qubit,plus_qubit,minus_qubit,
                  clockwisearrow_qubit,counterclockwisearrow_qubit,
                  ten_ninety_qubit]
    for separated_state in itertools.product(basis_states,
                                              repeat=get_nqubits_quantum_state(state)):
        candidate_state=create_quantum_state(separated_state)
        if np.allclose(candidate_state,state):
            return separated_state
```

Now things work!

`get_qubits_from_state_with_additional_guess(ten_ninety_qubit)` returns `ten_ninety_qubit` and `get_qubits_from_state_with_additional_guess(create_quantum_state([ten_ninety_qubit,ten_ninety_qubit]))` returns `[ten_ninety_qubit,ten_ninety_qubit]`. But we'd need a lot of guesses to be able to consider all possible scenarios with this simple algorithm. We would have to modify the function to consider all possible permutations of all possible states, not just those permutations of the few basis states we have studied. That would be a tough order! It turns out this problem is something called **NP-hard**, which essentially means it is a highly difficult problem to solve classically. We won't solve it in this text, and will only separate states that are composed of one of the basis states we have chosen to work with, as in `get_qubits_from_state`. The important thing to take away is that any state that can in principle be reduced to a  $\otimes$  of single qubits is called a separable state, although sometimes finding whether this separation is possible and the separation itself can be hard to compute classically.

# Entanglement

In this section, you'll learn what entanglement is and how to simulate it. We'll also discuss the power of entanglement, and why it is such an important resource in quantum computation. Entanglement is nothing more than a non-separable state. The neat thing is that, even though we can't describe any of the component qubits separate from the others, physically in the real world, we can drag the individual qubits apart. This doesn't change the fact that none of the qubits can be considered without considering the system as a whole.

For example, let's consider our `non_separable_state_00_plus_11`, which we defined in the previous section. Writing out what this is in words/algebra, this Python code equates to the following:

$$|\text{"non\_separable\_state"}\rangle = \frac{1}{\sqrt{2}}(|\text{"00"}\rangle + |\text{"11"}\rangle)$$

Now, imagine I take one of the qubits, and you the reader take the other, and then we move them apart by a large physical distance. If you perform a measurement in the  $|\text{"0"}\rangle$  and  $|\text{"1"}\rangle$  basis, you will get  $|\text{"0"}\rangle$  with 50% probability and  $|\text{"1"}\rangle$  with 50% probability. Now, imagine you get  $|\text{"0"}\rangle$ ; what will happen if I measure my qubit afterward? I will also get  $|\text{"0"}\rangle$  with 100% probability. Imagine you get  $|\text{"1"}\rangle$  as the result of your measurement. Well, then I will get  $|\text{"1"}\rangle$  with 100% probability. There is no scenario where you get a qubit in the  $|\text{"0"}\rangle$  and  $|\text{"1"}\rangle$  basis and I get the opposite. Now, what if, instead, I were first to measure my qubit instead of you measuring your qubit first. Then, the roles would be reversed, and I would get  $|\text{"0"}\rangle$ .

with 50% probability and  $|1\rangle$  with 50% probability, and when you measured, you would get whatever I got with 100% probability. As we discussed in the last chapter, after you perform a measurement on a qubit, any subsequent measurements will return the same result with 100% probability. Moreover, there's no way to copy or clone a qubit, so you can't try to extract information by copying the same qubit several times and performing measurements on each of the copies. This inability to make copies is one of the wonderful and different things about quantum mechanics, the underlying physics which powers quantum computing.

One natural question to ask would be this: couldn't we use this to send information at an infinite speed? The answer is no, we cannot use this to send information faster than the speed of light. Imagine if we took each of our qubits and then sailed away in spaceships to be separated by many light years, so that the speed of light would take 10 years to travel from me to you. In the absence of any attempts to change the qubits, whoever measures first will always get  $|0\rangle$  with 50% probability and  $|1\rangle$  with 50% probability.

Imagine I keep the qubits as is, and that there's some way to be sure that I perform a measurement, and then you perform a measurement in that order; I couldn't send you any information. Because my measurement had a 50/50 chance of being either 0 or 1, so would yours. I wouldn't be able to determine which of the two you got, so I would be unable to send you any information.

Now, imagine I try to manipulate my qubit to force the measurement into definitely being 0 or definitely being 1. We can see mathematically that this would break the dependence between our two qubits, so that then on my end I could measure a 1 100% of the time, and on your end

you'd have a 50/50 chance of either 1 or 0. Alternatively on my end, I could measure a 0 100% of the time, and on your end you'd have a 50/50 chance of either 1 or 0. So, no information is sent beyond randomness.

That's all assuming we manage to agree on an order of measurements, because whoever measures first will always get randomness. But there's not even a good way to do that. Even if we agree on an order of measurement, and have each some sort of accurate clock to attempt to adhere to the order, there'd never be a way for a person who is to attempt to receive information by performing a measurement, that is the person to go second, to be sure that the other person had performed their measurement already. Since they only have one shot at the measurement, they get either a 0 or a 1, and that's it. Does that mean that I, on the other end, have already measured and also gotten the same result? Or does that mean that you are the first person to go, and either result was equally likely? Unless we could signal to confirm who went first, there's no way to tell. And signaling will take the speed of light.

So unfortunately, although our measurements are correlated, there's no way for us to use those correlations to send messages to each other.

# Quantum measurement and entanglement

In this section, we will discuss quantum measurement for multiple qubits and present an algorithm to simulate quantum measurement in Python.

In the previous chapter, we discussed quantum measurement for a single qubit.



*Quantum measurement is a two-step process. First, we choose what basis we are performing the measurement in, and finally we choose what we are measuring and perform the measurement in that basis. The measurement of a qubit will always give us a state in the basis.*

Quantum measurement for multiple qubits is no different. After measuring five qubits in the  $|0\rangle$  and  $|1\rangle$  basis, we will get five qubits, each either  $|0\rangle$  or  $|1\rangle$ . So, there are  $2^5$  possible results of a measurement. The probability of each result depends on how the five qubits are entangled. After we measure the qubits, they are no longer entangled and we will always get the same result.

# An algorithm for simulating quantum measurement in Python

This subsection provides an algorithm for simulating quantum measurement in Python. First, we consider the steps needed to simulate quantum measurement on a classic computer. These are as follows:

1. To implement an algorithm for quantum measurement, we will also choose a basis. In this example, I chose the  $|0\rangle$  and  $|1\rangle$  basis.
2. Next, we need to figure out how many qubits the state represents. For a state of  $n$  qubits, we need  $2^n$  numbers to represent it. Essentially, each of these numbers helps codify the probability of each of the  $2^n$  possibilities of an  $n$  qubit state. So, given a state, we can compute how many numbers it contains then take the logarithm, base 2, to recover  $n$ , the number of qubits the state represents.
3. To compute the probability of each of the  $2^n$  possibilities of an  $n$  qubit state, we will take the number that represents it in the state and square it to get the probability. Because the number is complex, we multiply the number by its complex conjugate to get the value of the number squared.
4. We simulate the randomness of the outcome by picking a random number.
5. For  $n$  qubits, there are  $2^n$  different possible measurements, one for each possible binary value. For example, for 3 qubits, there are  $2^3=8$  possibilities for a

measurement: 000, 001, 010, 011, 111, 101, 110, 100.

We go through each of these  $2^n$  possibilities.

6. The sum of the probabilities of all states will add to one. To find the state "measured" in our simulation, we check each state in order. If the first state for which the random number chosen is less than the sum of the probabilities of the states we have checked so far, we return a string representing that state.

The following algorithm implements what we have described:

```
from math import log
from random import random
import itertools
def measure_in_01_basis(state): #1.
    n_qubits=int(log(state.shape[0],2)) # 2.
    probabilities=[(coeff*coeff.conjugate())
                   .real for coeff in state.flat] # 3.
    rand=random() #4.
    for idx,state_desc in enumerate([''.join(map(str,state_desc))
                                      for state_desc in itertools.product(
                                          [0, 1], repeat=n_qubits)]): #5.
        if rand < sum(probabilities[0:(idx+1)]): # 6.
            return '|"%s">' % state_desc #7.
```

Now, let's try it out on a few

states. `measure_in_01_basis(create_quantum_state([one_qubit,zero_qubit,zero_qubit,one_qubit,one_qubit]))` will always return the string `"/"10011">`, as we have constructed a state of basis states where the probability of each qubit returning that basis state is 100%.

Measurement gets more interesting when we deal with multiple qubits that are entangled. Let's try the measurement out on the following state, which we'll call

`bell_state_phi_plus`:

$$\text{Bell state } \Phi^+ = \frac{|“00 ”\rangle + |“11 ”\rangle}{\sqrt{2}}$$

After writing some Python code to prepare this same state 10 times in a row, and then measure it, we have the following:

```
| for i in range(10):
|   bell_state_phi_plus=
|     (create_quantum_state([zero_qubit,zero_qubit])+create_quantum_state([one_
|       qubit,one_qubit]))/np.sqrt(2)
|   print(measure_in_01_basis(bell_state_phi_plus))
```

When we ran the preceding code, we got the following output:

```
| "11">
| "00">
| "00">
| "00">
| "11">
| "00">
| "11">
| "00">
| "00">
| "11">
```

However, each time you run the code, there are new random numbers chosen as part of the measurement, so it will print different results. My results matched our expectations, which were that we would get  $|"00">$  50% of the time and  $|"11">$  50% of the time. What are your results?

Now, this code is missing one piece inherent to quantum mechanics: after we measure a state once, we need to guarantee that every subsequent measurement produces the same results. For this, we imagine a new function, `measure_in_01_basis_collapse`, which could be written to ensure that property.

The function `measure_in_01_basis_collapse`, which you will have the chance to implement in the chapter exercises, would need to ensure that the following code would always print  $|"00">$  twice in a row, or  $|"11">$  twice in a row:

```

bell_state_phi_plus=
(create_quantum_state([zero_qubit,zero_qubit])+create_quantum_state([one_
qubit,one_qubit]))/np.sqrt(2)
print(measure_in_01_basis_collapse(bell_state_phi_plus))
print(measure_in_01_basis_collapse(bell_state_phi_plus))

```

Our current function doesn't guarantee that, but the sixth question at the end of this chapter will give you a chance to implement a function that does.

Note that if we prepare multiple `bell_state_phi_plus` then the measurements of one do not affect the other. So the following code:

```

bell_state_phi_plus=(create_quantum_state([zero_qubit,zero_qubit])
+create_quantum_state(
[one_qubit,one_qubit]))/np.sqrt(2)
print(measure_in_01_basis_collapse(bell_state_phi_plus))
print(measure_in_01_basis_collapse(bell_state_phi_plus))
bell_state_phi_plus=(create_quantum_state([zero_qubit,zero_qubit])
+create_quantum_state(
[one_qubit,one_qubit]))/np.sqrt(2)
print(measure_in_01_basis_collapse(bell_state_phi_plus))
print(measure_in_01_basis_collapse(bell_state_phi_plus))

```

This produces the following, for example:

```

|"00">
|"00">
|"11">
|"11">

```

# **Decoherence, T1, and T2**

To exploit the power of quantum computing, we need to exploit and control the power of entanglement. We need to choose which qubits are entangled with which other qubits, and help ensure they are entangled with nothing else. As soon as a qubit becomes entangled with something unexpected that is not part of the desired computation, whether that is another qubit or something in the environment, we will get noise and potentially computational error. One way to reduce this possibility is to make sure our computation is physically isolated from the outside, and in quantum computation this is done by making the system physically cold and by shielding it heavily from outside influences.

# Decoherence

We can talk about how well a quantum system is isolated by discussing the concepts of coherence and decoherence.

Coherence is a property of waves, and if waves are coherent they can in some senses "work together." Coherence can be any correlation between the physical quantities of a wave or a group of waves; this correlation proves that they are "working together" as it shows there is a relationship between the two physical quantities. This is important in quantum computing as quantum mechanics helps us understand that the physical nature of a qubit can also be represented by a wave, and so, if we want different qubits to "work together," they will need to be coherent.

Coherent waves have the same frequency and have a constant phase difference. Imagine two waves in the ocean. Two coherent waves have the same frequency, so they would have the same spacing between their peaks and troughs. Two coherent waves don't have to have the same positions of their peaks and troughs, however; that isn't part of the definition. So, let's imagine our two waves have peaks and troughs separated by about 3 feet, or 1 meter. Any two waves that don't have the same positions of their peaks and troughs have a phase difference. Our two waves have a phase difference. However, since the separation of the peaks and troughs is constant, that is, it doesn't ever increase or decrease, we are dealing with a constant phase difference. In this example, we have two waves with the same frequency and a constant phase difference, so we have two coherent waves. These waves can "work together." In this case, wherever both waves are peaking, we get a bigger peak than either alone; wherever both

waves have a trough, we get a bigger trough than either alone; and wherever one wave has a peak and one has a trough, the peak is reduced by the amount of the trough.

If we start out with coherent qubits in a simulation, they will stay that way forever. If we are working on a real quantum computer, they will not stay coherent. Qubits interact with the environment and this process causes the correlations they had due to coherence, which we need to represent the results of our computation, to be lost in this process. In other words the information from our computation degrades or is lost over time due to physical interactions with the environment. Decoherence is the loss of information due to noise from the environment. Physically, this can mean, for example, a state  $|1\rangle$  becoming a state  $|0\rangle$ . In that case, decoherence has caused us to lose the information about the state. If we performed a computation and the result was  $|1\rangle$  but decoherence has occurred, we will instead get a  $|0\rangle$ . Not good news!

If we recall the Bloch sphere, a coherent qubit will lie on the Bloch sphere, whereas a qubit that is experiencing decoherence will lie within the Bloch sphere. Every qubit lying on the Bloch sphere when visualized has a line length of one, so those qubits that are experiencing decoherence have a line length when visualized on the Bloch sphere of less than one.

# T<sub>1</sub> and T<sub>2</sub>

In our Python simulations of quantum computing, we have not simulated decoherence, but ignoring decoherence is not an option for real quantum computers. Every quantum computer experiences decoherence, but quantum computers that are successful at delaying and minimizing decoherence perform better. That's why, when discussing a quantum computer and its ability to do computation, we need to discuss how well it does at preventing decoherence. To quantify that, the parameters  $T_1$  and  $T_2$  are particularly important:

- $T_1$  helps to quantify how quickly the qubits experience energy loss due to environmental interaction (energy loss would result in a change in frequency, which would make coherent qubits experience decoherence).
- $T_2$  helps to quantify how quickly the qubits experience a phase change due to interaction with the environment, again a cause of decoherence.

**Energy relaxation** is the loss of energy from the system, for example the process of a state with more energy decaying into another state with less energy.  $T_1$  measures, for example, the time for a state  $|1\rangle$  with higher energy to become a state  $|0\rangle$  with lower energy. Energy relaxation will always happen in a real quantum computer, and this process happens via exponential decay from the more energetic state to the less energetic state. That is, we initially have 100% probability of being in state  $|1\rangle$  in this example; after some time  $t$  the probability of being in state  $|1\rangle$  will have decreased exponentially. So after zero time, we should get 100% probability of being in state  $|1\rangle$ , but

after time  $t$  the probability will decrease to a value  $e^{-t/T_1}$ , where  $T_1$  is some constant. In words/algebra, this means the following:

$$\text{Probability in state } |“\circlearrowleft”\rangle = e^{-t/T_1}$$

$T_1$  is a measurement of how long energy relaxation takes to occur. Later in the book, we will measure  $T_1$  directly on the IBM QX machine. We will do this by putting a qubit in a more energetic state,  $|“1”\rangle$ , then waiting for a period of time  $t$ , then measuring the probability of the qubit still being in the energetic state  $|“1”\rangle$ . Then, we'll have all the numbers in our equation except for  $T_1$ , and we can solve the equation for  $T_1$ .

Let's do one example here in Python, and you will do another example in the questions in this chapter. Let's imagine we have a state  $|“1”\rangle$ , and after 0.1 milliseconds (0.0001 seconds) the probability of our state still being  $|“1”\rangle$  has dropped to 10% ( $10/100 = 0.1$ ). What is the value of  $T_1$  for this computer? Let's see:

```
probability_state_one_after_point1millisecond=0.1
t=0.0001
# probability_state_one_after_point1millisecond = np.e**(-t/T1) so T1 = -t/np.log(probability_state_one_after_point1millisecond)
T1=-t/np.log(probability_state_one_after_point1millisecond)
print(T1)
```

This gives us a value of  $4.34 \times 10^{-5}$  seconds or 43.4  $\mu\text{s}$  for the value of  $T_1$  for this computer.  $T_2$  is the second important quantum computing measurement. It affects only superposition states, as the decoherence results from the phase difference between two or more qubits that are coherent, that is, are in a superposition. Any environmental disturbance that causes phase changes can cause this sort

of decoherence. Like  $T_1$ ,  $T_2$  is measured in terms of the exponential decay of our expected result over a period of time.

Quantum computing takes time, particularly computations that require many steps. It is exactly the algorithms that require many steps which will enable quantum computing to be practically useful. If, in the middle of the computation, the qubits experience decoherence, the computation is impaired.  $T_1$  and  $T_2$  are measurements that quantify the time it takes for qubits to experience decoherence. The bigger  $T_1$  and  $T_2$  are, the longer it is until qubits experience decoherence, and the more steps our quantum computation can take and still be useful. So, modern quantum computers are always trying to increase  $T_1$  and  $T_2$ , and seeing the value of  $T_1$  and  $T_2$  for a quantum computer you are interested in can help you compare it to other quantum computers and to predict how useful it may be.

# Summary

A quantum state can also be composed of more than one qubit. In a single quantum state, each of the qubits that compose it can be related, that is, entangled. Entanglement is key to the power of quantum computing. The measurement process for a multi-qubit state is the same as the process for a single qubit state.

Coherence is a property of waves, and if waves are coherent, they can in some senses "work together." Qubits interact with the environment, and this process causes the correlations they had due to coherence, which we need to represent the results of our computation, to be lost in this process. We measure  $T_1$  and  $T_2$  to quantify how long a quantum computer can maintain coherence for.

In the next chapter, we will learn about the quantum equivalent of classic gates, quantum gates, which enable us to perform quantum computation by changing the quantum state as we dictate.

# Questions

1. Find at least one other state that is a separable state, but for which `get_qubits_from_state` fails to work.
2. For the state you found in question 1, modify the code in `get_qubits_from_state` to work.
3. You have designed a new quantum computer, and the probability of finding a qubit still in state  $|1\rangle$  after 0.1 milliseconds is 0.7%. What is  $T_1$  for this computer?
4. Quantum computer A has a  $T_1$  of 63  $\mu\text{s}$  and a  $T_2$  of 60  $\mu\text{s}$ , while quantum computer B has a  $T_1$  of 70  $\mu\text{s}$  and a  $T_2$  of 78  $\mu\text{s}$ . Both A and B can operate on the same number of qubits. In the absence of further information, which quantum computer would you prefer?
5. Change the `measure_in_01_basis` algorithm to measure a state in the  $|0\rangle$  and  $|1\rangle$  basis to return the state itself, instead of printing the string (mathematical) description of the state.
6. Create a class or a function that ensures that once a state is measured, any subsequent measurement always returns the same result. This function should follow the specifications of `measure_in_01_basis_collapse` given in this chapter.

# Evolving Quantum States with Quantum Gates

This chapter goes over the quantum gate and provides an analog to classic gates. It gives an overview of the most commonly used gates in quantum computing, I, X, Y, Z, H, S,  $S^\dagger$ , T,  $T^\dagger$ , and CNOT, which form a universal gate set that can be combined to perform any quantum computation. It describes how gates act on states to change them, and why this process is important to both classic and quantum computation. It provides a Python implementation of these commonly used quantum gates, and goes over many examples within Python of these gates being applied to states we have examined so far.

The following topics will be covered in this chapter:

- Gates
- Gates acting on states
- Single qubit gates
- Multi qubit gates

# Technical requirements

Code for this chapter is available in a Jupyter Notebook under <https://github.com/PacktPublishing/Mastering-Quantum-Computing-with-IBM-QX> in Chapter04 folder.

# Gates

Just as a qubit is the quantum version of classic bit, and a quantum register is the quantum version of a classic register, a quantum gate is the quantum version of a classic logic gate.

We have outlined the analogy to the classic  $n$ -bit register, the  $n$ -qubit quantum register for keeping track of quantum data. Here, we will do the same with a classic gate and a quantum gate, which evolve classic and quantum states respectively. In classic computation, a classic gate operates on a classic register to evolve its state. In quantum computation, a quantum gate operates on a quantum register to evolve its state.

# Classical gates

A classic gate evolves or changes a classic state. For example, if we have two classic registers, containing the bits 0 and 1 respectively, an OR classic gate would take these two bits as input and produce the value of 1. Other examples of classic gates include AND, NOT, XOR (exclusive or), and NAND (negative and). Some gates, XOR, AND, and NAND, for example, require two bits as input, and output one bit. Other gates, NOT being an example, take one bit as input, and output one bit. Still other gates take  $m$ -bits as input, and output  $n$ -bits for some choice of integers  $m$  and  $n$ .

It turns out that, with just one type of gate, NAND, we can construct any other type of classic logic gate. Thus, any classic computation can be written to be made of only NAND gates. There are other subsets of classic gates where just those gates would be enough to perform any classic computation. Another example would be that we could perform any classic computation with the AND and NOT gates alone. We'll call a set of gates that is enough to perform any classic computation, a universal gate set. Two such sets we've already given examples of: the NAND gate forms a universal gate set and AND and NOT gates together form another universal gate set.

# Quantum gates

Quantum gates operate on a quantum register to change its state. Some quantum gates operate on one qubit alone. Others operate on two qubits, taking both qubits into consideration. Still other quantum gates operate on more than two qubits. In this book, we will consider quantum gates that operate on either one or two qubits only, because these gates can be expanded via mathematics or code to operate on any sized quantum state on just the one or two qubits of choice. All one-qubit gates can be visualized as taking a qubit lying on the Bloch sphere, and moving its position on the Bloch sphere. It is this movement that will allow us to perform computations, taking an input qubit, processing it by applying quantum gates, and ending up with a potentially different qubit at the end.

The first example of a gate I will give is the identity gate, also known as  $I$ . The identity gate takes in a qubit and outputs the qubit unchanged. If we were to visualize this gate operating on a qubit on the Bloch sphere, it simply wouldn't move after the operation. The identity or  $I$  gate is defined in Python as follows:

```
| identity_gate=np.eye(2,2)
```

Writing out the fact that a gate should be operating on a state is as simple as writing the gate before the state. So, the identity gate operating on state  $|0\rangle$  would be written as  $I|0\rangle$ .

One more thing to note is the fact that the one-or two-qubit gates we consider in this book can be rewritten in code/mathematics to be able to perform their operations on

quantum registers containing states of more than two qubits. For example, we could rewrite the identity gate, which operates on a single qubit state, to operate on the third qubit in a quantum register containing a state of twenty qubits. In general, we can rewrite any one-or two-qubit gates to operate on a register containing a state of arbitrary qubits and perform the operation on just the one or two qubits of choice. Note that the operation is performed then on one or two qubits of choice, but that if these qubits are entangled with other qubits, the effects of the operation will be seen by other qubits.

# Gates acting on states

Simulating any gate's operation in Python is as simple as using the multiplication symbol `*` between the gate and the state it is meant to be operating on. Here is an example of `I/"0">`:

```
| identity_gate*zero_qubit
```

Note that, unlike with multiplying integers or floating point numbers, where  $a * b$  is the same as  $b * a$  (we say that multiplying integers is **commutative**), in the case of states and gates, order of multiplication does matter; that is, it is not commutative. If you try to put the state `/"0">` before the gate `I` in the multiplication, as in `/"0">I`, in Python, it will throw an exception, and in written words/algebra it just doesn't make sense. This is because the mathematical structure of the states and gates are more complicated and nuanced than integers or floats, resulting in multiplication not being commutative. For more details about the mathematical structure of states and gates, see the Appendix.



*With the multiplication of integers or floating point (real) numbers, the order doesn't matter.  $x * y * z$  is the same as  $z * y * x$ . This is called commutative. When we write gates, however, order does matter, and we always perform the multiplication from right to left. So the gates  $XYZ|0\rangle$  are not in general equal to gates  $ZYX|0\rangle$ . In the case of  $XYZ|0\rangle$ , we first apply Z, then Y, then X. In the case of  $ZYX|0\rangle$  we first apply X, then Y, then Z. Quantum gates are associative, just like multiplications of integers or floating point (real) numbers. That means that, as long as we keep the order fixed, it won't be a problem if we group the multiplications however we like. So  $XYZ|0\rangle = (XYZ)|0\rangle = (XY)(Z|0\rangle) = X(YZ|0\rangle)$  for example.*

We know that `I/"0">` should be equal to `/"0">` and that `I/"1">` should be equal to `/"1">`. Let's check that in Python:

```
| print(np.array_equal(zero_qubit,identity_gate*zero_qubit))
| print(np.array_equal(one_qubit,identity_gate*one_qubit))
```

The preceding statements print the following:

| True  
| True

**i** States written as `I|"state">` always appear after the gate in multiplication order. We will not cover it in this book, but as you may see this in your further reading in quantum computation, I will mention that there is a way to transform a state so that it could be put before the gate in multiplication order. Such states are instead written in the form `<"state"|`. Thus, `<"state">I` would be a valid operation, just like `I<"state">` (while the following don't make sense/would throw an exception: `<"state">I` or `I<"state">`). For more details, see the Appendix.

Great! We can also chain many gates together, for example, `////"1">`. The identity gate is easier to chain than other gates; we can do it in our heads. It just leaves the state that follows the chain unchanged, so we could chain as many `I` gates as we liked and get the original state.

While `/"1">` is the same as `////"1">` in our quantum computer simulation, on a physical quantum computer the two might produce very different results. This is because, on a physical quantum computer, we will have environmental noise to contend with, and that noise, as we learned in the previous chapter, after a certain period of time, accumulates so much that the computation is no longer able to be performed reliably.

Because each gate operation takes time, that eats into the time you have before errors accumulate. So, the fewer gate operations an algorithm for a physical quantum computer has, the better it is, as the less noise the result of the final computation will contain.

# Single-qubit gates

In this section, you will learn to implement commonly used quantum gates in Python. We learned that certain sets of gates form a universal gate set that can perform any classic computation. Certain sets of quantum gates form a universal gate set that can perform any quantum computation. I will present one such set of nine gates in this chapter. With just this set, we can perform any quantum computation, as it is a universal gate set. This universal gate set is the same one employed by the physical quantum computer IBM QX. I will call it the IBM QX universal gate set.

In this entire section, we will define gates that sometimes contain  $\sqrt{-1}$ , a type of number known as a complex number, or also known by the value  $i$  or  $j$ . Python has inbuilt support for complex numbers, the complex portion created by putting a  $j$  after any number. Because we will deal with  $i$  so often, here I make a shorthand for it in code, using  $i_$  as the variable name because  $i$  is used so often as a loop variable, I don't want the possibility of confusion. So, throughout this chapter we will define the following:

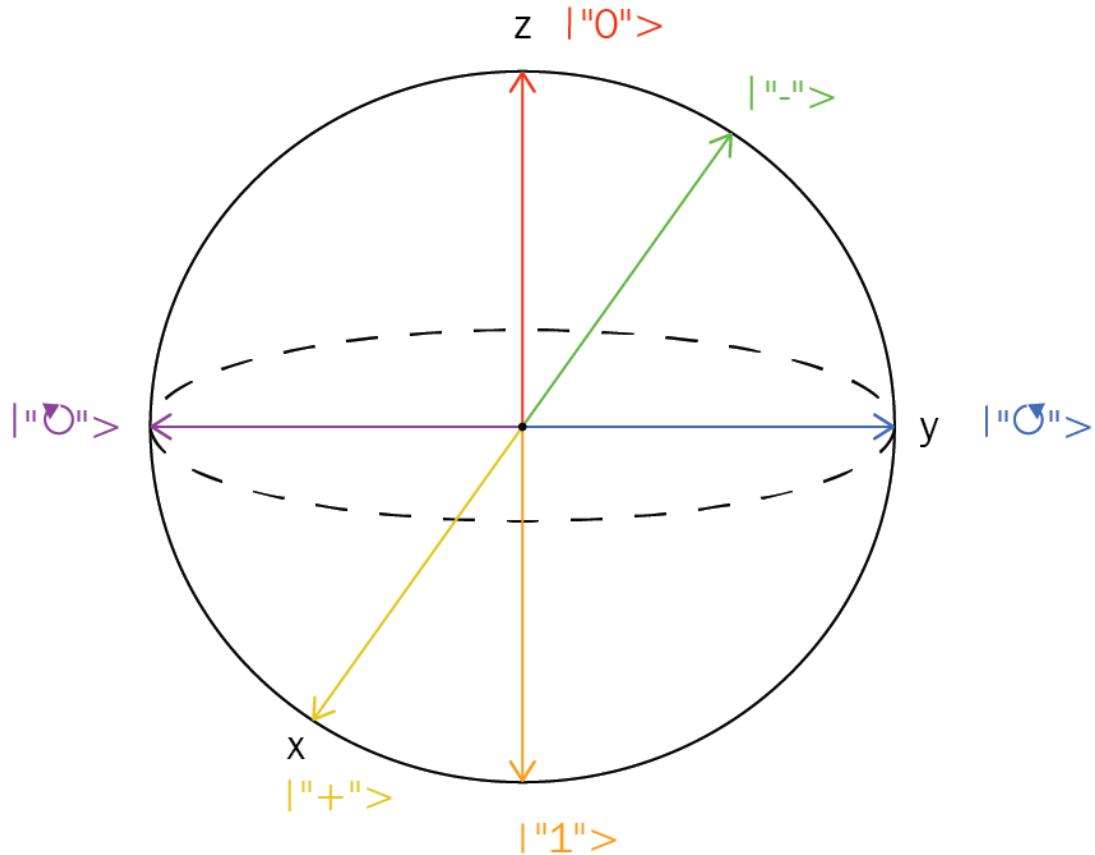
| $i_=1j$

Most of the gates in the IBM QX universal gate set are one-qubit gates. There are three gates, X, Y, and Z, which are called Pauli gates. There is a set of related gates called phase gates: S,  $S^\dagger$ , and a set of related gates called  $\pi/8$  gates, T,  $T^\dagger$ . Finally, there is the Hadamard gate (H). In this section, we will go through these single qubit gates one by one, define them in Python, and see what they do to a qubit on the Bloch sphere. Recall that single qubit gates

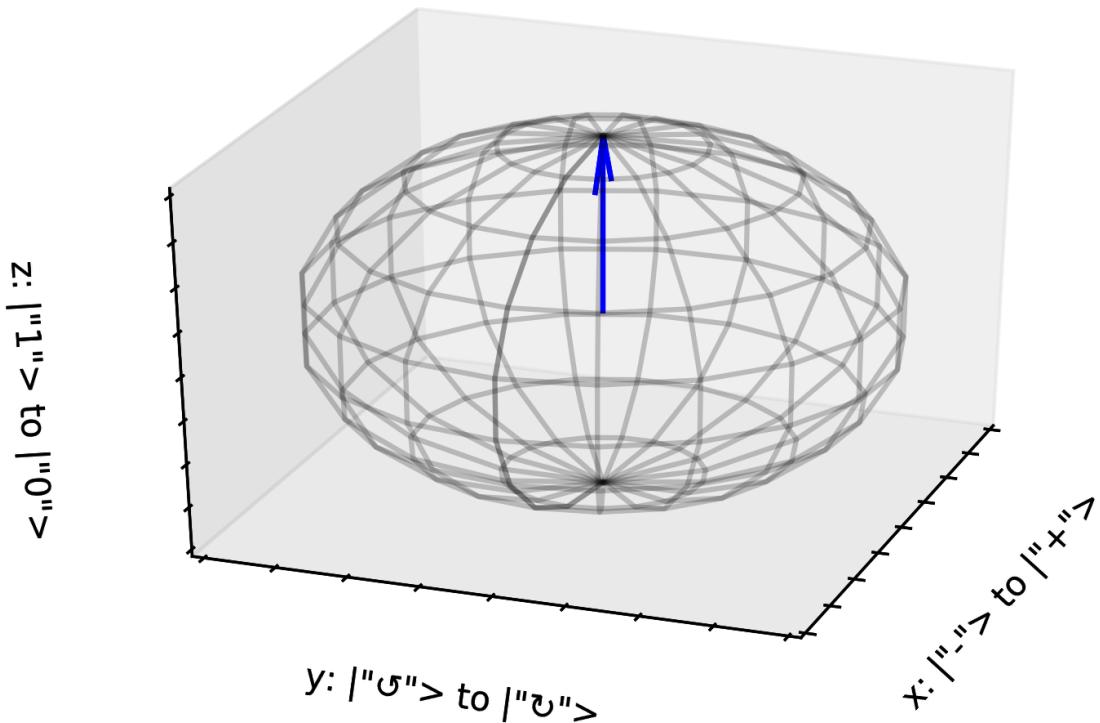
simply move a qubit around on the Bloch sphere in a predictable manner.

For each one-qubit gate, we will visualize its action on the  $|0\rangle$  or the  $|1\rangle$  qubit on the Bloch sphere. Each one-qubit gate will move the qubit around the Bloch sphere. To visualize the gate's action, we will use the `plot_bloch` method defined in [Chapter 2, Qubits](#), as well as the qubit definitions `zero_qubit` and `one_qubit`.

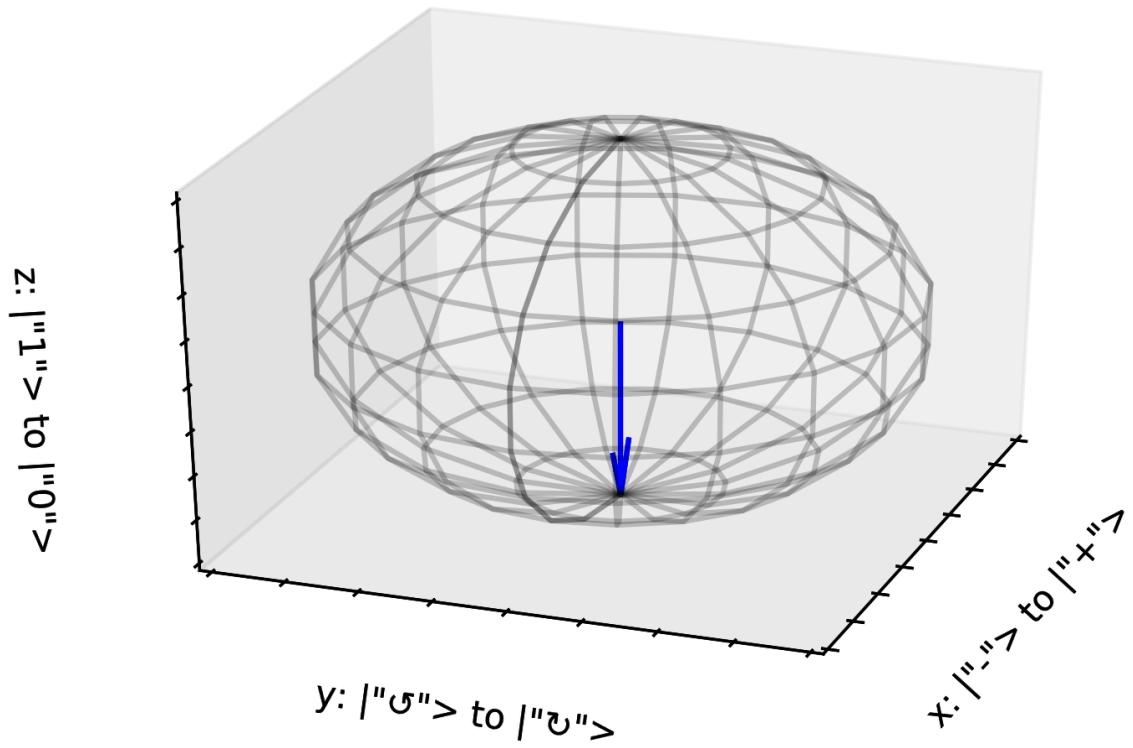
It might be helpful to review the six basis qubits we studied in [Chapter 2, Qubits](#), all the basis qubits we have learned so far,  $|0\rangle$ ,  $|1\rangle$ ,  $|+\rangle$ ,  $|-\rangle$ ,  $|0\rangle$ , and  $|1\rangle$ , and their positions on the Bloch sphere, as we will revisit each of them in this section:



The following Bloch sphere plots show `|"0">` and `|"1">` before we apply any gates, for reference. You can see `|"0">` points up with the `plot_bloch(zero_qubit)` statement as follows:



And `|"1">` points down with the `plot_bloch(one_qubit)` statement as follows:



# Hadamard gate (H)

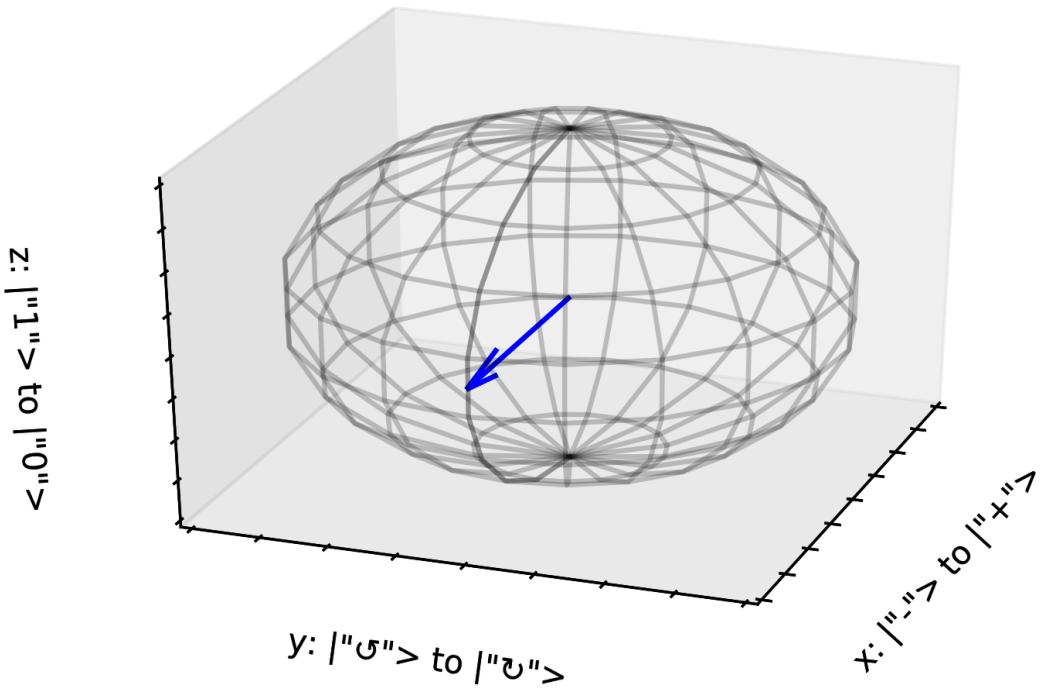
In Python, the Hadamard gate, also known as the H gate is defined as follows:

```
|H=1./np.sqrt(2)*np.matrix('1 1; 1 -1')
```

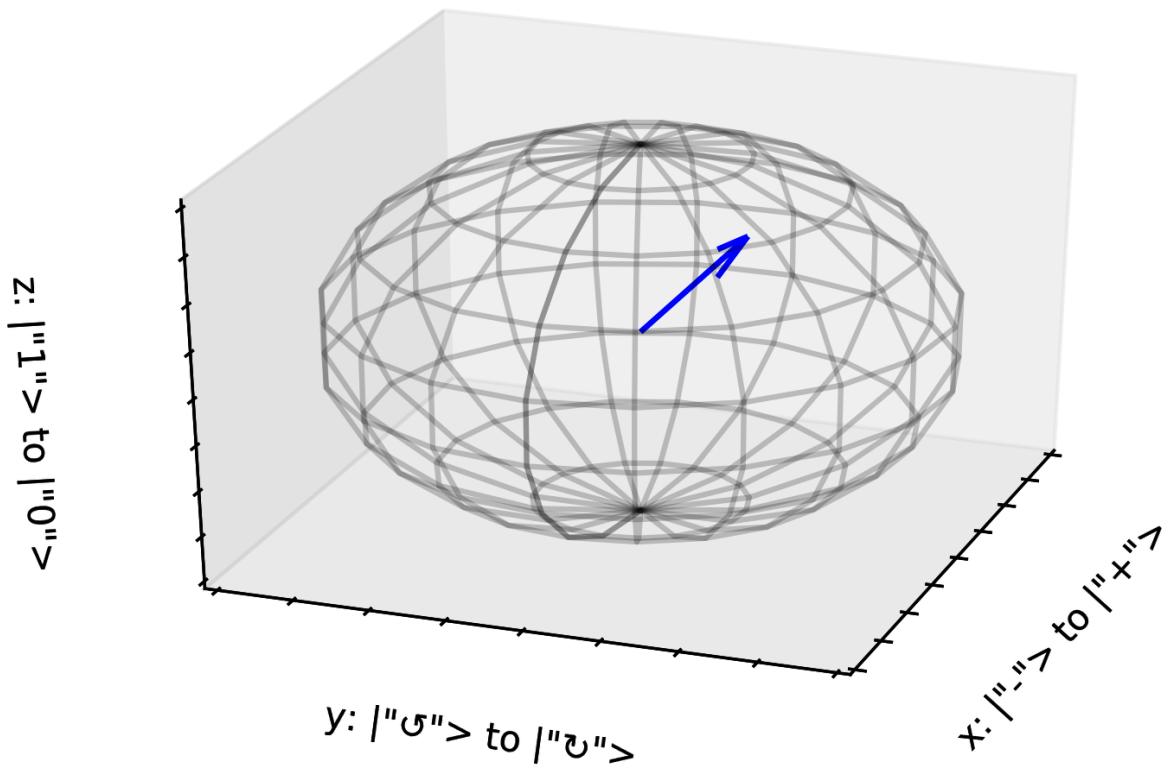
The Hadamard gate rotates the initial qubit by 180 degrees around the  $x$  axis, then 90 degrees about the  $y$  axis.

We know that the qubit that resides on the equator of the sphere would be in a superposition of  $|0\rangle$  and  $|1\rangle$ , and in equal parts of each. The Hadamard gate when applied to the  $|0\rangle$  qubit as in  $H|0\rangle$  transforms it to be in a superposition of equal parts  $|0\rangle$  and  $|1\rangle$ .

A plot of  $H|0\rangle$  can be seen with the `plot_bloch(H*zero_qubit)` statement as follows:



Here, we can see this is equal to  $|+\rangle$ . In words/algebra, that means that  $H|0\rangle = |+\rangle$ .  $H|1\rangle$  does likewise, though because of the starting point differing and rotations the gate performs, the result lies elsewhere, on the opposite end of  $H|0\rangle$  on the equator. A plot of  $H|1\rangle$  can be seen with the `plot_bloch(H*one_qubit)` statement as follows:



Here, we can see this is equal to  $|"/-\rangle$ . In words/algebra, that means that  $H|"/1\rangle = |"/-\rangle$ .

# Pauli gates (X, Y, Z)

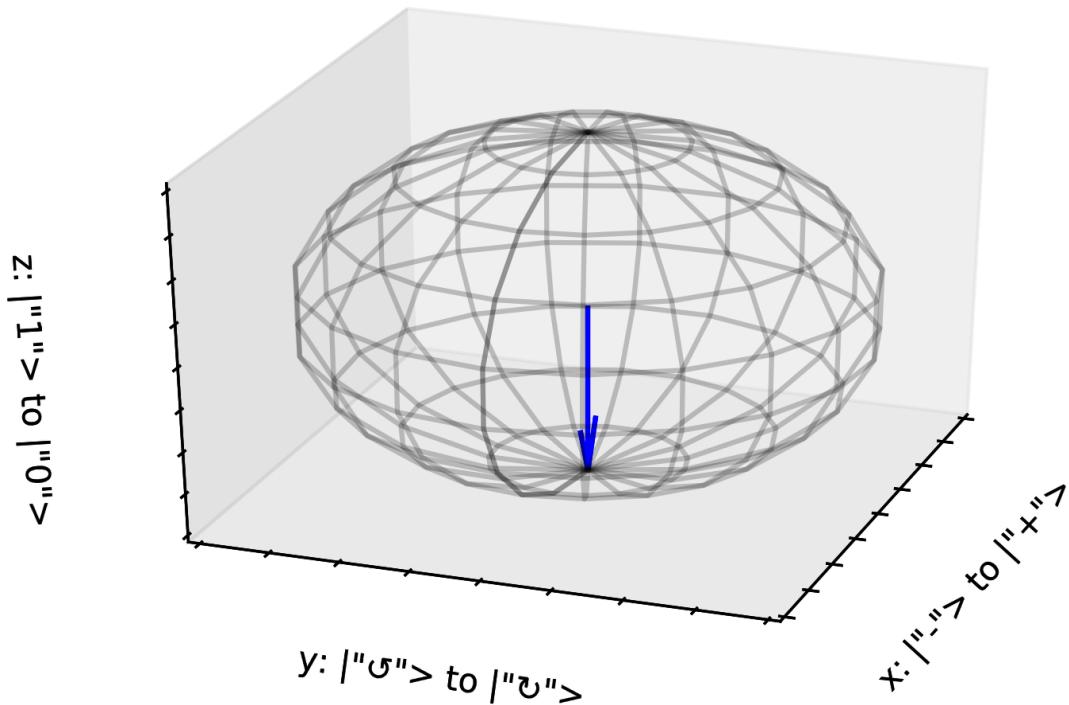
Each of these gates rotates the qubit they act on by 180 degrees. The rotation of the X gate is about the  $x$  axis, the rotation of the Y gate is about the  $y$  axis, and the rotation of the Z gate is about the  $z$  axis.

In Python, these gates are defined as follows:

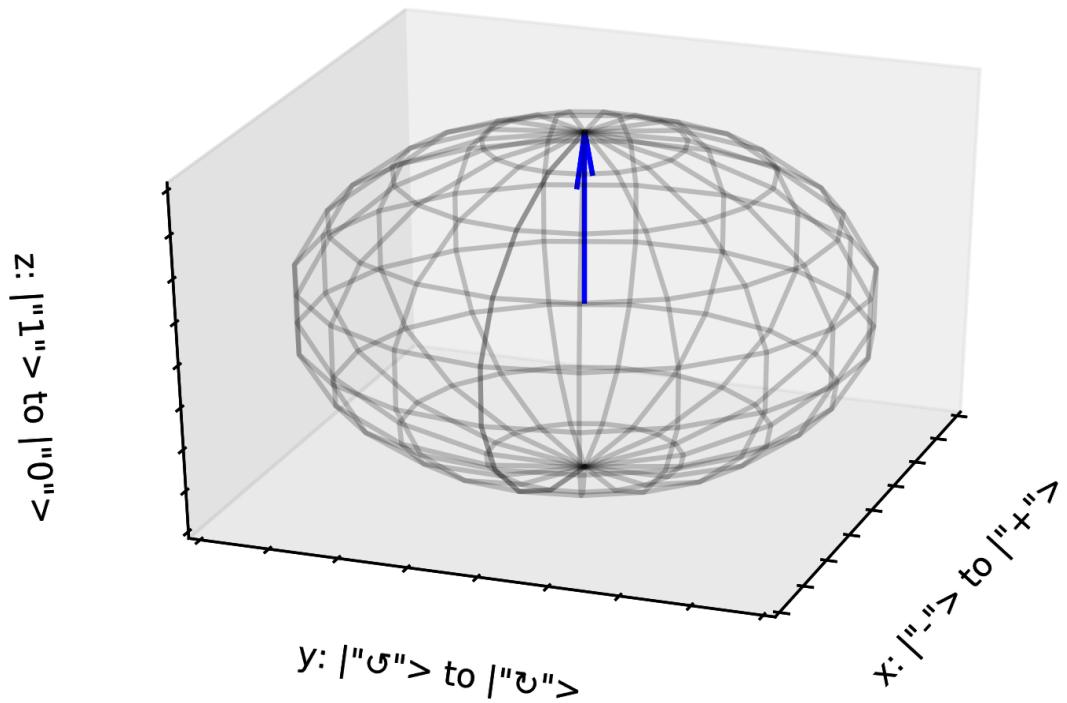
```
X=np.matrix('0 1; 1 0')
Y=np.matrix([[0, -i], [i, 0]])
Z=np.matrix([[1,0],[0,-1]])
```

# The X gate

The X gate is the quantum analogue of the classic NOT gate with respect to  $|0\rangle$  and  $|1\rangle$ . It rotates the qubit it acts on by 180 degrees around the x axis. Let's see it in action by operating it first on  $|0\rangle$  with  $X|0\rangle$  and viewing the results with the `plot_bloch(X*zero_qubit)` statement, as follows:



Sure enough, we see that  $X|0\rangle = |1\rangle$ . Next, we visualize  $X|1\rangle$  with the `plot_bloch(X*one_qubit)` statement to see that  $X|1\rangle = |0\rangle$ :



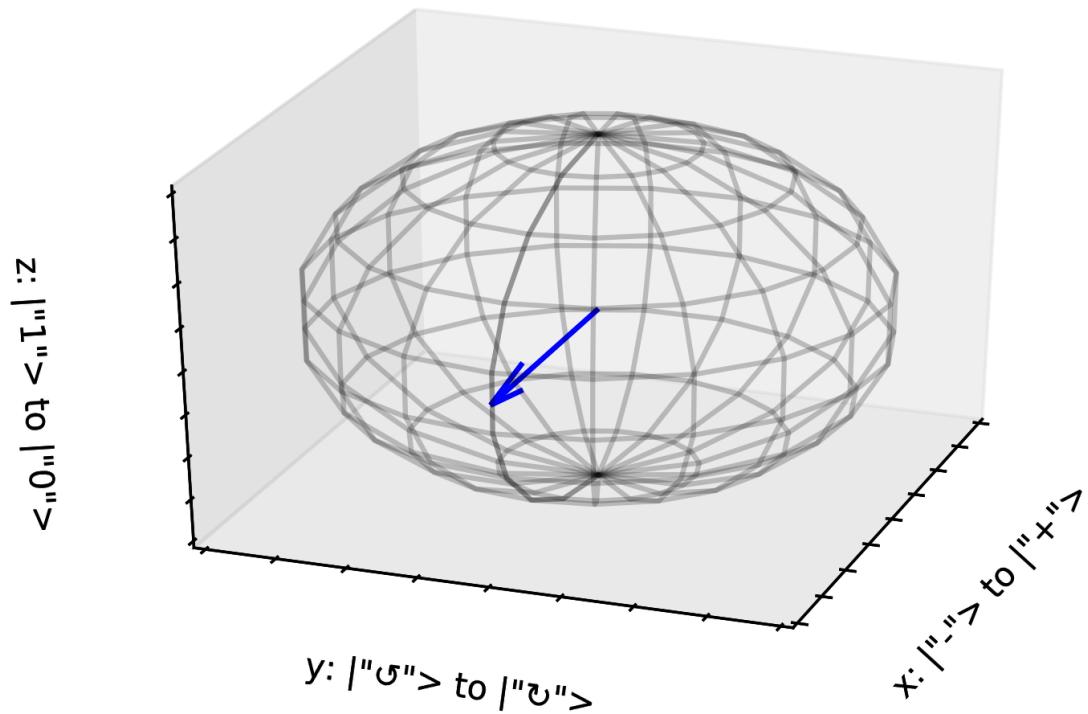
Sure enough, we can see the X gate acting as a NOT gate by flipping  $|"0">$  to  $|"1">$  and vice versa.

# The Y gate

The Y gate rotates the qubit it acts on by 180 degrees about the  $y$  axis. The results operating on the  $|0\rangle$  and  $|1\rangle$  qubit are the same as the X gate because these two qubits are aligned along the  $z$  axis, and, whether we rotate by 180 degrees around the  $x$  axis or the  $y$  axis, we will end up at the same place, on the opposite ends of the sphere. Written in words/algebra, that means  $X|0\rangle = Y|0\rangle$  and  $X|1\rangle = Y|1\rangle$ .

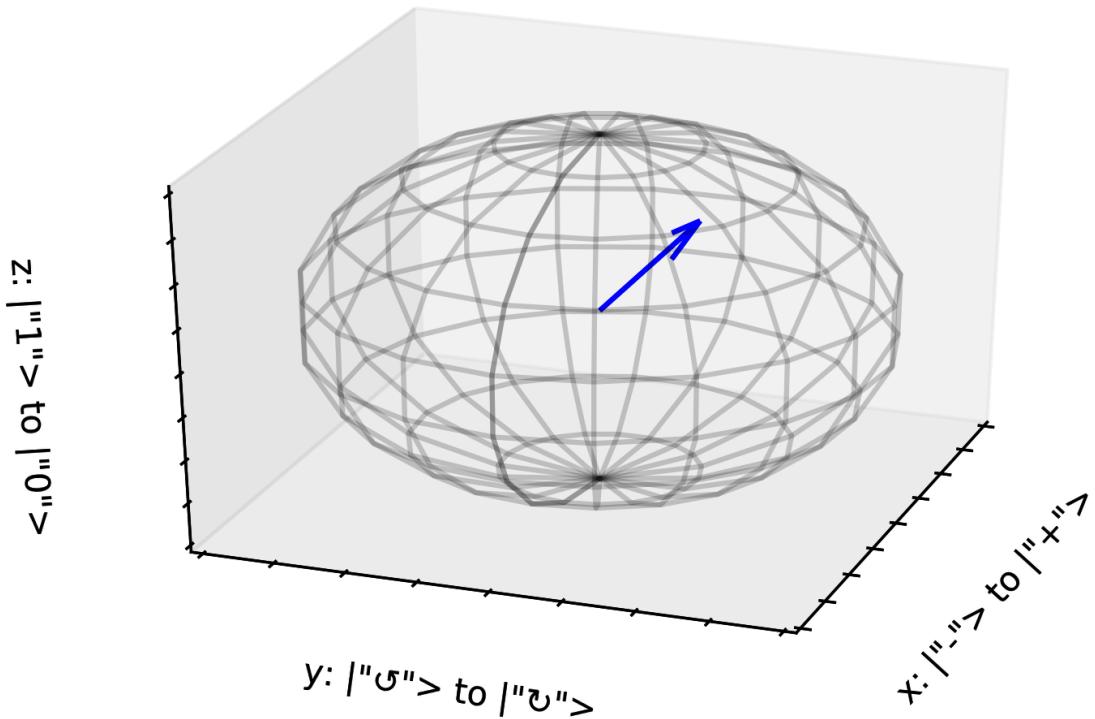
However, we can see differences in the two gates if we operate a qubit not aligned with the  $z$  axis  $|+\rangle$ , which as previously plotted aligns the qubit with the  $x$  axis

```
plot_bloch(plus_qubit):
```



Rotating  $|+\rangle$  around the  $x$  axis 180 degrees with the  $X$  gate seen in the last sub-section would do nothing to  $|+\rangle$ , because it is already aligned with the  $x$  axis. Written in words/algebra, that means  $X|+\rangle = |+\rangle$ .

But if we rotate  $|+\rangle$  around the  $y$  axis 180 degrees with the  $Y$  gate, we can see the effect. We visualize  $Y|+\rangle$  with the following `plot_bloch(Y*plus_qubit)` statement:



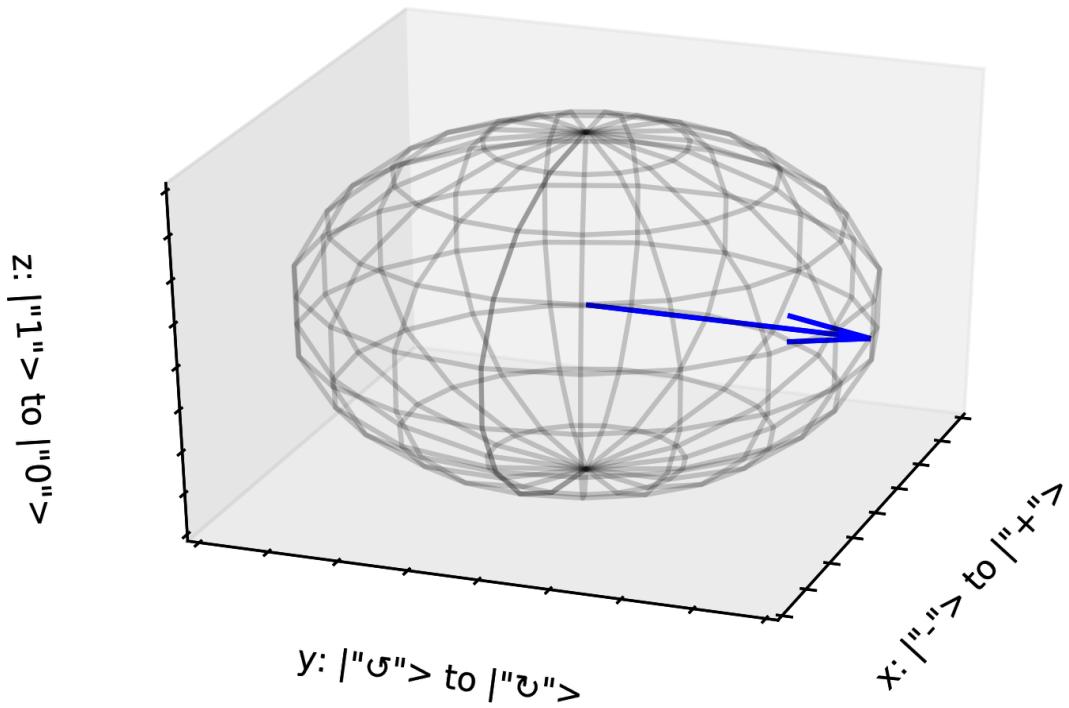
Here, we see  $|+\rangle$  has been flipped by 180 degrees around the  $y$  axis and is now equal to  $|-\rangle$ . In words/algebra, that means  $Y|+\rangle = |-\rangle$ . If we operated the  $Y$  gate on  $|-\rangle$ , we would see  $Y|-\rangle = |+\rangle$ .

The  $Y$  gate acts as a NOT gate with respect to  $|+\rangle$  and  $|-\rangle$ , flipping  $|+\rangle$  to  $|-\rangle$  and  $|-\rangle$  to  $|+\rangle$ .

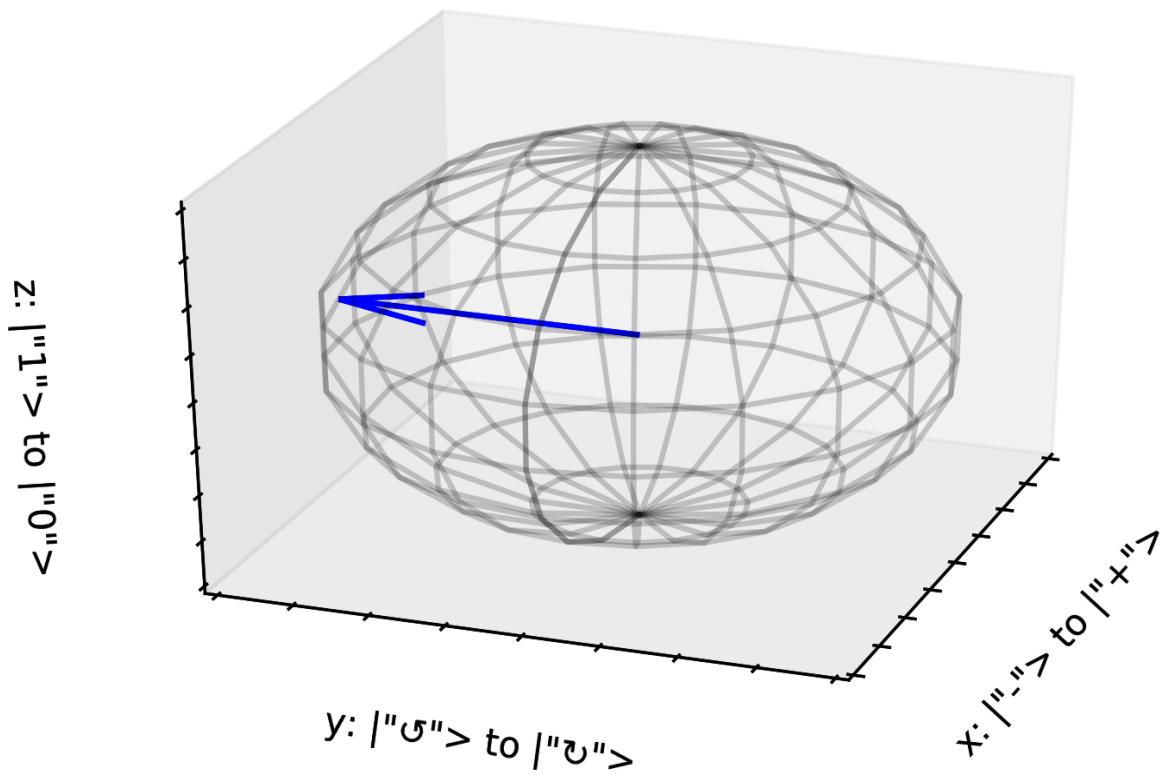
# The Z gate

The Z gate rotates the qubit it acts on by 180 degrees about the  $z$  axis. Since the qubits  $|0\rangle$  and  $|1\rangle$  are already aligned along the  $z$  axis, a rotation by 180 degrees with the Z gate won't change them. Using the Z gate on  $|+\rangle$  or  $|-\rangle$ , we would see an effect, because these qubits aren't aligned with the  $z$  axis. However, the effect is the same as applying the Y gate on  $|+\rangle$  or  $|-\rangle$ , as these qubits are aligned with the  $x$  axis whether we rotate around the  $y$  axis or the  $z$  axis; either way, we will flip  $|+\rangle$  to  $|-\rangle$  and  $|-\rangle$  to  $|+\rangle$ .

To see the Z gate having an effect on a qubit, and a different effect from the Y gate, we will consider it operating on the  $|0\rangle$  qubit, which as previously plotted aligns the qubit with the  $y$  axis as visualized with the `plot_bloch(clockwisearrow_qubit)` statement:



Since  $|\circlearrowleft\rangle$  is already aligned with the  $y$  axis, rotating it 180 degrees about the  $y$  axis with the  $Y$  gate will have no effect. In words/algebra, that means that  $Y|\circlearrowleft\rangle = |\circlearrowleft\rangle$ . However, rotating  $|\circlearrowleft\rangle$  by 180 degrees about the  $z$  axis will have an effect, as we can see from visualizing with the `plot_bloch(Z*clockwisearrow_qubit)` statement:



Here,  $|\psi\rangle$  has been flipped by 180 degrees and is now equal to  $|\bar{\psi}\rangle$ . In words/algebra, that means that  $Z|\psi\rangle = |\bar{\psi}\rangle$ . If we operated the Z gate on  $|\psi\rangle$ , we would see  $Z|\psi\rangle = |\bar{\psi}\rangle$ .

The Z gate acts as a NOT gate with respect to  $|\psi\rangle$  and  $|\bar{\psi}\rangle$ , flipping  $|\psi\rangle$  to  $|\bar{\psi}\rangle$  and  $|\bar{\psi}\rangle$  to  $|\psi\rangle$ .

# **Phase gate (S) and $\pi/8$ gate (T)**

After applying one of these gates, the probability of measuring  $|0\rangle$  or  $|1\rangle$  isn't changed; that is, the qubit keeps the same position relative to the z axis but the qubit rotates along the Bloch sphere by a certain amount around the z axis. Although the probability of measuring a  $|0\rangle$  or  $|1\rangle$  isn't changed, if we were to measure in a different basis, say the  $|+\rangle$  and  $|-\rangle$  basis or the  $|“\circlearrowleft\rangle$  and  $|“\circlearrowright\rangle$  basis, that measurement could change.

These gates are similar to the Z gate, in that Z, S, and T all perform rotations about the z axis; only the angle by which the qubit is rotated differs for each gate. For the S gate, that angle is 90 degrees and for the T gate that angle is 45 degrees, compared to 180 degrees for the Z gate.

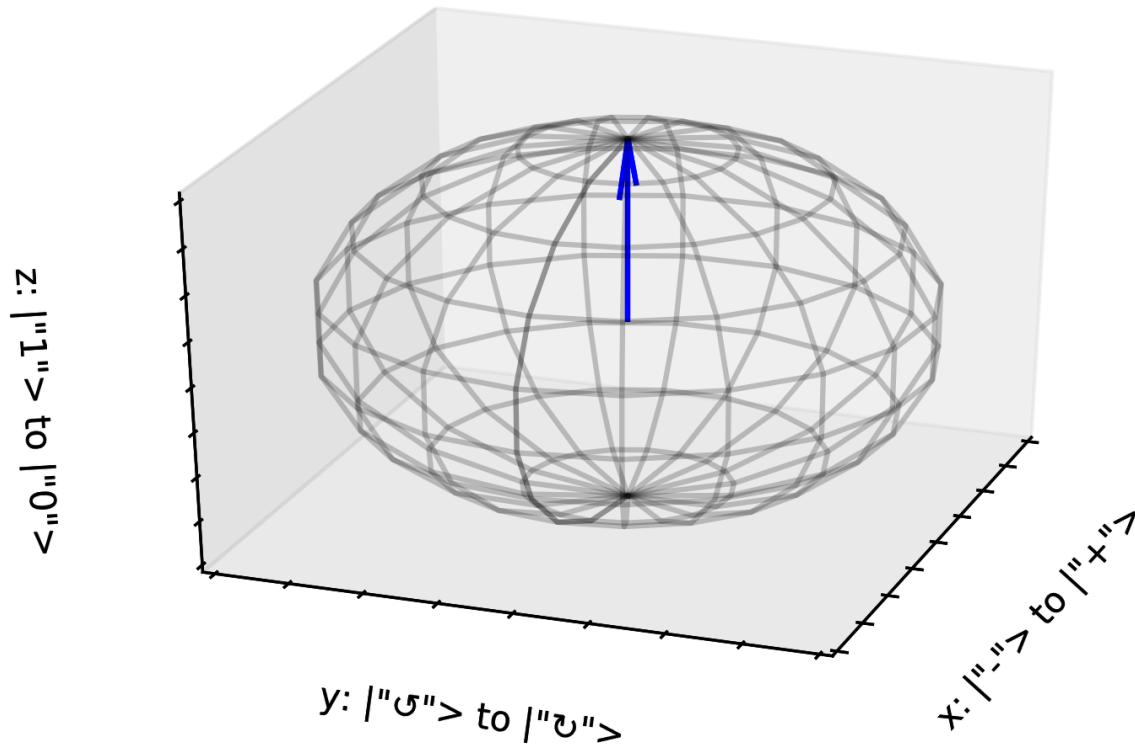
# Phase gate (S)

The S gate rotates around the  $x$ - $y$  plane by 90 degrees ( $\pi/2$  radians).

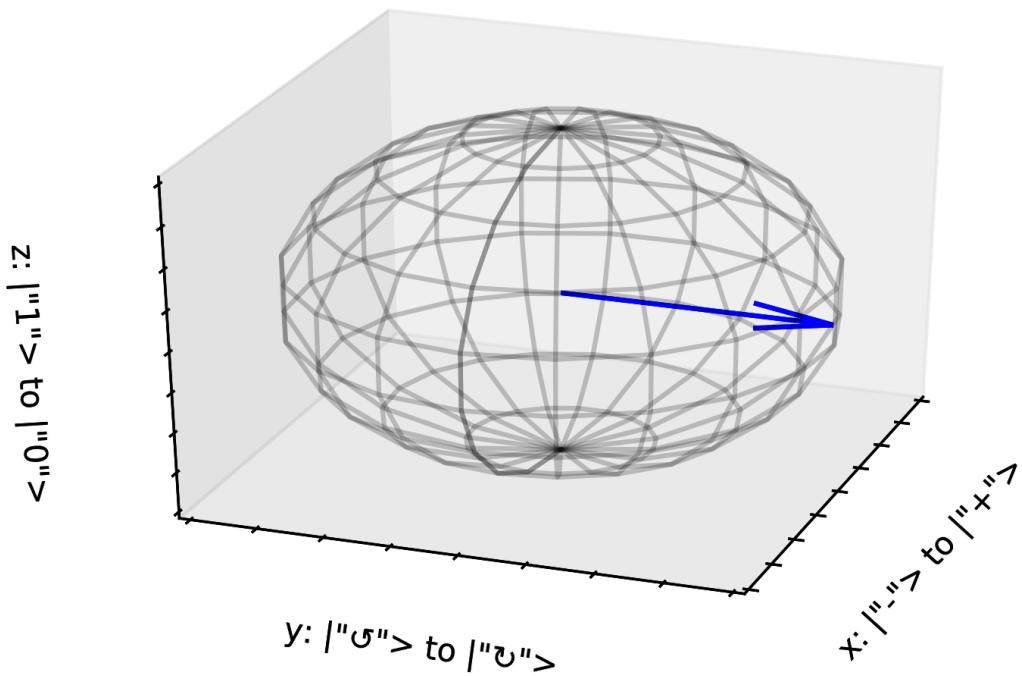
In Python it can be written as follows:

```
| S=np.matrix([[1,0],[0,np.e**(i_*np.pi/2.)]])
```

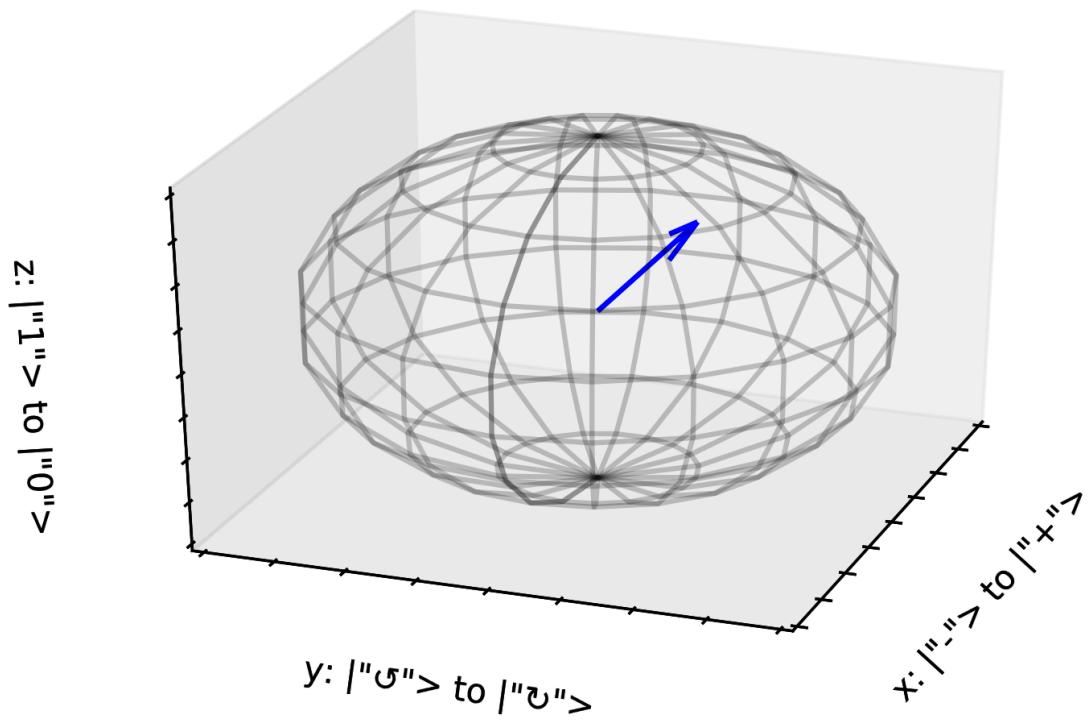
We can see the S gate has no effect on  $|0\rangle$  with the `plot_bloch(S*zero_qubit)` statement:



But if we pick a qubit in the  $x$ - $y$  plane, say  $|+\rangle$ , we can see the 90 degree rotation with the `plot_bloch(S*plus_qubit)` statement:



Here, we see  $S|+\rangle$  has rotated the  $|+\rangle$  qubit 90 degrees around the  $z$  axis to become  $|+\rangle$ . If we again apply the S gate, we will get another rotation, as can be seen with the `plot_bloch(S*S*plus_qubit)` statement:



Two applications of the S gate have rotated the  $|+\rangle$  qubit by  $90^\circ + 90^\circ = 180^\circ$ , turning it into the  $|-\rangle$  qubit. Four applications of the S gate to the  $|+\rangle$  qubit would rotate us  $90^\circ * 4 = 360^\circ$ , right back to where we started.

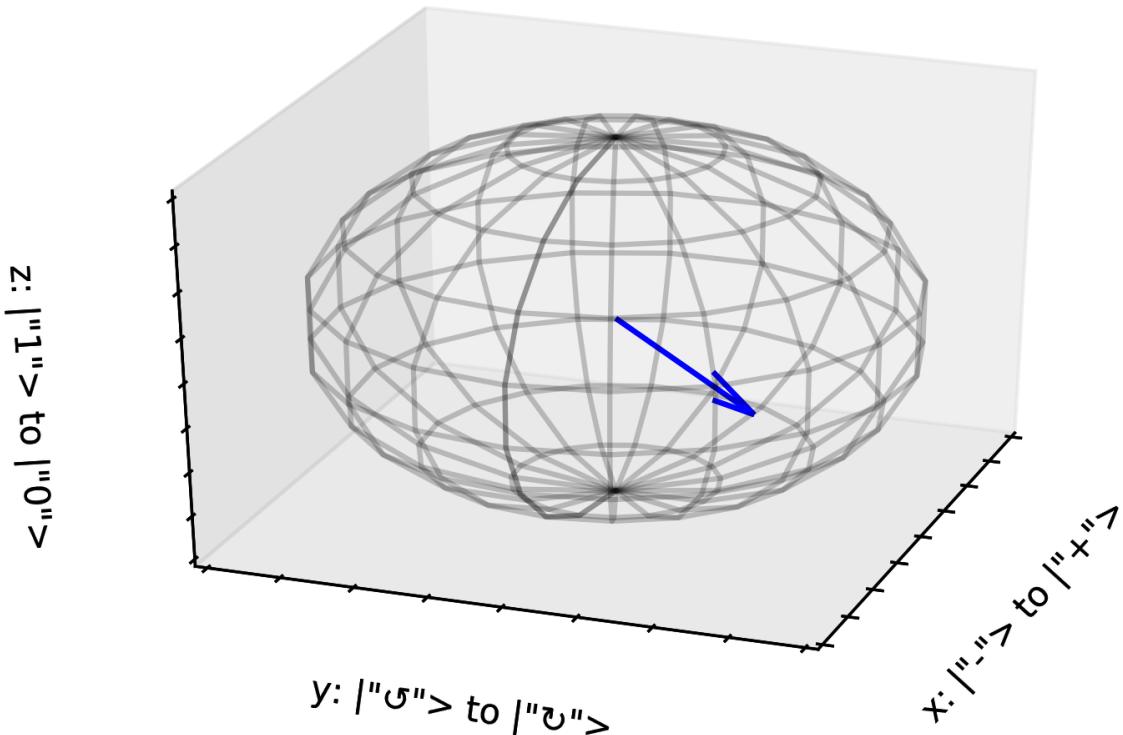
# $\pi/8$ gate (T)

The T gate rotates around the x-y plane by 45 degrees ( $\pi/4$  radians). Why this gate isn't just called the  $\pi/4$  gate is a historical artifact.

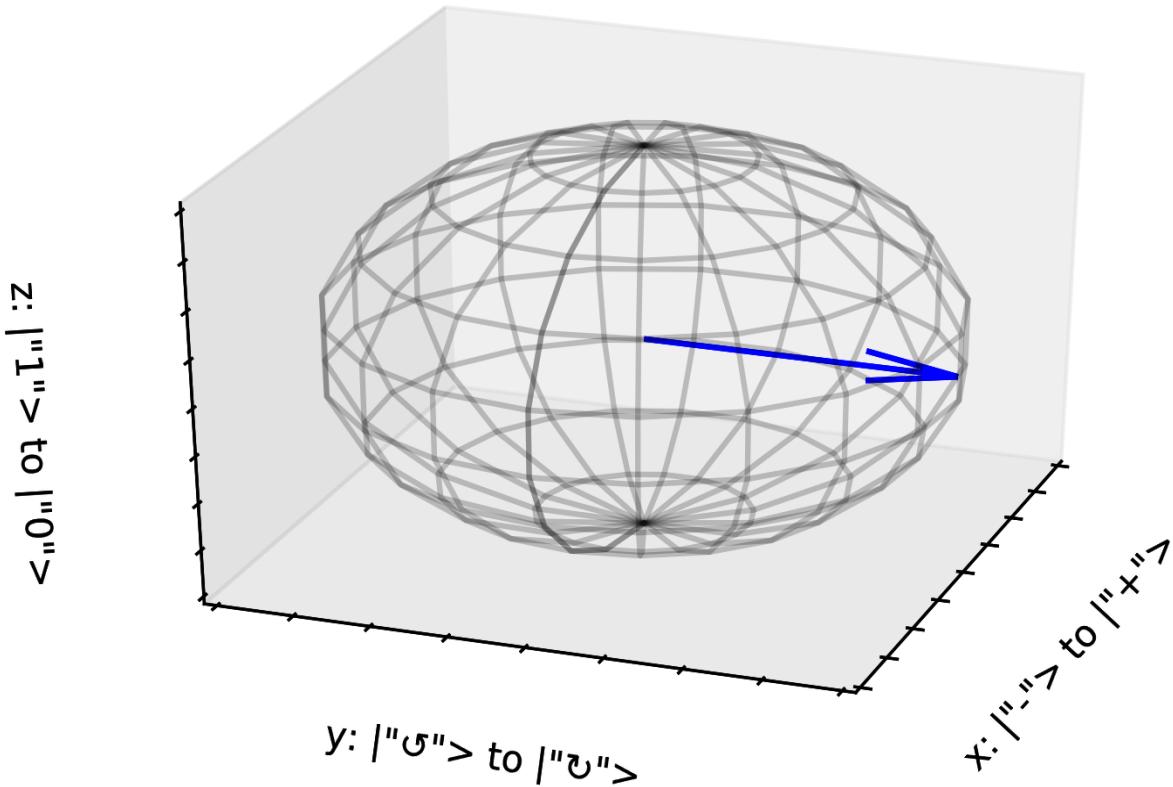
In Python it can be written as follows:

```
| T=np.matrix([[1,0],[0, np.e**(i_*np.pi/4.)]])
```

Again, this gate does nothing to a qubit on the z axis, but rotates all other qubits around the z axis. We can see this rotation on the  $|+\rangle$  gate by visualizing  $T|+\rangle$  with the `plot_bloch(T*plus_qubit)` statement:



Sure enough, the T gate has rotated  $|+\rangle$  by 45 degrees. Applying the T gate twice, we will get a  $45^\circ + 45^\circ = 90^\circ$  rotation about the z axis, which we can visualize with the `plot_bloch(T*T*plus_qubit)` statement:

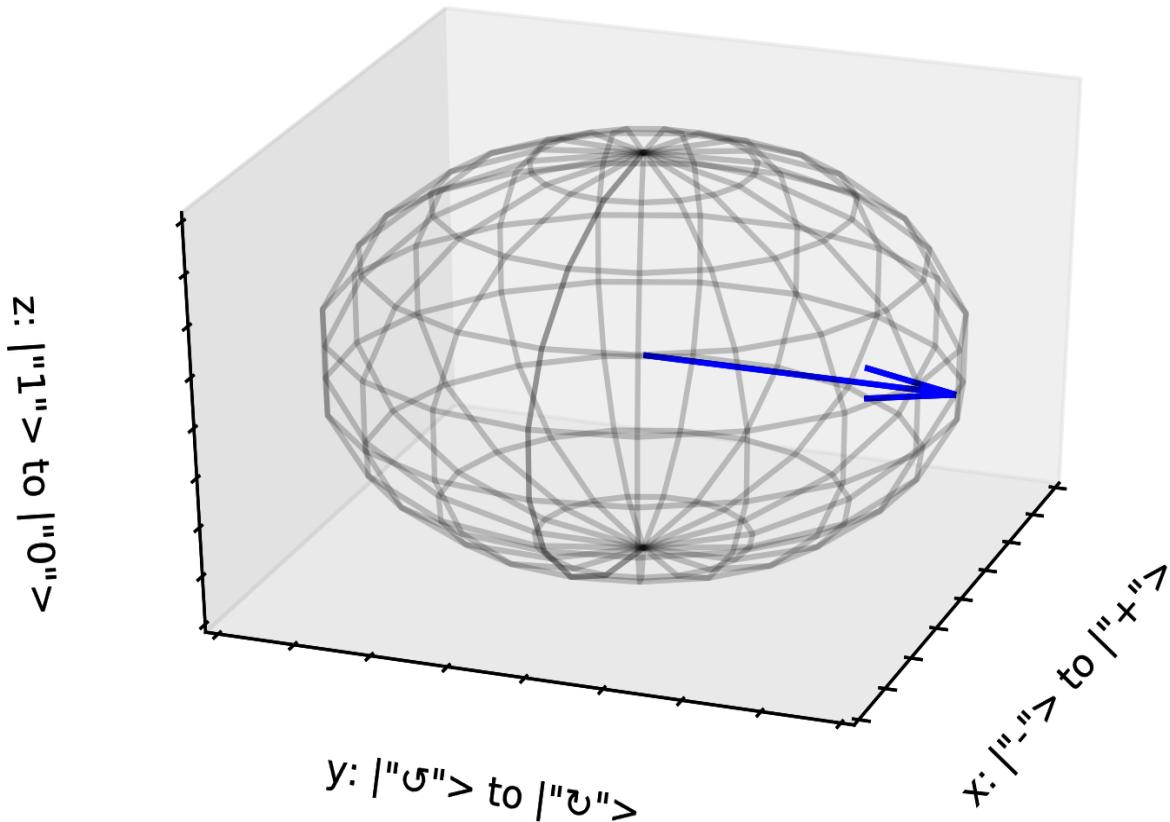


Here, we can see that  $T^2 |+\rangle = S |+\rangle = |\circlearrowleft\rangle$ .

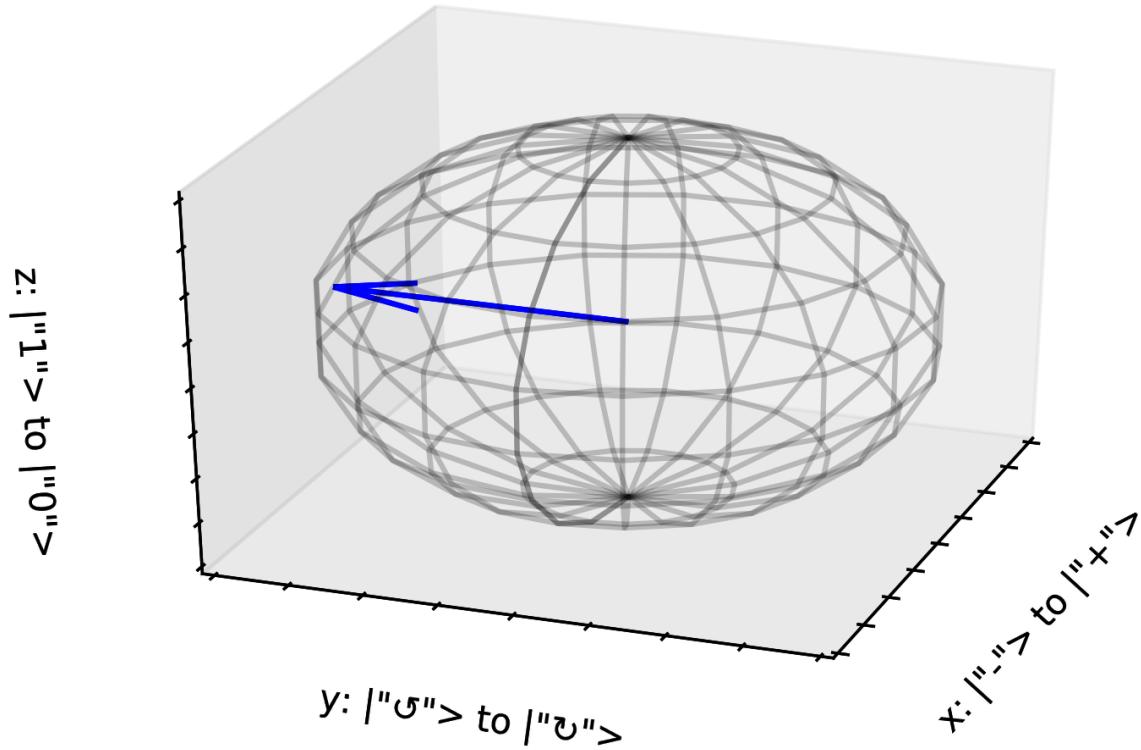
# The "dagger" gates $S^\dagger$ and $T^\dagger$

You will also see "dagger" gates  $S^\dagger$  and  $T^\dagger$ . The  $\dagger$  symbol is pronounced "dagger." In Python, these are defined in terms of the S and T gates as `Sdagger = S.conjugate().transpose()` and `Tdagger= T.conjugate().transpose()` respectively. These gates perform the same rotation about the z axis in terms of degrees as the S and T gates, only the rotation is in the opposite direction.

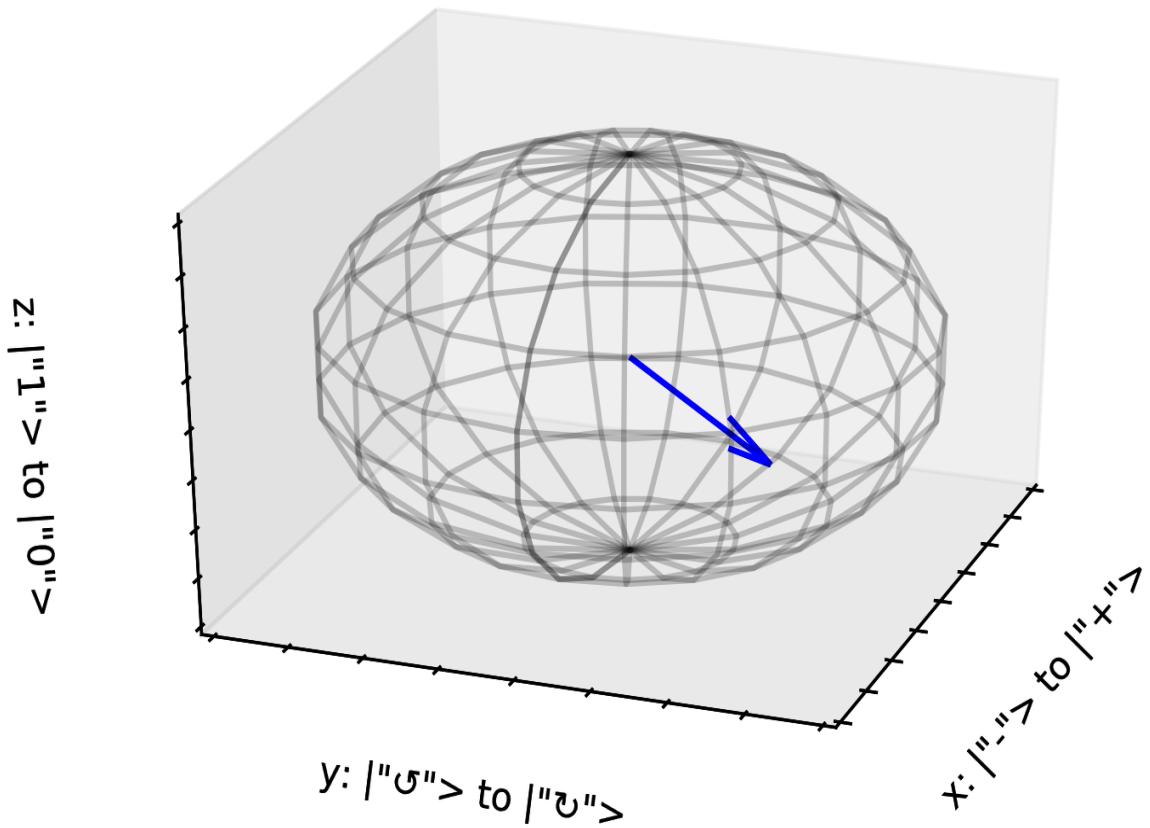
Compare  $S/\lvert + \rangle$  as visualized on the Bloch sphere by the Python `plot_bloch(S*plus_qubit)` statement:



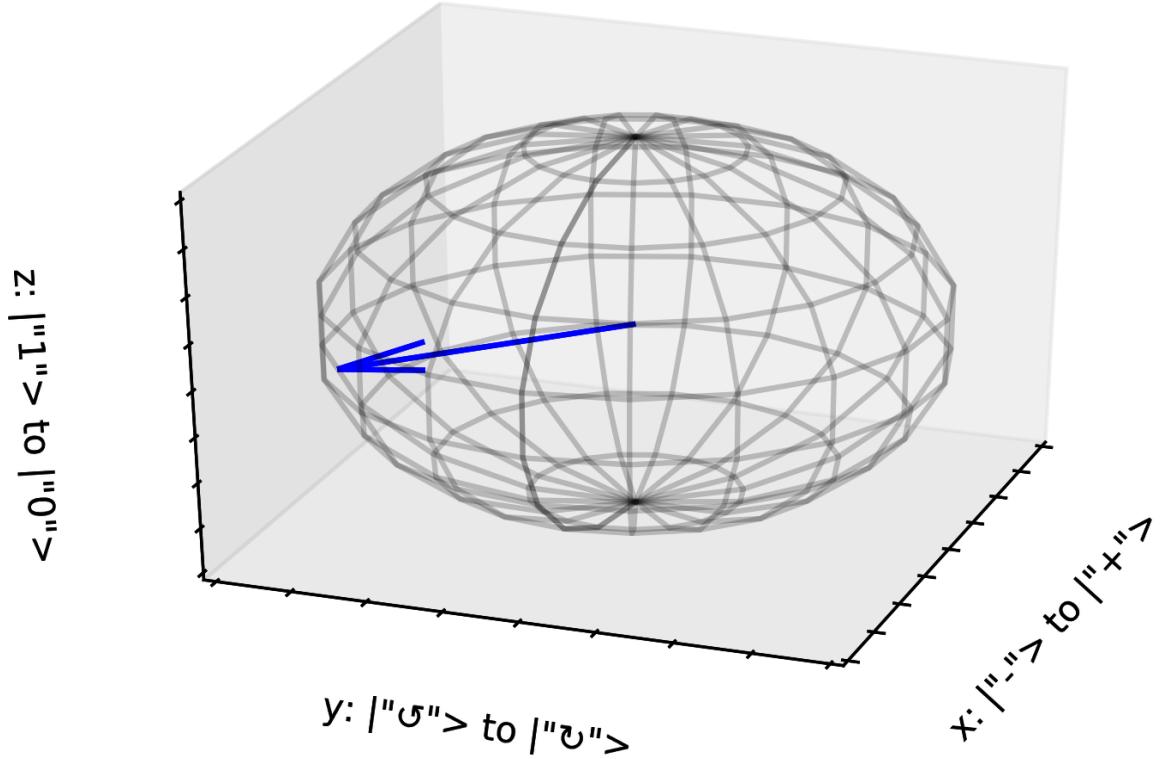
Compare it to  $S^\dagger |+\rangle$  as visualized on the Bloch sphere by the Python `plot_bloch(Sdagger*plus_qubit)` statement:



And, compare  $T|+\rangle$  as visualized on the Bloch sphere by the Python `plot_bloch(T*plus_qubit)` statement:



Compare it to  $T^\dagger |+\rangle$  as visualized on the Bloch sphere by the Python `plot_bloch(Tdagger*plus_qubit)` statement:



# **Multi-qubit gates**

In this section, we'll go over a multi-qubit gate, CNOT. CNOT is critical to quantum computation as it produces entangled states. Recall that entangled states are nothing more than states where each qubit cannot be discussed individually, but rather must be considered as a group.

# CNOT gate

The CNOT gate stands for the controlled-not gate and always operates on two qubits at once. This gate provides the quantum analogue of the classic XOR gate. The first qubit acts as a control qubit, and never changes as a result of the gate's operation. If this first qubit is  $|1\rangle$ , then a NOT operation is applied to the second qubit. If this first qubit is  $|0\rangle$ , then nothing is done to the second qubit.

This is easy enough to illustrate in a table for states  $|00\rangle$ ,  $|01\rangle$ ,  $|10\rangle$ , and  $|11\rangle$ :

Starting state	CNOT operating on stating state
$ 00\rangle$	$ 00\rangle$
$ 01\rangle$	$ 01\rangle$
$ 10\rangle$	$ 11\rangle$
$ 11\rangle$	$ 10\rangle$

The second qubit depends upon the first qubit, so if the first qubit's state changes before input to CNOT, the second qubit will shift. Since this gate is dealing with two qubits, it is not possible to visualize it generally on the Bloch sphere. We will need to make do with words.

If our control qubit is a starting state that is in a superposition between  $|0\rangle$  and  $|1\rangle$ , things get more difficult to envision in a chart, but we can still think of them in words/algebra or in code. We'll start out in words/algebra and then see how this plays out in code.

First, let's have our control qubit in a state that is halfway between zero and one, the  $|+\rangle$  qubit. Recall that

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$$

and measuring a  $|+\rangle$  qubit will result in  $|0\rangle$  approximately 50% of the time and  $|1\rangle$  approximately 50% of the time.

 *We can also create the  $|+\rangle$  state from  $|0\rangle$  by operating the Hadamard gate on  $|0\rangle$  as  $|+\rangle$ . Programming a physical quantum computer often requires starting with qubits in the  $|0\rangle$  state, so keeping in mind how to apply gates  $|0\rangle$  to reach other desired starting states is an important skill in quantum programming. It's part of the reason we've focused so much on visualizing the process of obtaining one qubit from another with a variety of gates.*

We'll take our second qubit to be the  $|0\rangle$  state. So we can write our two-qubit state, we start out with (naming it "starting") as:

$$|\text{"starting"}\rangle = |+0\rangle$$

The  $|+\rangle$  state isn't entangled, as we can separate it out into two qubits, the first qubit being  $|+\rangle$  and the second qubit being  $|0\rangle$ . For the next step, we'll use the fact that

$$|+\rangle = \frac{|0\rangle + |1\rangle}{\sqrt{2}}$$

to rewrite the  $|+\rangle$  state as:

$$|\text{"starting"}\rangle = \frac{|00\rangle + |10\rangle}{\sqrt{2}}$$

I've just chosen to write them together so that, in the next step, when we apply the CNOT gate, it is easier to see what happens.

Now, let's apply the CNOT gate to  $|"starting"\rangle$  to get a final state called  $"final"$ :

$$|"final"\rangle = \text{CNOT} |"starting"\rangle = \text{CNOT} \frac{|"00"\rangle + |"10"\rangle}{\sqrt{2}} = \frac{\text{CNOT} |"00"\rangle + \text{CNOT} |"10"\rangle}{\sqrt{2}} = \frac{|"00"\rangle + |"11"\rangle}{\sqrt{2}}$$

So, our final state is this:

$$|"final"\rangle = \frac{|"00"\rangle + |"11"\rangle}{\sqrt{2}}$$

We saw in the last chapter, under the *Separable states* section, that this state is a non-separable state, so we can no longer talk about the two qubits apart from each other. We saw in the last chapter, under the *Entanglement* section, that a state that is non-separable is an entangled state. So, we've generated an entangled state from two non-entangled qubits using the CNOT gate. If we measure this entangled state, as we saw in the last chapter, we will get  $|"00"\rangle$  approximately 50% of the time and  $|"11"\rangle$  approximately 50% of the time.

# Python code for the CNOT gate

In Python, we can define the CNOT gate as follows:

```
|CNOT=np.matrix('1 0 0 0; 0 1 0 0; 0 0 0 1; 0 0 1 0')
```

To see this CNOT in action in a simulation, we will need some code from a previous chapter, specifically the `create_quantum_state` and `measure_in_01_basis` functions from [Chapter 3, Quantum States, Quantum Registers, and Measurement](#). Consider the following state:

$$|\text{"starting"}\rangle = \frac{|\text{"00"}\rangle + |\text{"10"}\rangle}{\sqrt{2}}$$

We create the preceding state with the following code:

```
|starting_state=create_quantum_state([plus_qubit,zero_qubit])
```

Next, we operate CNOT on `/"starting">`:

```
|final_state=CNOT*starting_state
```

Since we can no longer use the Bloch sphere to visualize our final results, we can create a helper function that will visualize the probability of obtaining a certain state in the `/"0">` and `/"1">` basis in simulation:

```
|def probability_table_in_01_basis(state,n_measurements=1000):
    from collections import Counter
    measured=[measure_in_01_basis(final_state) for i in
    range(n_measurements)]
    for s,c in Counter(measured).items():
        print(s,"{:0.0%}".format(c/n_measurements))
probability_table_in_01_basis would be an impossible function outside of a simulation,
because measuring a state once means all subsequent measurements would be the
same. To get a similar function outside of a simulation, we would need to prepare
```





*multiple pairs of two qubits in an identical manner, then measure each pair individually, and then compute statistics afterward.*

Now, let's use `probability_table_in_01_basis` to show our results:

```
| probability_table_in_01_basis(final_state)
```

The preceding statement prints the following:

```
|| "11"> 48%  
|| "00"> 52%
```

For me, the first time I ran it, I got the following output:

```
|| "11"> 50%  
|| "00"> 50%
```

The second time, is identical to what we'd expect from our work in words/algebra before. Each time I run the code, it may produce a different result, as the measurement process is probabilistic. The higher the number of measurements I make, the closer the final result is to the result expected from words/algebra.

# **CNOT with control qubit of choice, target qubit of choice**

We can write a CNOT gate with a control qubit other than the first qubit, and a target other than the second qubit. For example, a CNOT gate with the control qubit being the second qubit and the target qubit being the first. In Python, this would be the following:

```
| CNOT_control1_target0=np.kron(H,H)*CNOT*np.kron(H,H)
```

In simulation, we can have any qubit in the register acting as the control and any acting as the target. In physical quantum computers, sometimes there are constraints based on the physical connections of the qubits, where only certain qubits in the state can act as a control, and, for each control, only certain qubits can act as the target qubit.

# Summary

Computation, both classic and quantum, relies on taking an input and transforming it in the desired way to achieve a certain output. In quantum computing, this is done with quantum gates.

Quantum gates operate on a quantum register to change its state. Quantum gates can operate on one, two, or more qubit states. In this chapter, we covered quantum gates operating on one or two qubits. The set of gates presented in this chapter form a universal gate set; that is, with just a combination of these gates, we can perform any quantum computation.

The action of a quantum gate operating on a single qubit state moves the single qubit to a new position on the Bloch sphere. In this chapter, we learned about a number of single-qubit gates. The identity gate  $I$  leaves the qubit's position on the Bloch sphere unchanged. The Hadamard gate  $H$  rotates the initial qubit by 180 degrees around the  $x$  axis, then 90 degrees about the  $y$  axis. The Pauli gates ( $X$ ,  $Y$ , and  $Z$ ) rotate the qubit they act on by 180 degrees. The rotation of the  $X$  gate is about the  $x$  axis, the rotation of the  $Y$  gate is about the  $y$  axis, and the rotation of the  $Z$  gate is about the  $z$  axis. The phase gate ( $S$ ), and the  $\pi/8$  gate ( $T$ ) are similar to the  $Z$  gate in that they all rotate the initial qubit about the  $z$  axis. Only the angle by which the qubit is rotated differs for each gate. For the  $S$  gate, that angle is 90 degrees and for the  $T$  gate that angle is 45 degrees, compared to 180 degrees for the  $Z$  gate. The gates  $S^\dagger$  and  $T^\dagger$  are identical in operation to the  $S$  and  $T$  gates, except their rotation is in the opposite direction.

The two-qubit CNOT gate generates entanglement through its operation on more than one qubit. This gate provides the quantum analogue of the classic XOR gate. The first qubit acts as a control qubit, and never changes as a result of the gate's operation. If this first qubit is  $|1\rangle$ , then a NOT operation is applied to the second qubit. If this first qubit is  $|0\rangle$ , then nothing is done to the second qubit.

With just the I, X, Y, Z, H, S,  $S^\dagger$ , T,  $T^\dagger$ , and CNOT gates, and a bit of understanding as to how they operate, we are armed to start programming for quantum computers.

# Questions

1. Find another classic universal gate set.
2.  $|0\rangle I$  is not an allowed operation. Try applying the gate after the qubit in Python; what happens?
3. What would be the result of  $|||||0\rangle$ ? What would be the result of  $||+\rangle$ ?
4. Visualize  $XH|0\rangle$  on the Bloch sphere. What gate would you have to apply to  $|1\rangle$  to get the same result?
5. Starting out with a  $|+\rangle$  qubit, which gate(s) could you apply to get  $|0\rangle$  as a result? To get  $|-\rangle$  as a result?
6. What gate do you get if you apply the X gate twice? The Y gate twice? The Z gate twice?
7. What is  $SS^\dagger|+\rangle$ ? What is  $TT^\dagger|+\rangle$ ?
8. What would you get by applying the CNOT gate to the state  $|++\rangle$ ?

# Quantum Circuits

This chapter expands on the idea of quantum gates to introduce quantum circuits, the quantum analogue of classical circuits. It goes over how classical gates can be reproduced by quantum circuits and proceeds to introduce a visual representation of quantum circuits that can be used to easily define a quantum circuit without reference to mathematics or use of a programming language. We discuss the concept of reversible computation, and learn how to combine gates to undo any quantum computation. We then go on to define several simple circuits that will later be useful as building blocks as we build up more complicated quantum algorithms.

The following topics will be covered in this chapter:

- Quantum circuits and quantum circuit diagrams
- Using Qiskit to generate quantum circuits
- Reversible computation
- Quantum circuit examples

# Technical requirements

The Jupyter Notebook for this chapter is available at <https://git.hub.com/PacktPublishing/Mastering-Quantum-Computing-with-IBM-QX>, under Chapter05.

# Quantum circuits and quantum circuit diagrams

In this section, we will simulate a quantum circuit using Python. The good news is that it's easy, as we've already done it. A quantum circuit is merely a sequence of quantum gates. One example is  $Y / "+ " >$ , another example is  $XH / "0" >$ . In the circuits we work with on IBM QX, the initial state will always be  $/ "0" >$  for a given qubit.

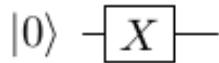
A **quantum circuit diagram** is a method to visualize quantum circuits. Typically, on the left-hand side of a quantum circuit diagram, we'll start with a series of  $/ "0" >$  qubits with lines coming out of them that look a bit like wires; hence the name quantum circuits. Quantum gates can be placed along the lines in the order they are to be applied to the  $/ "0" >$  qubit, with their name and a box drawn around it. For multiqubit-controlled gates, such as CNOT, the gate is put on the target qubit wire, and a line is drawn from the gate to the control qubit wire. The target qubit has a larger, open circle drawn around it, while the control qubit has a smaller black circle drawn on it.

When we write gates in words/algebra or in code, we put  $/ "0" >$  on the right side, as in  $XYZ / "0" >$ , and apply the gate closest to the qubit first, going from right to left. This is an artifact of the mathematical operations underlying the gate application. With quantum circuits, we have a setup that is more like reading in English. We go from left to right, applying gates in the order that we read them to the  $/ "0" >$  qubit. From now on, we'll primarily deal with visual quantum circuits as we program! So, just remember, in a

quantum circuit things go from left-to-right, but in a Python program or words/algebra, things go from right-to-left.

Let's first look at some quantum circuit diagrams for quantum circuits we have already seen in previous chapters:

- The quantum circuit diagram for  $X|0\rangle$  is as follows:

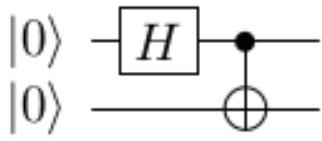


- The quantum circuit diagram for  $XH|0\rangle$  is as follows:



Remember that, CNOT is the controlled NOT gate, and always operates on two qubits at once. This gate provides the quantum analogue of the classical XOR gate. The first qubit acts as a control qubit, and never changes as a result of the gate's operation. If this first qubit is  $|1\rangle$ , then a NOT operation is applied to the second qubit. If this first qubit is  $|0\rangle$ , then nothing is done to the second qubit.

Now suppose we want to compute CNOT using  $|+\rangle$  as the control qubit and  $|0\rangle$  as the target qubit. Note that in the quantum circuit diagrams, we always start out with  $|0\rangle$ . So, we have to figure out how to get from  $|0\rangle$  to  $|+\rangle$ . Looking at our notes from [Chapter 4](#), *Evolving Quantum States with Quantum Gates*, we can see that  $|+\rangle = H|0\rangle$ . So, we just need to apply an *H* gate to  $|0\rangle$  before we apply CNOT. Following is the quantum circuit diagram:

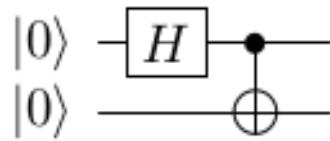


If we measure this entangled state, as we saw in the previous chapter, we will get  $|00\rangle$  approximately 50% of the time and  $|11\rangle$  approximately 50% of the time. Measurement can also be drawn into a quantum circuit, and our default measurement basis will be the  $|0\rangle$  and  $|1\rangle$  basis (also known, as we learned in [Chapter 3, Quantum States, Quantum Registers, and Measurement](#), as the  $z$ -basis). Since measurement of a given qubit will be the  $|0\rangle$  and  $|1\rangle$  basis, the measurement will always produce either a  $|0\rangle$  or a  $|1\rangle$ . We could then output the result of the measurement into a classical register, set to bit 0 if the measurement was  $|0\rangle$  and to the  $|1\rangle$  bit if the measurement was 1. We can represent the measurement

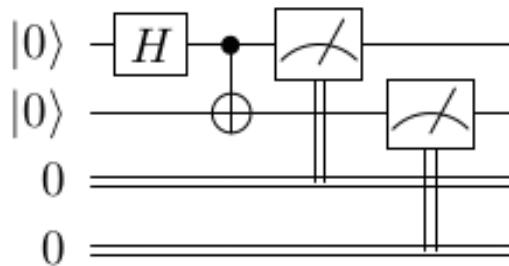
process with the

We can represent our classical registers as starting out containing bit 0, and with double wires to distinguish them from the quantum registers. We could also choose to represent them more succinctly with a single wire with a slash through it and the label  $0^n$  (zero to the  $n^{\text{th}}$  power), using  $n$  to indicate how many 0 bits the register should contain. This is just a shorthand, and one that you will see on IBM QX. For example, a classical register of five bits, each initialized to 0, could be visually represented by a single wire with a slash through it and the label  $0^5$ . Then we can draw a double line from the

symbol to the register in which we want to place the result of the measurement, thus measuring both the qubits in the circuit:

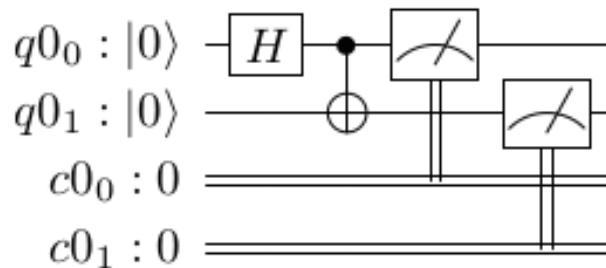


This will look as follows:



So, we know that if we prepare this circuit many times, about 50% of the time the classical registers will contain 00 after the run, and about 50% of the time the classical registers will contain 11.

Since we may want to refer to the quantum or classical registers by a name, so we can tell them apart, work with them in code, and talk about their results, we can also include a name for the register by indicating it on the left before  $|0\rangle$ . Here is an example of the previous circuit with labels. We label the two qubits as **q0<sub>0</sub>** and **q0<sub>1</sub>**, and the two classical bits as **c0<sub>0</sub>** and **c0<sub>1</sub>**:



IBM refers to its quantum circuit diagram as quantum scores. If you've ever seen music notation, a quantum score looks a bit like a musical score. In IBM QX, quantum scores can be created via the quantum composer, again a musical reference. All the gates we have learned so far are available in the quantum composer, plus a few additional gates. Although the gates we have learned so far form a universal gate set, that is, we can just use a combination of them to generate any quantum program, on a real quantum computer the fewer gates we use the better. This is because each gate takes time to execute and introduces errors.

# Using Qiskit to generate quantum circuits

**Qiskit** is the **Quantum Information Science Kit**. It is an SDK for working with the IBM QX quantum processors. It also has a variety of tools to simulate a quantum computer in Python. It is so important and useful it will get its own chapter later in this book. In this chapter, we are going to learn to use it to generate quantum circuits.

# Single-qubit circuits in Qiskit

First, let's import the tools to create classical and quantum registers as well as quantum circuits from `qiskit`:

```
| from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
```

Next let's make the  $X$  / "0" > circuit using `qiskit`:

```
| qr = QuantumRegister(1)
| circuit = QuantumCircuit(qr)
| circuit.x(qr[0])
```

Note that the argument to `QuantumRegister` is 1; this indicates that the quantum register is to contain one qubit.

The  $XH$  / "0" > circuit using `qiskit` becomes the following:

```
| qr = QuantumRegister(1)
| circuit = QuantumCircuit(qr)
| circuit.h(qr[0])
| circuit.x(qr[0])
```

# **Qiskit's QuantumCircuit class and universal gate methods**

We can see that the `QuantumCircuit` class allows us to execute a variety of gates on particular qubits in the quantum register used to initial it. The full gate set is available in the `QuantumCircuit` documentation, but I will give the equivalent of the gates we have learned so far:

<b>Gate</b>	<b>Qiskit QuantumCircuit class method name</b>
I	<code>iden</code>
X	<code>x</code>
Y	<code>y</code>
Z	<code>z</code>
H	<code>h</code>
S	<code>s</code>

$S^\dagger$	sdg
$T$	t
$T^\dagger$	tdg
CNOT	cx

# Multiqubit gates in Qiskit

Now suppose we want to use `qiskit` to construct a circuit for CNOT using `"/"+>` as the control qubit and `"/"0">` as the target qubit. We will need to create a quantum register to hold two qubits with `qr = QuantumRegister(2)`. We will also need to give each qubit in the register as an argument to the `cx` method of the `QuantumCircuit` class. The first qubit argument to `cx` is the control qubit; the second is the target qubit. The code is as follows:

```
|qr = QuantumRegister(2)
|circuit = QuantumCircuit(qr)
|circuit.h(qr[0])
|circuit.cx(qr[0],qr[1])
```

# Classical registers in Qiskit circuit

We can add a classical register to our quantum circuit. We will need a classical register to hold the output of a measurement. Here is an example of adding a classical register to the circuit for CNOT using `"/"+>` as the control qubit and `"/0">` as the target qubit:

```
|qr = QuantumRegister(2)
|cr = ClassicalRegister(2)
|circuit = QuantumCircuit(qr, cr)
|circuit.h(qr[0])
|circuit.cx(qr[0],qr[1])
```

Here we can see that just like creating an instance of the `QuantumRegister` class requires us to specify the length of the quantum register in qubits, creating an instance of the `ClassicalRegister` class requires us to specify the size of the classical register in bits. Here we can see that initializing a member of the `QuantumCircuit` class with a classical register means that we need to give the `ClassicalRegister` instance as a second argument to the `QuantumCircuit` constructor.

# Measurement in a Qiskit circuit

Now that we have a circuit with a two-qubit quantum register and a two-qubit classical register, we can perform a measurement of all the qubits in the circuit with the `measure` method of the `QuantumCircuit` class. This method takes as input the quantum register to measure as well as the classical register in which to place the result. Here is an example:

```
| qr = QuantumRegister(2)
| cr = ClassicalRegister(2)
| circuit = QuantumCircuit(qr, cr)
| circuit.h(qr[0])
| circuit.cx(qr[0],qr[1])
| circuit.measure(qr, cr)
```

Note that we can also decide to measure just an individual qubit, by specifying which qubit to measure and which bit to put the output result in the following:

```
| qr = QuantumRegister(2)
| cr = ClassicalRegister(2)
| circuit = QuantumCircuit(qr, cr)
| circuit.h(qr[0])
| circuit.cx(qr[0],qr[1])
| circuit.measure(qr[0], cr[0])
```

# Reversible computation

It turns out that any computation you do in a quantum computer can be undone prior to measurement, to get back to exactly what you started with, without knowing what the inputs were. Quantum computing is reversible, which is remarkable because some of the logical operations we might perform in a classical computer can't be reversed.

Consider the classical AND function, which is as follows:

<b>Bit 1</b>	<b>Bit 2</b>	<b>AND(Bit1Bit2)</b>
0	0	AND(00) = 0
0	1	AND(01) = 0
1	0	AND(10) = 0
1	1	AND(11) = 1

If this computation were reversible, we could find an `UNDO_AND` gate that could get us back to where we started, with two bits, from the output bit alone. If we had a function like that, then `UNDO_AND(AND(00))` would equal to 00, `UNDO_AND(AND(01))` would

equal 01,  $\text{UNDO\_AND}(\text{AND}(10))$  would equal 10, and  $\text{UNDO\_AND}(\text{AND}(11))$  would equal 11. Therefore,  $\text{UNDO\_AND}(\text{AND}(11))$  would work because we know the only combination of inputs that produces 1 is 11. But for the other arguments of AND, we'd have no idea how to make the function  $\text{UNDO\_AND}$  because AND(00), AND(01), and AND(11) all produce the same thing—0. So, how is the function  $\text{UNDO\_AND}$  going to decide which of 00, 01, or 11 to return? The answer is we simply can't make an  $\text{UNDO\_AND}$  function. The only way to undo it would be to memorize the inputs and store them elsewhere. The classical AND gate is not reversible; the information about the inputs is not discernible from the output.

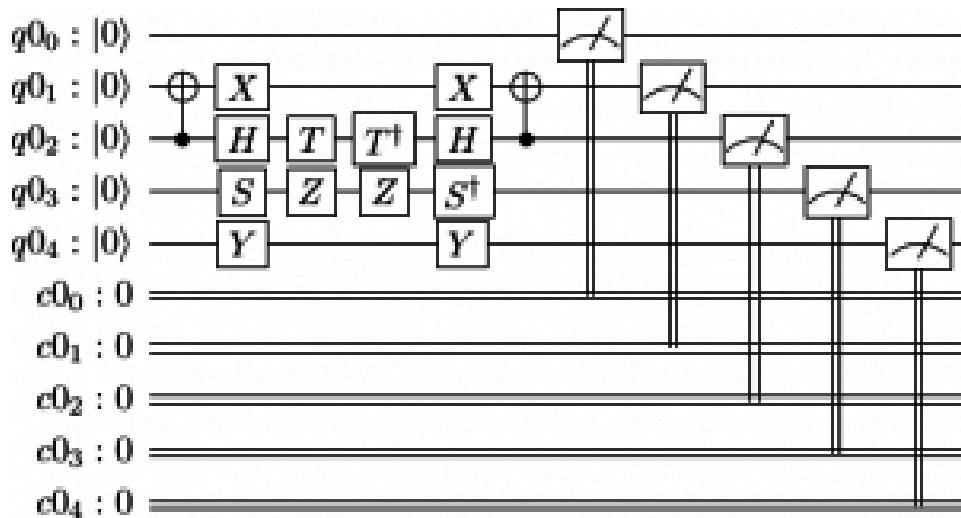
But in quantum computing, every gate is reversible. Depending on the gate, the method of reversing the gate is different. The following table illustrates each gate and how to reverse it:

Gate	Reverse
I	I
X	X
Y	Y
Z	Z
H	H

S	$S^\dagger$
$S^\dagger$	S
T	$T^\dagger$
$T^\dagger$	T
CNOT	CNOT

Here we can note that for most gates we have seen so far, the method to undo or reverse the gate is simply to apply the same gate again. The exceptions being, that for the S and T gates, the undo is the corresponding **dagger gate**  $S^\dagger$  and  $T^\dagger$ , undo is the corresponding gate without the dagger. These methods can be combined to reverse any circuit. When written in a circuit diagram it often then looks as if the reverse circuit is a mirror image of the original. When written in words/algebra it is the same, for example, for a single qubit gate  $XYZS^\dagger T^\dagger HHTSZYX |"0"\rangle = |"0"\rangle$  and  $XYZS^\dagger T^\dagger HHTSZYX |"1"\rangle = |"1"\rangle$ . The mirror starts right between the H gates in this example, and it is an exact mirror except for the cases of S and T, which switch to their dagger variants.

Here is an example of using every one of the gates we have used so far in a reversible circuit over five qubits:



In this example, all the qubits start off at  $|0\rangle$ , so they all end up measuring  $|0\rangle$  and placing 00000 in the classical register, but the circuit would work for any combination of inputs.  $|11111\rangle$  would place 11111 in the classical register,  $|10101\rangle$  would place 10101, and so on. We always get out exactly what we put in because we have followed the prescription for undoing the circuit. This prescription will work for any circuit, no matter how complicated.



*The reversibility is done on a per-qubit basis, so provided that along the qubit's wire there is a perfect mirror, we can conceivably place the reverse gates anywhere along the wire. In the preceding diagram, that means that either one or both  $Y$  gates could move along the wire prior to measurement in any direction, without affecting the result.*

# **Useful quantum circuits**

In this section, we'll go over a couple of useful quantum circuits where we can understand their output just on the basis of what we have learned so far. Later, when we begin to work with more complex quantum algorithms, we will use these circuits as building blocks.

# Using the X gate to prepare any binary input

Imagine we have  $n$  classical bits. Then we have  $2^n$  possibilities for what those bits could be. For  $n = 1$ , the answer is easy, as there are  $2^1 = 2$  possibilities for what the bits could be: 0 or 1. For  $n = 2$  bits, there are  $2^2 = 4$  possibilities for what the bits could be: 00, 01, 10, or 11. For  $n = 3$  bits, we have eight possibilities, and so on. If we have  $n$  qubits there are  $2^n$  possibilities of what measuring those qubits could output.

If we are going to make use of a quantum computer, we will need to give the quantum computer an input, have it compute something dependent on that input, and then extract an output. The input and output will be classical bits, which will need to be encoded as qubits. We do this by making sure that, for the example of an input consisting of two qubits, we would have four possibilities. Instead of 00, 01, 10, or 11, these are the states:  $|00\rangle$ ,  $|01\rangle$ ,  $|10\rangle$ , and  $|11\rangle$ .

We've seen that in the quantum circuits we have worked with, we always start at a state  $|0\rangle$  for each of the qubits. How then do we get the states containing  $|1\rangle$ ? This is easy, if we recall that the X gate acting on  $|0\rangle$  or  $|1\rangle$  simply swaps  $|0\rangle$  for  $|1\rangle$  and  $|1\rangle$  for  $|0\rangle$ . Using the X gate, we can then prepare an input with  $|1\rangle$  exactly where we want it by applying X to the  $|0\rangle$  qubit at that location. Here are some examples:

- $_{00}$  gets encoded as  $|"00">$  with this:

$$\begin{array}{c} |0\rangle \\ |0\rangle \end{array} \quad \text{---}$$

- $_{01}$  gets encoded as  $|"01">$  with this:

$$\begin{array}{c} |0\rangle \\ |0\rangle \end{array} \quad \begin{array}{|c|} \hline X \\ \hline \end{array}$$

- $_{10}$  gets encoded as  $|"10">$  with this:

$$\begin{array}{c} |0\rangle \\ |0\rangle \end{array} \quad \begin{array}{|c|} \hline X \\ \hline \end{array}$$

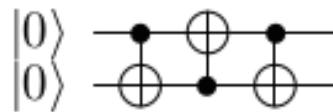
- $_{11}$  gets encoded as  $|"11">$  with this:

$$\begin{array}{c} |0\rangle \\ |0\rangle \end{array} \quad \begin{array}{|c|c|} \hline X \\ \hline X \\ \hline \end{array}$$

This trick works for registers of any qubit size. Simply apply the  $X$  to any qubit in the input, which should be a  $|"1">$  but currently is a  $|"0">$ . After your input is ready, the computation can be performed, and measurement can then place the output in a classical register.

# Swapping two qubits

This circuit swaps two qubits:



That means if we started out in the  $|01\rangle$  state, we would end up in the  $|10\rangle$  state and vice versa. Inputs of  $|00\rangle$  or  $|11\rangle$  would be unchanged as swapping would have no effect. How does this gate work? Note that for the first and third CNOTs, the control qubit is the first qubit and the target qubit is the second, while for the second CNOT, the control qubit is the second and the target qubit is the first. Let's trace it out step by step:

<b>Input</b>	<b>After first CNOT</b>	<b>After second CNOT</b>	<b>After third CNOT (output)</b>
$ 00\rangle$	$ 00\rangle$	$ 00\rangle$	$ 00\rangle$
$ 01\rangle$	$ 01\rangle$	$ 11\rangle$	$ 10\rangle$
$ 10\rangle$	$ 11\rangle$	$ 01\rangle$	$ 01\rangle$
$ 11\rangle$	$ 10\rangle$	$ 10\rangle$	$ 11\rangle$



*It's straightforward to see the swap when the two input qubits are in a state that is either 100%  $|1\rangle$  or 100%  $|0\rangle$  as we have seen here, but it is important to note that the swap will work just as well with qubits where one or both are instead a mix (superposition) between  $|1\rangle$  and  $|0\rangle$ . For example, if initially the first qubit is 90%  $|1\rangle$  and 10%  $|0\rangle$ , and the second qubit is 50%  $|1\rangle$  and 50%  $|0\rangle$ , then after the swap circuit is executed, the first qubit will be 50%  $|1\rangle$  and 50%  $|0\rangle$ , and the second qubit will be 90%  $|1\rangle$  and 10%  $|0\rangle$ .*

# Summary

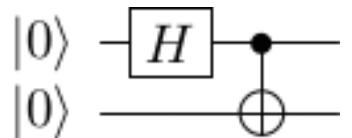
A quantum circuit is merely a sequence of quantum gates. A *quantum circuit diagram* is a method to visualize quantum circuits. Typically, on the left-hand side of a quantum circuit diagram, we'll start with a series of  $|0\rangle$  qubits with lines coming out of them, which look a bit like wires. Qiskit is the Quantum Information Science Kit. It is an SDK for working with the IBM QX quantum processors. In this chapter, we learned how to use Qiskit to write Python code to represent a variety of quantum circuits.

It turns out that any computation you do in a quantum computer can be undone prior to measurement, to get back exactly what you started with, without knowing what the inputs were. Quantum computing is reversible. For most gates we have seen so far, the method to undo or reverse the gate is simply to apply the same gate again. For the *S* and *T* gates, the undo is the corresponding **dagger gates**,  $S^\dagger$  and  $T^\dagger$ . For  $S^\dagger$  and  $T^\dagger$ , the undo is the corresponding gate without the dagger, *S* and *T*. These methods can be combined to reverse any circuit. Using the *X* gate, we can then prepare an input with  $|1\rangle$  exactly where we want it by applying *X* to the  $|0\rangle$  qubit at that location. In this way, we can produce any qubit combination corresponding to a sequence of classical bits.

In the next section of the book, we will move on to programming directly for the IBM QX. The next chapter introduces the IBM QX hardware and software.

# Questions

1. Draw the quantum circuit for  $S^{\dagger}/|-\rangle$ .
  2. Draw the quantum circuit for measuring  $S^{\dagger}/|-\rangle$ .
  3. Give labels to your qubits and bits in the circuit for measuring  $S^{\dagger}/|-\rangle$ .
  4. Write a Python program using `qiskit` to generate the circuit for  $S^{\dagger}/|-\rangle$ .
  5. Write a Python program using `qiskit` to measure a CNOT gate acting with a control qubit of  $S^{\dagger}/|-\rangle$  and a target qubit of  $|0\rangle$ .
  6. Reverse the computation  $XXYSZTS^{\dagger}HHTS |+\rangle$  in words/algebra.



7. Draw a quantum circuit that reverses [0]  .

8. Prepare the  $|01011101\rangle$  state in an 8-qubit quantum circuit. Then perform, either in code in words/algebra on paper, a computation on the state using the gates of your choice. Next add gates to reverse your computation. Finally, imagine you measure each qubit in the  $|0\rangle$  and  $|1\rangle$  basis and place the result in a classical register. What would this register contain?

# The Quantum Composer

This chapter is all about the Quantum Composer, which you will recognize as an interactive interface in IBM QX to create quantum circuits via quantum scores, the graphical manner of representing circuits introduced earlier. Through the Quantum Composer, you can define your own circuits for implementation on the IBM QX hardware or software simulator. The chapter proceeds to translate many of the examples previously coded in Python to the Quantum Composer representation and gives you the opportunity to run these on IBM QX hardware. Finally, we will perform runs in simulation and, optionally, on a real quantum computer.

The following topics will be covered in this chapter:

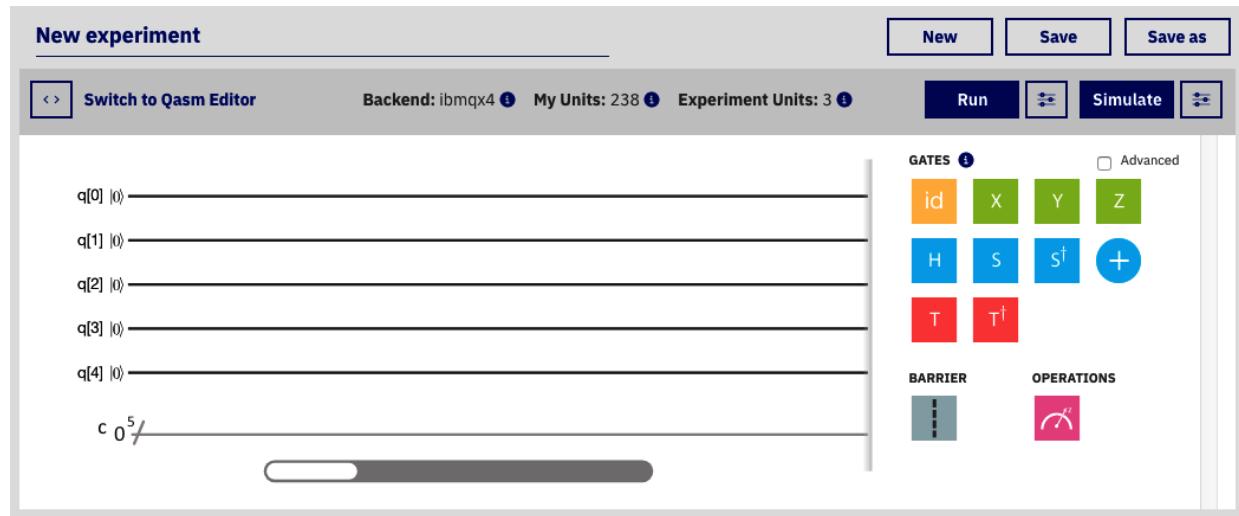
- The Quantum Composer
- Translating quantum circuits into the Quantum Composer
- Running in simulation or hardware from the Quantum Composer

# Technical requirements

You will need a modern web browser and the ability to sign into IBM QX (<https://quantumexperience.ng.bluemix.net/>) as we set up in [Chapter 1, What is Quantum Computing?](#).

# The Quantum Composer

In [Chapter 1](#), *What is Quantum Computing?* we set up a login for the IBM QX. Once we log in, we have a variety of options. Click on the Composer tab to see the Quantum Composer, and click New to create a new score, at which point, you will be allowed to choose which of IBM's quantum computers you will want your code to run on. I chose `ibmqx4`. We will work with a 5-qubit computer or simulator throughout this book; they are big enough to illustrate big concepts but small enough to follow their operations easily. IBM has devices with more qubits available, and every skill acquired from this book will transfer to running on those devices as well. The Composer tab is shown in the following screenshot:



On the left-hand side, we see five named qubits, numbered starting from zero:  $q[0]$ ,  $q[1]$ ,  $q[2]$ ,  $q[3]$ , and  $q[4]$ . Each of these is initialized to be in the  $|0\rangle$  state and has a wire coming out where we can place gates, barriers, and operations:



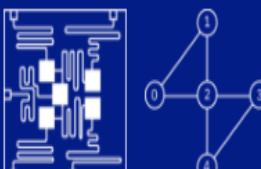
**i** Note that in IBM's notation the quotation marks clarifying that "0" is only a label for the state and not a number itself are dropped, and state  $|"0"\rangle$  is written as  $|0\rangle$ . This style of notation is more common in quantum computing, and hopefully the book has trained you to recognize whatever is inside  $|>$  as a label, not as a number.

On the bottom we see a c, indicating that it is a classical register, a  $0^5$  indicating that the register contains five bits all initialized to zero, and a wire with a line through it indicating that the register contains multiple bits. When we perform a measurement within the Quantum Composer, the measurement will draw a line down to the classical register, indicating that the measurement is stored there.

# Hardware

Here, we see that this Quantum Score can be run on `ibmqx4`. IBM provides details as to the current state of the hardware you will run on:

IBM Q 5 Tenerife [ibmqx4] ACTIVE: USERS



	Q0	Q1	Q2	Q3	Q4
<b>Frequency (GHz)</b>	5.24	5.31	5.35	5.41	5.19
<b>T1 (μs)</b>	50.80	56.30	42.00	33.10	52.30
<b>T2 (μs)</b>	13.90	57.70	49.90	15.30	26.20

Last Calibration: 2018-06-28 02:59:35  
Fridge Temperature: [] -

**More details**

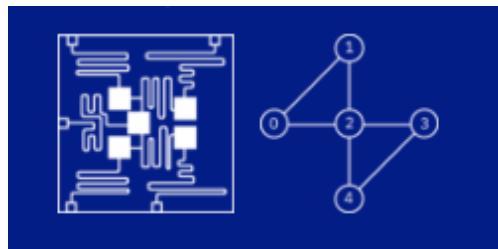
	Gate error ( $10^{-3}$ )	Readout error ( $10^{-2}$ )	CX1_0	CX2_0	CX3_2	CX4_2
<b>Gate error (<math>10^{-3}</math>)</b>	0.77	1.63	1.20	3.01	0.94	
<b>Readout error (<math>10^{-2}</math>)</b>	6.40	6.10	6.00	11.00	4.90	
<b>MultiQubit gate error (<math>10^{-2}</math>)</b>	2.82	2.76	11.93	5.68		
			<b>CX2_1</b>	<b>CX3_4</b>		
			2.71	8.61		

Here, we can see various calibration parameters of each qubit on the right, including the T1 and T2 parameters we studied in [Chapter 3, Quantum States, Quantum Registers, and Measurement](#):

	Q0	Q1	Q2	Q3	Q4
<b>Frequency (GHz)</b>	5.24	5.31	5.35	5.41	5.19
<b>T1 (μs)</b>	50.80	56.30	42.00	33.10	52.30
<b>T2 (μs)</b>	13.90	57.70	49.90	15.30	26.20
<b>Gate error (<math>10^{-3}</math>)</b>	0.77	1.63	1.20	3.01	0.94
<b>Readout error (<math>10^{-2}</math>)</b>	6.40	6.10	6.00	11.00	4.90
<b>MultiQubit gate error (<math>10^{-2}</math>)</b>	2.82	2.76	11.93	5.68	
			<b>CX2_1</b>	<b>CX3_4</b>	
			2.71	8.61	

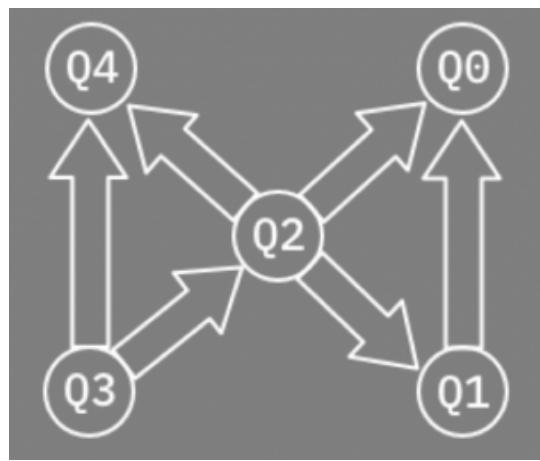
Remember that T1 and T2 quantify how long a quantum computer can maintain coherence for, and the bigger T1 and T2 the better, as that means we can perform more computations over a longer period of time before expecting errors in the computation from decoherence.

We also see a diagram sketching the computer's setup, a chip diagram, on the left as well as a connection diagram on the right:



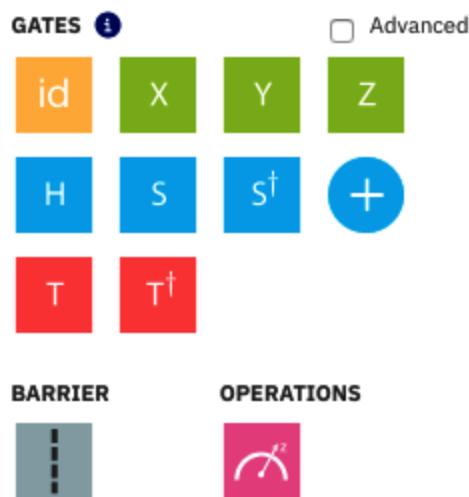
This connection diagram is a new concept, as T1 and T2, relevant to the physical implementation of a quantum computer. When we learned about the two-qubit gate, CNOT, we were using the gate in simulation via the Python code we wrote. Any qubit could serve as the control qubit, and any qubit could serve as the target qubit. In an ideal scenario, this would hold for a physical quantum computer as well. In practice, it is difficult for every qubit to remain connected to every other qubit in a physical implementation. Depending on the hardware setup, this means often the designers limit which qubits can serve as control qubits to any given qubit. This connection diagram depicts this.

The diagram is more schematic as it doesn't have an arrow from control qubit to target qubit. The lines in the diagram are not bidirectional. To fully use the diagram, we would need to know the direction of the gate. To do this, IBM releases more information on their GitHub, where we can see in the release notes that the `ibmqx4` quantum computer at this stage has a connection diagram like this:



# Gates, operations, and barriers

Let's go back to our quantum score. On the right of the quantum score, we see the gates we have learned about so far:



Here, **id** is the identity gate I and the gate marked **+** is the CNOT gate. All other gates are labeled consistent with the labels we have studied so far. For the CNOT gate, we will

place the **+** on the qubit we wish to be the control qubit, and a line will appear that we can drag to the qubit we wish to be the target qubit. All the other gates are single qubit gates, so we simply drag them to the qubit we wish the gate to act on.

There is a section called operations, with one item. The **⊗** operation performs a measurement. For the measurement

operation, we will place  on the qubit we wish to measure. The z in the box reminds us we are going to perform the measurement in the z-basis (the standard basis also known as the  $|0\rangle$  and  $|1\rangle$  basis). Any measurement we perform will then produce either a  $|0\rangle$ , which we can store as a bit 0 in our classical register, or a  $|1\rangle$ , which we can store as a bit 1 in our classical register. In principle, a quantum computer could implement and provide an interface to measure in another basis set; IBM at this time does not, which makes things simple for us.

One item appears that we have not seen before, the

 . This prevents transmissions across the line, which doesn't affect the theoretical result of the circuit but may affect the running time or accuracy (as a circuit running for more time is less accurate). This is because the barrier's function is to prevent any backend tools from optimizing the circuit over that barrier. For example, we learned in the last chapter that to reverse the X gate, we simply apply another X gate. Then if an optimizer saw the following circuit (which applies the X gate twice to the  $|0\rangle$  qubit residing in the q quantum register at index 0):



It could turn it into the following, which applies the identity gate to the same qubit, as applying the X gate twice is equivalent to the identity gate:



It could be nothing at all, as the identity gate is equivalent to taking no action, prior to running:

q[0] |0> —————

An efficient optimizer wouldn't put a gate there at all, as even the I gate takes time to run on a quantum computer. However, this may not be what we as a programmer want. Perhaps we are testing something, for example, trying to probe T1 or T2, where we don't want the circuit to be changed before being run at all. In that case we could put a barrier between the two X gates:

q[0] |0> — 

This ensures no optimizer will be able to group the two into an optimization. We will not use the barrier much in this book, but it is good to know what it is when you see it. To use the barrier in the Quantum Score interface, drag it to the qubit you want the barrier to begin at, and then drag the line to the qubit you want the barrier to end at.

Gates, operations, and barriers can be deleted by dragging a gate from the score to the upper left of the score:

 Delete 

q[0]  $|0\rangle$  —————●————●————

q[1]  $|0\rangle$  —————●————●————

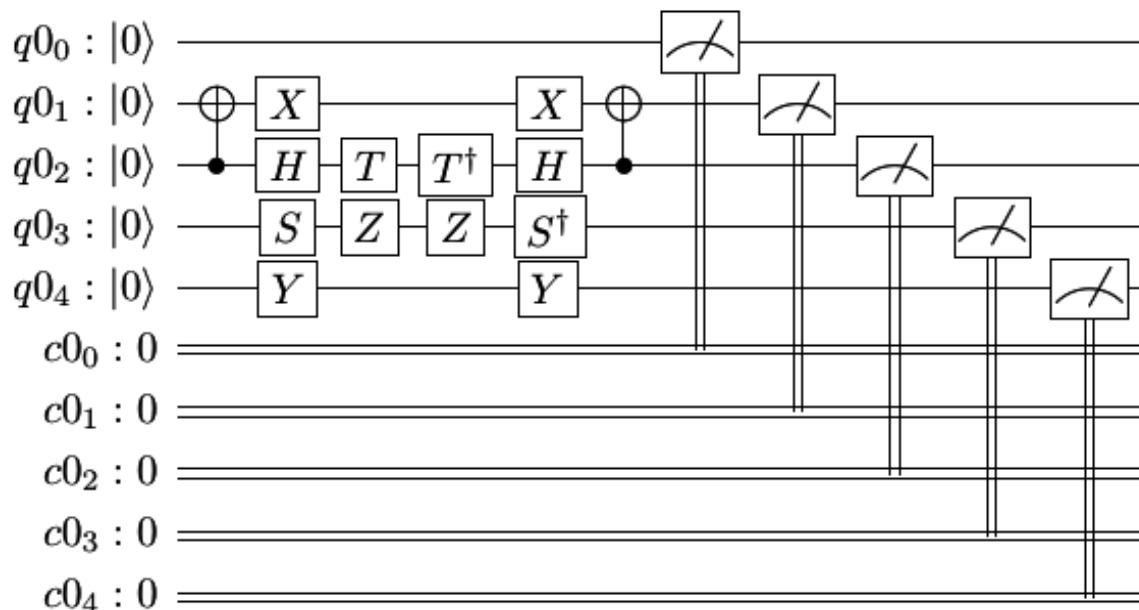
q[2]  $|0\rangle$  —————●————●————

q[3]  $|0\rangle$  —————●————●————

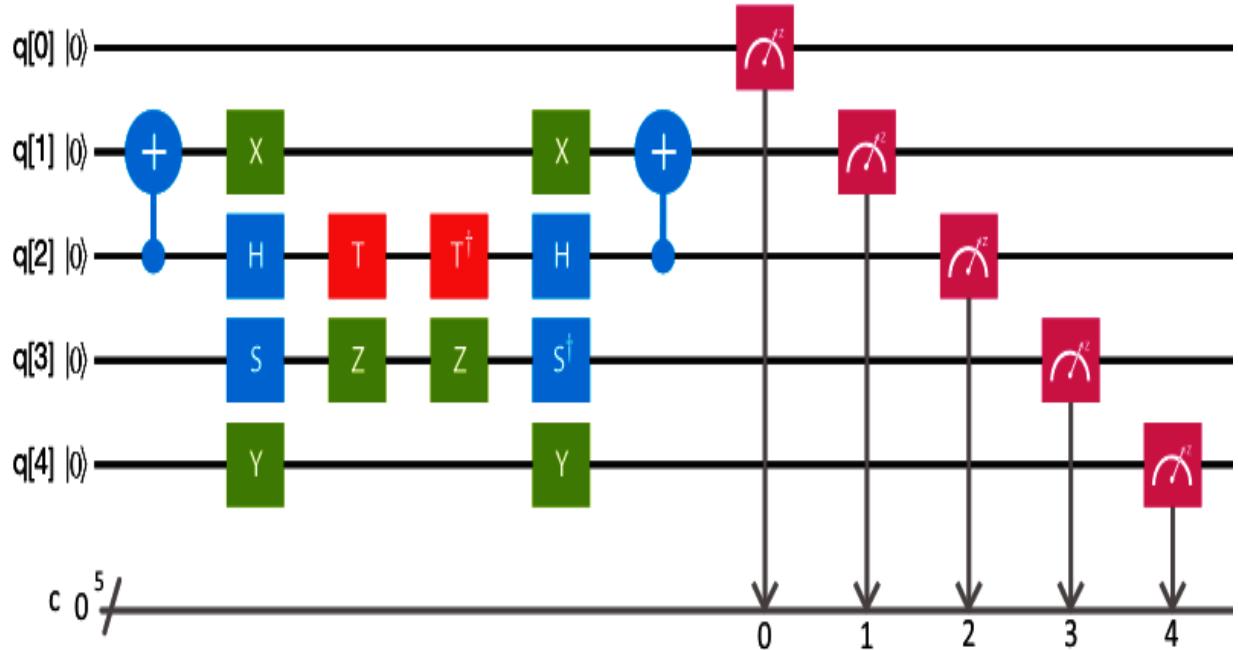
q[4]  $|0\rangle$  —————●————●————

# Translating quantum circuits into the Quantum Composer

In this section, we will take existing quantum circuits and implement them in the Quantum Composer. In the previous chapter, we dealt with a circuit:

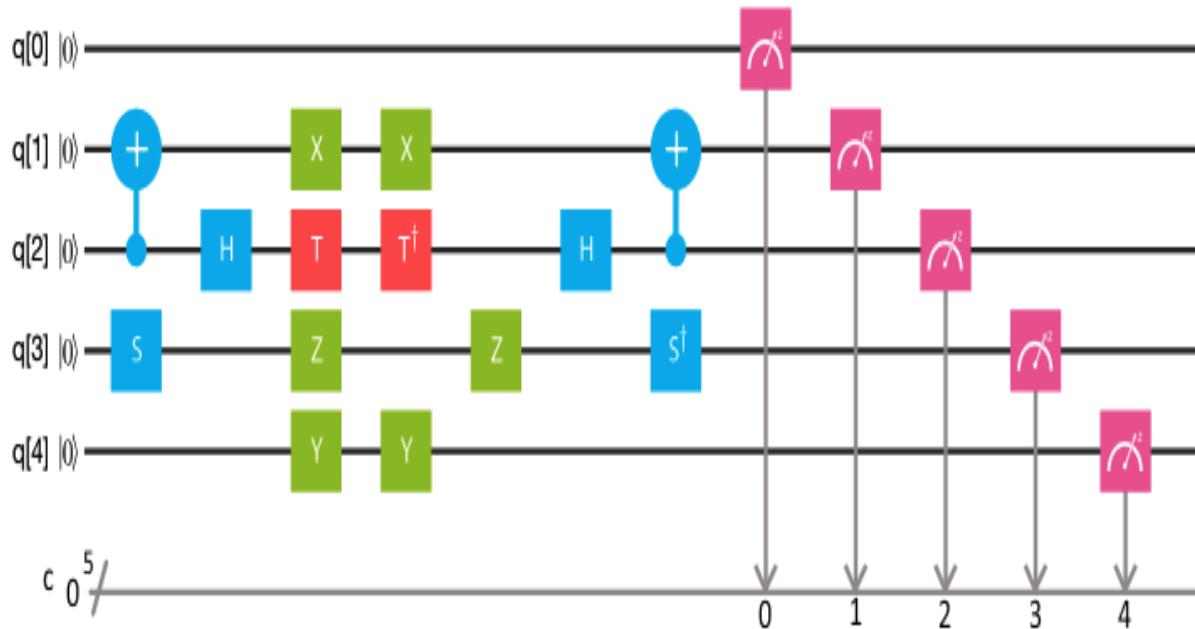


Here we see that it's easy to implement this circuit using the Quantum Composer:

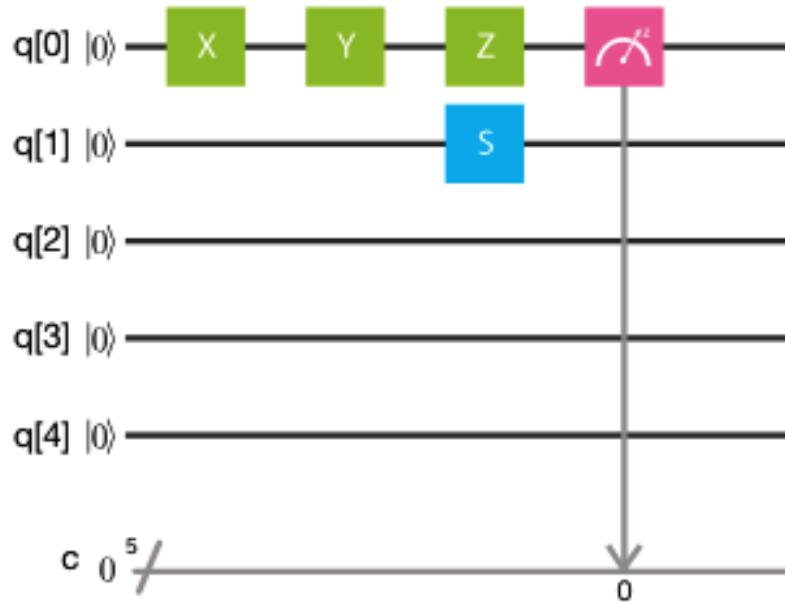


From a given picture of a circuit to implementing it in the Quantum Composer, most things are similar. The notation to label each qubit might vary, as will the notation for the classical register. However, the gates and measurement operations will stay the same. All that might vary is the exact symbol used to label the gate.

We also have the option to space gates, barriers, and operations out or put them closer together on a given wire, as long as their order doesn't change. The computation will remain the same. For example, the following circuit is exactly the same as the previous one:



This example uses all five qubits of the quantum computer, but we aren't obligated to use all of the qubits. We do need at least one measurement operation to simulate the circuit or to run on the IBM hardware. Without at least one measurement operation, there will be no output, so the run will have no point. Note that not every wire in the quantum circuit needs a measurement gate, just at least one. Here's a circuit in the Quantum Composer that illustrates that:



In these examples, our input has always been the classical five-bits of zeros "00000" translated into five qubits. These can be thought of together, as  $|"00000">\rangle$ , and at this point they are also separable into five separate qubits  $|"0"\rangle$ ,  $|"0"\rangle$ ,  $|"0"\rangle$ ,  $|"0"\rangle$ , and  $|"0"\rangle$ . If we had any ones in our input, for example, "10000", we would need to place the gates to prepare the input in a state of  $|"10000">\rangle$ , before we placed the gates for our circuit. We learned in the last chapter that the  $X$  gate flips state  $|"0"\rangle$  to state  $|"1"\rangle$ . So all we need is to place an  $X$  gate before any other gate on a wire where the qubit input should be 1. For the example of  $|"10000">\rangle$  the circuit then begins:

$q[0] |0\rangle$  — 

$q[1] |0\rangle$  —————

$q[2] |0\rangle$  —————

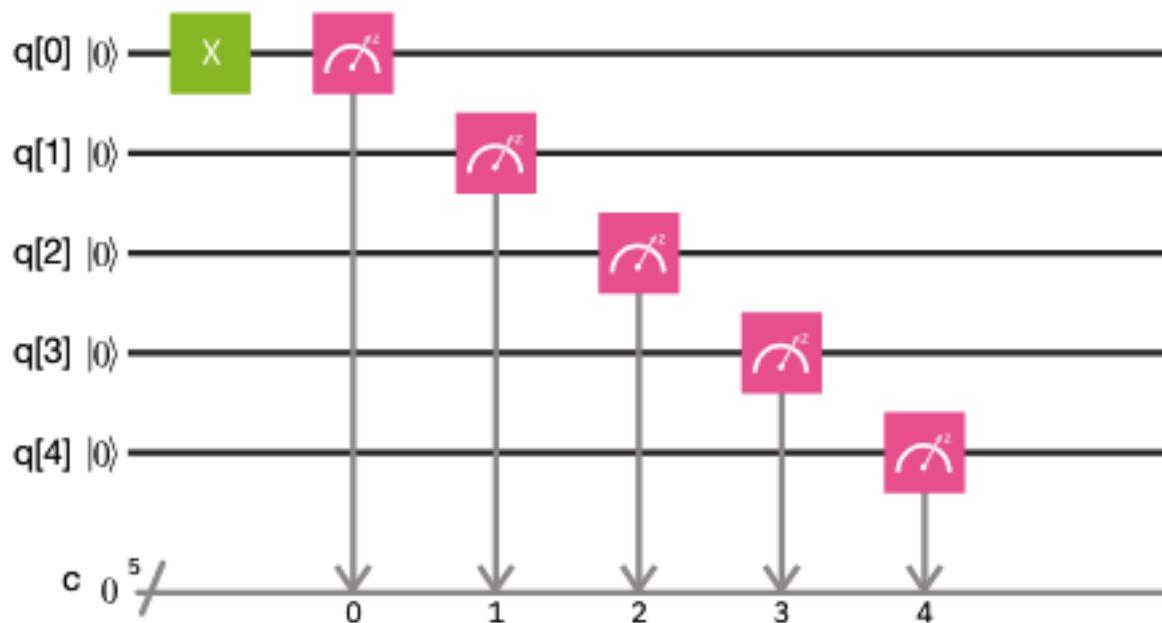
$q[3] |0\rangle$  —————

$q[4] |0\rangle$  —————

c 0  —————

# Executing quantum circuits in simulation or hardware from the Quantum Composer

Now comes the fun part. In this section, we will get to run our first program on a quantum computer. Let's test the circuit preparing state  $|10000\rangle$  that we just went over. Create a new quantum circuit in the Quantum Composer and put an X gate on the first qubit. Then put a measurement gate on each wire. Your circuit should look like this:



# Executing a quantum circuit in simulation

Now, click the Simulate button. Give your experiment the name "10000". The default simulation parameters, editable next to the Simulate button, specify that the circuit will be run 100 times. Remember that in quantum computing, each time we prepare an input and run an identical circuit, we may get different results as, for measurements, there is a certain probability of 1 and a certain probability of 0. Therefore, any one run might not tell us much about the probability of the output. Luckily, either in simulation or on the real quantum computer hardware, we can choose to reset, set up the qubits again, and run the same circuit on those qubits. With enough runs, we will be able to infer the probability of each output in the first place. This should be easy in our case with this circuit, as we expect that the qubits labeled  $q[1]$ ,  $q[2]$ ,  $q[3]$ , and  $q[4]$  should be 0 after measurement with 100% probability, and the qubit labeled  $q[0]$  should be 1 after measurement with 100% probability.

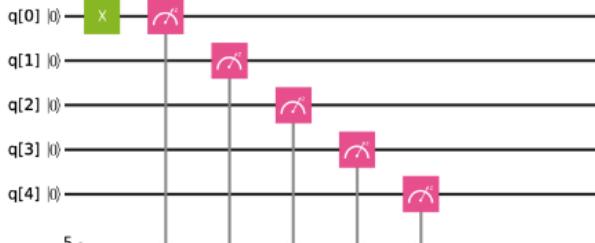
Depending on whether resources are currently available, a window might pop up that the execution is pending. In that case, you can check on the execution under the Quantum Scores section. You may need to click **Refresh** and expand your circuit "10000" to see whether your execution is planned or ready. Here you can see on the right that I have two past completed execution and one execution pending:

## Quantum Scores (109 scores)

[Refresh](#)[Remove All](#)

Pages 1 2 3 4 5 6 7 8 9 10 11

▼ 10000 v1

[Add a description](#)[Save](#)

### Executions

Jun 30, 2018 10:55:46 PM



Jun 30, 2018 10:39:12 PM



Jun 30, 2018 10:37:48 PM



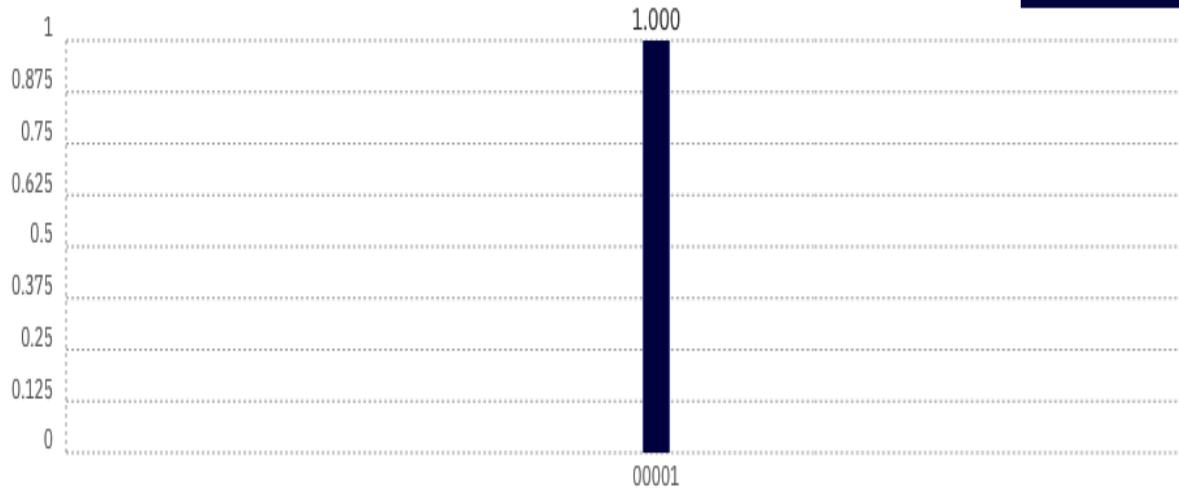
When the results are ready, a large window will pop up:

 10000

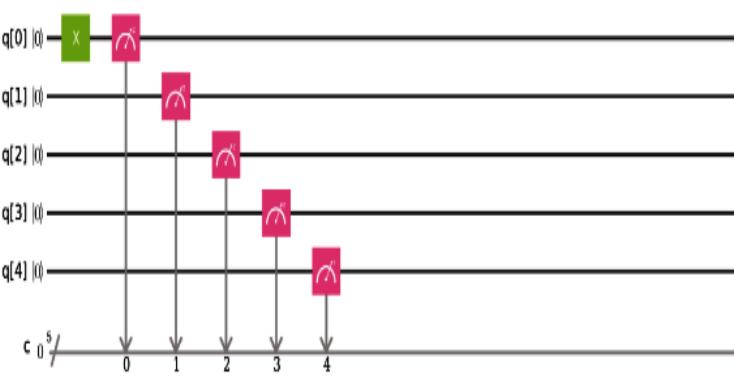
Device: Simulator

## Quantum State: Computation Basis

[Download CSV](#)



## Quantum Circuit



A quantum circuit diagram with five qubits (q[0] to q[4]) and one classical register c[0]. The circuit consists of two main parts: a sequence of CNOT gates between adjacent qubits followed by a sequence of single-qubit rotation gates (R\_y) on each qubit. The controls for the CNOT gates are labeled 0, 1, 2, 3, and 4 below the wires. The rotation angles are also labeled 0, 1, 2, 3, and 4 below the rotation symbols.

**OPENQASM 2.0**

```
1 include "qelib1.inc";
2
3 qreg q[5];
4 creg c[5];
5
6 x q[0];
7 measure q[0] -> c[0];
8 measure q[1] -> c[1];
```

[Open in Composer](#)

[Edit in QASM Editor](#)

Note that, although it's a bit confusing the results are listed in the opposite order with the qubit labeled q[0] appearing last, and the qubit labeled q[4] first, so the  $_{10000}$  that we expect is written as 00001:



I will underline the bits when I discuss reading the measurement output from the IBM interface to emphasize that we must remember they are in the opposite order from q[0], q[1], q[2], q[3], and q[4] and, instead, go from the highest numbered qubit to the lowest. It is unfortunate that IBM chose this convention, as it easily leads to confusion.

The number on top is the fraction of the total number of runs, in this case 100, which returned the bits on the bottom, for example,  $100/100$  or 1.000 fraction of runs, that is, 100% of runs returned 00001. This is exactly as we expect.

# Executing a quantum circuit on quantum computing hardware

Let's run our circuit on real hardware! We have learned from simulation, from our studies of quantum gates, and from writing quantum scores, what to expect from the output of this circuit. We expect  $|10000\rangle$  to be the output. But we also know as we learned in [Chapter 3, Quantum States, Quantum Registers, and Measurement](#), that a real quantum computer is subject to noise. That means that while the vast majority of runs on the quantum computer hardware will output  $|10000\rangle$ , we expect a percentage of them to output something else. Since the circuit is small, it won't take long to execute, and there is not a lot of time for errors to occur. So the percentage of runs that output something other than  $|10000\rangle$  should be small.

Click Run to run the circuit on real hardware. If your circuit has been recently executed by another user, you may then get the option to use their cached results. Don't do that for this example, because we want to run it for ourselves. By default the circuit will be run on identically prepared qubits 1,024 times. This number can be changed via the button next to Run. Run your circuit. Again, as in simulation, you may have to wait and refresh to see when your results are ready. You should also receive an email when the results are ready. This may take a while as there are sometimes users ahead of you in the queue to run on the hardware.

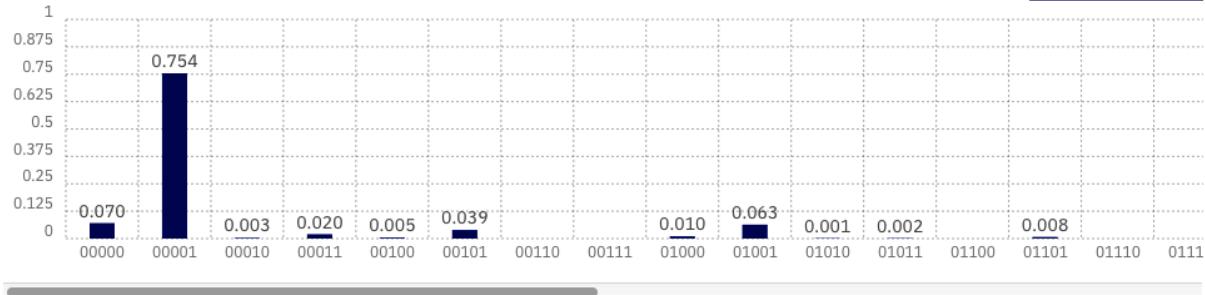
Let's see what the results are:

## ⚙️ 10000

Device: ibmqx4

Quantum State: Computation Basis

[Download CSV](#)



This is only a portion of the results that fits on screen. Here we see that the vast fraction of the 1,024 runs, 0.754 or 772/1024 runs returned what we expected: 00001, or rewriting in the correct order, 10000. Since we are running on real hardware, each time this percentage may vary. The rest of the 252 runs returned something different, usually off from what we expect by just one bit although not always.

At the bottom of the output, IBM shares the device calibration at the time of the run, including the values for T1 and T2 for each qubit. This will help you get a sense how qubits are prone to error, as the smaller these values, the more error prone a qubit is.

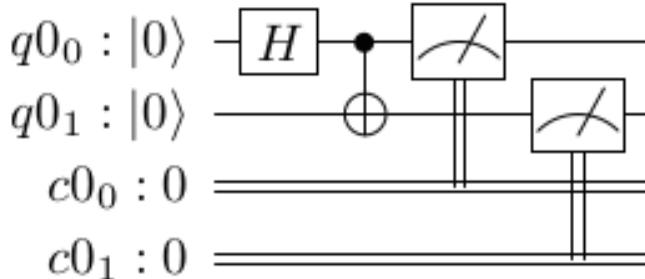
# Summary

In this chapter, we learned about the Quantum Composer translating quantum circuits into the Quantum Composer and running Quantum Composer scores in simulation or hardware. In the score, every qubit starts in the `/"0">>` state with a wire coming out of it, where gates can be placed to evolve the state. To flip the state to `/"1">>` prior to input to the quantum circuit, we can execute the X gate. To use a single qubit gate, simply drag it to the qubit we wish the gate to act on. Running in simulation or hardware is easy. Just click on the **Run** or **Simulate** buttons respectively.

In the next chapter, we will cover working with Open Quantum Assembly Language (Open QASM, pronounced "open kazm"), a custom programming language designed specifically to minimally describe quantum circuits.

# Questions

1. Use the Quantum Composer to implement the following



circuit:

2. Use the Quantum Composer to implement the  $XYZS^{\dagger}T^{\dagger}HHTSZY X |"0"\rangle$  circuit.
3. Use the Quantum Composer to create an input corresponding to the five "01010" classical bits.
4. Make a circuit of your own devising, where you know with 100% certainty what a simulation should return. Run the simulation.

# Working with OpenQASM

This chapter describes the **Open Quantum Assembly Language (OpenQASM)**, pronounced *open kazm*, a custom programming language designed specifically to minimally describe quantum circuits. It goes over the motivation behind introducing this language, where it can be used within IBM QX and other quantum computing work it can be used for. The chapter revisits some of the quantum circuits defined in previous chapters and redefines them within the OpenQASM language. It then provides you with the opportunity to rerun these circuits on the IBM QX, using OpenQASM directly, instead of the Quantum Composer. The chapter focuses on the specifications and implementations of OpenQASM, which is the version of OpenQASM used by the IBM QX at the time of writing.

The following topics will be covered in this chapter:

- Reading and writing OpenQASM programs
- Presenting quantum scores in OpenQASM programs
- Translating OpenQASM programs into quantum scores

# Technical requirements

You will need a modern web browser and the ability to sign into IBM QX (<https://quantumexperience.ng.bluemix.net/>) as we set up in [Chapter 1, What is Quantum Computing?](#).

# OpenQASM

The Quantum Composer is a tool to specify quantum programs graphically, and many SDKs and APIs exist to write compute code to represent a quantum program in a modern language (Python being a common choice). Like the Quantum Composer, OpenQASM is a higher-level language for specifying quantum programs than computer code, but unlike the Quantum Composer, it is neither graphical nor user interface specific, so it can be much easier to specify longer programs that can be directly copied in to the many quantum simulators or into IBM QX for use. The Quantum Composer can take as input, programs in OpenQASM, and translate them into the graphical view. Likewise, for every program specified in the Quantum Composer it is easy to access the equivalent in OpenQASM within the IBM QX user interface.

Just as a book on Python cannot be expected to outline every feature of Python, the scope of our description of OpenQASM will be limited and focused on OpenQASM 2.0. In this chapter, we will look in detail just at the portion of its syntax relevant to interfacing with IBM QX for the programs and algorithms described in this book. Other features, such as declaring custom gates, are beyond the scope of this book, but can be found in the full specifications of OpenQASM 2.0 (<https://github.com/Qiskit/openqasm/blob/master/spec/qasm2.rst>) or are briefly touched upon in the final section of this chapter.

OpenQASM is similar in syntax to C:

- Comments are one per line and begin with `//`
- White space **isn't** important

- Case **is** important
- Every line in the program must end in a semicolon ;

Additionally, the following points apply:

- Every program must begin with `OPENQASM 2.0;` (this book and IBM QX at the time of writing uses version 2.0, but this can be updated to whichever version of OpenQASM you are using).
- When working with IBM QX, the `include "qelib1.inc";` header must be given. Any other file can be included with the same syntax; what OpenQASM does is simply copies the content of the file at the location of include. The path to the file is a relative path from the current program.

Reading and writing OpenQASM 2.0 programs for the IBM QX will involve the following operations:

Include header	<code>include "qelib1.inc";</code>
Declaring a quantum register ( <code>qregname</code> is any name you choose for the quantum register)	<code>qreg qregname[k];</code>
Referencing a quantum register	<code>qregname[i];</code>
Declaring a classical register  ( <code>cregname</code> is any name you choose for the quantum register)	<code>creg cregname[k];</code>
Referencing a classical register	<code>cregname[i];</code>

One-qubit gate list, available with inclusion of <code>qelib1.inc</code> on IBM QX	<code>h, t, tdg, s, sdg, x, y, z, id</code>
One-qubit gate action syntax	<code>gate q[i];</code>
Two-qubit CNOT gate list, available with inclusion of <code>qelib1.inc</code> on IBM QX	<code>cx</code>
Two-qubit CNOT gate action (control and target both qubits in a previously declared quantum register)	<code>cnot control, target;</code>
Measurement operations available	<code>measure, bloch</code>
Measurement operation action syntax	<code>measure q[i] -&gt; c[j];  bloch q[i] -&gt; c[j];</code>
Barrier operation ( <code>args</code> are a comma-separated list of qubits)	<code>barrier args;</code>
Primitive gates (OpenQASM standard but not used on IBM QX or within this book)	<code>CX, U</code>

I will not go over each operation in detail in this section. Rather, in the following sections, we will learn by doing and will practice reading OpenQASM programs and translating them into quantum scores as well as translating quantum scores to OpenQASM programs.



*Note that  $i$  and  $j$  are integer counters, starting at 0, which specifies which qubit/bit in the quantum or classical register the program would like to reference;  $k$  is an integer counter greater than 0 which specifies the size of of a classical or quantum register at declaration*

# **Translating OpenQASM programs into quantum scores**

Later in this chapter, there is a section where we will work with IBM QX to automatically translate an OpenQASM program into a quantum score. In this section we will do the translation by hand to practice reading OpenQASM code.

# OpenQASM to negate one qubit

Consider the following program:

```
|OPENQASM 2.0;  
|include "qelib1.inc";  
|qreg q[1];  
|x q[0];
```

The following lines are the standard headers for working with IBM QX:

```
|OPENQASM 2.0;  
|include "qelib1.inc";
```

Then the following line declares a quantum register of size one named `q`:

```
|qreg q[1];
```

Quantum registers are automatically initialized to contain `/"0">`.

Finally, the next line operates the X gate on the first (and only) qubit in the `q` quantum register:

```
|x q[0];
```

Putting this all together, we can create the following equivalent quantum score:

$q[0] |0\rangle$  —  —

$q[1] |0\rangle$  —————

$q[2] |0\rangle$  —————

$q[3] |0\rangle$  —————

$q[4] |0\rangle$  —————

$c_0^5$  —————

# OpenQASM to apply gates to two qubits, and measure the first qubit

Next consider the OpenQASM program:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
creg c[1];
x q[0];
y q[0];
z q[0];
s q[1];
measure q[0] -> c[0];
```

The first two preceding lines are the standard header to declare a program and OpenQASM program and the standard import header to interface with the IBM QX. The next two lines declare a quantum register of two qubits initialized to  $|00\rangle$  and a classical register of one bit initialized to 0:

```
|qreg q[2];
|creg c[1];
```

The next three lines apply gates, in order, to the first qubit in the `q` quantum register:

```
|x q[0];
|y q[0];
|z q[0];
```

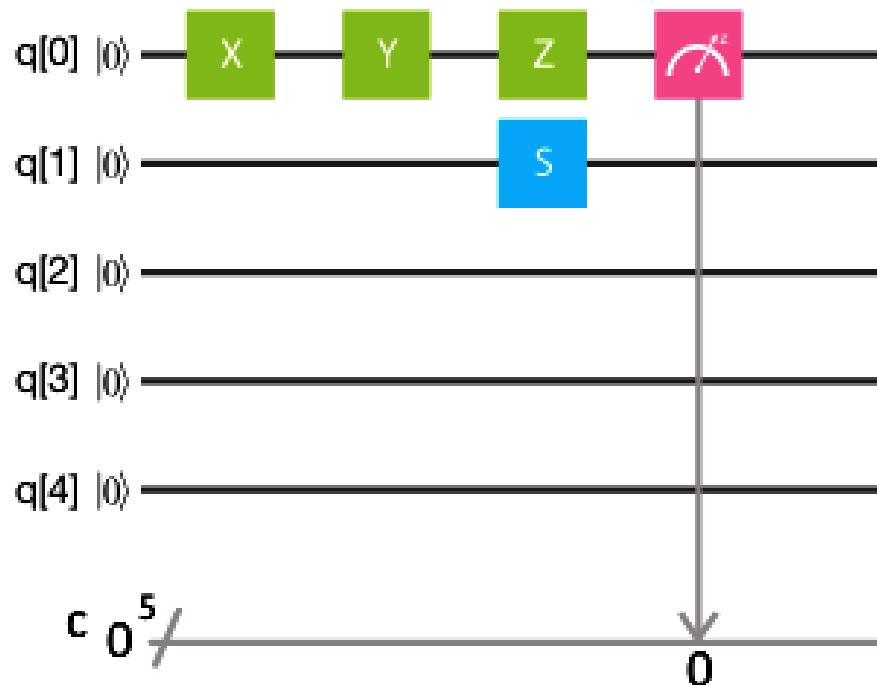
The next line applies a gate to the second qubit in the `q` quantum register:

```
|s q[1];
```

And the final line measures the first qubit in the `q` quantum register and places the result in the first (and only) bit in the `c` classical register:

```
|measure q[0] -> c[0];
```

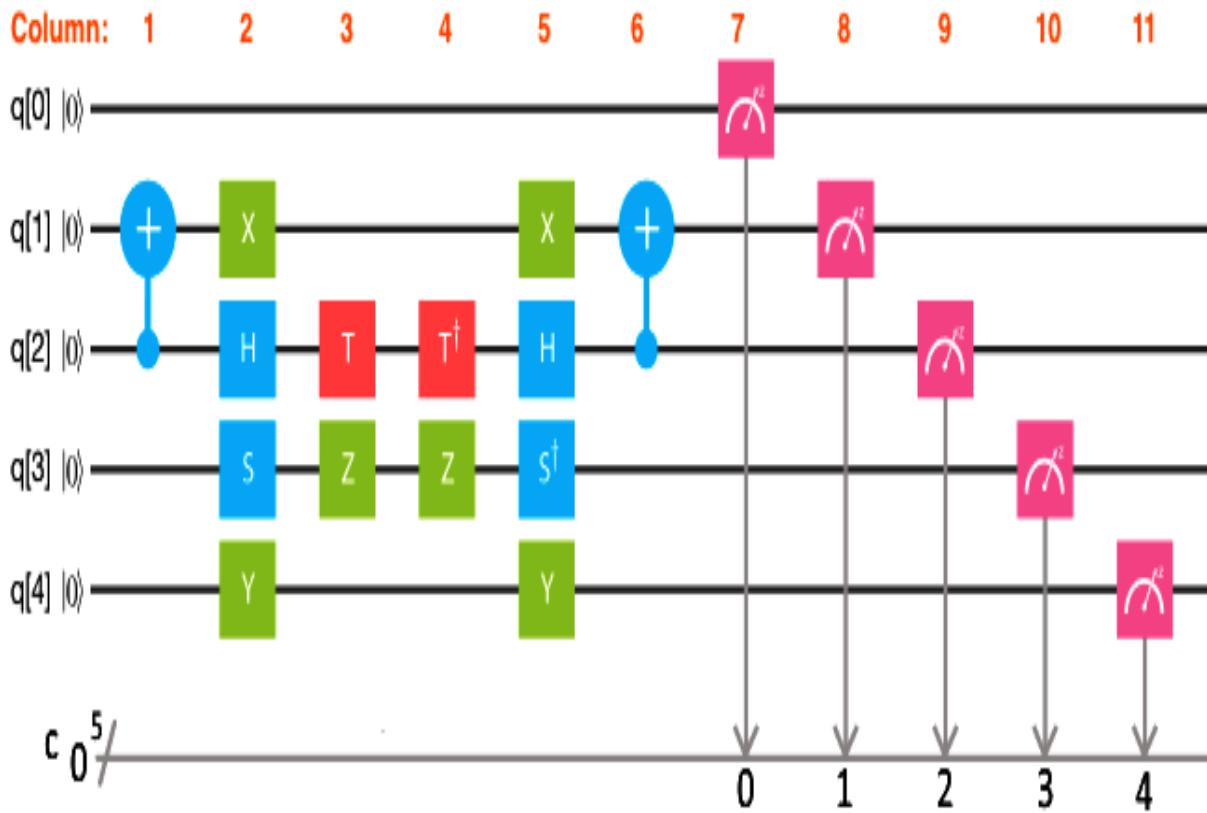
Putting this all together, we can create the following equivalent quantum score:



# Representing quantum scores in OpenQASM 2.0 programs

Here is an example of writing an OpenQASM 2.0 program from a quantum score. I have broken it down into columns from top to bottom of the score for clarity, and annotated these in the diagram by indicating column numbers in orange.

We saw the following circuit in a previous chapter where it illustrated the reversibility of quantum computations:



Let's dissect the OpenQASM that generates this circuit.

The first lines are, as usual, the headers, indicating the code is OpenQASM and that we will be using the standard IBM QX header:

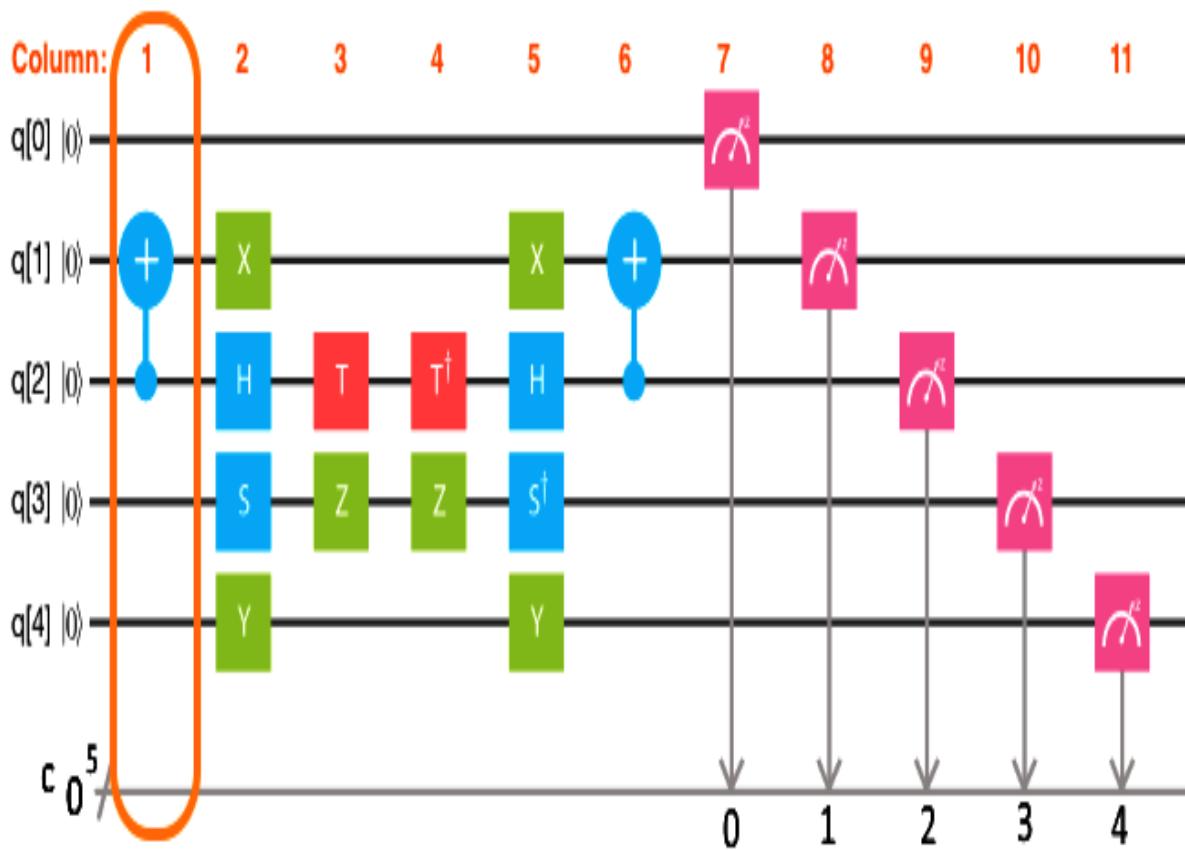
```
| OPENQASM 2.0;  
| include "qelib1.inc";
```

The next lines declare a quantum register named `q` of 5 qubits initialized to `"/00000">` and a classical register name `c` of 5 bits initialized to `00000`:

```
// declare the quantum and classical registers we will use  
qreg q[5];  
creg c[5];
```

The next lines will go column by column in the circuit diagram, creating the code for each column in order.

We will start with the first column:



The first column we can see only contains a CNOT gate, with its control qubit being the third qubit in the  $q$  quantum register,  $q[2]$  and the target qubit being the second qubit in the  $q$  quantum register,  $q[1]$ . Looking up the OpenQASM syntax for the CNOT gate in the table in the previous section, we see that it is `cnot control, target;`, which means that the first column will be coded as:

```
//column 1
cx q[2],q[1];
```

Next we will move to the second column, which has a number of gates specified. The code for the second column is:

```
//column2
x q[1];
h q[2];
```

```
| s q[3];  
| y q[4];
```

Each successive column should now be straightforward to encode from looking at the quantum score in OpenQASM. The full program is as follows:

```
OPENQASM 2.0;  
include "qelib1.inc";  
// declare the quantum and classical registers we will use  
qreg q[5];  
creg c[5];  
  
//column 1  
cx q[2],q[1];  
//column2  
x q[1];  
h q[2];  
s q[3];  
y q[4];  
//column 3  
t q[2];  
z q[3];  
//column 4  
tdg q[2];  
z q[3];  
//column 5  
x q[1];  
h q[2];  
sdg q[3];  
y q[4];  
// column 6  
cx q[2],q[1];  
// column 7  
measure q[0] -> c[0];  
// column 8  
measure q[1] -> c[1];  
// column 9  
measure q[2] -> c[2];  
// column 10  
measure q[3] -> c[3];  
// column 11  
measure q[4] -> c[4];
```

The previous code exactly reproduced the quantum score as depicted, but we could make several quantum scores, which are equivalent (and thus several variations on the OpenQASM program that are equivalent), as we saw in previous sections. Here are a couple of things to keep in mind:

- Each column could be in any order, for example column 3 could be:

```
| t q[2];  
| z q[3];
```

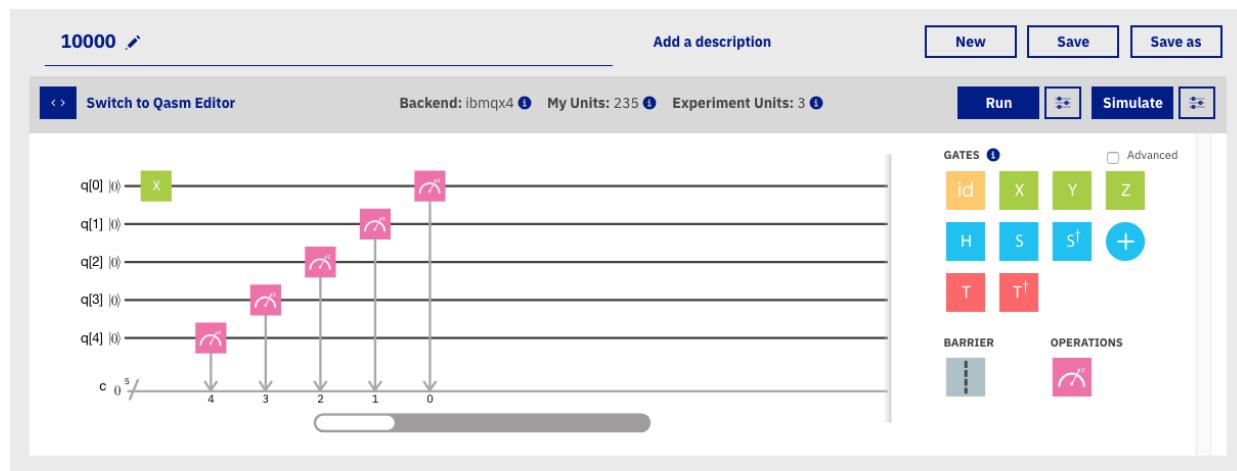
- Or it could be:

```
| z q[3];  
| tdg q[2];
```

In addition, any gate operating on a qubit in any column where there is no gate in the previous column on the qubit can be moved to the previous column, without affecting the computation.

# Using OpenQASM to interface with IBM QX

In the Quantum Composer, there is always a Switch to Qasm Editor button:

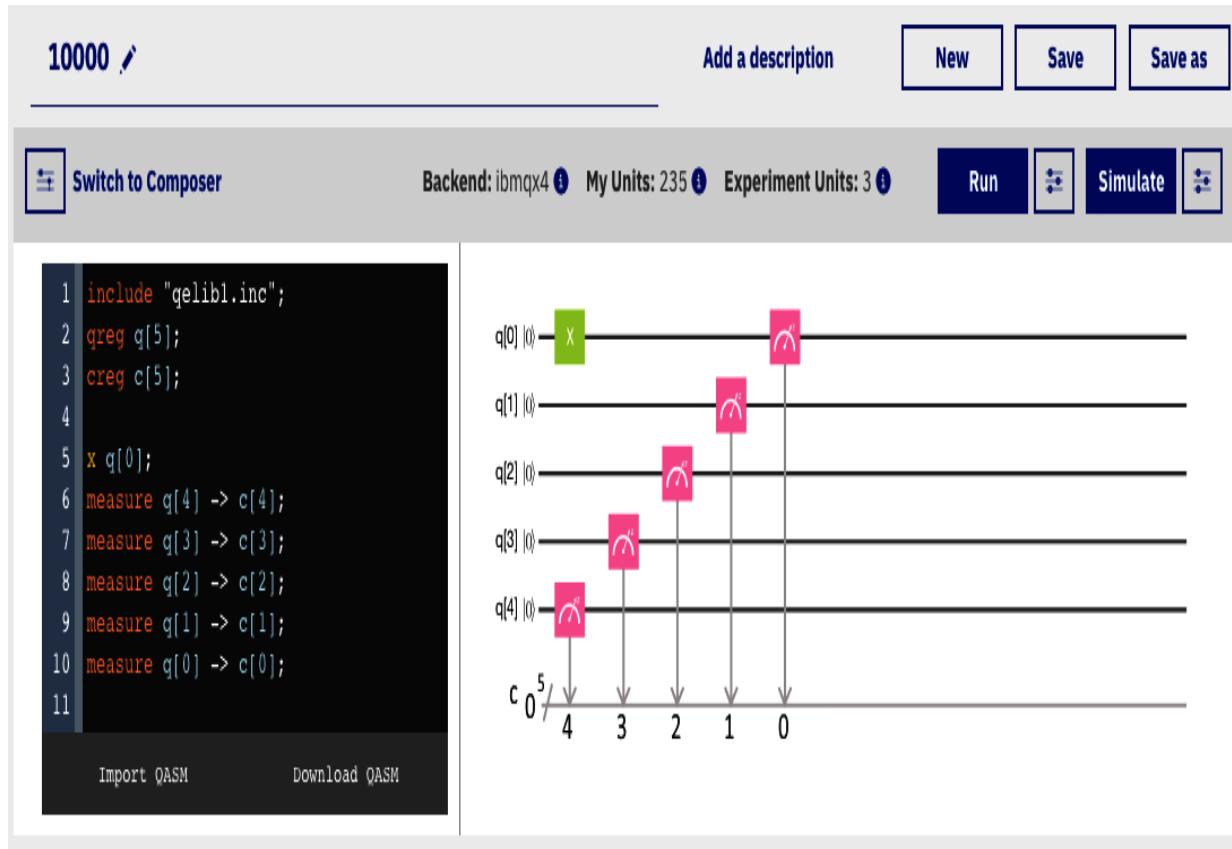


Click the button and you will get a warning that working in the editor could affect your quantum score, click OK:

Warning, moving to the qasm editor could modify the arrangement of gates in your score.  
Do you want to continue?

**OK**      Cancel

Then you are brought into the OpenQASM editor:



The editor helpfully has a Quantum Composer view of the circuit on the right, but you are no longer able to drag and drop gates. Instead, you can edit the OpenQASM directly, and see the modifications in the Quantum Composer, or import OpenQASM from an external file on your computer. Once you are happy with your circuit, the view gives you the opportunity to download the OpenQASM via the Download button as well.

Let's edit our circuit to change it from representing 10,000 to 11,000 by adding in an X gate onto the qubit labeled q[1]. We do this by adding `x q[1];` to the code in between the register declarations and the measurements. As soon as this line is added, the user interface updates to display the modified quantum score:

10000 ↗

Add a description

New

**Save**

**Save as**

5

## Switch to Composer

Backend: ibmqx4 | My Units: 235 | Experiment Units: 3

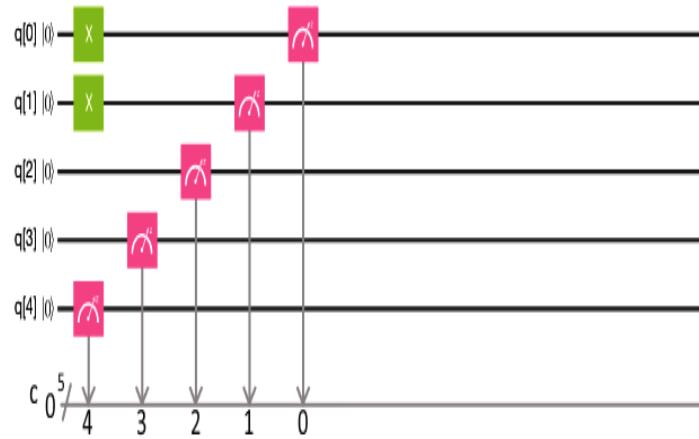
Run



**Simulate**



```
1 include "qelib1.inc";
2 qreg q[5];
3 creg c[5];
4
5 x q[0];
6 x q[1];
7 measure q[4] -> c[4];
8 measure q[3] -> c[3];
9 measure q[2] -> c[2];
10 measure q[1] -> c[1];
11 measure q[0] -> c[0];
12
```



## Import QASM

Download QASM

# Advanced OpenQASM usage

There are several features of OpenQASM that are not enabled on the IBM QX, and thus are not covered in detail in this book. These include:

- Resetting a qubit
- `if` statements
- User-defined gates
- Built-in/physical/opaque gates

However, as some OpenQASM programs specified in the literature use these language features, and IBM QX may use them in the future, I will go over their syntax and usage briefly here.

# Resetting a qubit

The OpenQASM language supports resetting a qubit or quantum register, which prepares the qubit or all the qubits in the quantum register to the `/"0">` state. The syntax for this is as follows:

```
| reset q[0];
```

This will reset the first qubit in the `q` quantum register to `/"0">`, or use the following:

```
| reset q;
```

This will reset every qubit in the `q` quantum register to `/"0">`. The IBM QX does not at this time support the reset operation.

# if statements

OpenQASM has a mechanism for `if` statements, which allow the outcome of one or more quantum measurements, placed in a classical register, to determine future program execution. The syntax is the following:

```
| if(creg==int) qop;
```

Note that `creg` cannot be indexed, so we check the entire register to ensure that, where the first bit in the classical register at index 0, is the lowest-order bit. Furthermore, `qop` is simply any valid line of OpenQASM that corresponds to a quantum operation of a gate. One example of the `if` statement in action is:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[5];
creg c[5];
x q[0];
measure q[0] -> c[0];
if(c==1) x q[1];
measure q[1] -> c[1];
```

The `if` statement in this program executes only if the classical register is 1. Since measuring `q[0]` will result in 1 as the program is specified, the `if` statement will execute and change `q[1]` to 1. After the final line of the program, which measures the value in `q[1]`, the classical register will now hold 00011, the integer value of 3.

The `if` statement of OpenQASM is not currently supported by IBM QX.

# User-defined gates and primitive gates

In the OpenQASM language, the users can define any gate they choose. The syntax is as follows:

```
gate name(params) qargs
{
    body
}
```

`qargs` here is a comma-separated list of qubits, and at least one argument is required. IBM at the time of the writing of this book does not support this feature for users.

# Primitive gates CX and U

The OpenQASM standard is only specified with gates, CX and U. All the rest of the gates we have worked with are user defined, within the IBM QX environment, available with the inclusion of `qelib1.inc`. IBM QX does not allow the direct access of these primitive gates.

If you look at the content of the `qelib1.inc` file (<https://github.com/Qiskit/openqasm/blob/master/examples/generic/qelib1.inc>), you can see how these definitions are made. In the following sections, we will go over the primitive CX and U gates to give you enough understanding.

# 2-qubit gate CX

CX is the controlled NOT gate. IBM's implementation of this as a user-defined gate just renames it to be lowercase:

```
gate cx c,t
{
    CX c,t;
}
```

# 1-qubit gate U

We learned in [Chapter 4](#), *Evolving Quantum States with Quantum Gates*, how every 1-qubit gate can be generalized as taking a qubit on the Bloch sphere to another position on the Bloch sphere. The 1-qubit U gate is the generalization of this. The U gate takes in three angles as arguments,  $U(\theta, \phi, \lambda)$  which collectively specify the rotation the input qubit undergoes on the Bloch sphere. The details of this specification are available in the Appendix.

The important thing to remember is that, with the U gate, we can rotate any qubit at any position on the Bloch sphere to any other position on the Bloch sphere thus, any 1-qubit gate, including naturally all the ones available through the IBM QX, can be specified in terms of U alone. This is done through the `qelib1.inc` file included at the top of every IBM QX program. If you are curious as to how each 1-qubit gate, I, X, Y, Z, H, S,  $S^\dagger$ , T, and  $T^\dagger$ , is defined in terms of U, look at this file to see the details.

# Opaque gates

The OpenQASM standard allows the definition of so-called opaque gates, which are simply placeholders for future definitions of a gate, meant to be used in situations where the gate's operation is currently unspecified. If you know C or C++, you could think of an opaque gate as a header declaration. The syntax is the same as for a user-defined gate, but without the body. Opaque gates are not used within the IBM QX, and are beyond the scope of this book.

# Summary

The Open Quantum Assembly Language, a custom programming language, is designed specifically to minimally describe quantum circuits. OpenQASM can be used to write a quantum program, and then execute it on IBM QX. It can also be used as a non-graphical way to describe any circuit specified in the Quantum Composer on IBM QX. Since it is a general, non graphical language, it is useful to describe quantum algorithms. Several of OpenQASM's advanced features are not available on IBM QX, including resetting qubits, `if` statements, user-defined gates, and access to the only gates in OpenQASM's specification: CX and U. Instead IBM QX imports a header file with some user-defined gates (defined by IBM) called `qelib1.inc`, which allows the user to have access to all the gates this book has covered in detail: I, X, Y, Z, H, S,  $S^\dagger$ , T,  $T^\dagger$ , and CNOT.

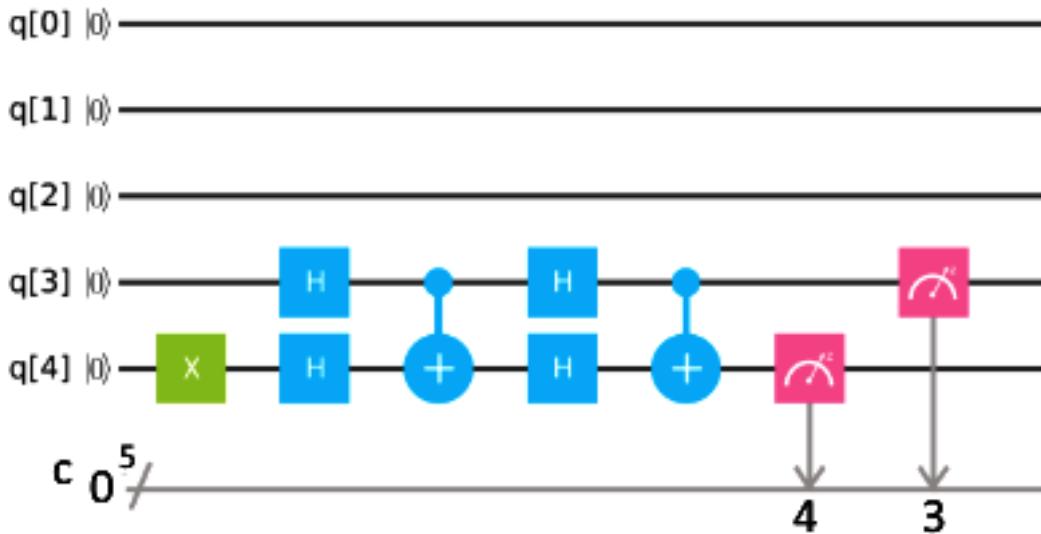
In the next chapter, we will go over Qiskit and quantum computer simulation.

# Questions

1. Translate the following OpenQASM program into a quantum score:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[5];
creg c[5];
x q[4];
h q[3];
h q[4];
cx q[3],q[4];
h q[3];
h q[4];
cx q[3],q[4];
measure q[3] -> c[3];
measure q[4] -> c[4];
```

2. Translate the following quantum score to an OpenQASM program:



3. Challenge problem: run the following OpenQASM program on IBM QX twice in the simulator, once with the third  $q[2]$  qubit initialized to  $|1\rangle$ , as in the following

example code, and once with the third qubit initialized to  $|0\rangle$  (hint: remove the `x q[2];` line to do this). Make sure to set the parameters of the simulator, or of the experiment, to do as many shots as possible to get as close to the theoretical result as possible. What does the program return? What is its purpose? Refer to the following OpenQASM code:

```
include "qelib1.inc";
qreg q[5];
creg c[5];

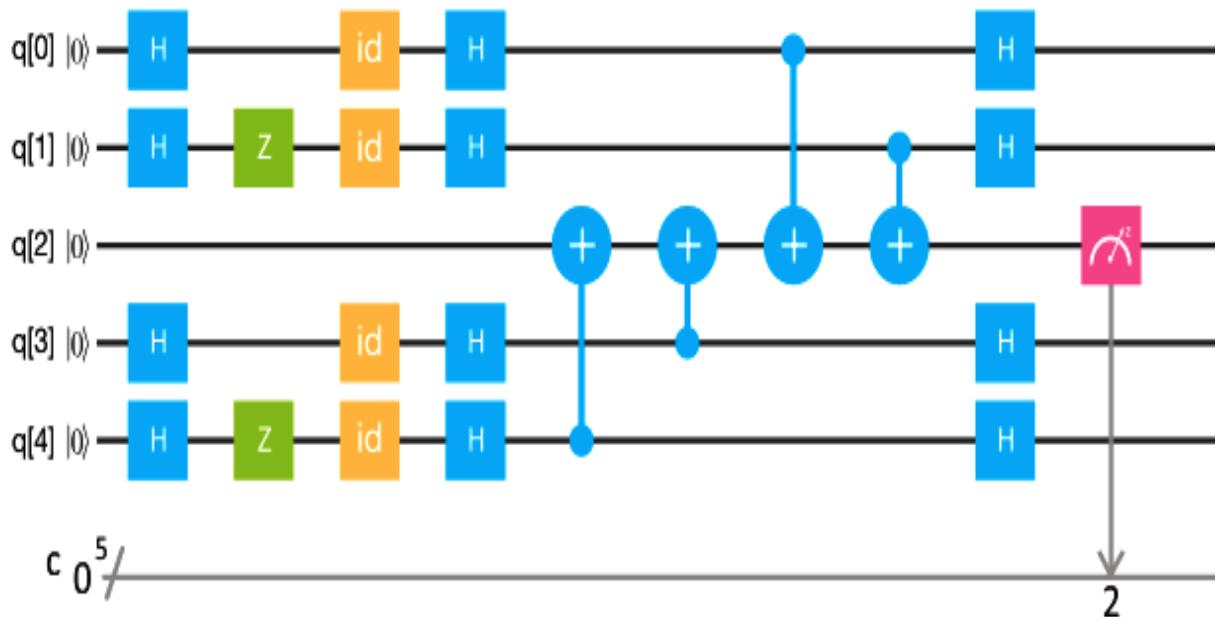
x q[2];
h q[1];
sdg q[1];
cx q[2],q[1];
h q[1];
t q[1];
cx q[2],q[1];
t q[1];
h q[1];
s q[1];
x q[1];
s q[2];

measure q[1] -> c[1];
```



*In each of the two runs, write out a program with no more than one gate, which would produce the equivalent output. Think about the similarities to the CNOT gate in terms of controlled behavior.*

4. Implement the  $XYZS^{\dagger}T^{\dagger}HHTSZY|0\rangle$  circuit in OpenQASM.
5. Translate the following circuit to OpenQASM:



6. Draw the quantum score that corresponds to the following OpenQASM program:

```

include "qelib1.inc";
qreg q[5];
creg c[5];

x q[0];
x q[1];
id q[2];
h q[2];
cx q[1],q[2];
tdg q[2];
cx q[0],q[2];
t q[2];
cx q[1],q[2];
tdg q[2];
cx q[0],q[2];
t q[1];
t q[2];
h q[2];
cx q[1],q[2];
h q[1];
h q[2];
cx q[1],q[2];
h q[1];
h q[2];
cx q[1],q[2];
cx q[0],q[2];
t q[0];
tdg q[2];

```

```
| cx q[0],q[2];
| measure q[0] -> c[0];
| measure q[1] -> c[1];
| measure q[2] -> c[2];
```

# Qiskit and Quantum Computer Simulation

This chapter introduces **Qiskit (Quantum Information Software Kit)**, focusing on how it can be used to run programs in IBM QX as well as its capabilities for quantum simulation. These capabilities are highly useful in development. This chapter gives an overview of how to obtain, install, and work with Qiskit, as well as how to set up Qiskit to interact with IBM QX. The chapter then moves on to a capstone project using Qiskit to illustrate, in one project, the concepts of quantum circuits, measurement, and Qiskit usage, while producing a useful demo of using a quantum computer to represent musical chords. This chapter also discusses more advanced quantum computing algorithms with Qiskit, available via the Qiskit Aqua package.



*We learned how to install Qiskit in [Chapter 1, What is Quantum Computing?](#), and some basics in [Chapter 5, Quantum Circuits](#). In this chapter, we will review Qiskit and the basics of working with it, as well as learning more advanced usage. The intention of this chapter is to be standalone, so that those who have not read or not recently read [Chapter 1, What is Quantum Computing?](#), and [Chapter 5, Quantum Circuits](#), should be able to start from this chapter. For readers who are already comfortable with Qiskit basics, feel free to skim or skip the subsections of this chapter that focus on them.*

This chapter will cover the following topics:

- Qiskit installation and usage
- Qiskit Terra and Qiskit Aqua
- Using Qiskit with OpenQASM
- Qiskit Capstone project – quantum chords

# Technical requirements

The code for this chapter is available in a Jupyter Notebook under <https://github.com/PacktPublishing/Mastering-Quantum-Computing-with-IBM-QX>, in the `Chapter08` directory. Data files for the chapter are available in the same directory.

# **Qiskit installation and usage**

This section will go over Qiskit installation and usage. First we'll test the installation, which was performed as part of [Chapter 1, \*What is Quantum Computing?\*](#).

# Testing Qiskit installation

Check your installation of Qiskit by running the following code, and refer to [Chapter 1](#), *What is Quantum Computing?*, in the *Setup and run Qiskit examples* section, as required for troubleshooting:

```
from qiskit import Aer
from qiskit import IBMQ
# Authenticate an account and add for use during this session. Replace
string
# argument with your private token.
IBMQ.enable_account("INSERT_YOUR_API_TOKEN_HERE")
```

 Recall that to get your API token, log into IBM QX at <https://quantumexperience.ng.bluemix.net/qx> and then go to My Account. The API token will be in the upper right of the user interface. Click on Advanced, then on Copy API Token to insert it here.

Run the following code:

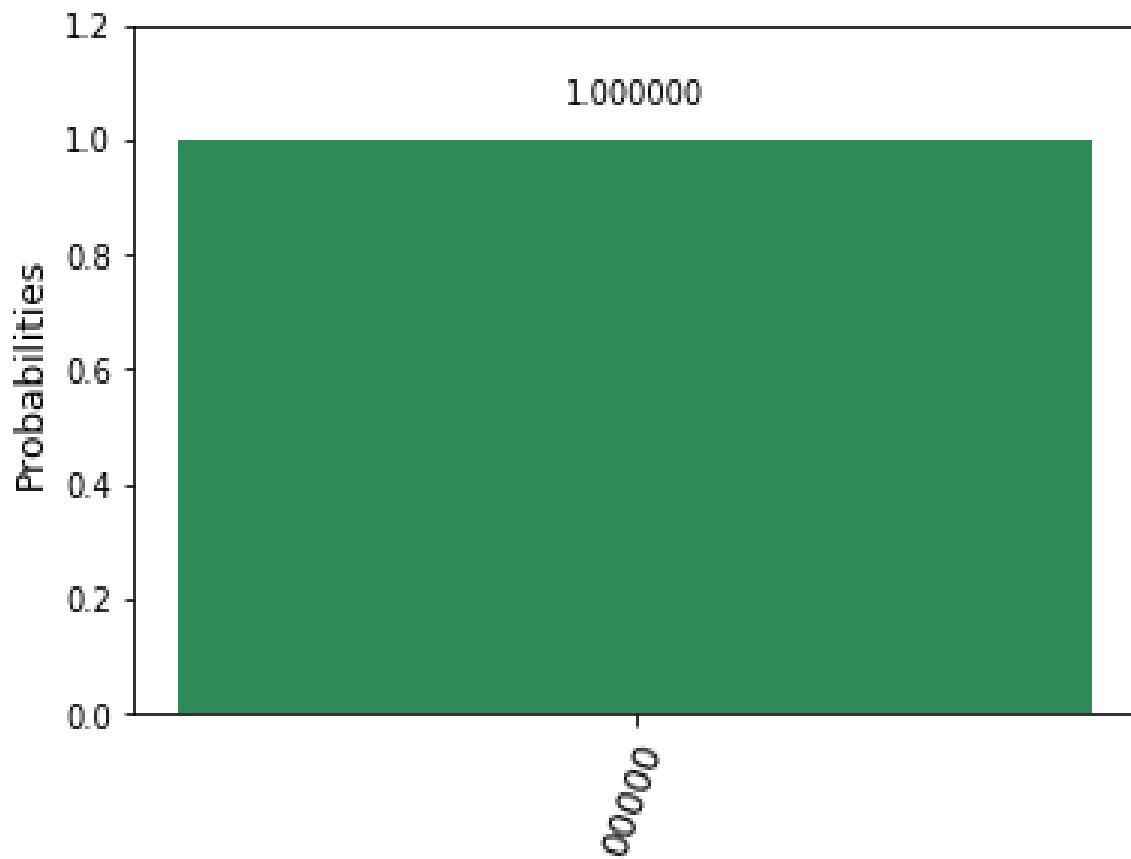
```
import qiskit
from qiskit.tools.visualization import plot_histogram

# Pick an available backend
backend = IBMQ.get_backend('ibmq_qasm_simulator')

# Setup 5 quantum and 5 classical registers, performing a measurement
q = qiskit.QuantumRegister(5)
c = qiskit.ClassicalRegister(5)
qc = qiskit.QuantumCircuit(q, c)
qc.measure(q, c)

# Executing the job on IBM QX
job_exp = qiskit.execute(qc, backend=backend)
plot_histogram(job_exp.result().get_counts(qc))
```

You should get, as you did in [Chapter 1](#), *What is Quantum Computing?* 00000 in the classic register with 100% probability:



*Note, if you are at a loss for a backend, Qiskit always has a local simulator available, with the backend name of `local_qasm_simulator`, which doesn't require registering with the IBM QX API. If you have registered, then the backend of `ibm_qasm_simulator` is always available.*

# Using OpenQASM with Qiskit

Although directly programming in Qiskit via Python is useful, often quantum algorithms are specified in OpenQASM. We went over the details of OpenQASM in [Chapter 7, Working with OpenQASM](#), and, in this subsection, I will provide a few examples of integrating OpenQASM programs into Qiskit code.

# Load OpenQASM from a file

In [Chapter 7](#), *Working with OpenQASM*, we worked with a circuit, illustrating the reversibility of quantum computation. Specifically, the circuit, which starts out at `|"00000">`, should output `|"00000">` even after all the gates are applied as each gate is successfully reversed (undone). In this chapter's code directory, the OpenQASM for this circuit is saved as the `reversible.qasm` file; for convenience, the OpenQASM is also reproduced here:

```
OPENQASM 2.0;
include "qelib1.inc";
qreg q[5];
creg c[5];
cx q[2],q[1];
x q[1];
h q[2];
s q[3];
y q[4];
t q[2];
z q[3];
tdg q[2];
z q[3];
x q[1];
h q[2];
sdg q[3];
y q[4];
cx q[2],q[1];
measure q[0] -> c[0];
measure q[1] -> c[1];
measure q[2] -> c[2];
measure q[3] -> c[3];
measure q[4] -> c[4];
```

This file can be loaded in a way it can be run on IBM QX's local or remote infrastructure, using the `load_qasm_file` method as follows:

```
import qiskit
qc = qiskit.wrapper.load_qasm_file("reversible.qasm")
# Run on local simulator
```

```
|sim = qiskit.execute(qc, "local_qasm_simulator")
|result = sim.result()
|# Output result
|print(result.get_counts(qc))
```

This should print the following:

```
|{'00000': 1024}
```

The output should be identical to the input ( $00000$ ) for this circuit.

# Working with OpenQASM loaded from a string

We can just as easily work with OpenQASM loaded from a string. Thus, if you prefer programming in OpenQASM instead of pure Python, you can simply place your program in a string and then easily run it from Python. Here, we illustrate a short program for readability, where we change each input qubit from `"/0">` to `"/1">`:

```
import qiskit
qasm_string="""OPENQASM 2.0;
include "qelib1.inc";
qreg q[2];
creg c[2];
x q[0];
x q[1];
measure q[0] -> c[0];
measure q[1] -> c[1];
"""
qc = qiskit.wrapper.load_qasm_string(qasm_string)
sim = qiskit.execute(qc, "local_qasm_simulator")
result = sim.result()
print(result.get_counts(qc))
```

As expected, the result is the following:

```
|{'11': 1024}
```

# Qiskit Aqua introduction and installation

Qiskit, as we have set it up so far, focuses on lower level direct manipulation of qubits. This portion of the `qiskit` library is also known as Qiskit Terra (<https://qiskit.org/terra>) to contrast it with QISKit Aqua (<https://qiskit.org/aqua>). It is likely that, as quantum computers become more useful, the interfaces programmers have to quantum computers will not be through writing low level code (the equivalent of assembly language in classic computing) as this book focuses on, where such low level code manipulates qubits and quantum registers as assembly language manipulates classic bits and classic registers. Rather, as with all computing, the field will evolve to use higher level libraries and languages to abstract this away.

As an attempt to design such an abstraction, IBM has released Qiskit Aqua, which focuses on working with existing algorithms that can be run on quantum computers at a higher level. These algorithms can have a hybrid component, running on both the CPU and the quantum processor. Many algorithms, particularly those focused on quantum chemistry, are available. Qiskit Aqua is also designed so that users can write code to plug into the framework, with the idea that users can add their own algorithms to the collection available.

If you installed the requirements for the book using pip from the `requirements.txt` file, Qiskit Aqua is already installed. If not, Qiskit Aqua can be installed with the following:

```
| pip install qiskit-aqua qiskit-aqua-chemistry
```

Test your Qiskit Aqua installation by running the following code:

```
# Testing QISKit Aqua installation
import qiskit_aqua
import qiskit_aqua_chemistry
```

Qiskit Aqua will not be extensively used in this book, which aims to teach quantum computing at a lower level, but after mastering the material in this book, Qiskit Aqua is a great next step to grapple with. At the time of this writing, the quantum computers available are too small for any of the Qiskit algorithms to be practical, but learning to work with Qiskit Aqua and writing algorithms within its framework means that once quantum computers good enough for the particular problem become available, code for the particular problem will already be prototyped and ready to be tested. One optional exercise of this chapter encourages you to run and explore an algorithm of Qiskit Aqua in depth.

For reference, at the time of this book's writing, Qiskit Aqua contains modules for the following:

- **Quantum Grover Search (QGS)**
- **Support Vector Machine Quantum Kernel (SVM Q Kernel)**
- **Quantum Dynamics (QD)**
- **Quantum Phase Estimation (QPE)**
- **Variational Quantum Eigensolver (VQE)**
- **Iterative Quantum Phase Estimation (IQPE)**
- **Support Vector Machine Variational (SVM Variational)**

It also contains classic algorithms such as the following:

- **Support Vector Machine Radial Basis Function Kernel (SVN RBF Kernel)**
- CPLEX

- **Exact Eigensolver (EE)**

It also includes certain optimizers. More functionality will certainly be added over the coming years. For more information about Qiskit Aqua, check out the website at <https://github.com/Qiskit/aqua>.

# **Qiskit Terra - Capstone project**

In previous chapters, we learned the basics of quantum computing with simple circuits meant to illustrate academic points. In this section, we will work with some more advanced usage and create a real project, leading into the third part of the book on algorithms. The idea is to do a fun and useful project with Qiskit to illustrate and practice some of the concepts the book has covered so far, without having to get into advanced algorithm description or design.

IBM has some great tutorials on short quantum programs that illustrate the nature of superposition ([https://github.com/Qiskit/qiskit-tutorials/tree/master/community/hello\\_world](https://github.com/Qiskit/qiskit-tutorials/tree/master/community/hello_world)) through ASCII characters and through sound clips. In this section, we'll illustrate the nature of superposition through encoding musical notes via the MIDI code. We'll make chords, groups of notes that play together, which represent the superposition of a quantum state, and be able to listen to the results of our measurements. To do this, we'll need to briefly study how to represent musical notes on the computer, which can be done with the MIDI specification.

# A brief introduction to the MIDI specification

MIDI is a specification that allows digital musical equipment to communicate. MIDI messages consist of a status byte followed optionally by additional data bytes. Status bytes always begin with 0, and data bytes always begin with 1, leaving each data byte seven remaining bits, or  $2^7 = 128$  different values to describe its data.

For the `NOTE ON` message, this consists of three bytes. The first byte indicates that it is a `NOTE ON` together with a channel (which can be used to differentiate between different instruments); the second byte is the pitch (specifying the musical note); and the third byte is the velocity (specifying how loud the musical note should be played). Out of the eight bits in the byte encoding the pitch of a note, the first bit is always a 0. You can read more at <https://www.midi.org/specifications-old/item/table-1-summary-of-midi-message>.

# Quantum computing with MIDI

In this example, we will use our quantum computer to represent a note in superposition. We will have the identical status byte (`NOTE ON`, channel 0: `10010000`) and an identical velocity (loud, value of 80: `01010000`) for each of our notes. So, the first and the third byte of our message will remain fixed. For the middle byte, we know it will begin with `0`, and the last seven bits will determine the note.



*Note names include A, B, C, D, E, F, and G, each of which has a distinct frequency that makes the notes sound different. There are also notes in between some of these basic notes, which are indicated with a # after the note name. For example, C# is a note with a frequency in between C and D. Notes can sound the same and be higher and lower in register. This is referred to as being separated by an octave and corresponds to a factor of 2 up or down in frequency. Notes that are separated by an octave have the same name. In our notation, we will indicate the octave by the number following the name. For example, C1, C2, C3, are all the note C, with C2's frequency being twice that of C1, and C3's frequency being three times that of C1. The exact spacing of the various notes is a subject we won't cover in this book.*

For convenience, the code for this chapter includes a MIDI conversion dictionary represented as a pickled dictionary containing a variety of dictionaries with mappings from 7-bit binary to frequency or note name, from frequency to 7-bit binary or note name, and from note name to frequency or 7-bit binary. In addition, mappings are maintained from/to the integer equivalent of the 7-bit binary. Since the output of our measurements from IBM QX is returned as strings, for example, '`0111100`', all of our dictionary keys and values are themselves strings for convenience. For example, we make the 7-bit binary a string of 0s and 1s instead of a true binary number in Python. Finally, only 88 possible values are encoded, those corresponding to notes on a piano.

This dictionary can be loaded with the following:

```
| midi_conversion_tables=pickle.load(  
|     open('midi_conversion_tables.p','rb'))
```

There are two tables we will rely on frequently. The first is the table that maps the 7-bit binary MIDI code to a note name:

```
| midi_to_note_bin=midi_conversion_tables['midi_to_note_bin']
```

The second table maps 7-bit binary MIDI code to a frequency in Hz:

```
| midi_to_frequency_bin=midi_conversion_tables['midi_to_frequency_bin']
```

These can be used as follows. For example, if we didn't know that the binary `0111100` corresponds to *C4* and a frequency of 277.1826 Hz, we could look that up with the following:

```
| print(midi_to_note_bin['0111100'])
```

This would print `c4`.

Then, we can instead print the frequency instead of the note with the following:

```
| print(midi_to_frequency_bin['0111100'])
```

This would print `277.1826`.

# Synthesizing notes

For the next part, we will need some helper functions that will allow us to use Python to synthesize a note.

Fundamentally, a note is a wave (the type of wave will change how the note sounds to our ears, similar to the way the same note on different instruments sounds different) with a frequency (which produces the distinct note) and an amplitude (indicating its volume). Python has a variety of options for playing notes and chords, many of which involve a large package, a sound pack, and/or external installations. To avoid this, we used the `pygame` module together with a simple choice of wave (sine wave) to make a simple note synthesizer in just a few lines:

```
import numpy
import pygame, pygame.sndarray
import pickle

def play_notes(freqs,volumes):
    """
        freqs: a list of frequencies in Hz
        volumes a list of volumes: (1 highest 0 lowest)
        example usage:
        play_notes([440,880],[0.6,0.2])

    """
    pygame.mixer.init()

    sample_wave=sum([numpy.resize(volume*16384*numpy.sin(numpy.arange(int(44100/float(hz)))*numpy.pi*2/(44100/float(hz))), (44100,)).astype(numpy.int16) for hz,volume in zip(freqs,volumes)])
    stereo = numpy.vstack((sample_wave, sample_wave)).T.copy(order='C')
    sound = pygame.sndarray.make_sound(stereo)
    sound.play(-1)
    pygame.time.delay(1000)
    sound.stop()
    pygame.time.delay(1000)
```

# **Playing notes and chords to represent quantum measurement**

Our end goal will be to create a quantum circuit that represents a superposition between different notes. Since our output will be 7-bit, our circuit will be 7-qubits. The measurement outcome of a single preparation of the circuit will be one of the notes in the superposition, and if we perform many measurements, we expect to get each of the notes in the superposition with a certain probability. In simulation or on the IBM QX, we can do the identical preparation of the quantum circuit many times and measure it to extract these probabilities.

The code present in following sections will help us to hear the results of a single measurement, and the results of many measurements on identical preparations of the circuit together.

# Playing a note for each quantum measurement

For each quantum measurement of all seven qubits, we will get one note. We can perform this measurement, get the outcome, and play the note with the following code:

```
import qiskit
def quantum_play_notes(qc,shots):
    """
        qc: the quantum circuit of 7 qubits
        shots: the number of times to prepare and perform the circuit
        computation

        The quantum state is prepared <shots> number of times.

        The result of the measurement each time is played as a single note.

    """
    midi_conversion_tables=pickle.load(open(
                                'midi_conversion_tables.p','rb'))
    midi_to_note_bin=
        midi_conversion_tables['midi_to_note_bin']
    midi_to_frequency_bin=
        midi_conversion_tables['midi_to_frequency_bin']

    for i in range(shots):
        # Note shots=1 may result in a deprecation warning,
        # which will go away in a future code version.
        sim = qiskit.execute(qc, "local_qasm_simulator",shots=1)
        result = sim.result()
        final=result.get_counts(qc)
        [print(midi_to_note_bin[k]) for k in final.keys()]
        play_notes([float(midi_to_frequency_bin[k])
                    for k in final.keys()],[1.0])
```

# Playing a chord (group of notes at the same time) to represent the results of many quantum measurements

When we prepare a quantum circuit the same way multiple times, and then measure it each time, our results will be notes within the superposition of the quantum circuit's state, each with a certain probability. To represent them together, we play each note of the result with volume scaled by its probability. This quantum run can be done with the following code:

```
import qiskit
def quantum_play_chords(qc, shots):
    """
    qc: the quantum circuit of 7 qubits
    shots: the number of times to prepare and perform the circuit computation

    The quantum state is prepared <shots> number of times.

    This method helps to hear the outcome of the <shots> measurements together by producing
    a chord where each note in the chord is a measurement result and the maximum volume of the note
    is reduced the fraction of the times the note appears in <shots> measurement.
    """
    midi_conversion_tables=
        pickle.load(open('midi_conversion_tables.p','rb'))
    midi_to_note_bin=
        midi_conversion_tables['midi_to_note_bin']
    midi_to_frequency_bin=
        midi_conversion_tables['midi_to_frequency_bin']

    sim = qiskit.execute(qc, "local_qasm_simulator",shots=shots)
    result = sim.result()
```

```
final=result.get_counts(qc)

freqs=[]
volumes=[]
for k,v in final.items():
    try:
        freqs+=[float(midi_to_frequency_bin[k])]
        volumes+=[int(v)/shots]
        print('%f percent' % (int(v)/shots*100)
              ,midi_to_note_bin[k])
    except:
        print('%f percent' % (int(v)/shots*100),k)
        play_notes(freqs,volumes)
```

# **Creating a superposition of notes**

To create a superposition of two notes, we want to find two notes whose MIDI codes are off by one bit, and then use our quantum gates to create a quantum circuit in which the qubit corresponding to the bit that varies is in a superposition between 0 and 1, and the rest of the qubits are fixed to correspond to the fixed bits in the MIDI code. To create a superposition of four notes, we would find four notes that vary only in two bits among them, and create a circuit with a superposition in each of those two corresponding qubits. To create a superposition of eight notes, we would find 8 notes which vary only in three bits among them.

# Two note chord

One example is *F4* and *A4*. *F4* is 65 in MIDI code, or  $1000001$  in binary. *A4* is 69 in MIDI code or  $1000101$  in binary. Counting the rightmost bit as index 0, *F4* and *A4* differ only by the bit at index 2. These also form two of the three notes of a major *F* chord, so they are a bound to sound nice together.

The following code creates a superposition between *F4* and *A4* accordingly:

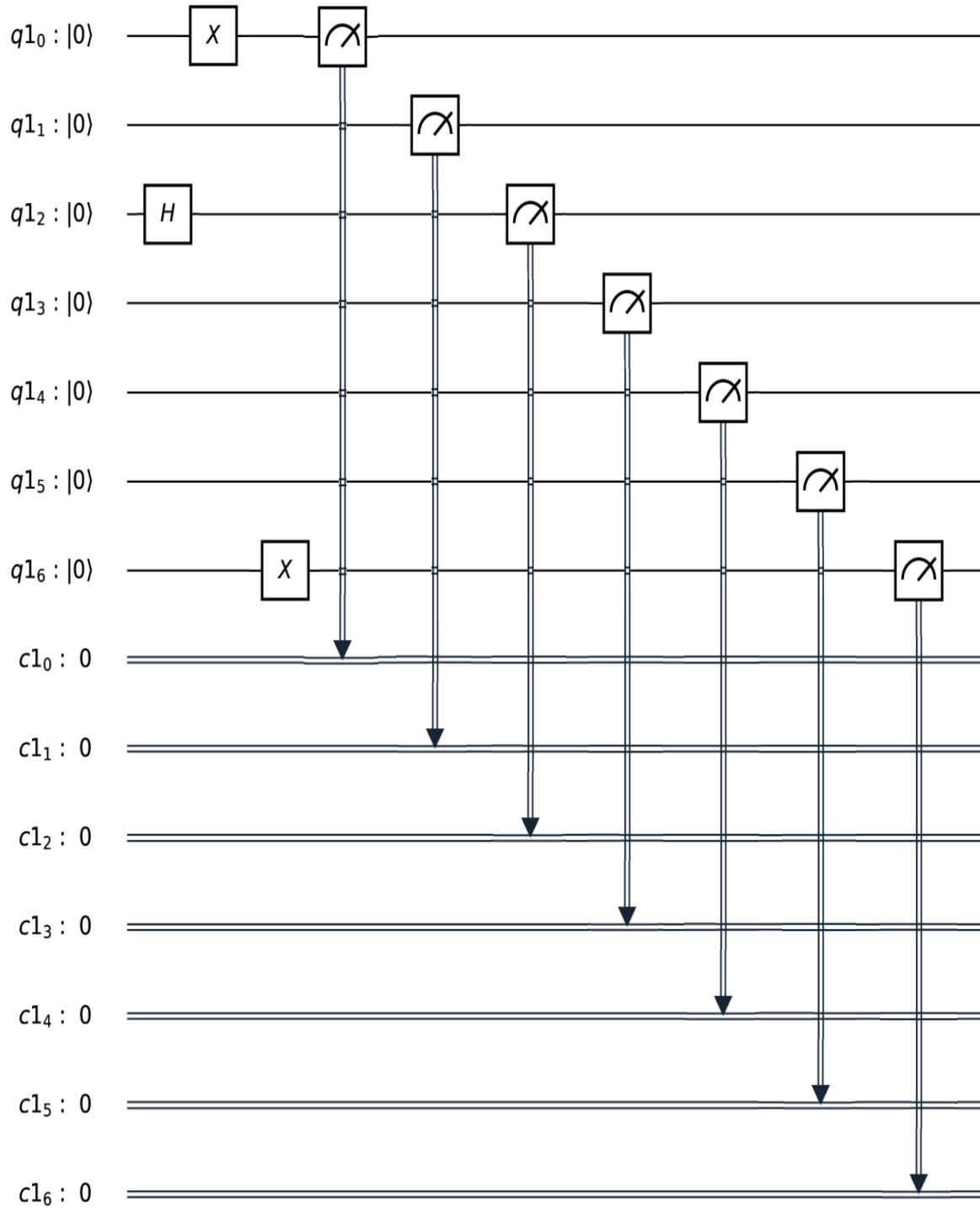
```
from qiskit import ClassicalRegister, QuantumRegister
from qiskit import QuantumCircuit

# set up registers and program
qr = QuantumRegister(7)
cr = ClassicalRegister(7)
qc = QuantumCircuit(qr, cr)

# F4 and A4 together:
qc.h(qr[2]) # create superposition on 2
qc.x(qr[0]) # 1
qc.x(qr[6]) # 1

for j in range(7):
    qc.measure(qr[j], cr[j])
```

It may help to visualize the circuit:



Then, the results of a single measurement can be represented via sound with the following:

```
| quantum_play_notes(qc, 4)
```

Or, the results of a series of measurements can be represented via sound with the following:

```
| quantum_play_chords(qc,4)
```

The argument `4` in the previous example indicates how many times to identically prepare and measure the circuit, in this case four times. The fewer the times, the less likely that we will see an even distribution over the equal superposition. For example, if this argument were equal to `1`, we'd only ever see one of the two notes pop up.

# Four note chord

Let's next build a four note chord. Consider the following four notes, which differ just in their bits at index 2 and index 3:

Note	MIDI code	MIDI code (binary)
C3	48	0110000
E3	52	0110100
G#3	56	0111000
C4	60	0111100

This is an augmented C chord, so it is bound to sound interesting. Read more about augmented triad chords here: [h](https://en.wikipedia.org/wiki/Augmented_triad)  
[https://en.wikipedia.org/wiki/Augmented\\_triad](https://en.wikipedia.org/wiki/Augmented_triad).

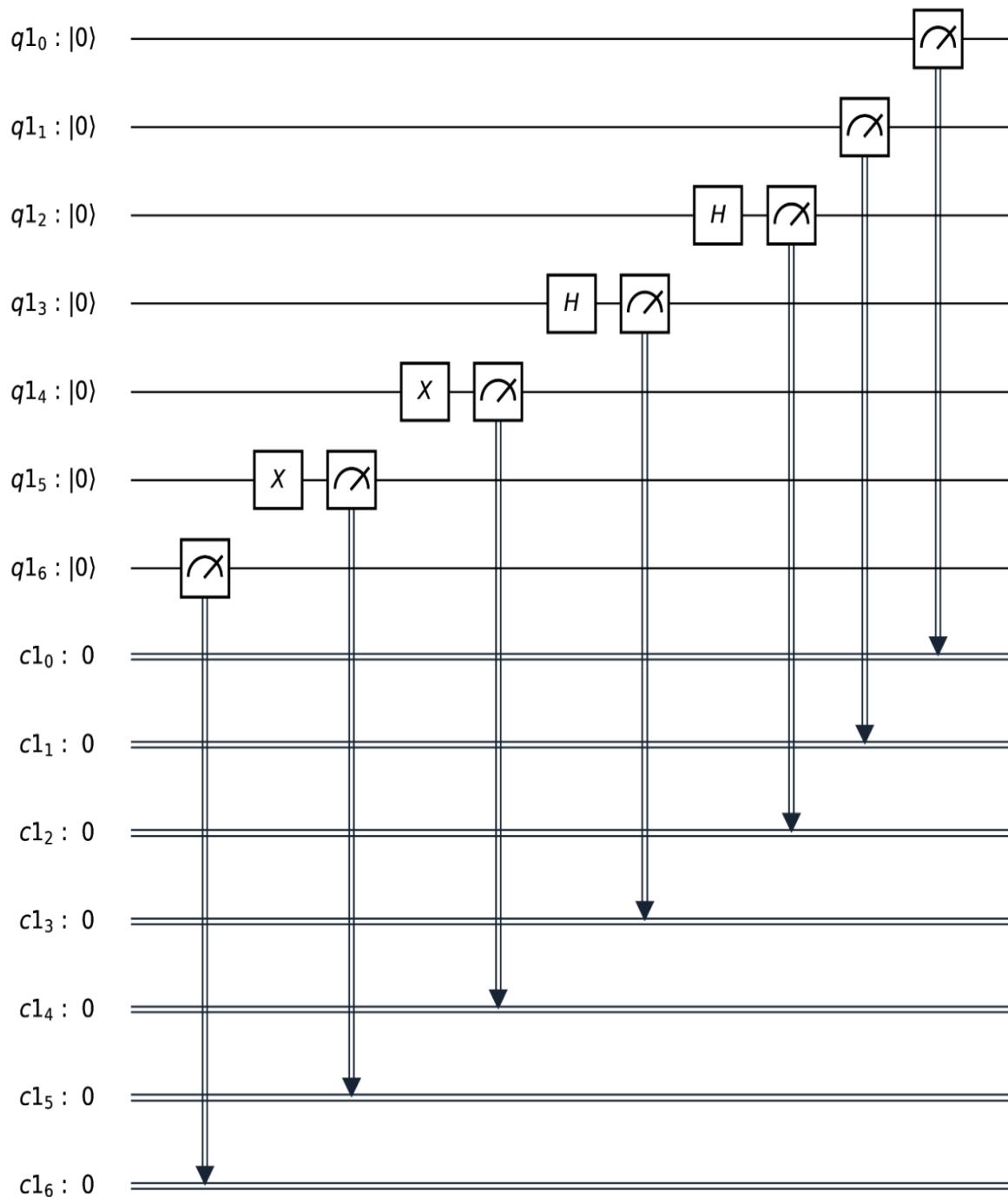
Now for the code that produces this chord:

```
import qiskit
from qiskit import ClassicalRegister, QuantumRegister
from qiskit import QuantumCircuit

# set up registers and program
qr = QuantumRegister(7)
cr = ClassicalRegister(7)
```

```
| qc = QuantumCircuit(qr, cr)
| qc.x(qr[4]) # 1 at qubit 4
| qc.x(qr[5]) # 1 at qubit 5
| qc.h(qr[2]) # create a superposition on qubit 2
| qc.h(qr[3]) # create a superposition on qubit 3
|
| for j in range(7):
|     qc.measure(qr[j], cr[j])
```

It may help to visualize the circuit:



Then, the results of a single measurement can be represented via sound with the following:

```
| quantum_play_notes(qc,4)
```

Or, the results of a series of measurements can be represented via sound with the following:

```
| quantum_play_chords(qc,45)
```

This time, we have increased the argument indicating how many times to identically prepare the circuit and repeat the experiment. Since we have four notes, we want to have a good probability of having each of the four notes represented in the course of our experiment, so increasing this argument should give us a better chance.

# Summary

Qiskit can be used to run quantum computing code in a local or remote simulator, or directly on the IBM QX hardware. There are two main code repositories in Qiskit. The one commonly referred to by Qiskit alone is also known as Qiskit Terra, and focuses on low level quantum programming via directly manipulating qubits and quantum and classic registers. Qiskit can be used by programming in Python directly, or by loading OpenQASM files. The code repository in the Qiskit release is called Qiskit Aqua and contains a library of higher level quantum and classic algorithms useful in building larger quantum programs within the framework, and in specific cases where the algorithms directly apply. Qiskit Aqua will likely be similar to the way we program for quantum computers in the future, as ones that are practical for real-world use cases come online.

In the third part of the book, on quantum algorithms (next chapters), we will learn about the common quantum algorithms that can form subroutines of larger programs and offer the potential to demonstrate the superiority of quantum computing over classical computing as quantum hardware improves. The next chapter will go over the quantum AND and quantum OR gates necessary to power these quantum algorithms.

# Questions

1. Modify the `quantum_play_notes` and `quantum_play_chords` functions to optionally run on IBM QX. Note that, not all 128 possible values are represented in the `midi_conversion_tables` we provided, just the 88 possible values from a piano keyboard. Ignore results that are outside of this range in your code for now.
2. Create a quantum circuit that represents a superposition between the notes *A4* and *C5*. Listen to your superposition.
3. Create a quantum circuit that represents a superposition between the notes *F4*, *A4*, and *C5*, and optionally other notes. Listen to your superposition.
4. Instead of encoding the volume of a note by its probability, as derived from a series of quantum measurements of a quantum circuit in which the note appears as a superposition, encode the duration of the note. You will have to modify the `play_notes`, `quantum_play_notes`, and `quantum_play_chords` functions.
5. Since running on IBM QX will potentially produce noise that will cause us to have binary output corresponding to one of the unrepresented values, modify your code to make full use of the full 128 values a 7-bit binary value can encode. Choose what to do in this scenario.  
Previously, we ignored such a value. Now, choose to extend the conversion tables to have all 128 possible notes, or to use the spillover into binary values outside of the table's range to encode additional information, for example, note duration, relative to the previous note.
6. Pick one particular algorithm from Qiskit Aqua to investigate further and run it. Vary the parameters. Can

- you write a short description of your results?
7. The MIDI code isn't optimal for encoding superpositions of notes in a chord, as notes within a common chord aren't necessarily close to each other in the MIDI binary code. Design a code that takes 5 bits or fewer and can easily encode any major chord.

# **Quantum AND (Toffoli) Gates and Quantum OR Gates**

This chapter explores some of the quantum equivalents of classic Boolean logical gates, with the aim of being able to specify logic problems to be solved by a quantum computer for use with Grover's algorithm and other algorithms.

The following topics will be covered in this chapter:

- The 3SAT function and the process of inverting it
- A classic algorithm for inverting the 3SAT function
- The Toffoli gate: The quantum equivalent of the AND gate
- The quantum equivalent of the OR gate
- Specifying 3SAT problems using quantum circuits

# Technical requirements

Code for this chapter is available in a Jupyter Notebook under <https://github.com/PacktPublishing/Mastering-Quantum-Computing-with-IBM-QX>, in the `Chapter09` directory. Data files for the chapter are available in the same directory.

# Boolean satisfiability problem (SAT)

In computer programming, we often deal with logical conditions, `if/else` statements and loops being two prominent examples that evaluate expressions to determine whether they are true or not, and thereby determine future actions of the code. Most frequently in the code we write, the expressions being evaluated for the logic of the code are short and simple. Here's an example:

```
a=True  
b=False  
if a and b:  
    print("Ready for the next section.")  
else:  
    print("Change one variable to be ready to move on.")
```

This prints "Change one variable to be ready to move on". It should be obvious to any programmer which variable to change.

An expression with variables joined by the operators AND, OR, and NOT as well as parentheses is called a Boolean expression. Boolean expressions can get as lengthy as desired and, in addition to being useful in programming logic flow, can be useful to encode information about a particular situation.

For example, if we wanted to make a program to determine every possible combination of documents that are acceptable to bring on the first day of a new job to fill out paperwork (see the US regulations here: <https://www.uscis.gov/i-9-central/acceptable-documents>), we could encode that with the following:

```

def
documents_ok(passport,permanent_resident_card,drivers_license,voter_registration,
under_18,report_card,doctor_record,daycare_record,school_id,social_security_card,birth_certificate):
    return (passport or permanent_resident_card) or \
        ((drivers_license or school_id or voter_registration) or \
        (under_18 and \
        (report_card or doctor_record or daycare_record)) and \
        (social_security_card or birth_certificate))

```

Note that, for the US regulations case, I have already simplified things considerably by leaving out many possible documents. In this case, a document checker reading the `if` statement would probably be able to easily determine whether any particular document combination is valid, from working through the code with the set of documents they have in their hand. But imagine the document checker's boss wants a table of all possible valid combinations on one side, and all possible invalid combinations on the other. In this case, the document checker would have a nearly impossible job unless they were very systematic about checking every possible case and had a lot of time on their hands. There are 11 variables so there are  $2^{11} = 2048$  possible combinations of documents a person can bring in their first day of the job to give to the document checker.

To make this task more feasible, we could design an algorithm which validates the acceptable document combinations by iterating through every possible combination of true/false for the

**variables** `passport, permanent_resident_card, drivers_license, voter_registration, under_18, report_card, doctor_record, daycare_record, school_id, social_security_card, and birth_certificate` and checks whether or not the `documents_ok` function returns true for that combination. If it does, then that particular combination of documents is valid. Then, the document checker would only need to give the algorithm the documents, and the algorithm would spit out yes or no.

Here is some code to do this:

```

import itertools
works=[]
doesnt_work=[]
for combo in itertools.product([True, False], repeat=11):
    if
documents_ok(combo[0], combo[1], combo[2], combo[3], combo[4], combo[5], combo[6], combo[7], combo[8], combo[9], combo[10]):
    works+=[combo]
else:
    doesnt_work+=[combo]
print(len(works), len(doesnt_work))

```

For this setup, we get that 2005 combinations of documents work, and only 43 possible combinations of documents do not work.

Now, imagine the government gets together and implements new legislation in which there are thirty different possible documents, which means

$2^{30} = 1,073,741,824$  possible combinations and which the formula to determine whether a set of documents is valid is long and difficult but still takes only a few minutes per document set for the document checker to check. After a few days on the job with the new legislation, the document checker checks a few hundred people's documents per day, but no one's documents are valid. Their boss is upset, and both of them begin to suspect there is a flaw in the legislation, and that perhaps there is no combination of valid documents.

The boss orders the document checker to see if there are any valid combinations. There seems to be fewer than in the last scenario, so if they could just list the few combinations that are valid, they could save everyone time. This could take a while, but luckily the document checker has a computer program. In the worst case, the computer program will need to go through every one of the 1,073,741,824 possibilities for combinations of documents to see if each is valid before reaching the end and giving up. The document checker thinks that if we study

the formula, we can see whether there might be vast classes of possibilities they could rule out all at once, saving the code some time.

Before we can get to running the code, we get interference. The government got wind of the boss and the document checker's plan to discredit them and so no longer publish the formula, which they claim could be taken advantage of if widely known. Instead, each time the document checker wants to check a combination of documents, they must call a toll free number and wait on hold, tell the operator the combination of documents, and the operator will say yes or no. It takes 4 minutes to check each combination of documents, so the document checker can still reach their target of 100 documents checked per day, but it would now take more than 30,000 years working at 40 hours a week for the document checker to be able to prove the government wrong in this situation; this example is admittedly contrived to illustrate what a checker function such as `documents_ok` could be used for. The document checker now suspects there is no possible combination of documents that are valid, but there is no way to prove that.



*In the quantum computing literature, a function that you don't have access to which can check whether a given input is the correct input is known as an **oracle** function. I won't use this terminology but it might help when interpreting other writings on the subject.*

The process of figuring out whether any possible combination of Boolean variables could satisfy a Boolean expression is known as the **Boolean satisfiability problem** or **SAT**. In this section, we will see that a quantum computer could let the document checker prove whether or not there is any combination of valid documents in just 1 year of full-time work, instead of more than 30,000. It's not perfect, but it definitely gives the quantum computer a computational advantage and makes a problem that classically would be infeasible tractable.

# 3SAT classical implementation

Any Boolean satisfiability problem can be rewritten in the form of distinct sections of variables containing `or` and `and` functions between them. The individual variables can be negated with `not` wherever needed in this form, called a conjunctive normal form. We can further rewrite any expression in this form to contain at most three variables in each section, which will be useful in algorithmic analysis (dummy variables otherwise not used can be introduced to help). This is done to simplify the algorithmic analysis. For example, the following is a Boolean expression over four variables in the conjunctive normal form restricted to three variables per section:

```
| (not a or b or d) and \
| (not b or c or d) and \
| (a or not c or d) and \
| (not a or not b or not d) and \
| (b or not c or not d) and \
| (not a or c or not d) and \
| (a or b or c) and \
| (not a or not b or not c)
```

The Boolean satisfiability problem (SAT) associated with this expression would ask whether there is any combination of `a`, `b`, `c`, and `d` values for which the expression evaluates to true. Since we've restricted our sections to contain at most three variables, this variation of the Boolean satisfiability problem is known as 3SAT.

Now, we can write a function to determine whether the Boolean expression given before is satisfiable:

```
| def checker(f,candidate):
|     """
```

```

        f: can be any function which takes as a string of 1s and 0s which is
at least as long as the number of variables in f. Right most character is
the 0th bit.
    candidate: should be a string of 1s and 0s which is at least as long
as the number of variables in f. Right most character is the 0th bit.
    """
    return(f(candidate))

def try_all_inputs(f,len_input):
    import itertools
    result=[]
    for candidate in ["".join(seq) for seq in itertools.product("01",
repeat=len_input)]:
        if checker(f,candidate):
            result+=[candidate]
    return result # list of inputs which the function
                  # f validates to true

def is_satisfiable(f,len_input):
    return len(try_all_inputs(f,len_input))>0

def a_3sat_function_4(binary_string):
    """
    binary_string is a string that is at least 4 characters long with 1s
and 0s with the rightmost character representing the 0th bit
    """
    binary_list=[int(i) for i in binary_string]
    binary_list.reverse()
    a,b,c,d=binary_list[0:4]
    return (not a or b or d) and \
           (not b or c or d) and \
           (a or not c or d) and \
           (not a or not b or not d) and \
           (b or not c or not d) and \
           (not a or c or not d) and \
           (a or b or c) and \
           (not a or not b or not c)

```

Using the previous code, we can see if our function `a_3sat_function` is satisfiable, and get a list of all possible inputs which satisfy it with the following code:

```

print(is_satisfiable(a_3sat_function_4,4))
print(try_all_inputs(a_3sat_function_4,4))

```

With the previous code, we see that `a_3sat_function` is satisfiable and the inputs which satisfy it are `1010` and `1110` as the following is printed:

```

True
['1010', '1110']

```

# **3SAT - why is it so interesting?**

3SAT is an especially interesting problem to solve as many other problems can be mapped to its formulation. 3SAT is also a well known NP-complete problem. For more about NP-completeness, check out <http://mathworld.wolfram.com/NP-CompleteProblem.html>.

# Quantum AND and OR

We've already met the quantum equivalent of the classic NOT gate: the X gate. 3SAT requires the logical AND and logical OR of variables, so we will need the quantum equivalent of the logical AND and logical OR to formulate 3SAT in a quantum computer. All quantum gates are reversible, which was no problem for translating the classic NOT gate directly into a single quantum gate, as the classic NOT gate is itself reversible. To reverse the classic NOT gate, simply apply it twice to get back to where you started. The same goes for the quantum gate.

However, the classic OR gate and the classic AND gate are not reversible. To see this, we can draw out their truth tables:

<b>a</b>	<b>b</b>	<b>a AND b</b>
0	0	0
0	1	0
1	0	0
1	1	1

And the truth table for the OR gate is as follows:

<b>a</b>	<b>b</b>	<b>a OR b</b>
0	0	0
0	1	1
1	0	1
1	1	1

We can see that in the case of the classic AND gate, the output 0 could have resulted from bits a and b being 00, 01, or 10 respectively, so given only the output information, it is impossible to uniquely determine the input (unless the output is 1). We can see that for the OR gate, the output 1 could have resulted from bits a and b being 01, 10, or 11. This means, that given only the output information, it is impossible to uniquely determine the input (unless the input is 0). Neither of these functions is fully reversible and thus we would have to extend them to be, so we translate them into their quantum equivalents. The way we will do this is to introduce an additional third input.

# Toffoli gate - quantum AND gate

Let's consider a classic reversible AND gate. We could do this by using three input bits and three output bits. Here, the gate will flip the third bit if the first two bits are 1, leaving the first two bits alone. It is an AND gate in the sense that it performs an action only if the AND of the first two bits evaluates to true.

We can implement this gate by making the last output bit equal to the last input bit XOR with: this first input bit AND the second input bit. That is to say, our reversible AND gate will map input bits  $a$  to output bit  $a'$ , input bit  $b$  to output bit  $b'$ , and input bit  $c$  to output bit  $c' = c \text{ XOR } (a \text{ AND } b)$ .

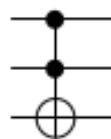
We can see its truth table as follows:

<b>Input - abc:</b>			<b>Output - a'b'c':</b>		
<b>a</b>	<b>b</b>	<b>c</b>	<b>a'</b>	<b>b'</b>	<b><math>c' = c \text{ XOR } (a \text{ AND } b)</math></b>
0	0	0	0	0	0
0	0	1	0	0	1

0	1	0	0	1	0
0	1	1	0	1	1
1	0	0	1	0	0
1	0	1	1	0	1
1	1	0	1	1	1
1	1	1	1	1	0

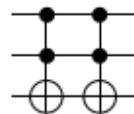
Since the input is three bits and the output is three bits, we can easily reverse the output to get back the input. To map output to input, we leave the first two bits ( $a$  and  $b$ ) unchanged and equal to  $a'$  and  $b'$  respectively, and set the bit  $c$  to be equal to  $c' \text{ XOR } (a' \text{ AND } b')$ . With this, we can get back exactly the input from the output, reversing the gate.

The Toffoli Gate is the quantum AND gate and it implements the previous truth table, taking three qubits as input, and outputting three qubits. Its symbol is as follows:



Here, the two black dots indicate the bits we have referred to as  $a'$  and  $b'$ , and the open circle indicates the bit we have referred to as  $c'$ . The wires coming into the symbol indicate  $a$ ,  $b$ , and  $c$ . The Toffoli gate is also known as the **controlled-controlled-NOT** gate (**CCNOT** gate). Since the CNOT gate within Qiskit is known as `cx`, this means that the CCNOT gate is known as `ccx`.

The Toffoli gate can be enacted, then reversed, with the following circuit:



It can be imported from Qiskit as follows:

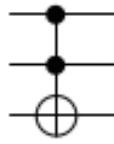
```
| from qiskit.extensions.standard import ccx
```

The gate can then be called with the following, for quantum registers of  $i$ ,  $j$ ,  $k$  indices respectively:

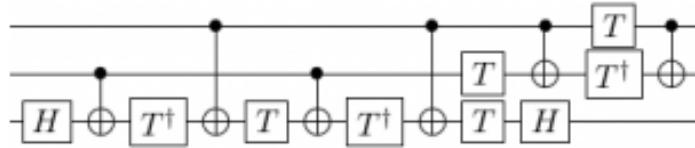
```
| qc.ccx(qr[i], qr[j], qr[k])
```

Implementing the Toffoli gate in terms of the elementary gates we have learned so far is quite a procedure. I will not go into details about how we arrived at the implementation, but I will provide the circuit and the OpenQASM for it. This is especially important if you interact with IBM QX via the Quantum Composer as the Toffoli `ccx` gate is not available there. This means that in the IBM Quantum Composer each instance of the Toffoli gate must be implemented using a combination of the available gates.

The Toffoli circuit is as follows:



The preceding circuit is equivalent to the following:



In OpenQASM, this translates to the following (for inputs a, b, and c residing on qubits 0, 1, and 2 respectively; otherwise indices will need to be modified accordingly):

```

include "qelib1.inc";
qreg q[3];
creg c[3];
// input is here:
// x q[0]; // uncomment if a=True
// x q[1]; // uncomment if b=True
// x q[2]; // uncomment if c=True

// Toffoli Gate on input:
h q[2];
cx q[1],q[2];
tdg q[2];
cx q[0],q[2];
t q[2];
cx q[1],q[2];
tdg q[2];
cx q[0],q[2];
t q[1];
t q[2];
h q[2];
cx q[0],q[1];
t q[0];
tdg q[1];
cx q[0],q[1];

// Measuring output:
measure q[0] -> c[0];
measure q[1] -> c[1];
measure q[2] -> c[2];

```

**i** Although I provide the OpenQASM for the Toffoli gate in terms of the fundamental gates we have learned so far, for use directly in the UI of the IBM QX, I will use the shorthand `ccx` for the gate for the rest of the book, and leave it to the user who wants to implement the gate directly in the UI of the IBM QX to refer back to this code to do

*so. We will primarily work with the `ccx` gate from Qiskit, where we can use the `ccx` notation shorthand directly.*

In this chapter's code examples in the Jupyter Notebook on GitHub, there are full examples using both Qiskit Python `ccx` function and the full OpenQASM implementation to print the truth table for the Toffoli gate.

# Quantum OR gate

Next, let's design a classic, reversible OR gate. As with the classic reversible AND gate, the key to enable reversibility is to have three input bits and three output bits. Here, we will flip the third bit in the output only if one, two, or both of the first two bits are 1. It is an OR gate in the sense that it performs an action only if the OR of the first two bits evaluates to true. We can implement this gate by making the third bit equal to the negation of the first bit XOR with: the negation of a AND the negation of b. That is to say, our reversible AND gate will map input bit a to output bit  $a'$ , input bit b to output bit  $b'$ , and input bit c to output bit  $c'=(\text{NOT } c) \text{ XOR } (\text{NOT } a \text{ AND NOT } b)$ .

In this case, the truth table will be the following:

Input - abc:			Output - $a'b'c'$ :		
a	b	c	$a'$	$b'$	$c'=(\text{NOT } c) \text{ XOR } (\text{NOT } a \text{ AND NOT } b)$
0	0	0	0	0	0
0	0	1	0	0	1

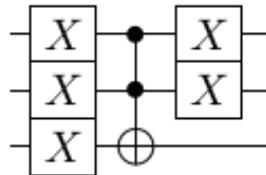
0	1	0	0	1	1	
0	1	1	0	1	0	
1	0	0	1	0	1	
1	0	1	1	0	0	
1	1	0	1	1	1	
1	1	1	1	1	0	

Now, we can see that, if we just look at a and b as the inputs and c as the output, variable c after the computation will contain the equivalent of a OR b. However, since the input is three bits and the output is three bits, we can easily reverse the output to get back the input. To map output to input, we leave the first two bits (a and b) unchanged and equal to  $a'$  and  $b'$  respectively, and set bit c to be equal to  $c' \text{ XOR } (a' \text{ or } b')$ . With this, we can get back exactly the input from the output, reversing the gate.

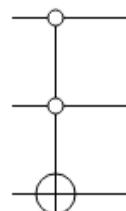
How can we actually implement this as a quantum gate? Well, notice that the reversible OR classic gate looks identical to the reversible classic AND gate, except each of its arguments is negated. So, we can use the Toffoli gate to implement the OR gate quantum mechanically. First, we must just negate each of the three input qubit with the X gate, then run the Toffoli gate, and then, after our

computation, be sure to reverse the negation on the first and second qubits by applying the X gate to each to get the desired output.

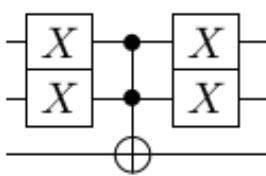
The quantum circuit for the quantum OR gate is the following:



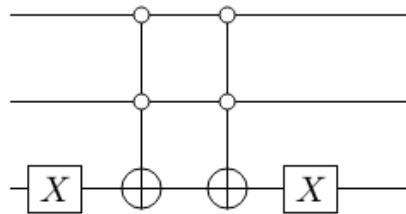
We can shorten the notation for the quantum OR gate by specifying just the symmetric part in a shorter notation, which I will call the partial quantum OR gate. This will allow us to use the notation to enact the OR gate or reverse the OR gate, with the only addition being an X gate on the third bit, and the only change between enacting and reversing being which side of the compact symmetric notation the X gate goes on (on the left side when enacting; the right side when reversing). This notation is as follows:



This is equivalent to the following:

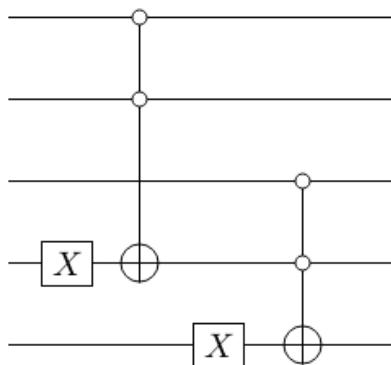


Thus, the quantum OR gate can be enacted then reversed with the following circuit:



*The quantum OR gate isn't symmetric. One side has three X gates, the other only two. To reverse a quantum operation, we need to carefully track the number of X gates. This is why our notation shorthand is only the symmetric part of the quantum OR, and the extra X gate is written separate to the notation.*

For our 3SAT problem, we will need to compute the OR of three qubits. We can do this by stacking quantum OR gates. We do this by noting that  $(a \text{ OR } b \text{ OR } c)$  is logically equivalent to  $((a \text{ OR } b) \text{ or } c)$ , so we can group OR results for two bits using our existing quantum OR gate, before ORing the result with the final qubit. Here is an example circuit that computes a quantum OR over three qubits (with two intermediate output qubits) by breaking the circuit up into two parts. The first, outputs to a qubit which serves as one of the input qubits to the second part:

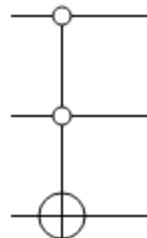


The OpenQASM for the full quantum OR gate is as follows (for inputs  $a$ ,  $b$ , and  $c$  residing on qubits 0, 1, and 2

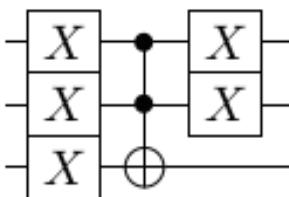
respectively; otherwise indices will need to be modified accordingly):

```
include "qelib1.inc";
qreg q[3];
creg c[3];
// input is here:
// x q[0]; // uncomment if a=True
// x q[1]; // uncomment if b=True
// x q[2]; // uncomment if c=True
// Quantum OR gate:
x q[0];
x q[1];
x q[2]; // comment out this line if the gate is to be symmetric and match
the shortened notation for the partial quantum OR gate.
h q[2];
cx q[1],q[2];
tdg q[2];
cx q[0],q[2];
t q[2];
cx q[1],q[2];
tdg q[2];
cx q[0],q[2];
t q[1];
t q[2];
h q[2];
cx q[0],q[1];
t q[0];
tdg q[1];
cx q[0],q[1];
x q[0];
x q[1];
measure q[0] -> c[0];
measure q[1] -> c[1];
measure q[2] -> c[2];
```

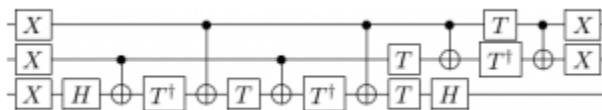
Note that the line `x q[2];` should be commented out for the gate to match the symbol:



The full gate is shown as follows:



The preceding gate is equivalent, using the fundamental gates we have learned so far, to the following circuit:



We will primarily work with the quantum OR gate from Qiskit, where we can use the `ccx` notation shorthand directly for clarity. When the `ccx` gate shorthand is not available, refer to this section for its implementation in terms of more fundamental gates.

In this chapter's code in the Jupyter Notebook available on GitHub, under the section *Showing the quantum OR gate truth table*, there is an example of using this OpenQASM implementation to print the truth table for the quantum OR gate.

# Quantum AND and Quantum OR over multiple qubits

To properly compute a 3SAT function, we would need to have a quantum OR that functioned over four qubits. Creating a quantum OR or a quantum AND over more qubits is certainly possible. Classic AND and classic OR are both associative. That means we can group an expressions into pairs however we choose, and compute intermediate results, rewriting the computation of AND or OR over many variables. Via this rewrite, we can transform the computation into one where just two variables are operated on at a time, via an AND or an OR operation. For example, to classically rewrite a series of ANDs over four variables as an AND over pairs of variables, we could do the following:

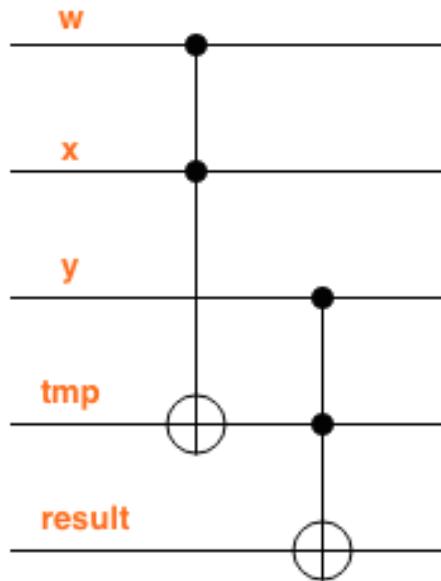
$$|w \text{ AND } x \text{ AND } y = ((w \text{ AND } x) \text{ AND } y)$$

We could write the parentheses however we choose, and there are a variety of ways to do so while keeping the AND working over pairs of variables alone. This works for OR as well, for example:

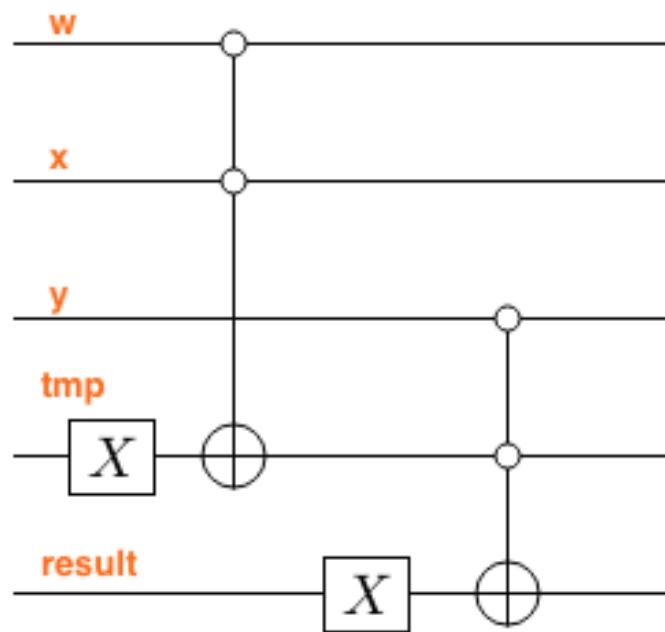
$$|w \text{ OR } x \text{ OR } y = ((w \text{ OR } x) \text{ OR } y)$$

We can use the same strategy with the quantum gates to create quantum AND and a quantum OR that work over as many qubits as we want, simply by working over three qubits at a time and using some auxiliary qubits as scratch space to store the intermediate results. For example, the

quantum version of ((w AND x) AND y) can be implemented as this quantum circuit:



The quantum version of ((w OR x) OR y) can be implemented as this quantum circuit:



# 3SAT quantum circuit implementation

Let's put everything together to implement a 3SAT problem in a quantum circuit. In the next chapter on Grover's algorithm, we will use the formulation we develop in this section as input to Grover's algorithm to solve for the unique input that satisfies it.

The 3SAT problem we will implement in a quantum circuit is as follows:

$$(a \vee b \vee \neg c) \wedge (a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$$

Here,  $\vee$  stands for `or`,  $\wedge$  for `and`, and  $\neg$  for `not`. Logical problems such as 3SAT are often specified in this notation, so I introduce it here. But don't worry, I will also translate it directly to Python since that is more familiar to the majority of readers:

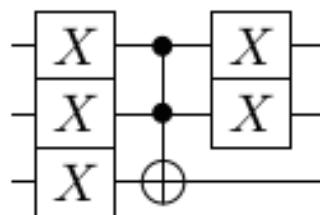
```
| (a or b or not c) and \
| (a or b or c) and \
| (a or not b or c) and \
| (a or not b or not c) and \
| (not a or b or not c) and \
| (not a or b or c) and \
| (not a or not b or not c)
```

Now, we want to implement this as a quantum circuit. We will need the quantum AND (Toffoli) gate as well as the quantum OR gate, along with the ability to compute these over multiple qubits. Based on the previous formula, we need to take a quantum OR of three qubits eight times, and

then take the quantum AND of the eight qubits representing these eight results. Not surprisingly, given that the quantum AND and quantum OR gates operate on two qubits at a time, and when they are reformulated to operate on more than two qubits require temporary qubits to store intermediate results, we will need a lot of intermediate qubits in this circuit (a total of 23 qubits if we don't reuse intermediate qubits, meaning three qubits to store the logical values of a, b, and c; one qubit to store the result; and 19 qubits to store intermediate results).

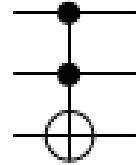
The good news for when we run this on a practical quantum computer is that, since quantum gates are reversible, we can generally clean up the temporary qubits after their use for reuse later in the computation. Since real-world quantum computers have a limited number of qubits available for use (and simulators begin to slow the more qubits you use), this reuse will come in handy.

However, for clarity to start out with, before we build in the reusability of the temporary qubits, we will first specify the quantum circuit using a new temporary qubit each time one is needed. This will allow us to more clearly see what is going on in the circuit. Moreover, since we will use the circuit to implement directly in IBM QX or with Qiskit, I will specify in the quantum OR in full notation using only quantum AND (Toffoli) gates and the X gates, as we initially learned it:

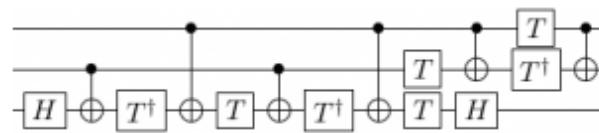


So, the circuit is more easily directly translated into IBM QX. We also note that, even this circuit isn't simple enough, and

that we will have to expand all of quantum AND elements in our circuit appearing as:



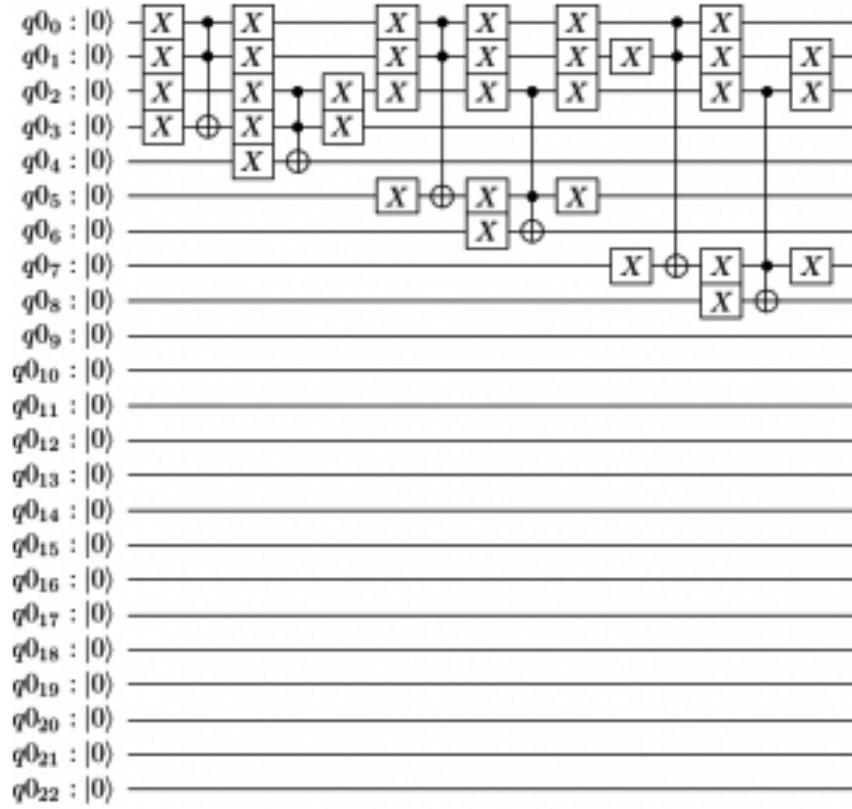
The elements will need to be replaced with their full representation:



But, already we have a very complicated circuit! I will divide the circuit into three parts, meant to be run sequentially, so that the circuit even fits on the page. The first part will be the computation of the first three clauses:

$$(a \vee b \vee \neg c), (a \vee b \vee c), (a \vee \neg b \vee c)$$

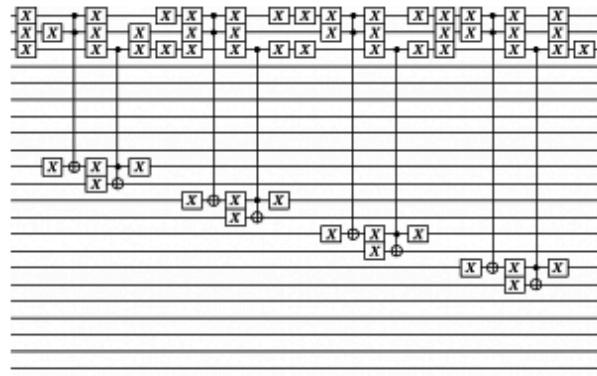
This computation of the first three clauses is stored into intermediate qubits via the following circuit:



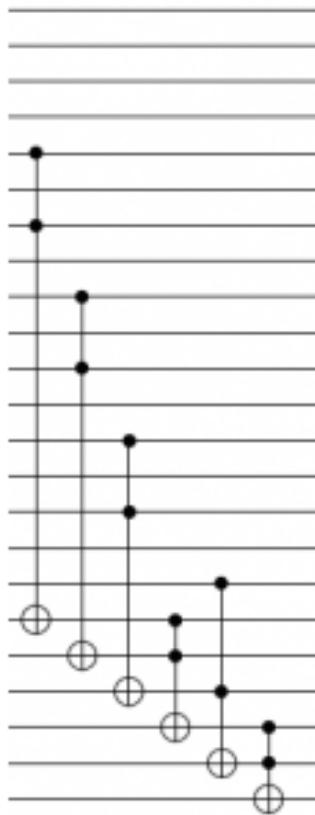
The second will be the next four clauses:

$$(a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$$

This computation of the next four clauses will be stored into intermediate qubits in the following circuit:



The final step will be to compute the quantum AND of each of these temporary variables together to get the final result in the very last wire:



# Summary

In this chapter, we learned about the Boolean satisfiability problem (SAT) and a specific formulation of a special case of it called 3SAT, in which each clause is a logical OR of at most three variables, and the clauses are connected by logical AND functions. We learned how to reformulate the classic AND and OR gates to be reversible and used this to formulate the quantum gate equivalent of the classic AND and classic OR gates, the quantum AND (Toffoli) gate and the quantum OR gate. We then learned how to use these gates to operate on more than two qubits at once by chaining them and using temporary variables.

3SAT is an especially interesting problem to solve as many other problems can be mapped to its formulation, and in the next chapter we will use a quantum algorithm, Grover's algorithm, to solve an instance of 3SAT. In this chapter, we used our knowledge of quantum AND and OR gates to encode a 3SAT problem in a quantum circuit, in preparation for using this quantum circuit as input to Grover's algorithm in the next chapter.

# Questions

1. Challenge problem: Rewrite the `documents_ok` function provided in this chapter to be in the 3SAT form. It turns out that any logic problem can be rewritten in the 3SAT form. Sometimes, however, this rewriting takes a while to do by hand or by algorithm.
2. Challenge problem: design a reversible classic NAND gate.
3. Challenge problem: design a quantum OR gate that operates over four qubits.
4. Design a quantum circuit that represents the logic problem a AND NOT b.
5. Challenge problem: draw the circuit for
$$(a \vee b \vee \neg c) \wedge (a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee c)$$
using fewer temporary qubits. Hint: after finishing using a temporary qubit, figure out how to reset it so that it can be used again.
6. Challenge problem: write up the circuit for
$$(a \vee b \vee \neg c) \wedge (a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee c)$$
using OpenQASM and only the gates accepted by IBM QX, as well as your formulation, using the fewest number of temporary qubits possible. Hint: you might want to write some Python code to help you write the OpenQASM consistently, particularly substituting the Toffoli gate shorthand for its full expansion in terms of gates available on the IBM QX.

# Grover's Algorithm

Grover's algorithm is a quantum algorithm that can be used to invert a function more quickly than any classic algorithm. Essentially, that means if there are  $N$  possible solutions to a function, only one of which is correct, Grover's algorithm works to efficiently find the one solution out of the pack. Grover's algorithm is sometimes referred to as quantum search or even as quantum database search. This chapter first introduces Grover's algorithm in depth.

It then goes over a specific use case of Grover's algorithm to illustrate its potential: that of solving the Boolean satisfiability problem to figure out which inputs correspond to satisfying a certain Boolean function, with an example of using 3SAT that returns true for exactly one combination of inputs.

To show Grover's algorithm at work for 3SAT, we will have to extend our work with quantum OR gates in the previous chapter to work over three inputs. The chapter will close with a few words about using Grover's algorithm more generally to solve the case of inverting other types of functions and provide some comments on its generality.

The following topics are covered in this section:

- Introduction to Grover's algorithm
- Implementing a 3SAT function to use as a Grover's algorithm checker
- Implementing Grover's algorithm

# Technical requirements

The Jupyter Notebook for this chapter is available at: <https://github.com/PacktPublishing/Mastering-Quantum-Computing-with-IBM-QX>, under Chapter10.

# An overview and example use case of Grover's algorithm

In the previous chapter we went over the Boolean satisfiability problem (SAT) and the special case when each clause contains at most three variables (3SAT), and is satisfied by exactly one combination of variables (exactly-1 3SAT). We will use Grover's algorithm to find the solution to an exactly-1 3SAT problem, `3sat_mystery` defined as follows:

$$\begin{aligned} \text{3sat\_mystery}(a, b, c) = \\ (a \vee b \vee \neg c) \wedge (a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee c) \end{aligned}$$

For the above, we drew the quantum circuit in the previous chapter directly in Qiskit. Grover's algorithm will find the unique combination of  $a$ ,  $b$ , and  $c$  for which this function returns true.

Classically, if we had no further information about the function, we would have to check all possible combinations of  $a$ ,  $b$ , and  $c$ , each of which could be true or false, leading to  $N = 2^3 = 8$  possible solutions to check, each of which involves invoking the `3sat_mystery(a, b, c)` function and seeing whether it returns true or false. That means that on a classical computer, to get the result, our algorithm would take on the order of  $N$  function calls. Luckily on a quantum computer, we can create an algorithm which requires fewer function calls: we can use Grover's algorithm

to get the correct result in the order of  $\sqrt{N}$  function calls. In this case, that would mean we would have to call the quantum equivalent of the `3sat_mystery(a,b,c)` function approximately  $\sqrt{N} = \sqrt{8} = 2.8$  times.

This might not seem like such a big deal when there are only eight possible solutions to check, but generally we could imagine a 3SAT function over many variables, say 30. Then, we would have to, in the worst case, check  $2^{30}$  possible inputs (more than a billion) and call the function a billion times classically, whereas Grover's algorithm can do this in closer to 32,000 function calls, far fewer than a billion. The gap only widens as the number of inputs grows, showing the quantum advantage.

To show Grover's algorithm at work for 3SAT, we will have to extend our work with quantum OR gates in the previous chapter, to work over three inputs (as each clause in the 3SAT specification is an OR over three inputs), and develop a strategy to take the quantum AND over as many qubits as necessary (Here, our 3SAT function has an AND among seven clauses, so we will need a seven-way AND, but the strategy we develop will be able to be extended to an AND over an arbitrary number of qubits.) We will also have to develop gates to reverse the quantum AND and quantum OR gates, for two and three inputs as needed. Then, we will use these in combination to specify the `3sat_mystery(a,b,c)` function on a quantum computer, as well as the steps needed to reverse the computation of this function. The variables in the function will be mapped to qubits and the clauses in the function will be mapped to quantum gates. We will then develop the gates to perform the second step in Grover's algorithm, which gradually shifts the inputs toward the final solution. Then, we will run Grover's algorithm on our `3sat_mystery(a,b,c)` function and show that it produces the

correct result in just two evaluations. In this example, `3sat_mystery(a,b,c)` is simple enough that we can efficiently compute the solution classically: the solution is  $a = 1$ ,  $b = 1$ ,  $c = 0$ , or, written three bits in a row: 110.

There are two primary ingredients to Grover's algorithm:

- The `checker` function: Checks whether or not an input satisfies a function
- The `mover` function: Moves the portion of the inputs that are incorrect closer to the portion of the input that is correct



*Grover's algorithm can also work on checker functions for which more than one input satisfies their conditions, but for simplicity, we will focus on checker functions where exactly one input satisfies the function.*

If our input to the checker step was 100% wrong, the mover step wouldn't have any hope of working. For example, if our `checker` function operated on three input qubits, outputting 1 only for the input  $|110\rangle$ , and we give it input  $|101\rangle$ , things would be hopeless. Our `mover` function would have no idea where to move. The key is to use the fact that inputs in quantum computing can in general be superpositions. Now, imagine we give the checker function as input a superposition of all possible inputs; in the case of three qubits, that would be this state:

$$|“\text{all three qubit inputs }”\rangle =$$

$$\frac{1}{\sqrt{8}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle)$$

This state can be created by placing an H gate on each input qubit, which puts that input qubit in a superposition between 50%  $|0\rangle$  and 50%  $|1\rangle$ . That would mean that, if this input were to be measured, there would be a 1/8 chance of any of the eight outputs happening.

Now, if we feed `/"all three qubit inputs"` as input to our checker function, the `checker` function will have a different output for the portion of `/"all three qubit inputs"` that is the solution to `3sat_mystery(a,b,c)` (`/"110"`), than for the portion of `/"all three qubit inputs"` that corresponds to any other qubit combination. The `checker` function operates on the entire `/"all three qubit inputs"`, so the full output is the superposition of these portions.

Then, it is the `mover` function's job to move the `/"all three qubit inputs"` input closer to solution of the `3sat_mystery(a,b,c)` function, `/"110"`, which is the desired output. This is done by reducing the chance that we measure anything but `/"110"`. To start out with, we had a 1/8 chance of observing any input. The `mover` function will use the output of the `checker` function to modify the input state and to increase the probability of observing `/"110"` and decrease the probability of observing anything else. To increase the probability enough so that it is likely in just one measurement to get the correct input that satisfies the `checker`, we will in general need to run the `checker` function and then the `mover` function steps multiple times, until the chance that we get the correct input when we measure, is very high.

# Grover's algorithm steps

Now that we have defined the `checker` and the `mover` functions, in this subsection we will see how they can be used together to solve the `3sat_mystery(a,b,c)` in Grover's algorithm. The steps to Grover's algorithm are the following:

1. Setup step
2. Checker step
3. Mover step

The following subsections will define each step in detail.

# Setup step

During the setup step, we put the input qubits, which start out as  $|0\rangle$ , in a superposition by applying the H gate to each of them. This is done with the following code:

```
| for i in range(num_inputs):
|     qc.h(qr[i])
```

Recall that  $H|0\rangle = |+\rangle$ , so we have put all the input qubits in the plus state. If we have three input qubits over which our `checker` function will operate, `num_qubits` in this example would be equal to 3.

We will also put the output of the checker function in the minus state, recalling that  $HX|0\rangle = H|1\rangle = |-\rangle$ , with the following code:

```
| # Setting up the output of the checker function
| qc.x(qr[num_registers-1])
| qc.h(qr[num_registers-1])
```

We will need many temporary quantum registers to get to the output to store intermediate results. `num_registers` will be set to the total number of quantum registers needed, the sum of the number of input registers, the register in which the checker output resides, as well as the number of temporary registers needed.

# Checker and mover steps

To use Grover's algorithm, we will then need to run both the checker and the mover steps in order at least once so that the mover step can move the input superposition toward the correct input. In general, we will need to run the check and mover steps  $\mathcal{O}(\sqrt{N})$  times, where  $N$  is the number of possible inputs. For example, if  $N = 8$  as in our example, the checker and mover steps will have to be run  $\approx 2.8$  times. Thus, we have `iterations`  $\approx 2$  in our example:

```
# Do the Grover's steps
for it in range(iterations):
    checker(qr,qc)
    mover(qr,qc,num_inputs)
```

# Naming conventions

In the quantum computing literature, the `checker` function is known as the **oracle** function. The term "oracle" can be confusing as it is used many places in computer science, so I will call this function the "checker" since, given an input, it checks whether the input is satisfactory and it is clearer from its name what it does.

In the quantum computing literature, the `mover` function is known as the **diffusion** function, which is a physics description for the operation. Since we aren't focusing on the physics in this book, I call the diffusion function the `mover` function so that it is clear what it does: it moves the input toward the correct solution.

# Measurement step

Finally, after the checker and mover steps have been run for enough iterations, they will have morphed our input /"all three qubit inputs"/ into a final state /"Grover final"/.

The /"three qubit input which satisfies checker"/ is returned with very high probability. In our example, the /"three qubit input which satisfies checker"/ is /"110"/. We can measure this state with the following code:

```
| for j in range(num_inputs):  
|     qc.measure(qr[j], cr[j])
```

This will give us the correct input that satisfies the checker with a very high probability. We will see, when we run the code using our 3SAT example, we get /"110"/ with more than 95% probability. Each function, number of inputs, and number of iterations will in general produce a different probability at the end, but Grover's algorithm operates to increase the probability from equal chance over all inputs to high chance for the correct input, and it is merely a question of how high that chance is.

# 3SAT as a Grover's algorithm checker

To implement any 3SAT function, we will need quantum AND and quantum OR gates that work on more than two qubits. We will then need to implement these gates as well as their equivalents that work over two qubits in Qiskit. To implement Grover's algorithm, we will need gates that reverse the quantum AND and quantum OR gates. In the previous chapter, we went through the OpenQASM to design a quantum AND and a quantum OR that function over two input qubits using one auxiliary qubit, storing the output as necessary for reversibility, for a total of three inputs and three outputs.

In this section, we will review these gates and implement them in Qiskit. We will extend the gates to work on three input qubits using two auxiliary qubits, which are necessary to store the output and for reversibility. We will then develop and test the combination of gates necessary to reverse each of these computations.

Note that the two-qubit quantum AND has a primitive available for use in the IBM simulator, both in OpenQASM and in Qiskit, called `ccx`. It currently is not enabled on the IBM QX hardware itself. If this gate is needed to be implemented on the hardware, refer to its equivalent in terms of the fundamental I, X, Y, Z, H, S,  $S^\dagger$ , T,  $T^\dagger$ , and CNOT gates in OpenQASM, and translate it to Qiskit. The computations done in this chapter are done in the simulator only, for two reasons. One, the number of qubits they require is more than the current quantum computers available (although with some careful work, this could be reduced to just be in

the range). Two, the number of gates required to perform the computation means that the computation will take far more time than the typical decoherence and dephasing time, meaning that running on current hardware would produce essentially random results. However, the results in the simulator will show that the code will work on the IBM QX hardware at some point, after the number of qubits available and the decoherence and dephasing times increase appropriately.

The multi-qubit logic gates necessary to implement a 3SAT function as specified in this section will all take as input a quantum circuit and a quantum register instance over which to operate, followed by the register numbers, in order, over which to operate. The convention will be that register numbers that are the logical qubits over which the gate is operating are specified by the names  $w$ ,  $x$ , and  $y$ , and that output registers necessary for output and reversibility are specified in terms of  $t_1$  and  $t_2$ . The "t" prefix is meant to indicate that they are in some sense "temporary" or "transient" and have nothing to do with the quantities  $T_1$  and  $T_2$  defined in [Chapter 3, Quantum States, Quantum Registers, and Measurement](#). These temporary registers are needed as, in a lengthy computation combining many AND and OR gates, many such registers will need to be used before outputting to the final register containing the single result.

# Two- and three-qubit quantum AND (Toffoli) in Qiskit

In the previous chapter, we learned about the two- and three-qubit quantum AND functions. This subsection gives their implementations in Qiskit.

The two-qubit quantum AND is simply the CCX gate called from Qiskit:

```
| def quantumand_2(qr,qc,w,x,t1):  
|     qc.ccx(qr[w],qr[x],qr[t1])  
|     return t1
```

The three-qubit quantum AND combines two of the two-qubit quantum ANDs with the help of a temporary register. This takes advantage of the fact that  $a \wedge b \wedge c = (a \wedge b) \wedge c$ , meaning that we can first compute  $a \wedge b$ , put the result in the temporary register  $t_1$ , and then compute  $t_1 \wedge c$  to get the final result, which we store in the temporary register  $t_2$ :

```
| def quantumand_3(qr,qc,w,x,y,t1,t2):  
|     qc.ccx(qr[w],qr[x],qr[t1])  
|     qc.ccx(qr[y],qr[t1],qr[t2])  
|     return t2
```

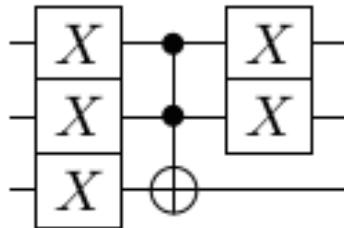
# **Quantum AND reverse**

The good news is that the quantum AND is its own reverse function, so just calling it twice on the same registers as were specified on its initial call, will reverse the inputs back to what they were before the call.

# Two- and three-qubit quantum OR in Qiskit

In the previous chapter, we learned about the two- and three-qubit quantum OR functions. This subsection gives their implementations in Qiskit.

The circuit for the two-qubit quantum OR was given in the last chapter and is the following:



Here, we can see that it can be implemented in Qiskit as the following:

```
def quantumor_2(qr,qc,w,x,t1):
    qc.x(qr[w])
    qc.x(qr[x])
    qc.x(qr[t1])
    qc.ccx(qr[w],qr[x],qr[t1])
    qc.x(qr[w])
    qc.x(qr[x])
    return t1
```

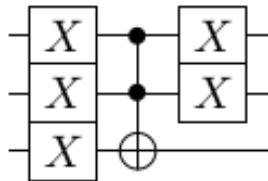
The three-qubit quantum OR takes a similar form to the quantum AND function, and combines two of the two-qubit quantum ORs with the help of a temporary register. This takes advantage of the fact that  $a \vee b \vee c = a \vee b \vee c$ , meaning that we can first compute  $a \vee b$ , put the result in the temporary register  $t_1$ , and then compute  $t_1 \vee c$  to get the final result, which we store in the temporary register  $t_2$ :

```
def quantumor_3(qr,qc,w,x,y,t1,t2):
    qc.x(qr[w])
    qc.x(qr[x])
    qc.x(qr[t1])
    qc.ccx(qr[w],qr[x],qr[t1])
    qc.x(qr[w])
    qc.x(qr[x])

    qc.x(qr[y])
    qc.x(qr[t1])
    qc.x(qr[t2])
    qc.ccx(qr[y],qr[t1],qr[t2])
    qc.x(qr[y])
    qc.x(qr[t1])
    return t2
```

# Quantum OR reverse

We can see that the following circuit diagram for quantum OR is not symmetric. As a reverse operation involves applying the same gates in the opposite order, we won't be able to use it as its own reverse function. However, the circuit can be easily implemented to exactly reverse the steps in the two input and three input quantum ORs we specified in the last subsection. The code to do so is given in this section.



The two-qubit quantum OR reverse is the following:

```
def quantumor_2_reverse(qr, qc, w, x, t1):
    qc.x(qr[x])
    qc.x(qr[w])
    qc.ccx(qr[w], qr[x], qr[t1])
    qc.x(qr[t1])
    qc.x(qr[x])
    qc.x(qr[w])
    return t1
```

And the three-qubit quantum OR reverse is the following:

```
def quantumor_3_reverse(qr, qc, w, x, y, t1, t2):
    qc.x(qr[t1])
    qc.x(qr[y])
    qc.ccx(qr[y], qr[t1], qr[t2])
    qc.x(qr[t2])
    qc.x(qr[t1])
    qc.x(qr[y])

    qc.x(qr[x])
    qc.x(qr[w])
    qc.ccx(qr[w], qr[x], qr[t1])
    qc.x(qr[t1])
```

```
| qc.x(qr[x])
| qc.x(qr[w])
return t2
```

# **Testing gates and their reversibility**

Now that we have specified all the necessary gates in Qiskit, including their reverse gates, it is important to test our specifications. To do that, this section provides some example code to run the gates over all possible inputs to see the result in the quantum simulator on your local computer. Using this code, we can produce the truth tables for these gates. With this truth table, we can verify that our reverse gates function correctly.

# General framework

The general idea with the test code is to run on the local simulator:

```
import itertools
def run_local_sim_one_result(qc):
    backend = Aer.get_backend('qasm_simulator')
    job_exp = qiskit.execute(qc,backend=backend)
    result = job_exp.result()
    final=result.get_counts(qc)
    result_in_order=list(final.keys())[0][::-1]
    return result_in_order
```

For the two-qubit logic functions, a general structure can be used, namely to compute the function over two qubits and one auxiliary qubit, measure the result, and then compute the reverse over the same qubit set:

```
def test_logic_function_2(f,frev):
    print("inputs","forward","reverse")
    print("abc","a'b'c'", "a''b''c'''")
    for combo in itertools.product([0,1],repeat=3):
        # forward
        qr = QuantumRegister(3)
        cr = ClassicalRegister(3)
        qc = QuantumCircuit(qr,cr)
        setup_input(qr,qc,combo[0],combo[1],combo[2])
        f(qr,qc,0,1,2)
        for i in range(3):
            qc.measure(qr[i],cr[i])
        forward_result=run_local_sim_one_result(qc)
        # forward then reverse
        qr = QuantumRegister(3)
        cr = ClassicalRegister(3)
        qc = QuantumCircuit(qr,cr)
        setup_input(qr,qc,combo[0],combo[1],combo[2])
        f(qr,qc,0,1,2)
        frev(qr,qc,0,1,2)
        for i in range(3):
            qc.measure(qr[i],cr[i])
        reverse_result=run_local_sim_one_result(qc)
        print('%d%d%d %s %s'.
              (combo[0],combo[1],combo[2],forward_result,reverse_result))
```

The setup for the three-qubit logic function test is similar:

```

def test_logic_function_3(f,frev):
    print("inputs","forward","reverse")
    print("abcd","a'b'c'd'","a''b''c''d''")
    for combo in itertools.product([0,1],repeat=4):
        # forward
        qr = QuantumRegister(5)
        cr = ClassicalRegister(5)
        qc = QuantumCircuit(qr,cr)
        setup_input(qr,qc,combo[0],combo[1],combo[2],combo[3])
        f(qr,qc,0,1,2,3,4)
        for i in range(5):
            qc.measure(qr[i],cr[i])
        forward_result=run_local_sim_one_result(qc)
        # forward then reverse
        qr = QuantumRegister(5)
        cr = ClassicalRegister(5)
        qc = QuantumCircuit(qr,cr)
        setup_input(qr,qc,combo[0],combo[1],combo[2],combo[3])
        f(qr,qc,0,1,2,3,4)
        frev(qr,qc,0,1,2,3,4)
        for i in range(5):
            qc.measure(qr[i],cr[i])
        reverse_result=run_local_sim_one_result(qc)

        forward_result=forward_result[0:3]+forward_result[4]
        reverse_result=reverse_result[0:3]+reverse_result[4]

        print('%d%d%d%d %s %s'%
              (combo[0],combo[1],combo[2],combo[3],forward_result,reverse_result))

```

In both the two-qubit and the three-qubit tests, we code the result to print the input and the output, as well as its reverse, in a simple string in a chart for easy reading.

We can then use this code to test each of our gates, four tests that produce four printouts:

```

print("Testing two qubit quantum AND")
test_logic_function_2(quantumand_2,quantumand_2)
print()

print("Testing two qubit quantum OR")
test_logic_function_2(quantumor_2,quantumor_2_reverse)
print()

print("Testing three qubit quantum AND")
test_logic_function_3(quantumand_3,quantumand_3)
print()

print("Testing three qubit quantum OR")
test_logic_function_3(quantumor_3,quantumor_3_reverse)

```

The following is not a code snippet; it is just a printout of the results of the previous code snippet:

```
Testing two qubit quantum AND
inputs forward reverse
abc a'b'c' a''b'''c''
000 000 000
001 001 001
010 010 010
011 011 011
100 100 100
101 101 101
110 111 110
111 110 111
```

Scenario showing two-qubit quantum OR:

```
Testing two qubit quantum OR
inputs forward reverse
abc a'b'c' a''b'''c''
000 000 000
001 001 001
010 011 010
011 010 011
100 101 100
101 100 101
110 111 110
111 110 111
```

Scenario showing three-qubit quantum AND:

```
Testing three qubit quantum AND
inputs forward reverse
abcd a'b'c'd' a''b'''c'''d'''
0000 0000 0000
0001 0001 0001
0010 0010 0010
0011 0011 0011
0100 0100 0100
0101 0101 0101
0110 0110 0110
0111 0111 0111
1000 1000 1000
1001 1001 1001
1010 1010 1010
1011 1011 1011
1100 1100 1100
1101 1101 1101
1110 1111 1111
1111 1110 1110
```

Scenario showing three-qubit quantum OR:

```

Testing three qubit quantum OR
inputs forward reverse
abcd a'b'c'd' a''b''c''d''
0000 0000 0000
0001 0001 0001
0010 0011 0010
0011 0010 0011
0100 0101 0100
0101 0100 0101
0110 0111 0110
0111 0110 0111
1000 1001 1000
1001 1000 1001
1010 1011 1010
1011 1010 1011
1100 1101 1100
1101 1100 1101
1110 1111 1110
1111 1110 1111

```

Notice that, in each scenario, if the classic equivalent of the AND or the OR of the input bits is true, then the auxiliary bit is flipped, and otherwise it is left the same. In addition, the third column of each truth table, which corresponds to running the function then reversing it, is identical to the first column corresponding to the input, as expected. Our functions work, and can be used as building blocks for the next steps.

# Solving a 3SAT problem with Grover's algorithm

As we learned in the section of this chapter specifying Grover's algorithm, it consists of some setup, then two repeating parts. One repeating part is a section that computes the `checker` (also known as the oracle) function, that is, the function for which we are trying to determine the input that makes it evaluate to true; the other is the "diffusion" step, which causes Grover's algorithm to converge toward the solution of the input for which the oracle function evaluates to true. In this section, we will implement the setup, the oracle, and the diffusion steps in Qiskit, and formulate Grover's algorithm as a coherent whole. We will get a chance to see Grover's algorithm find the input `a, b, c` to our `3sat_mystery(a, b, c)` function that causes it to evaluate to true, in just two evaluations of the `3sat_mystery(a, b, c)` function's quantum equivalent.

# Oracle implementation in Qiskit

Let's review `3sat_mystery(a,b,c)`. Since the solution space is still small enough (only eight possible solutions), it is straightforward to implement the function classically and check which input satisfies the function by brute force. In this section, we will show that we can do this in fewer function evaluations on a quantum computer, but, by first going through the exercise classically, it will help us understand the difference in the approach on a quantum computer.

# Test classic logic

Recall the following:

$$\begin{aligned} \text{3sat\_mystery}(a, b, c) = \\ (a \vee b \vee \neg c) \wedge (a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c) \end{aligned}$$

We implement `3sat_mystery(a,b,c)` classically as follows:

```
def _3sat_mystery3_classic(a,b,c):
    return int((a or b or not c) and (a or b or c) and (a or not b or c)
and (a or not b or not c) and (not a or b or not c) and (not a or b or c)
and (not a or not b or not c) )
```

Then, we can check all possible combinations of inputs and see which one returns true:

```
import itertools
for combo in itertools.product([0,1], repeat=3):
    print(combo,'->',_3sat_mystery3_classic(combo[0],combo[1], combo[2]))
```

Here, this prints the following:

```
(0, 0, 0) -> 0
(0, 0, 1) -> 0
(0, 1, 0) -> 0
(0, 1, 1) -> 0
(1, 0, 0) -> 0
(1, 0, 1) -> 0
(1, 1, 0) -> 1
(1, 1, 1) -> 0
```

We can see that the function returns true for just one input,  
<sub>110</sub>. We expect to get the same result on the quantum computer, only with fewer function evaluations.

# Quantum 3sat\_mystery implementation logic

Recall the definition of `3sat_mystery(a,b,c)`:

$$3\text{sat\_mystery}(a, b, c) =$$

$$(a \vee b \vee \neg c) \wedge (a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$$

To implement the `3sat_mystery(a,b,c)` function on a quantum computer, we will need to use the multi-qubit AND and OR that we developed earlier in this chapter. The general strategy will be to compute each clause of 3SAT, which is an OR over three variables, separately and to store the result in a separate quantum register for temporary use. For example, the first clause of `3sat_mystery(a,b,c)` is  $(a \vee b \vee \neg c)$ , so we can compute the quantum OR over the three input qubits, with the third qubit negated, and store this result in a temporary quantum register. If we do this for each of the seven clauses in `3sat_mystery(a,b,c)`, at the end, we can compute the quantum AND over all of the seven temporary registers (two at a time, using temporary registers) to get the final result.

# Setup or teardown logic function

Prior to the computation of each clause, we will need to set up the logic to compute that clause. For example, prior to computing  $(a \vee b \vee \neg c)$ , we will need to negate the qubit corresponding to  $c$ , the third qubit. To do that succinctly in our Qiskit code, we create a function

called `setup_or_teardown_logic`, which takes in a quantum register and quantum circuit object, as well as a truth value for each variable in the clause, set to `False` if the variable is negated in the clause, and set to `True` if it is left alone. The function is defined as follows:

```
def setup_or_teardown_logic(qr,qc,is_a,is_b,is_c):
    """
    is_a,is_b, and is_c: False indicates the variable should be negated,
    True left as is.
    Negation is done with the X gate.
    """
    if not is_a:
        qc.x(qr[0])
    if not is_b:
        qc.x(qr[1])
    if not is_c:
        qc.x(qr[2])
```

For the  $(a \vee b \vee \neg c)$  computation, the setup would then be `setup_or_teardown_logic(qr,qc,True,True,False)`. After the setup, the three-qubit quantum OR is computed with the `quantumor_3` function and the final result is stored in the temporary register of our choice. This will be repeated for each clause, so prior to setting up the next clause, we will need to tear down the logic for this clause.

Since the X gate reverses itself, this can be done by calling the `setup_or_teardown_logic` function with exactly the same

arguments as in setup; thus the name of the function, which implies that it can be used to either set up or to tear down the logic of a clause. In the case of  $(a \vee b \vee \neg c)$ , that means we will call `setup_or_teardown_logic(qr,qc,True,True,False)` before and after the call to the `quantumor_3` function.

# Compute clauses one by one

Now that we understand how to compute a single clause, let's go over computing the logical result of each clause in turn. We will have to carefully manage our quantum registers. The input qubits will reside in the first three registers, denoted by the indices  $0, 1, 2$ . For each `quantumor_3` function, we will need to specify these three input registers and additionally specify two temporary registers, one of which will store the final result of the OR. For convenience, the `quantumor_3` function, as we specified earlier in the chapter, returns the index of the register that contains the final result. Thus, we can easily keep track of where that is when the time comes to do the quantum AND between these registers. We will use unique temporary registers for each call to `quantumor_3` for clarity. The full code for computing the seven clauses is as follows:

```
# (a or b or not c)
setup_or_teardown_logic(qr,qc,True,True,False)
first_clause=quantumor_3(qr,qc,0,1,2,3,4)
setup_or_teardown_logic(qr,qc,True,True,False)

# (a or b or c)
setup_or_teardown_logic(qr,qc,True,True,True)
second_clause=quantumor_3(qr,qc,0,1,2,5,6)
setup_or_teardown_logic(qr,qc,True,True,True)

# (a or not b or c)
setup_or_teardown_logic(qr,qc,True,False,True)
third_clause=quantumor_3(qr,qc,0,1,2,7,8)
setup_or_teardown_logic(qr,qc,True,False,True)

# (a or not b or not c)
setup_or_teardown_logic(qr,qc,True,False,False)
fourth_clause=quantumor_3(qr,qc,0,1,2,9,10)
setup_or_teardown_logic(qr,qc,True,False,False)

# (not a or b or not c)
setup_or_teardown_logic(qr,qc,False,True,False)
```

```
fifth_clause=quantumor_3(qr,qc,0,1,2,11,12)
setup_or_teardown_logic(qr,qc,False,True,False)

# (not a or b or c)
setup_or_teardown_logic(qr,qc,False,True,True)
sixth_clause=quantumor_3(qr,qc,0,1,2,13,14)
setup_or_teardown_logic(qr,qc,False,True,True)

# (not a or not b or not c)
setup_or_teardown_logic(qr,qc,False,False,False)
seventh_clause=quantumor_3(qr,qc,0,1,2,15,16)
setup_or_teardown_logic(qr,qc,False,False,False)
```

The results of each clause are stored in the quantum registers corresponding to the indices `first_clause`, `second_clause`, `third_clause`, `fourth_clause`, `fifth_clause`, `sixth_clause`, and `seventh_clause`. Examining the code, we can see these are the registers at indices  $4, 6, 8, 10, 12, 14$ , and  $16$ .

# Combine clauses

Now that we have computed the final result of each clause, to compute the final result for the function as a whole, we will need to combine each of these temporary results with a multi-qubit quantum AND. We will do so two results at a time, using the `quantumand_2` function, until we get to the final result. This takes advantage of the fact that  $t, u, v, w, x, y$ , and  $z$  are all the results of the seven individual clauses  $t \wedge u \wedge v \wedge w \wedge x \wedge y \wedge z = (((((t \wedge u) \wedge v) \wedge w) \wedge x) \wedge y) \wedge z$ , and we can in general regroup an AND function over many variables into sub-functions over two variables at a time with the help of temporary variables to store the result. In fact, we can do the grouping however we like, and use the `quantumand_2` function, so long as it is two at a time. Here, I chose to group the results two at a time,  $t \wedge u \wedge v \wedge w \wedge x \wedge y \wedge z = ((t \wedge u) \wedge (v \wedge w)) \wedge ((x \wedge y) \wedge z)$ , for clarity of the code (each of which corresponds to our quantum circuit requiring an additional qubit).

Each time we call `quantumand_2`, we will need to specify the indices of the quantum registers containing the input qubits over which to compute the quantum AND, as well as the index of a temporary quantum register in which to store the result. Thus, our computation will require six more temporary registers, the sixth of which is actually not temporary (index 22) as it will (finally!) contain the final result of the computation. For convenience, the `quantumand_2` function returns the index of the temporary register in which the result is stored.

Thus, in our choice of variables corresponding to the results of each of the seven clauses, the quantum register at index `intermediate_and_pair1` (index 17) contains the result of  $(t \wedge u)$ ,

the quantum register at index `intermediate_and_pair2` (index 18) contains the result of  $(v \wedge w)$ , and the quantum register at index `intermediate_and_pair3` (index 19) contains the result of  $(x \wedge y)$ . Then, the quantum register at index `intermediate_and_pair_12` (index 20) contains the result of  $((t \wedge u) \wedge (v \wedge w))$ , the quantum register at index `intermediate_and_pair_34` (index 21) contains the result of  $((x \wedge y) \wedge z)$ , and the quantum register at index `final_result_and_pair_1234` (index 22) contains the final result:

$$((t \wedge u) \wedge (v \wedge w)) \wedge ((x \wedge y) \wedge z) =$$

$$(a \vee b \vee \neg c) \wedge (a \vee b \vee c) \wedge (a \vee \neg b \vee c) \wedge (a \vee \neg b \vee \neg c) \wedge (\neg a \vee b \vee \neg c) \wedge (\neg a \vee b \vee c) \wedge (\neg a \vee \neg b \vee \neg c)$$

This corresponds to the result of the entire `3sat_mystery(a,b,c)` function evaluated on a given input.

The code to do the quantum AND combination of the individual clauses is then as follows:

```
# Let's whittle down
intermediate_and_pair1=quantumand_2(qr,qc,first_clause,second_clause,17)
intermediate_and_pair2=quantumand_2(qr,qc,third_clause,fourth_clause,18)
intermediate_and_pair3=quantumand_2(qr,qc,fifth_clause,sixth_clause,19)

# Now whittling down further
intermediate_and_pair_12=quantumand_2(qr,qc,intermediate_and_pair1,intermediate_and_pair2,20)
intermediate_and_pair_34=quantumand_2(qr,qc,intermediate_and_pair3,seventh_clause,21)

# Now whittling down to 1 result
final_result_and_pair_1234=quantumand_2(qr,qc,intermediate_and_pair_12,intermediate_and_pair_34,22)
```

# Reverse logic so that we are back to the original

In Grover's algorithm, we will need to evaluate our 3SAT function in general something like  $\sqrt{N}$  times, where  $N$  is the number of possible input combinations, with steps in between. The steps in between may want to take advantage of the temporary qubits, and when we rerun the function, we want all the temporary qubits to be reset to zero. Luckily, since all quantum computation is reversible, we can simply run the code in reverse using the identical arguments to the appropriate function, which reverses the original AND or OR. This is `quantumand_2` in place of `quantumand_2` (as it is its own reverse) and `quantumor_3_reverse` in place of `quantumor_3`. With this, the full reverse of the logic so far would be the following:

```
final_result_and_pair_1234=quantumand_2(qr,qc,intermediate_and_pair_12,in  
termediate_and_pair_34,22)  
intermediate_and_pair_34=quantumand_2(qr,qc,intermediate_and_pair3,sevent  
h_clause,21)  
intermediate_and_pair_12=quantumand_2(qr,qc,intermediate_and_pair1,interm  
ediate_and_pair2,20)  
intermediate_and_pair3=quantumand_2(qr,qc,fifth_clause,sixth_clause,19)  
intermediate_and_pair2=quantumand_2(qr,qc,third_clause,fourth_clause,18)  
intermediate_and_pair1=quantumand_2(qr,qc,first_clause,second_clause,17)  
# (not a or not b or not c)  
setup_or_teardown_logic(qr,qc,False,False,False)  
seventh_clause=quantumor_3_reverse(qr,qc,0,1,2,15,16)  
setup_or_teardown_logic(qr,qc,False,False,False)  
# (not a or b or c)  
setup_or_teardown_logic(qr,qc,False,True,True)  
sixth_clause=quantumor_3_reverse(qr,qc,0,1,2,13,14)  
setup_or_teardown_logic(qr,qc,False,True,True)  
# (not a or b or not c)  
setup_or_teardown_logic(qr,qc,False,True,False)  
fifth_clause=quantumor_3_reverse(qr,qc,0,1,2,11,12)  
setup_or_teardown_logic(qr,qc,False,True,False)  
# (a or not b or not c)  
setup_or_teardown_logic(qr,qc,True,False,False)  
fourth_clause=quantumor_3_reverse(qr,qc,0,1,2,9,10)
```

```

| setup_or_teardown_logic(qr,qc,True,False,False)
| # (a or not b or c)
| setup_or_teardown_logic(qr,qc,True,False,True)
| third_clause=quantumor_3_reverse(qr,qc,0,1,2,7,8)
| setup_or_teardown_logic(qr,qc,True,False,True)
| # (a or b or c)
| setup_or_teardown_logic(qr,qc,True,True,True)
| second_clause=quantumor_3_reverse(qr,qc,0,1,2,5,6)
| setup_or_teardown_logic(qr,qc,True,True,True)
| # (a or b or not c)
| setup_or_teardown_logic(qr,qc,True,True,False)
| first_clause=quantumor_3_reverse(qr,qc,0,1,2,3,4)
| setup_or_teardown_logic(qr,qc,True,True,False)

```

After running the code to compute `3sat_mystery(a,b,c)`, we will have the first three quantum registers contain the value of the input (any one of eight combinations: 000, 001, 010, 011, 100, 101, 110, or 111), and the fourth through the twenty-second registers contain the result of temporary calculations, with the twenty-third register (at index 22) containing the final result of the computation of the function on the chosen three inputs (which can be measured to be either 0 or 1 depending on the input). After the reverse code is run, the first three quantum registers will still contain the value of the input, but the fourth through the twenty-third registers will all measure to bit 0.

For Grover's algorithm, we know exactly one of the eight possible inputs will result in a register 22 measuring bit 1, and the algorithm relies on that register remaining 1 for that input. Thus, for Grover's algorithm, we do not fully reverse the computation and leave out the reverse, which puts the final result in the quantum register at index 22, which corresponds to leaving out the following in the code:

```

| final_result_and_pair_1234=quantumand_2(qr,qc,intermediate_and_pair_12,in
| termediate_and_pair_34,22)

```

Since we'll want to test the function with and without reversibility, and with and without this final reverse, I will leave these as an option to the specification of the `3sat_mystery(a,b,c)` function in code. The Jupyter Notebook

associated with this chapter has the entire `_3sat_mystery_3` function as a convenient code block.

# Testing the \_3sat\_mystery\_3 function

Let's test our `_3sat_mystery_3` function in the IBM QX simulator by manually setting the input bits and running the function. We can try all eight possible inputs by using the built-in `itertools` module and the code `itertools.product([0,1], repeat=3)`. Here is some code to test our function for a variety of options:

```
import time
from qiskit.tools.visualization import plot_histogram
def
try_input_combination(input_combination,shots=1,reverse=False,full_revers
e=True):
    backend = IBMQ.get_backend('ibmq_qasm_simulator') # remote simulator
    qr = QuantumRegister(23)
    cr = ClassicalRegister(23)
    qc = QuantumCircuit(qr,cr)
    # setting up the input
    for i in range(3):
        if input_combination[i]:
            qc.x(qr[i])
    # calling the function on that input
    _3sat_mystery_3(qr, qc, reverse=reverse, full_reverse=full_reverse)
    # measuring every qubit as we will want to verify reversibility
    for i in range(23):
        qc.measure(qr[i],cr[i])

    # Executing the job on IBM QX
    job_exp = qiskit.execute(qc, backend=backend,shots=shots)
    result = job_exp.result()
    final=result.get_counts(qc)
    if not len(final)==1:
        print(input_combination,final)
    else:
        # note that due to IBM's choice the result returned is in
        opposite order with last register coming first
        # and the first register coming last. For clarity we reverse the
        output so the first register is first
        # and the last register is last.

        result_in_order=list(final.keys())[0][::-1]
        print(input_combination,'->',result_in_order[-1],'(measured bits:
'+result_in_order+')')
```

# Testing the \_3sat\_mystery\_3 function without reverse

If all goes well, our function without the reverse option should return 1 100% of the time in the classic register at index  $_{22}$  just for the input  $_{110}$ , and 0 100% of the time in the classic register and index  $_{22}$ , with the 4<sup>th</sup> register through the 22<sup>nd</sup> register filled with temporary results. The code to test is then the following:

```
| import itertools
| for combo in itertools.product([0,1], repeat=3):
|     try_input_combination(combo)
```

This outputs the following:

```
(0, 0, 0) -> 0 (measured bits: 0000100111111111011010)
(0, 0, 1) -> 0 (measured bits: 0010001111111111011010)
(0, 1, 0) -> 0 (measured bits: 010111000111111101010)
(0, 1, 1) -> 0 (measured bits: 0111111010011111101010)
(1, 0, 0) -> 0 (measured bits: 1001111111010011110100)
(1, 0, 1) -> 0 (measured bits: 1011111111000111110100)
(1, 1, 0) -> 1 (measured bits: 1101111111111101111111)
(1, 1, 1) -> 0 (measured bits: 1111111111111100111100)
```

We can see, as we expect, that input  $110$  is the only input that results in a  $1$  in the final register, and the fourth through the twenty second registers all contain intermediate results.

# Testing the `_3sat_mystery_3` function's reversibility

If all goes well with reversibility, running the function and then reversing it should result in the input remaining in the first, second, and third registers, with the rest of the registers measuring  $\oplus$  100% of the time. Here is the code to execute that test:

```
| import itertools
| for combo in itertools.product([0,1], repeat=3):
|     try_input_combination(combo, reverse=True, full_reverse=True)
```

This outputs the following:

```
| (0, 0, 0) -> 0 (measured bits: 000000000000000000000000)
| (0, 0, 1) -> 0 (measured bits: 001000000000000000000000)
| (0, 1, 0) -> 0 (measured bits: 010000000000000000000000)
| (0, 1, 1) -> 0 (measured bits: 011000000000000000000000)
| (1, 0, 0) -> 0 (measured bits: 100000000000000000000000)
| (1, 0, 1) -> 0 (measured bits: 101000000000000000000000)
| (1, 1, 0) -> 0 (measured bits: 110000000000000000000000)
| (1, 1, 1) -> 0 (measured bits: 111000000000000000000000)
```

We can see that, indeed, the first three registers contain the input, and the rest contain  $\oplus$ . The reverse works!

# **Testing \_3sat\_mystery\_3 function reversibility (except for final result)**

Finally, let's test the reverse of all but the output register. We expect the first three registers to contain the input, the final register to contain the output (which should be `1` only in the case of input `110`, and `0` otherwise), and all registers in between to contain `0`. Here is the code to execute that test:

```
| import itertools
| for combo in itertools.product([0,1], repeat=3):
|     try_input_combination(combo, reverse=True, full_reverse=False)
```

This outputs the following:

```
| (0, 0, 0) -> 0 (measured bits: 000000000000000000000000)
| (0, 0, 1) -> 0 (measured bits: 001000000000000000000000)
| (0, 1, 0) -> 0 (measured bits: 010000000000000000000000)
| (0, 1, 1) -> 0 (measured bits: 011000000000000000000000)
| (1, 0, 0) -> 0 (measured bits: 100000000000000000000000)
| (1, 0, 1) -> 0 (measured bits: 101000000000000000000000)
| (1, 1, 0) -> 1 (measured bits: 110000000000000000000001)
| (1, 1, 1) -> 0 (measured bits: 111000000000000000000000)
```

We can see that the first three registers do, indeed, contain the input, and the last the output, which is `0` except for input `110`, where it is `1`, and all other registers contain `0` consistently. The reverse of all but the final register works perfectly!

# Mover step implementation

The mover step's goal is to move the chance of observing the correct input up while moving the chance of observing the incorrect input down. Let's see how it does so by looking at an example:

$$|\text{"all three qubit inputs"}\rangle =$$

$$\frac{1}{\sqrt{8}}(|000\rangle + |001\rangle + |010\rangle + |011\rangle + |100\rangle + |101\rangle + |110\rangle + |111\rangle)$$

If the correct input is  $|110\rangle$  and the others are wrong, we can have the `checker` function mark that input. The `checker` function can't change the chance of observing that input, which means that the `checker` function can't simply set the incorrect inputs to 0. Remember, that the chance of observing a particular input is the square of the number in front of it. So in this case, that is the  $(1/\sqrt{8})^2 = 1/8$  for each input. How could we mark the correct input with the `checker` without changing the probability? We could simply mark it with a negative number. Because the square of any negative number is positive, this won't change the chance of observing that input, but it will provide information the mover function can use to do so; since  $(1/\sqrt{8})^2 = (-1/\sqrt{8})^2$ , we can mark the correct input with a negative without changing the final result.

Thus, the checker when run on /"all three qubit inputs"> would give the following:

$$\text{checker} |“\text{all three qubit inputs }”\rangle =$$

$$\frac{1}{\sqrt{8}}(|“000 ”\rangle + |“001 ”\rangle + |“010 ”\rangle + |“011 ”\rangle + |“100 ”\rangle + |“101 ”\rangle + |“110 ”\rangle - |“111 ”\rangle)$$

Note the negative number before /"110">.

Now, we are ready for the first mover step. The mover will need to use this information from the checker to move all the chances of the other states down, while moving the chance for /"110"> up.

It will do this by taking the average of the numbers out in front of each individual state. Since most of them are  $1/\sqrt{8}$ , this will be very close to  $1/\sqrt{8}$ , but not exactly. The average is calculated as follows:

$$\frac{\left(\frac{1}{\sqrt{8}} + \frac{1}{\sqrt{8}} + \frac{1}{\sqrt{8}} + \frac{1}{\sqrt{8}} + \frac{1}{\sqrt{8}} + \frac{1}{\sqrt{8}} - \frac{1}{\sqrt{8}} + \frac{1}{\sqrt{8}}\right)}{8} = \frac{\frac{6}{\sqrt{8}}}{8}$$

The average is  $\approx 0.265$  compared to  $1/\sqrt{8}$ , which is  $\approx 0.35$ . It then takes twice this average, and flips each state's numbers over this average. Twice 0.265 is 0.53, so every state but the correct state changes its number to the following:

$$2 \cdot \frac{\frac{6}{\sqrt{8}}}{8} - \frac{1}{\sqrt{8}} \approx 0.53 - 0.35 = 0.18$$

Then, the correct state changes its number to the following:

$$2 \cdot \frac{\frac{6}{\sqrt{8}}}{8} - \frac{1}{\sqrt{8}} \approx 0.53 - 0.35 = 0.18$$

So, our new state after one iteration of the `checker` function, then the `mover` function, is the following:

$$\text{mover}(\text{checker}|“all three qubit inputs ”\rangle) = \\ (2 \cdot \frac{\frac{6}{\sqrt{8}}}{8} - \frac{1}{\sqrt{8}})|“000 ”\rangle + |“001 ”\rangle + |“010 ”\rangle + |“011 ”\rangle + |“100 ”\rangle + |“101 ”\rangle + |“111 ”\rangle) + (2 \cdot \frac{\frac{6}{\sqrt{8}}}{8} + \frac{1}{\sqrt{8}})|“110 ”\rangle$$

Alternatively, it is the following:

$$\text{mover}(\text{checker}|“all three qubit inputs ”\rangle) = \\ 0.18(|“000 ”\rangle + |“001 ”\rangle + |“010 ”\rangle + |“011 ”\rangle + |“100 ”\rangle + |“101 ”\rangle + |“111 ”\rangle) + 0.88|“110 ”\rangle$$

Now, we can see that, after just one iteration of the `checker` and the `mover`, we have a  $0.88^2 \approx 0.77$  or 77% chance of seeing the correct input  $|“110”\rangle$ , and a  $0.18^2 \approx 0.03$  or about 3% chance of seeing each of the other inputs. (Since there are seven other inputs, we have about a 21% chance of seeing one of the other outputs in total.) Since all these numbers are rounded, they don't add up quite to 100%, but if we do the calculation from the original probabilities, we will see that they do.

The `mover` function has done its job! We have successfully increased the chance of observing the input  $|“110”\rangle$ , while

decreasing the chance of seeing the others. Successive applications up to the ideal number of movers and checkers will increase the chance of observing /"110"/ further. If we apply more than the ideal number, we may find ourselves straying from the correct solution.

# Full implementation of the mover function

How the `mover` function is implemented is beyond the scope of the mathematics of this book, but the implementation is provided here (Note that this currently only works over two or three inputs, and it is future work to extend this to work over more qubits.):

```
def diffusion_step(qr,qc,num_inputs):
    if num_inputs not in [2,3]:
        raise Exception("currently only supports 2 or 3 inputs")
    for i in range(num_inputs):
        qc.h(qr[i])
    for i in range(num_inputs):
        qc.x(qr[i])

    control_Z(qr,qc,num_inputs)

    for i in range(num_inputs):
        qc.x(qr[i])
    for i in range(num_inputs):
        qc.h(qr[i])
```

The `mover` function relies on a controlled Z gate, which is implemented as follows:

```
def control_Z(qr,qc,num_inputs):
    if num_inputs not in [2,3]:
        raise Exception("currently only supports 2 or 3 inputs")
    if num_inputs==2:
        qc.h(qr[1])
        qc.cx(qr[0],qr[1])
        qc.h(qr[1])
    elif num_inputs==3:
        qc.h(qr[2])
        qc.ccx(qr[0],qr[1],qr[2])
        qc.h(qr[2])
```

# Full algorithm setup

Now that we have information about how to implement the `checker` and the `mover` function, here is a full implementation of Grover's algorithm:

```
def grovers_algorithm(checker,num_inputs,num_registers,num_iterations=None):
    if num_iterations == None:
        from math import floor,sqrt
        iterations=floor(sqrt(2**num_inputs))
    else:
        iterations=num_iterations
    print("Running Grover's algorithm for %d iterations"%iterations)
    qr = QuantumRegister(num_registers)
    cr = ClassicalRegister(num_registers)
    qc = QuantumCircuit(qr,cr)
    # Configuring the input
    for i in range(num_inputs):
        qc.h(qr[i])
    # Setting up the output of the checker function
    qc.x(qr[num_registers-1])
    qc.h(qr[num_registers-1])

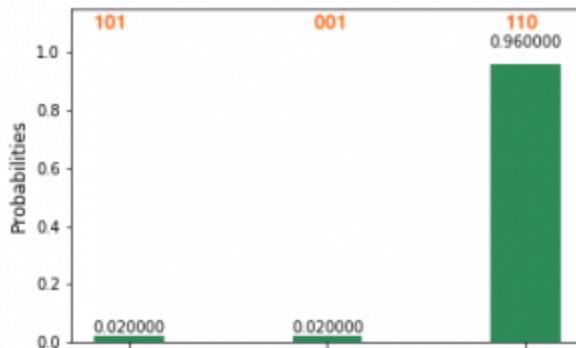
    # Do the Grovers steps
    for it in range(iterations):
        checker(qr,qc)
        mover(qr,qc,num_inputs)
    # Measure the inputs
    for j in range(num_inputs):
        qc.measure(qr[j], cr[j])
    return cr,qr,qc
```

# Running the algorithm on Qiskit

Let's run the algorithm on Qiskit using our `_3sat_mystery_3` as the checker function and see what we get out!

```
from qiskit.tools.visualization import plot_histogram
backend=IBMQ.get_backend('ibmq_qasm_simulator')
shots=50
cr,qr,qc = grovers_algorithm(
    _3sat_mystery_3,3,23,num_iterations=num_iterations)
job_exp = qiskit.execute(qc, backend=backend, shots=shots)
result = job_exp.result()
final=result.get_counts(qc)
plot_histogram(final)
```

Here, we see we get the correct solution, `|"110">` with a 96% chance:



This algorithm tried 50 times to try to get a picture of what the probabilities were (`shots=50`). However, we likely didn't need to try 50 times: with a good chance we could have just tried once. A single try would result in calling the `checker` function just twice, which is far fewer times than the 8 times a classical algorithm would need to call the checker function, in the worst case scenario.

# Summary

Grover's algorithm works to efficiently find the one solution to a function out of all possible inputs. Grover's algorithm is implemented by creating a superposition over all possible inputs and then iterating a **checker** step, which tags the unique input that satisfies the function, with a **mover** step that changes the probabilities of observing each input to increase the chance of seeing the correct input and decrease the chance of seeing the other inputs. The checker step is called an oracle step and the mover step is called a diffusion step in most of the quantum computing literature.

Grover's algorithm typically takes  $\mathcal{O}(\sqrt{N})$  calls to the <sup>checker</sup> function, where  $N$  is the number of possible inputs to the checker to find the solution, whereas a classic algorithm would need  $N$  calls to <sup>checker</sup> function.

In the next section, we will move on to learning about Shor's algorithm, an algorithm for integer factorization that has important implications for current cryptographic implementations.

# Questions

1. Run Grover's algorithm using different iteration numbers to solve `3sat_mystery(a,b,c)`. What happens when we use only one iteration? Zero iterations? More than the recommended number?
2. Switch variables `b` and `c` in the `3sat_mystery(a,b,c)` function to create a new mystery function. What result do you expect? Verify that you get this result classically, and that Grover's algorithm works for this new function to reproduce the classic result.
3. Challenge problem: modify the implementation of `3sat_mystery(a,b,c)` to use at least one fewer temporary qubits. Verify your modification still produces the correct result. Continue modifying the code to use as few temporary qubits as possible while producing the correct result. What is the smallest number you can possibly use? The fewer qubits used, the smaller the number of qubits in the quantum computer necessary to solve the problem.
4. 2SAT Grover:
  1. Implement the following 2SAT function classically, where each clause contains the OR of two variables, and the clauses are combined with an AND:  
$$(a \vee b) \wedge (\neg a \vee \neg b) \wedge (a \vee \neg b)$$
. Verify your function returns true only for the case of  $a=\text{True}$ ,  $b=\text{False}$ . How many inputs did you need to check classically?
  2. Implement your function on a quantum computer using multi-qubit AND and multi-qubit OR.
  3. Modify the code in this chapter to execute Grover's algorithm to find the input for this 2SAT function that satisfies it. How many iterations did you need? What

happens if you increase or decrease the number of iterations?

4. Challenge problem: modify the code to run on IBM QX hardware. You will need to make sure to use no more than the number of qubits available on the current hardware you have access to, to make sure the program is of the appropriate length, and to take care that, on IBM hardware, it is in general possible to compute a CNOT gate between two qubits only for some pairs of qubits.
5. Apply the `mover` and `checker` functions once more to the output by manually calculating their results:

**`mover(checker|“all three qubit inputs ”) =`**

$$0.18(|“000 ”\rangle + |“001 ”\rangle + |“010 ”\rangle + |“011 ”\rangle + |“100 ”\rangle + |“101 ”\rangle + |“111 ”\rangle) + 0.88|“110 ”\rangle$$

What is the probability of observing the correct state  $|“110”\rangle$  after this second iteration?

# Quantum Fourier Transform

This section describes the **Quantum Fourier Transform (QFT)**, which is a sub-routine of many important quantum algorithms, including Shor's algorithm. It first gives an overview of the **Classical Fourier Transform (CFT)**, which decomposes a signal, any function of time, into the frequencies which make it up. This section will require a basic knowledge of algebra. The chapter uses analogies from music to understand what a Fourier transform does and provides applications where Fourier transforms are commonly used in computation. The chapter next shows the use a classical algorithm to compute the Discrete Fourier Transform, that is, a Fourier transform of a signal represented on a classical computer. Applications of the QFT are discussed, and a QFT is compared to a Classical Fourier Transform. Finally, a QFT algorithm is given in OpenQASM, Qiskit, and in a quantum score, and the reader is given the opportunity to run this quantum algorithm on IBM QX or within the Qiskit simulator.

The following topics are covered in this chapter:

- An overview of the Classical Fourier Transform
- A demonstration of the Fast Fourier Transform algorithm for computing the Classical Fourier Transform
- The Quantum Fourier Transform
- An implementation of the Quantum Fourier Transform

# Classical Fourier Transform

Oftentimes in life, we can describe something two different ways. For example, if you are giving directions to the local coffee shop to someone from out of town, you could say something like "*take the first left, go two blocks, and then take a right*", or you could give directions in terms of street names, or in terms of landmarks. Both sets of directions are equivalent, and will get you to the same place in the end. Depending on the type of person you are talking to, they might find one method easier to follow than another. Some people like landmark directions (turn right after the third palm tree by the old McDonald's); others are hopelessly confused by them. Other people love cardinal directions (go north two blocks, then turn west) others hate them.

Likewise, in mathematics, there are often several competing ways of describing something, some of which are easier to work with mathematically in certain scenarios than others. Given any mathematical function  $y$  of some variable  $x$ , if our function has a specific beginning point and a specific ending point (that is, the function is bounded), and has no breaks (that is, the function is continuous), then we can rewrite the function to be composed of the sum of sine functions: waves of different frequencies, amplitudes, and zero crossings. Sometimes, this is very convenient! For example, if our function represents a musical sound of piano keys played for a certain amount of time, rewriting it as the sum of waves of different frequencies, amplitudes, and zero crossings can help us identify the notes on the piano key that compose

the sound. What was a tough problem has become simple just by rewriting it mathematically.

As another analogy, you can think of the final function as the output of a recipe, say a cake, and the sinusoidal functions as the ingredients that went into making the recipe. If you want a recipe that serves twice as many people, the cake itself won't be of much help for figuring out how to bake that; you'd need the original recipe. Since either the whole cake, or the recipe with the instructions to bake a cake, will produce something the same, you could give either as a gift. One or the other will be more useful depending on the circumstance.

Imagine the function instead describes an image. Most images change very little on small scales (that is one pixel is likely to be the same as the pixel next to it), but have big changes on larger scales. If we wipe out the smaller scale changes entirely, we'd get a blurry image. If we are careful to wipe out only the smaller scale changes that are below the resolution of our computer display, then we could create a small image without any difference to the viewer. The process of representing the image instead of bit by bit, by the sum of waves of different frequencies, amplitudes, and zero crossings can help us do this. First, we compute the new representation of the image. Then we throw away the items of the representation that are of higher frequency. Then we compute the original visual representation of the image again and display it. Voila—we have an image compression algorithm.

The JPEG algorithm for compressing images uses Fourier techniques, for example. These techniques can be used in other types of data as well. The MP3 compression algorithm for compressing audio uses Fourier techniques as well.

The process of going from a function to the representation in terms of a sum of waves is called the Fourier transform; it goes from a function to the "ingredients" that compose the function. This is one of the most useful operations in mathematics, because the wave representation allows us to perform certain operations more easily than the original, and we can always go back and forth between the two. Most often, the Fourier transform is just a component of another algorithm or operation, which makes the end goal easier to achieve.

# Sine waves

We can represent a periodic (that is, repeating) oscillation that is smooth (that is, it has no breaks) by a sine function.

Any sine wave can be written as follows:

$$y(x) = A \sin(2\pi f x + \frac{\pi}{180} \phi)$$

$f$  is the number of oscillations per one  $x$  (for the rest of this chapter, let's consider time in units of seconds, so  $x = 1$  corresponds to one second),  $A$  is the amplitude or height of the wave, and  $\phi$  is the phase, indicating where in the cycle the wave is, in degrees ( $^\circ$ ). A full cycle is  $360^\circ$ .

We can plot this with the following code:

```
import numpy as np
import matplotlib.pyplot as plot
def plot_wave(A,f,phi,name=''):
    plot.clf()
    x = np.arange(0, 1, 0.01);
    y = A*np.sin(2*np.pi*f*x + np.pi/180*phi)
    plot.plot(x,y)
    plot.xlabel('x')
    plot.ylabel('y')
    plot.title(name)
    plot.grid(True, which='both')
    plot.axhline(y=0, color='k')
    plot.show()
```

We can plot a wave with an amplitude of 1, a frequency of 1, and a phase of 0, with the following line of code:

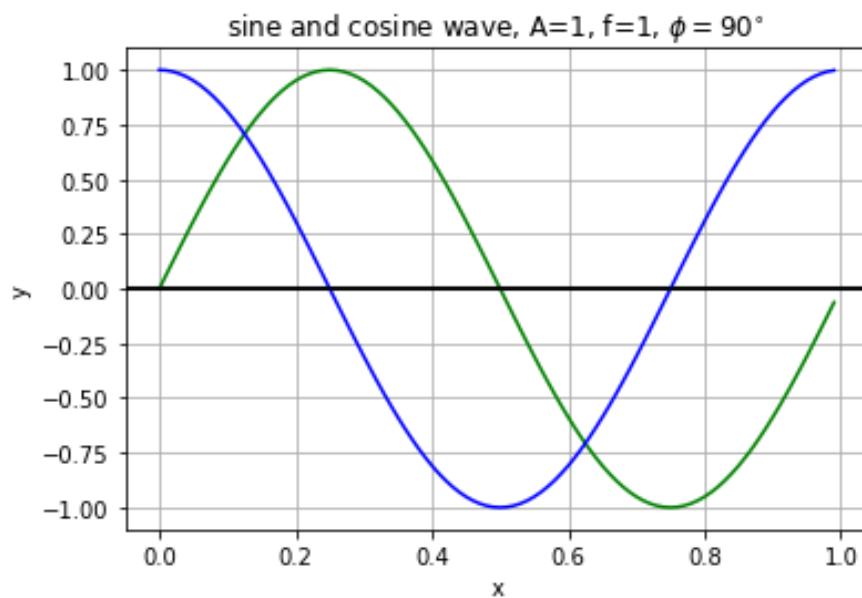
```
|plot_wave(1,1,0,'sine wave, A=1, f=1, omega=0')
```

We can create a similar wave that is a quarter of a cycle out of phase with our original sine wave, or  $\phi = 360^\circ / 4 = 90^\circ$ .

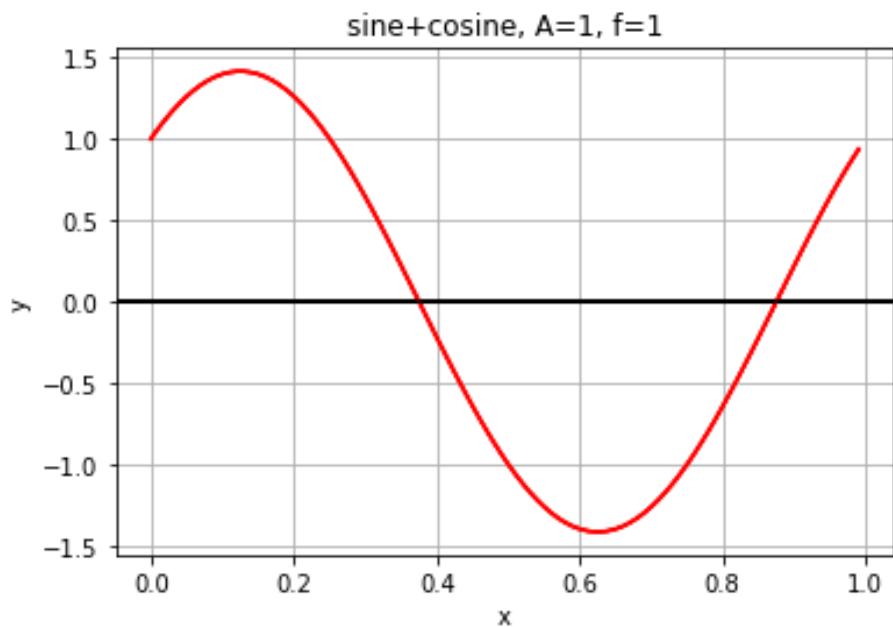
We can create the wave with the same code, so long as we input these parameters:

```
|plot_wave(1,1,90,'cosine wave, A=1, f=1, omega=0')
```

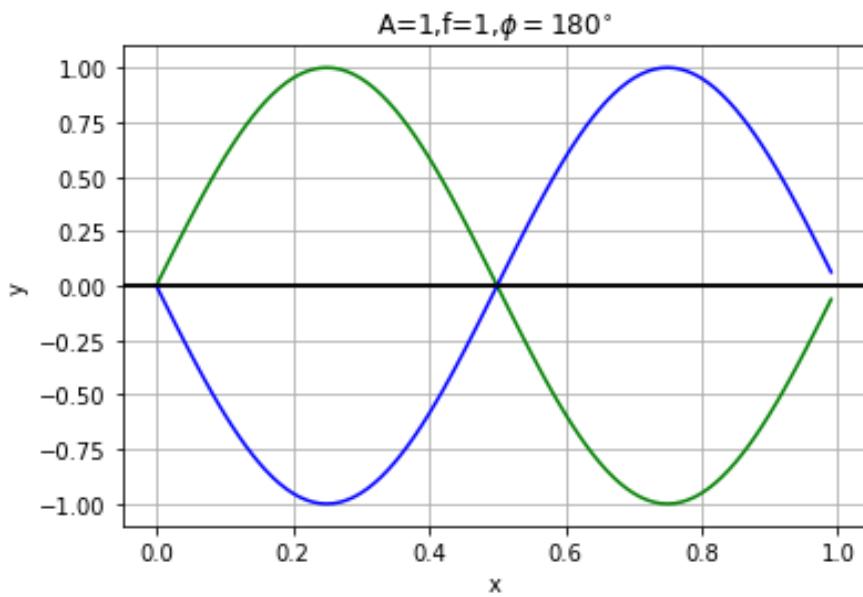
Alternatively, there is a shorthand for this type of wave, `cos`, called the cosine wave. Displayed next to each other, we have the sine wave in green and the cosine wave in blue:



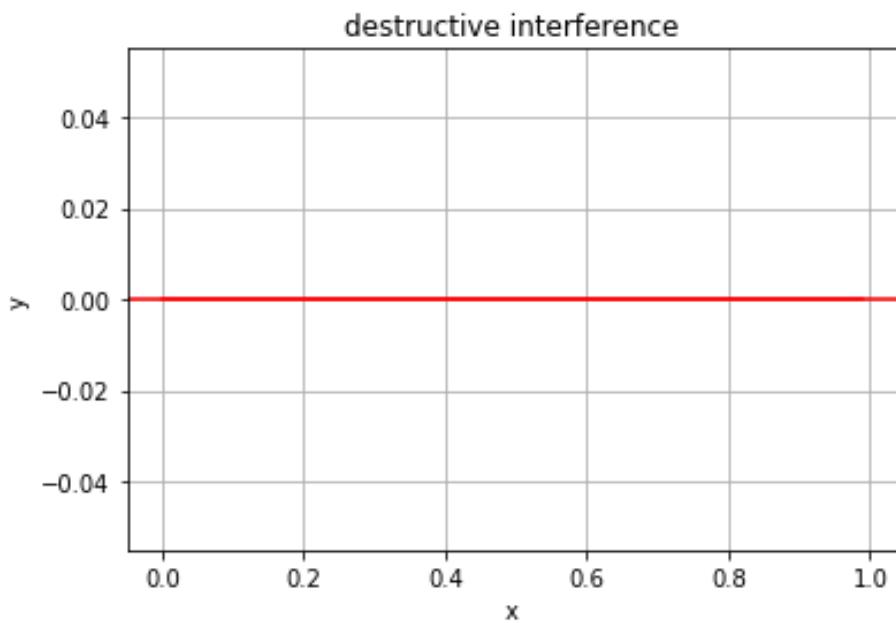
If we add these two waves, the sine and the cosine, together, we'd get the following:



Now, imagine that instead we have a wave with the same amplitude as the original sine wave, only  $180^\circ$  out of phase. Displayed next to each other, with the original sine in green and the  $180^\circ$  out-of-phase version in blue, we have the following:



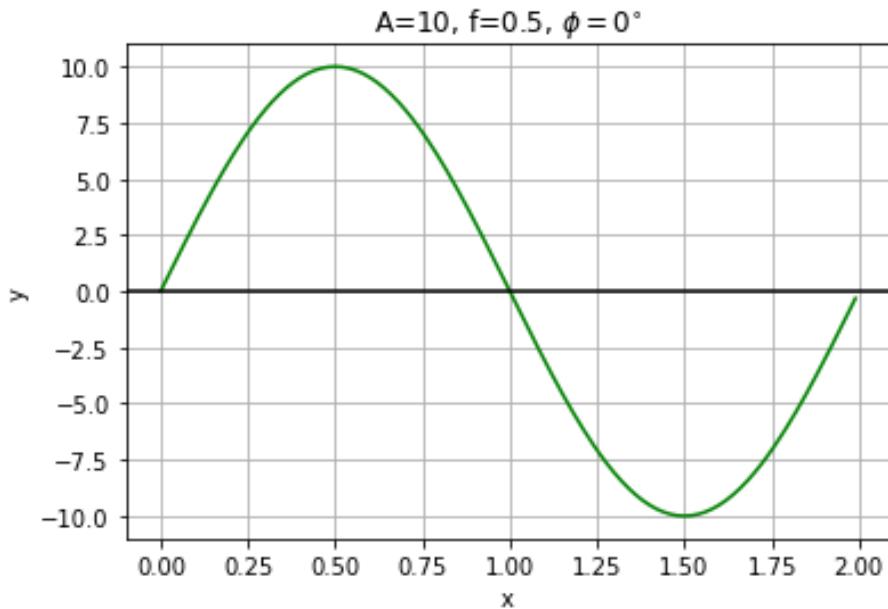
Every time the original wave has a positive value, this wave has the exact same value, only negative. So they completely cancel each other out. If we added the two, we would just get a line at zero, so we say that these two waves **destructively interfere**:



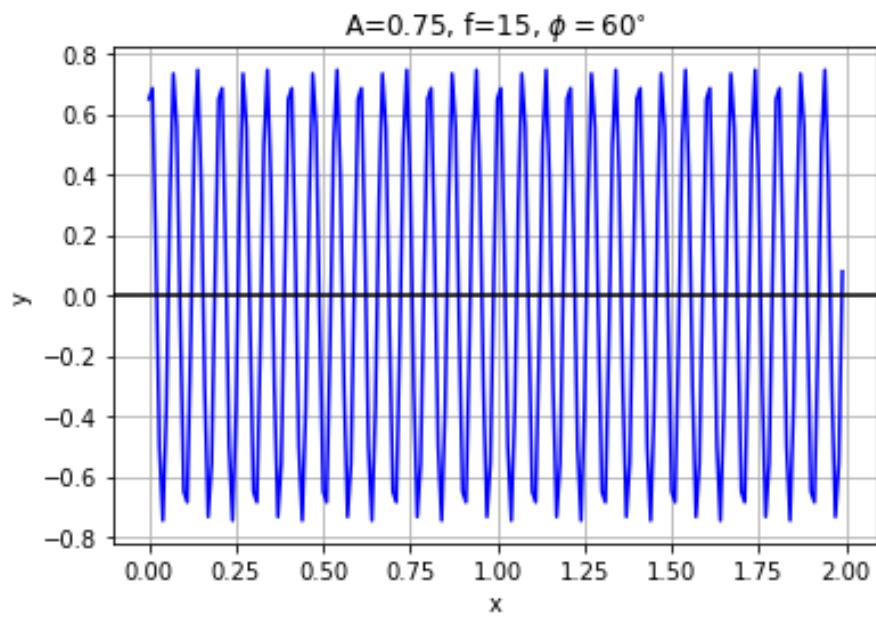
**i** Because computers have a finite accuracy, a wave and the same wave  $180^\circ$  out of phase will not add up to exactly zero, but close to it, to within numerical accuracy.

# The Fourier transform in action

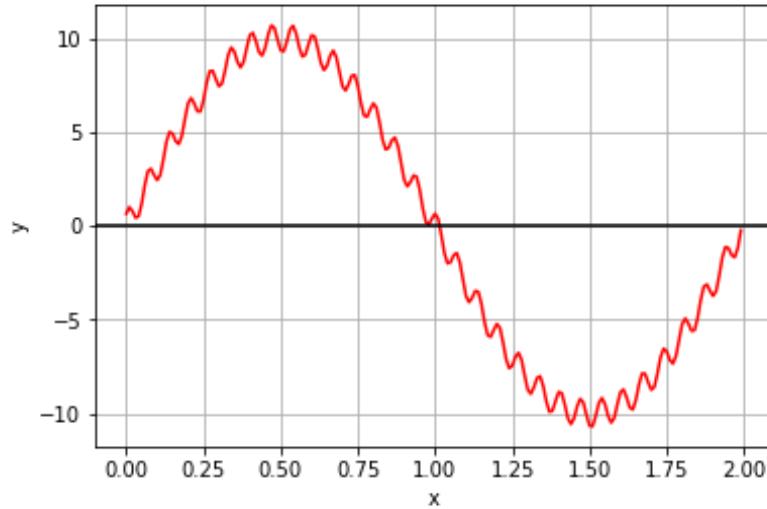
In the previous section, we saw several examples of sine waves that were added to each other. It turns out that any bounded continuous function can be rewritten as the sum of sine waves of different amplitudes, phases, and frequencies. In the previous subsection, each example was itself a sine wave, so that was easy to do. Let's consider a more difficult example, with waves of different amplitudes, phases, and frequencies. The first, in green, has an amplitude of **10**, a frequency of **0.5**, and a phase of **0°**:



The second, in blue, has an amplitude of **0.75**, a frequency of **15**, and a phase of **60°**:



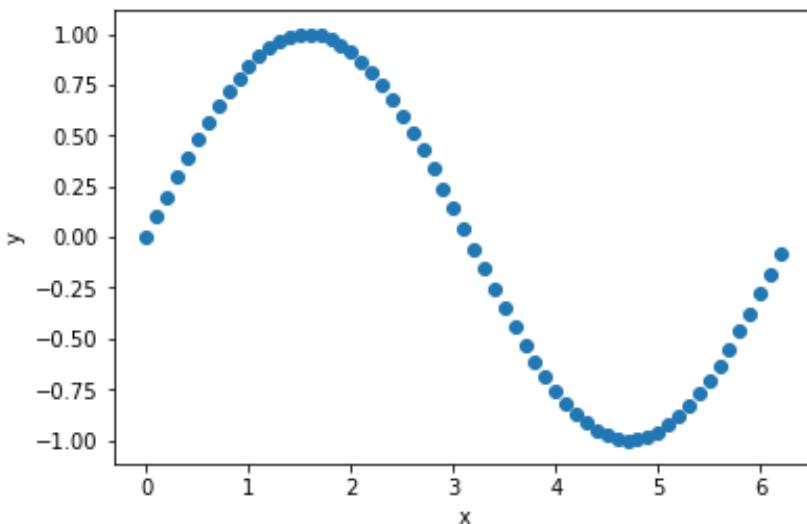
If we add these two together, we get the following graph in red:



The Fourier transform will take us between the red function, to give us the blue and green function parameters,  $A = 10$ ,  $f = 0.5$ , and  $\phi = 0^\circ$  and  $A = 0.75$ ,  $f = 15$ , and  $\phi = 60^\circ$ . If the red function is the final output, the blue and

green functions are the ingredients that comprise that output.

We can undo a Fourier transform with the **Inverse Fourier Transform (IFT)**. The IFT takes us from the mathematical values corresponding to the parameters, in this example,  $A = 10$ ,  $f = 0.5$ , and  $\phi = 0^\circ$ , and  $A = 0.75$ ,  $f = 15$ , and  $\phi = 60^\circ$ , back to the original red function above. Essentially, we break up a final function into its ingredients, and that is the power of the Fourier transform. There are two versions of the Fourier transform: continuous and discrete. Continuous means we have access to the original function in its entirety; this is only possible theoretically, as with any system, whether on a computer or physically, we can only sample portions of the function. Discrete means there are some gaps in our knowledge of the function; that is, that we sample the value of the function at certain portions. As one example, we may only know the value of the function every 0.1 seconds, instead of knowing it at all possible times, as depicted in the following graph:



This example has gaps every 0.1 seconds between the places where we know the function can be fixed or varied.

These fixed gaps are called the sampling interval. So this example has a sampling interval of 0.1 seconds.

We will focus on the **Discrete Fourier Transform (DFT)** here. How does the Fourier transform operate? One way would be to mathematically try out a variety of frequencies, and then determine the energy the function has in that frequency (that is, the amount the frequency contributes to the recipe for the final function, which is proportional to its amplitude) by taking an average of the final function rotated around at that frequency. After this procedure is repeated for enough frequencies, we can write down the final signal as a sum of sine waves at the frequencies and amplitudes we have determined. Since the transformation is discrete, we can choose how many frequencies we try out, and we want to ensure we try out enough frequencies to adequately capture the final function.

In practice, there are quicker algorithms to compute the Fourier transform, which involve additional mathematical tricks. The **Fast Fourier Transform (FFT)** is most commonly used, and runs in complexity  $O(n \log n)$ .



*There's an even faster algorithm called the **Sparse Fourier Transform (SFT)** which runs in  $O(k \log n)$ , where  $k$  is the number of frequencies that ultimately come into play in the solution, but the FFT is the most commonly used.*

Let's see the FFT algorithm in action for our signal in red. Since the transform is a discrete transform, we will need to define a sampling rate and a time interval over which to sample the function:

```
|sampling_rate = 100.  
|time_interval=2.
```

The number of samples is then equal to the sampling rate multiplied by the time interval:

```
|n_samples = sampling_rate*time_interval
```

Now let's define a general wave function that we will use to define our waves:

```
| def wave(A,f,phi):
|     sampling_interval = 1/sampling_rate
|     x = np.arange(0, time_interval, sampling_interval);
|     y = A*np.sin(2*np.pi*f*x + np.pi/180*phi)
|     return y
```

Then we can create our green wave with  $A = 10$ ,  $f = 0.5$ , and  $\phi = 0^\circ$ :

```
| green_wave=wave(10,0.5,0)
```

And our blue wave with  $A = 0.75$ ,  $f = 15$ , and  $\phi = 60^\circ$ :

```
| blue_wave=wave(0.75,15,90)
```

The red wave is just the sum of these two:

```
| red_wave=green_wave+blue_wave
```

Then, we will use the `fft` module under `numpy` to help us recover the parameters of the green wave and the blue wave (the ingredients of the red wave) from the red wave alone:

```
| from numpy import fft
| fft_wave = fft.fft(red_wave)/n_samples
```

The Fourier transform tries a variety of frequencies to see whether the wave has any amplitude at that frequency. Those frequencies tried are as follows:

```
| frequencies_tried=np.arange(n_samples)/time_interval
```

If the frequency of an ingredient to the wave isn't exactly one of the ones tried, we won't get exactly the right answer as the ingredient. This is why sampling the wave frequently enough is important if we are to recover its ingredient waves. For real data in this implementation, we only have to consider the first half of the frequencies, as the second half

are the complex conjugates of the first and represent redundant information. We can compute the amplitude with the following:

```
| amplitude=2*np.absolute(fft_wave)
```

And then, keeping in mind we only need to look at half the range, define the following: `half_range=range(int(n_samples/2))`. Finally, making sure to ignore very small coefficients, which will be just due to numerical noise, we can print the amplitudes as follows:

```
| for f,a in zip(frequencies_tried[half_range],amplitude[half_range]):  
|     if a>1e-6: # ignore those coefficients which are so small, so as to  
|     be numerical noise  
|         print("the amplitude at f=% .2f is A=% .2f"%(f,a))
```

This prints the following:

```
| the amplitude at f=0.50 is A=10.00  
| the amplitude at f=15.00 is A=0.75
```

Which is just our original blue and green wave parameters. From the red wave alone, the FFT algorithm has recovered the ingredient wave parameters.

# The Quantum Fourier Transform

The Quantum Fourier Transform performs a similar operation, except instead of performing a DFT on data, it performs a Discrete Fourier Transform on a quantum state itself. Moreover, it runs much faster:  $O((\log n)^2)$  time, compared to the **Classical Discrete Fourier Transform (CDFT)**, which runs in  $O(n \log n)$  or  $O(k \log n)$  time. Recall in [Chapter 1](#), *What is Quantum Computing?* when we introduced quantum states, where, to find the probability of a state in a superposition of other basis states as a particular output of one of those basis states, we would take the coefficient in front of that basis state and square it. This coefficient is called the *amplitude* of the state, and in general can be a complex number. It is these coefficients that the QFT operates on.

Uses of the QFT include as a subroutine to Shor's algorithm, used to factor integers. This is because, using Shor's algorithm to find the factorization, at a certain point the period of a function needs to be computed and the QFT can be used to do so. So, learning to implement the QFT will help us to implement Shor's algorithm. There are also a variety of other uses of the QFT as a subroutine to other quantum algorithms.

When I introduced the CDFT, I described the idea behind it but I did not provide the details of the algorithm used to implement it; in fact, many such algorithms exist to compute the Classical Discrete Fourier Transform, and understanding the mathematical details isn't important, as

libraries to perform the Classical Discrete Fourier Transform exist in all widely-used programming languages.

Quantum computing isn't quite at the level where all quantum computers provide a one-liner to call the QFT, but we still don't need to go into the details of the mathematics to use the QFT. In this section, I will provide the circuit that implements the QFT, the background necessary to implement this circuit, and its implementation in Qiskit 2.0. We will then see it working on an example quantum state.

# Implementing the Quantum Fourier Transform

The QFT is built out of Hadamard (H) gates and a new gate we will define,  $R_k$ , which is a type of controlled rotation gate. We will also need a gate that reverses the order of the qubits, which we will call REV.

# Implementing a controlled rotation gate, R<sub>k</sub>, in Python

Recall the Pauli gates: X, Y, and Z, which rotate a qubit about the x, y, or z axis respectively by 180°, and the phase gate, S, which rotates about the z axis by 90°. For the QFT, we will want a gate that rotates about the z axis by the number of degrees of our choice. To get some insight into how to do this, recall the definition of the S gate:

The S gate rotates around the z axis, about the x-y plane by 90° ( $\pi/2$  radians).

In Python, recall that this is the following:

```
| S=np.matrix([[1,0],[0,np.e**(i_*np.pi/2.)]])
```

We see that  $\pi/2$  in the preceding equation determines the number of radians by which S will perform its rotation. So, for an arbitrary number of radians, we would want to make that a function of some parameter, for example, let's define a function, called `simple_rotation`, which performs a rotation by a specified parameter, `lambda`:

```
| def simple_rotation(lambda):
|     np.matrix([[1,0],[0,np.e**(i_*lambda)]])
```

That will then rotate around the z axis by `lambda` (the symbol for `lambda` is  $\lambda$ ) degrees. With the QFT we will be performing a transform on n qubits, and each rotation will be controlled by another qubit. Recall the CNOT gate that performed an operation on a target qubit only if the control

qubit is `/"1">`. For the QFT we will not only want to rotation gate the be controlled by a control qubit, but at index  $k$ , we will also want to change the amount of the rotation based on the qubit that is the control qubit. The formula we will use to do so is as follows, in radians:

$$\lambda = \frac{2\pi}{2^k}$$

The preceding formula can be written in degrees as follows:

$$\lambda = \frac{2\pi}{2^k} \frac{180}{\pi} = \frac{360^\circ}{2^k}$$

To this end, we define a rotation gate, `Rk`, that is a function of  $k$ :

```
|def Rk(k):
    return np.matrix([[1,0][0,np.e**(2*np.pi*1j/2**k)]])
```

This rotates around the axis by a certain amount that changes depending on the value of  $k$ . For  $k = 0$  this rotation is  $0^\circ$  (0 radians); for  $k = 1$  this rotation is  $180^\circ$  ( $\pi$  radians); for  $k = 2$  this rotation is  $90^\circ$  ( $\pi/2$  radians); for  $k = 4$  this rotation is  $45^\circ$  ( $\pi/4$  radians); for  $k = 5$  this rotation is  $22.5^\circ$  ( $\pi/8$  radians); and for  $k = 6$  this rotation is  $11.25^\circ$  ( $\pi/16$  radians), for example. Notice that the  $k = 0$  case is identical to the `I` gate; the  $k = 1$  case is identical to the `Z` gate;  $k = 2$  case is identical to the `S` gate; and the  $k = 3$  case is identical to the `T` gate. For  $k = 4$  and above, we do not have a primitive gate that we have studied thus far in IBM QX, which is the equivalent, so we will expect to need a type of gate we haven't yet studied in order to implement this in IBM QX. This gate, called the `U1` gate, will be introduced in the IBM QX implementation section of this chapter.

What is left then is to make  $R_k$  a controlled gate. To do this, let's look to the CNOT gate as inspiration. Recall that the CNOT gate was defined as the following:

```
|CNOT=np.matrix('1 0 0 0; 0 1 0 0; 0 0 0 1; 0 0 1 0')
```

The first two columns of the first two rows of the CNOT gate, separated by semi-colons in the Python syntax, correspond to what amounts to the identity gate, meaning to leave the output unchanged if the control qubit is zero. The second two columns of the second two rows correspond to the X gate, that is, the NOT gate, meaning to flip the output if the control qubit is one. In this case, to define our controlled rotation gate, we will want to leave the first two rows the same as the CNOT gate, but change the second two rows to incorporate the  $R_k$  gate. Thus, we have the following:

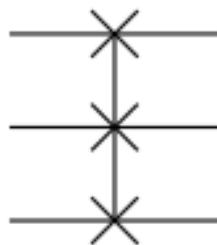
```
def CRk(k):
    return np.matrix([[1,0,0,0],[0,1,0,0],[0,0,1,0],[0,0,0,np.e**  
    (2*np.pi*1j/2**k)]])
```

# Reverse gate - REV

This REV gate swaps the order of the qubits. On a two qubit state, it does the following, for example:

<b>Input</b>	<b>Output</b>
$ 00\rangle$	$ 00\rangle$
$ 01\rangle$	$ 01\rangle$
$ 10\rangle$	$ 01\rangle$
$ 11\rangle$	$ 11\rangle$

The symbol for this gate is a cross over every wire of the circuit where the qubits are to be reversed. For example, for a three qubit reverse gate operating on a circuit of three qubits, it would be drawn as follows:

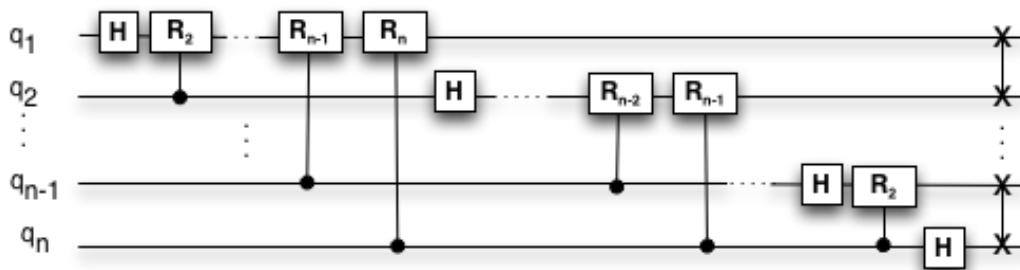


# QFT circuit

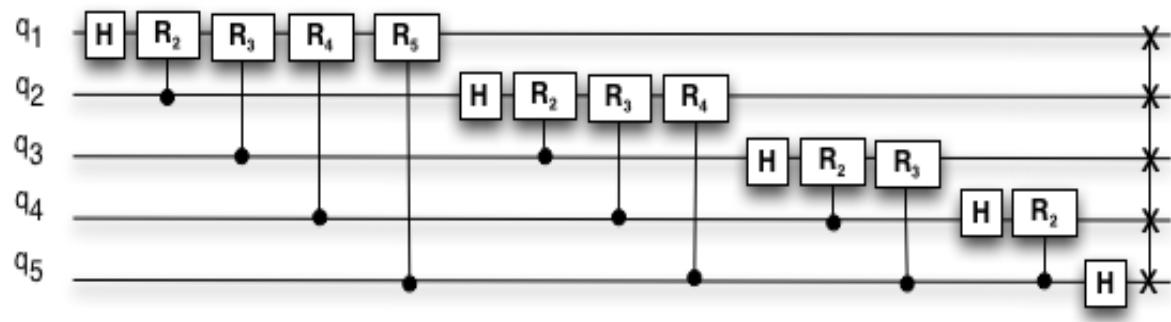
The QFT circuit will work as follows for the first qubit:

1. Apply a Hadamard gate to the first qubit.
2. Apply a controlled rotation gate  $R_k$  to the first qubit with parameter  $k = 2$  targeted on the following (second) qubit.
3. Continue applying controlled rotation gates  $R_k$  to the first qubit, increasing the parameter  $k$  each time by one, and moving the control qubit to the following qubit each time until we run out of control qubits. For example:
  - Apply a controlled rotation gate  $R_k$  to the first qubit with parameter  $k = 3$  controlled by the third qubit
  - Apply a controlled rotation gate  $R_k$  to the first qubit with parameter  $k = 4$  controlled by the fourth qubit

The entire QFT circuit will repeat the procedure above for each successive qubit until we run out of qubits. The general circuit diagram is as follows:



Specifically, here is an implementation with five qubits:



# **QFT circuit implementation in IBM QX**

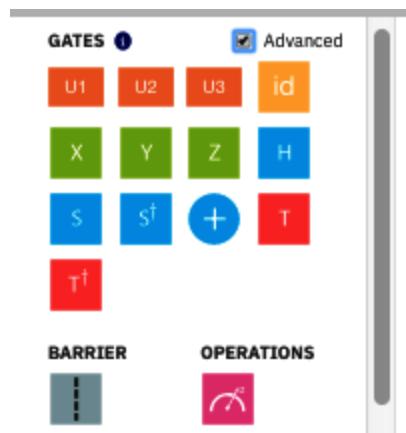
In this section, we'll do a QFT implementation for IBM QX for 1-qubit, 2-qubits and 3-qubits. Since the  $R_k$  gate and the REV gates aren't available natively in IBM QX, we will need to create these gates or find alternatives to them.

# Implementing the REV gate in IBM QX

Although conceptually, we could build a quantum gate out of the IBM QX primitive gates to perform a reverse, we could do something much simpler. We can just swap the bits after they are measured, classically. So if we measure bits  $_{001}$ , we can simply interpret that as the reverse,  $_{100}$ . A classical reversal, such as the Python `reverse` function on a list, will do just fine, and there is no need to add something additional to the quantum circuit. Recall that each additional gate in a circuit can introduce noise and extra computation time, so minimizing the number of gates is in our best interest.

# Implementing the R<sub>k</sub> gate in IBM QX

The standard gates we have worked with so far that rotate around the z axis are the Z, S, S<sup>†</sup>, T and T<sup>†</sup> gates, which rotate  $\pi$ ,  $\pi/2$ ,  $-\pi/2$ ,  $\pi/4$ ,  $-\pi/4$  or  $180^\circ$ ,  $90^\circ$ ,  $-90^\circ$ ,  $45^\circ$ , and  $-45^\circ$  respectively. To implement R<sub>k</sub> in IBM QX, as in Python, we will need a way to rotate around the z axis, about the x-y plane, by the number of degrees of our choice, not just a selection of  $\pi$ ,  $\pi/2$ ,  $-\pi/2$ ,  $\pi/4$ ,  $-\pi/4$ . To do this, we will use one of the advanced IBM QX gates. Via the quantum composer, click Advanced to see the gates:

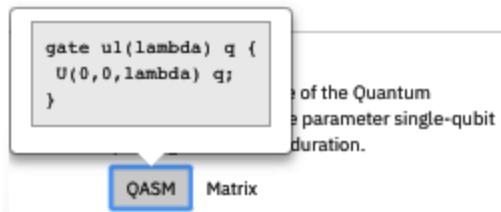


Then look at the description of the U<sub>1</sub> gate:

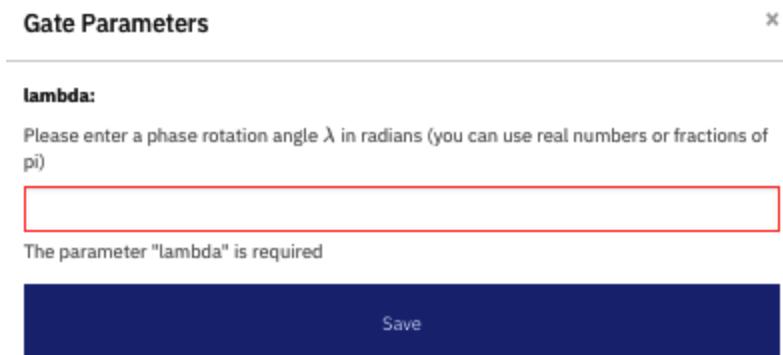
**U1** The first physical gate of the Quantum Experience. It is a one parameter single-qubit phase gate with zero duration.  
QASM Matrix

In OpenQASM, this gate can be created by using the u gate with parameters  $0, 0, \lambda$ , where  $\lambda$  will correspond to the

number of radians to rotate by (you can convert from degrees to radians by multiplying by  $\pi/180$ ):



You can use the  $U_1$  gate in the quantum composer by dragging it to the score, and the user interface will bring up a dialog to select the number of radians that the gate should perform the rotation about the  $z$  axis (in the  $x$ - $y$  plane) for:



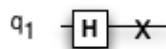
In the end, we will want not only the  $U_1$  gate but the controlled  $U_1$  gate,  $CU_1$ , which will perform the role of the controlled  $R_k$  gate. This is not available directly in the IBM QX GUI, although with some mathematics we could figure out the equivalent of the  $CU_1$  gate in terms of  $U_1$  and CNOT gates. Specifically,  $CU1(\lambda, control, target)$  is equivalent to the following:

```
u1(lambda/2) control;
cx control,target;
u1(-lambda/2) target;
cx control,target;
u1(lambda/2) target;
```

Here, lambda is in radians. Luckily the  $\text{CU}_1$  gate is available directly from Qiskit, so that makes things easy: it can simply be called with `cu1(lambda,control,target)`. To find the value of lambda to plug in for a given  $k$ , we need to plug in the value for  $k$  into the  $\lambda = 2\pi/2^k$  equation.

# 1-qubit QFT in IBM QX

Let's do a QFT implementation for 1-qubit. The general circuit diagram for the QFT over 1-qubit is simply as follows:



For the 1-qubit implementation, we don't have to worry about reversing (as it'd be the same), so the diagram is just the following:



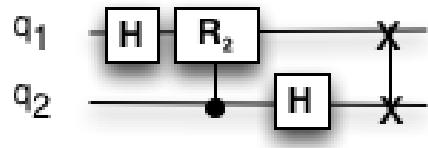
The code for this in Qiskit is as follows:

```
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
qr = QuantumRegister(1)
cr = ClassicalRegister(1)
circuit = QuantumCircuit(qr, cr)
circuit.h(qr[0])
```

Now, let's see an example of this QFT in action in the quantum simulator.

# 2-qubit QFT in IBM QX

Let's do a QFT implementation for 2-qubits. The general circuit diagram for the QFT over 2-qubits is as follows:



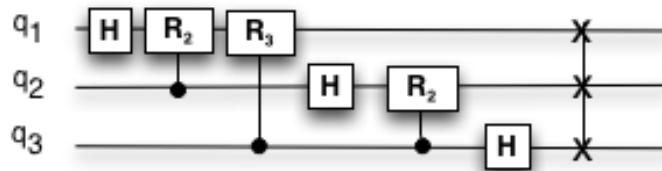
For IBM QX, we won't bother with the REV gate; we'll just do that classically after measurement switching 10 to 01 and 01 to 10 (00 stays the same when reversed, as does 11). So the task falls to implement the gate  $R_2$ . Here,  $\lambda = 2\pi/2^2 = 2\pi/4 = \pi/2$ , and so we will use the gate  $CU_1(\pi/2)$  where the control qubit is the second qubit, the target qubit is the first qubit, and the angle is  $\pi/2$ .

The code for this in Qiskit is as follows:

```
import math
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
qr = QuantumRegister(2)
cr = ClassicalRegister(2)
circuit = QuantumCircuit(qr, cr)
circuit.h(qr[0])
circuit.cu1(math.pi/2., qr[1], qr[0])
```

# 3-qubit QFT in IBM QX

Let's do a QFT implementation for 3-qubits. The general circuit diagram for the QFT over 3-qubits is as follows:



We saw in the previous section the implementation for  $R_2$ ; here, we will also need the implementation for  $R_3$  on IBM QX. Here,  $\lambda = 2\pi/2^3 = 2\pi/8 = \pi/4$ , and so we will use the gate  $CU_1(\pi/4, 1, 0)$  where the control qubit is the third qubit, the target qubit is the first qubit, and the angle is  $\pi/4$ .

The code for this in Qiskit is as follows:

```
import math
from qiskit import QuantumRegister, ClassicalRegister, QuantumCircuit
qr = QuantumRegister(3)
cr = ClassicalRegister(3)
circuit = QuantumCircuit(qr, cr)
circuit.h(qr[0])
circuit.cu1(math.pi/2., qr[1], qr[0])
circuit.cu1(math.pi/4., qr[2], qr[0])
circuit.h(qr[1])
circuit.cu1(math.pi/2., qr[2], qr[1])
circuit.h(qr[2])
```

# Generalizations

In the questions section of this chapter, you will have the opportunity to generalize the Qiskit code to compute the QFT on an arbitrary qubit number, as well as to implement the QFT in the IBM QX Composer.

# Summary

In this chapter, we learned about the Fourier transform, an algorithm that allows us to mathematically switch the representation of a periodic bounded function to represent it as a sum of sine waves of various frequencies, amplitudes, and phases. The inverse Fourier transform goes from the representation in terms of a sum of sine waves, back to the original function.

The Fourier transform has a quantum analogue, called the Quantum Fourier Transform, which is the Fourier transform of the amplitude of the quantum state. Since we can't measure the amplitude of a quantum state directly, the QFT is not typically used on its own, but rather as a useful subcomponent of many key quantum algorithms. In the next chapter, we will see the QFT in action when it is used as a component in Shor's algorithm.

# Questions

1. Sketch two different sine waves of different frequencies, and the wave you would get by summing the two.
2. Write out the circuit diagram for the 4-qubit QFT using H, REV, and R<sub>k</sub> gates.
3. Write out the circuit diagram for the 5-qubit QFT using H, REV, and R<sub>k</sub> gates.
4. Create a one qubit implementation for the QFT on the IBM QX Composer.
5. Create a two qubit implementation for the QFT on the IBM QX Composer.
6. Advanced: Create a three qubit implementation for the QFT on the IBM QX Composer.
7. Use the `h` and `cu1` gates, available in Qiskit, to write a QFT function in Python that takes as input a quantum circuit and quantum registers, along with a number  $n$  that will perform a QFT on  $n$  qubits on the quantum register in the circuit.

# Shor's Algorithm

This chapter explores Shor's algorithm, a quantum algorithm that is used to find the prime factorization of a number. This chapter goes over what prime factorization is, which is the classical implementation of an algorithm to perform prime factorization compared to the quantum implementation that's given by Shor's algorithm. We will also give you detailed practical examples in which Shor's algorithm running on a quantum computer of sufficient power has disruptive consequences. Finally, we will provide an implementation of Shor's algorithm in Qiskit. These examples can be run in Qiskit simulation.

In this chapter, we will cover the following topics:

- An overview of Shor's algorithm
- Examples of Shor's algorithm
- Python implementation of Shor's algorithm

# Shor's algorithm

As indicated by the following quote, Shor's algorithm is one of the most important in quantum computing:

*"Shor's algorithm has been called by Klaus Hepp the most exciting result in theoretical physics of the respective decade. There are not so many quantum algorithms, and Shor's solves very prominent problems from cryptography. It is somewhat ironic that there is this two fold-connection, with quantum crypto, between encryption and quantum physics."*

-Dr. Stefan Wolf, Professor USI

Shor's algorithm efficiently factors an integer into two other integers, which when multiplied give the original integer. The practical use case of Shor's algorithm is factoring a large integer, which is the product of two primes, into the two primes it is a product of. For example, the algorithm could factor 1,624,637,792,837 into the primes 15,485,867 and 104,911 as  $15485867 * 104911 = 1624637792837$ . Many modern cryptography systems, such as **RSA** (short for **Rivest-Shamir-Adleman**) cryptosystem, rely upon the fact that factoring such large integers is computationally difficult, so Shor's algorithm renders the data that's encrypted by such systems potentially decryptable, causing a huge disruption to the current way we conduct our digital lives.

# Factoring large integers efficiently disrupts modern cryptography

Much of modern cryptography and the way we keep digital data, such as online banking transactions, secure from prying eyes, relies on what are called trapdoor one-way functions. **Cryptography** is the process of encrypting and decrypting data. We can think of encryption as a function and decryption as the inverse of that function.

A one-way function is a function that is quick to compute for any input, but which is computationally difficult to invert; that is to say given an output of the function, it is computationally difficult to find the input of the function that would result in that output. So, a one-way function is very useful in encrypting data. However, we need a way for some people—just the people we want—to be able to invert the function, that is, to decrypt the encrypted data.

A trapdoor one-way function is the key to such a cryptosystem. A trapdoor one-way function permits a hint as to how to invert it. With this hint, the one-way function is no longer one-way—it is in fact easy to invert. Otherwise, without the hint, the function is a one-way function that's easy to compute in one direction and hard to compute the inverse. This means to all but those who possess the hint data can be encrypted, and only those with the hint, known as a secret-key in a cryptosystem, can decrypt the data.

One particularly popular cryptography system, the RSA algorithm, relies on a trapdoor one-way function involving the product of two primes to create what is called a semiprime number. Semiprimes are a great candidate for a one-way function as they are the hardest factorization case.

It is easy to compute the product of two prime numbers. For example, the product of the prime numbers 533,000,401 and 86,028,157 can be quickly computed in Python to be 45,853,042,178,290,957, but figuring out which primes multiply together to yield 45,853,042,178,290,957 is computationally tricky; thus this is a one-way function. However, given a hint, for example, that one of the two prime factors is 533,000,401, it becomes quick to compute again. Simply divide 45,853,042,178,290,957 by 533,000,401 to get the other prime; thus, this is a trapdoor one-way function. RSA uses primes so large that to factor their product without a hint on a classical computer could take more than a quadrillion years, for example, much longer than the age of the universe. The best known classical algorithm for factoring a semiprime integer into the product of two primes runs in time  $O(\exp(d^{1/3}))$ , where  $d$  is the number of decimal digits of the integer to factor, for example, for 45,853,042,178,290,957,  $d = 17$ . The record for  $d$  at the time of writing this book is around 200.

Any algorithm that has computed the prime factorization of a large integer efficiently, figuring out which two primes multiplied generated it, would ruin the one-way nature of the product of two primes. The RSA cryptosystem relies on this one-way nature, and so its cryptography would be broken in this scenario. This would have real-world consequences: since the RSA cryptosystem powers many secure financial transactions on the internet, these transactions would no longer be secure. Shor's algorithm can run on a quantum computer that runs in polynomial

time instead of exponential time; specifically, it runs in  $O(d^3)$ , unlike the best-known classical algorithm, which runs in  $O(\exp(d^{1/3}))$ .

New cryptosystems cannot be designed and deployed that are not vulnerable to Shor's algorithm, but that has not stopped the flurry of research and hope on the part of nation states that a quantum algorithm could give them unique access to encrypted data for the purposes of national advantage for offensive or defensive cyber operations. Since Shor's algorithm's existence shows that quantum computing can have a disruptive impact on real-world classical technology, it has resulted in the hope that many more quantum computing algorithms can be designed to show such disruptive potential.

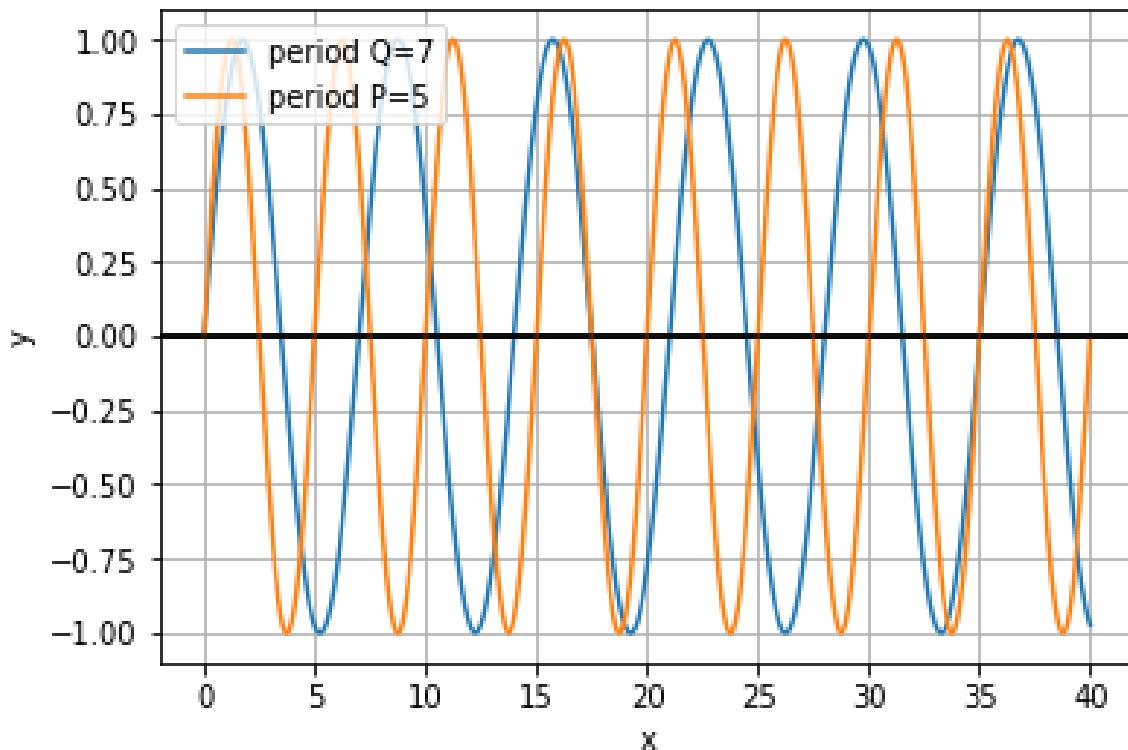
# Shor's algorithm overview

To follow Shor's algorithm exactly, we need to understand some mathematics related to the study of integers, which makes sense because Shor's algorithm is all about factoring integers into other integers. Some important concepts are the divisor, coprime, and the **greatest common divisor (GCD)**:

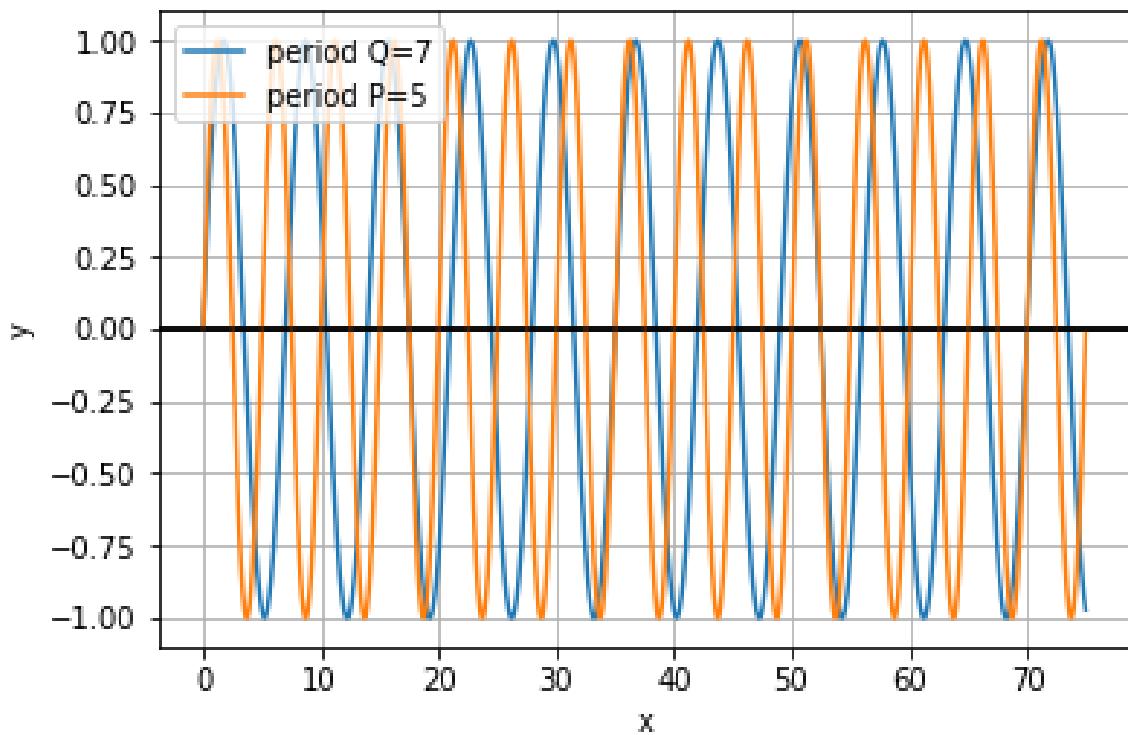
- **Divisor:** A number's divisors divide into it evenly, resulting in an integer. For example, divisors of the number 6 are 1, 2, 3, and 6, as  $6/1$ ,  $6/2$ ,  $6/3$ , and  $6/6$  are all integers. However, 4 is not a divisor as  $6/4$  is not an integer, and 5 is not a divisor as  $6/5$  is not an integer.
- **Coprime:** If two numbers are coprime, they don't share any divisors besides 1. For example, 3 and 6 are not coprime because they share a divisor—3 ( $3/3$  and  $6/3$  are both integers). Instead, consider 25 and 12, which are coprime. The divisors of 25 are 1 and 5, and the divisors of 12 are 1, 2, 3, 4, and 6. The only divisors they share are 1, and so they are coprime.
- **GCD:** The GCD between two numbers is the largest divisor they both share. If the GCD between two numbers is 1, then they are coprime as that means they don't share any divisors besides 1. For example, the GCD of 6 and 21 is 3, as the divisors of 6 are 1, 2, 3, and 6, and the divisors of 21 are 1, 3, and 7.

Given an integer  $N$ , we want to find another prime integer  $P$  between 1 and  $N$  where  $N$  divided by  $P$  is also a prime integer. If we call  $N/P$  or  $Q$ , then we know that  $N=P*Q$  (in mathematical language,  $M$  divides  $N$  or  $M$ , which is a divisor of  $N$ ). In the previous chapter, we learned about sine waves, phase, and the Quantum Fourier Transform. One way to think

intuitively about how a quantum computer might be useful in computing the factorization of  $N$  is if we have two sine waves of period  $P$  and period  $Q$ , then at a point at length  $N$  the phase of  $P$  and  $Q$  should be zero. Here is a visualization, imagining  $N = 35$ ,  $P = 5$ , and  $Q = 7$ :



Here, we can see where the two waves cross zero does not overlap at any integer from **0** to just before **35**, that is, they have a non-zero phase. At **35**, the waves overlap exactly, that is, they have a phase of **0**. After **35**, they will not overlap at any integer again until  $35 + 35 = 70$ :



This is why factorization relates to finding the period of two sign waves. Given  $N = 35$ , we can translate the problem of factorization to finding two waves having a phase difference of zero at multiples of  $35$  on the  $x$  axis, and the period of each of those waves are then the factors of  $N$ .

The basic idea behind Shor's algorithm is to rewrite the problem of finding the prime factorization of a number as something involving period finding, then do everything except for the period finding, and finally hand off the period finding portion of the algorithm to a quantum computer. Understanding the minute details of the mathematics of how this works is beyond the scope of this book, but if you are interested, have a look at *Quantum Computation and Quantum Information* by Nielsen and Chuang from Cambridge University Press (2000).

# Shor's algorithm described

Here, we will follow the steps for Shor's algorithm. The indented items all depend on the result computed one indentation in. For any integer greater than zero, our implementation will return a value. This implementation is described as follows:

1. Choose an integer random number less than the integer  $N$  we wish to factor.
2. Compute the GCD between our random number and  $N$ :
  3. If the GCD between our random number and  $N$  isn't one, great news—we've found a factor of  $N$ , which is the GCD we just computed. A second factor is the integer we want to factor divided by this GCD. The algorithm returns.
  4. Otherwise, if the GCD is one, then we consider the function: our random number to the  $x$  power modulo  $N$ , and compute its period (when the function repeats itself):
    5. If the period we find is odd, we're out of luck and need to choose another random number. We go back to step 1 and start over with a new random number.
    6. Else if the period we find to the power of our random number divided by two modulo  $N$  is the same as -1 modulo  $N$  (they are congruent modulo  $N$ ), we are also out of luck, so go back to the very beginning of the algorithm, and start over with a new random number.
    7. Else we are finally at a potential solution. We have found two factors of  $N$ .

- The first factor we have found can be computed by computing the GCD between the following:
  - $N$  the power of the period we found divided by two that quantity plus one, and  $N$
- The second factor can be computed by taking the GCD (greatest common divisor) between the following:
  - $N$  to the power of the period we found divided by two that quantity minus one, and  $N$
- If these two factors are simply  $N$  and 1, the solution isn't the one we are looking for, so we aren't done yet and go back to step 1 and choose a new random number. Otherwise, we're done and the algorithm returns the two factors we just found.

Shor's algorithm is mostly run on a classical computer. Only the step 4—finding the period when a function repeats is performed on a quantum computer is the quantum version of Shor's algorithm. We could also implement Shor's algorithm classically, doing all of these steps on a classical computer. If the function period finding of the step 4 is done on a quantum computer, the runtime of Shor's algorithm is  $O((\log N)^2(\log \log N)(\log \log \log N))$ , which is much (exponentially) more efficient than if that portion of the algorithm was done classically.

Why does Shor's algorithm work at all? Each algorithm can be mathematically proved using facts from number theory, a branch of math that tends to focus on the relationships between positive integers. This book isn't a book on number theory, so I won't go into the mathematical proofs that

underlie Shor's algorithm's utility, but there are a variety of resources on the internet. For the least mathematical introduction, I recommend the following post by Professor Scott Aaronson: <https://www.scottaaronson.com/blog/?p=208>.

Since this book *is* a book about quantum computing, I will focus on the details behind just the quantum part of the algorithm: that is, the step 4, and go into that in detail. First, we'll go through the whole of Shor's algorithm as being implemented completely classically, so that we can get a feel for various scenarios. Then, we'll change the implementation so that the step 4 is run on a quantum instead of a classical computer to realize the full power of the algorithm.

# Shor's algorithm described in symbols/mathematics

To code up Shor's algorithm in a classical computer, it will help to translate the words of the previous section by outlining the algorithm by the use of symbols. Calling the integer we wish to factor,  $N$ , Shor's algorithm in symbols is as follows:

1. Pick a random integer  $a$ , so that  $a < N$ .
2. Compute  $g = \gcd(a, N)$ :
  3. If  $g \neq 1$ ,  $g$  is a factor of  $N$ , so we're done: two factors of  $N$  are  $g$  and  $N/g$ .
  4. Or else find the period of  $f(x) = a^x \pmod{N}$ . That is, find the integer  $r$  after which  $f(x)$  repeats itself, meaning  $f(x) = f(x+r)$  or  $a^x \pmod{N} = a^{(x+r)} \pmod{N}$ :
    5. If  $r$  is odd, go back to the beginning.
    6. If  $a^{r/2} \pmod{N} = -1 \pmod{N}$ , go back to the beginning.
    7. We're in luck. Two factors of  $N$  are  $\gcd(a^{r/2} + 1, N)$  and  $\gcd(a^{r/2} - 1, N)$ . If these factors are simply 1 and  $N$ , go back to the beginning because we want more interesting factors.

The interesting part of Shor's algorithm is that only step 4 is done on a quantum computer. For demonstration purposes, we will first do all of the steps classically. Then, we will move on to the algorithm for step 4 on a quantum computer.

# **Shor's algorithm examples**

Note that because Shor's algorithm has randomness on the classical section in the portion where a random integer is chosen, it can be said that each example will turn out slightly shorter or slightly longer, depending on the random integer. However, to wrap our head around Shor's algorithm, it's useful to go through various examples, step by step.

# Example when N is prime, **N = 7**

Let's trace this through an example. Imagine  $N = 7$ . Therefore, its factors will be 7 and 1. Let's see what the algorithm does:

1. Pick a random integer  $a$ , so that  $a < 7$ . Our code picked  $a = 4$  in this example.
2. Compute  $g = \gcd(a, N) = \gcd(4, 7) = 1$ :
  3. Since  $g = 1$  we have to keep going!
  4. Find the period of  $f(x) = a^x \pmod{N} = 4^x \pmod{7}$ . This period is  $r = 3$ :
  5.  $r = 3$  is odd, so we go back to the beginning!

To go back to the beginning, you should do the following:

1. Pick a random integer  $a$ , so that  $a < 7$ . Our code picked  $a=0$  in this example.
2. Compute  $g = \gcd(a, N) = \gcd(0, 7) = 7$ :
  3. Since  $g = 7$  isn't equal to one, we're done. Two factors of 7 are  $g = 7$  and  $7/7 = 1$  (7 and 1).

# **Example when N is the product of two prime numbers, N is small, N = 15**

Let's say  $N = 15$ . Let's trace through this example:

1. Pick a random integer  $a$ , so that  $a < 15$ . Our code picked  $a = 6$  in this example.
2. Compute  $g = \gcd(a, N) = \gcd(6, 15) = 3$ :
3. Since  $g = 3$  is not one, we have found a non-trivial factor already. Two factors are  $g = 3$  and  $N/g = 15/3 = 5$ .

# **Example when N is the product of two prime numbers, N is larger, N = 2257**

Let's say  $N = 2257$ . Let's trace through this example.

1. Pick a random integer  $a$ , so that  $a < 2257$ . Our code picked  $a = 1344$  in this example.
2. Compute  $g = \gcd(a, N) = \gcd(1344, 2257) = 1$ :
  3. Since  $g = 1$ , we have to keep going!
  4. Find the period of  $f(x) = a^x \pmod{N} = 1344^x \pmod{2257}$ . This period is  $r = 180$ :
    5.  $r$  is even, so we keep going!
    6.  $a^{r/2} = 1344^{180/2} \pmod{2257} = 1768$  since;  $1768 \neq -1 \pmod{2257} = 2256$ , we keep going!
  7. We've found two factors of 2257.  
They are  $\gcd(a^{r/2} + 1, N) = \gcd(1344^{180/2} + 12257) = 61$  and  $\gcd(a^{r/2} - 1, N) = \gcd(1344^{180/2} - 12257) = 37$ .

Let's check our work:  $61 * 37 = 2257$ . We got it right!

# **Example when N is the product of one prime number and one non-prime number, N = 837**

Let's say  $N = 837$ . Let's trace through this example:

1. Pick a random integer  $a$ , so that  $a < 837$ . Our code picked  $802$  in this example.
2. Compute  $g = \gcd(a, N) = \gcd(802, 837) = 1$ :
  3. Since  $g = 1$ , we have to keep going!
  4. Find the period of  $f(x) = a^x \pmod{N} = 802^x \pmod{837}$ . This period is  $r = 30$ :
    5.  $r = 30$  is even, so we keep going!
    6.  $a^{r/2} = 802^{30/2} \pmod{837} = 433$  and  $433 \neq -1 \pmod{837} = 836$ , so we keep going!
  7. We've found two factors of 837. They are  $\gcd(a^{r/2} + 1, N) = \gcd(802^{30/2} + 1, 837) = 31$  and  $\gcd(a^{r/2} - 1, N) = \gcd(802^{30/2} - 1, 837) = 27$ .

Let's check our work:  $31 * 27 = 837$ . We got it right! Note that since one number isn't prime, there are a variety of different pairs of numbers that factor 837. The algorithm will always produce a valid pair, but which pair depends on the sequence of random numbers that have been chosen, so the algorithm may show can generically different results.

# Implementing Shor's algorithm in Python

Now, let's implement Shor's algorithm in Python. Recall that `%` is the mod operator in Python, and to check if an integer is even, we check if the integer *mod* 2 is equal to zero. The following code is Shor's algorithm in Python:

```
import math
import random
def shors_algorithm_classical(N):
    assert(N>0)
    assert(int(N)==N)
    while True:
        a=random.randint(0,N-1)
        g=math.gcd(a,N)
        if g!=1 or N==1:
            first_factor=g
            second_factor=int(N/g)
            return first_factor,second_factor
        else:
            r=period_finding_classical(a,N)
            if r % 2 != 0:
                continue
            elif a**(int(r/2)) % N == -1 % N:
                continue
            else:
                first_factor=math.gcd(a**int(r/2)+1,N)
                second_factor=math.gcd(a**int(r/2)-1,N)
                if first_factor==N or second_factor==N:
                    continue
                return first_factor,second_factor
```

# Shor's algorithm - classical implementation

Now, let's look at the classical subroutine. We know that to find the period of  $f(x) = a^x \pmod{N}$ , we just need to find the points along  $x$  that are the closest together for which the function repeats itself. We know  $f(0) = a^0 \pmod{N} = 1$ , so starting there, let's find the smallest  $x$  where the function is one again. In code this is as follows:

```
import itertools
def period_finding_classical(a,N):
    for r in itertools.count(start=1):
        if a**r % N == 1:
            return r
```

This algorithm assumes `a` is greater than zero. Note that the code only reaches this subroutine if the GCD of the random number we chose `a` and the number we wish to factor `N` isn't one and `N` isn't one. If `a` is zero,  $\gcd(a,N)$  will always be  $N$  and thus this period-finding algorithm will never be reached.

Let's run the algorithm in classical mode a few times for our examples  $N = 7, 15, 2257$ , and  $837$ :

```
print(shors_algorithm_classical(7))
print(shors_algorithm_classical(15))
print(shors_algorithm_classical(2257))
print(shors_algorithm_classical(837))
```

One example run gives us the following output:

```
(7, 1)
(3, 5)
(37, 61)
(3, 279)
```

Running it again, we can see that, because 837 isn't the product of primes, the two factors the algorithm returns can be different. For example, another run may produce the following output:

(7, 1)
(5, 3)
(37, 61)
(9, 93)

# Shor's algorithm - quantum implementation

Our goal is to implement step 4 of the algorithm, given the integer we want to factor  $N$  and a random number  $a$ . Solve the following problem:

*Find the period of  $f(x) = a^x \pmod{N}$ . That is, find the integer  $r$  after which  $f(x)$  repeats itself, that is, meaning  $f(x) = f(x+r)$  or  $a^x \pmod{N} = a^{(x+r)} \pmod{N}$ .*

Now, remember that our classical algorithm started at  $x = 1$  and kept counting up until  $a^x \pmod{N}$  equaled 1. At that location,  $x = r$ . So, we will always have to count up  $r$  integers to find the solution. With quantum parallelism, we will be able to find, once we implement the function  $f(x) = a^x \pmod{N}$  in the quantum computer,  $r$  in a single step using Quantum Fourier Transform. The steps to do this are described in the paper *Realization of a scalable Shor algorithm*: <https://arxiv.org/pdf/1507.08852.pdf>:

1. Create our counting up variable  $x$  by creating a quantum register that is a superposition of all possible integers it can possibly hold. For a quantum register of  $k$  qubits, that would mean it could represent a superposition of all possible values from 0 to  $2^k-1$  (each of the  $k$  qubits can be measured to have a value of either 0 or 1, so it's as if we have a binary value of length  $k$ , meaning it can represent integers from 0 to  $2^k-1$  ).

2. Using a computational quantum register, operate on  $x$  to compute all possible values of  $a^x \pmod{N}$  for  $x < 2^k$ .

Here's how we'd do this in a quantum computer:

1. Create our counting up quantum register  $x$  by using the Hadamard gate. The Hadamard gate ( $H$ ) when operating on each of the  $k$  qubits initialized to the "0" qubit creates a superposition between "0" and "1."
2. We design a quantum circuit to compute  $f(x) = a^x \pmod{N}$ . We could do this by mapping the problem onto classical logic gates, and then directly translating each gate into a quantum gate. For example, we replace the classical ANDs with quantum ANDs, the classical ORs with quantum ORs, and the classical NOTs with quantum NOTs. This would result in a lot of qubits and gates.  
Another option is to try and see if there are optimizations to reduce the number of qubits required.
3. We operate our  $f(x)$  circuit on our quantum register  $x$ . Then, we measure  $f(x)$  and see the result. Imagine that the result is  $f(x) = C$ . This will collapse the state of  $x$  into an equal superposition over all possible  $x$ 's that could have led to  $f(x) = C$ . Since  $f(x)$  is periodic, these values will all differ from each other by multiples of the period.
4. Our final task will be to extract the information about the period from the quantum register  $x$  that now contains the superposition over all possible numbers that could have led to  $f(x) = C$ . To do that, we will take the Quantum Fourier Transform of the state  $x$ , which will cause the result to have peaks in amplitude around multiples of the period. Measuring this result will help us extract the period.

# Example implementation on a quantum computer, $N = 15$ , $a = 2$

In this section, we will sketch the process for implementing Shor's algorithm, including the quantum computing sections, to factor  $N = 15$ . The steps for designing the preceding quantum circuit  $f(x) = a^x \pmod{N}$  involves designing a special quantum circuit for each  $a < N$  that the classical algorithm might randomly choose. To simplify our example, we will assume that the classical portion of the algorithm randomly chooses  $a = 2$ .

Let's say  $N = 15$ . Let's trace through the an example, beginning by doing the following on a classical computer:

1. Pick a random integer  $a$ , so that  $a < 15$ . Our code picked  $2$  in this example.
2. Compute  $g = \gcd(a, N) = \gcd(2, 15) = 1$ :
  3. Since  $g = 1$ , we have to keep going!
  4. Dispatch the problem of finding the period of  $f(x) = a^x \pmod{N} = 2^x \pmod{15}$  to a quantum computer.

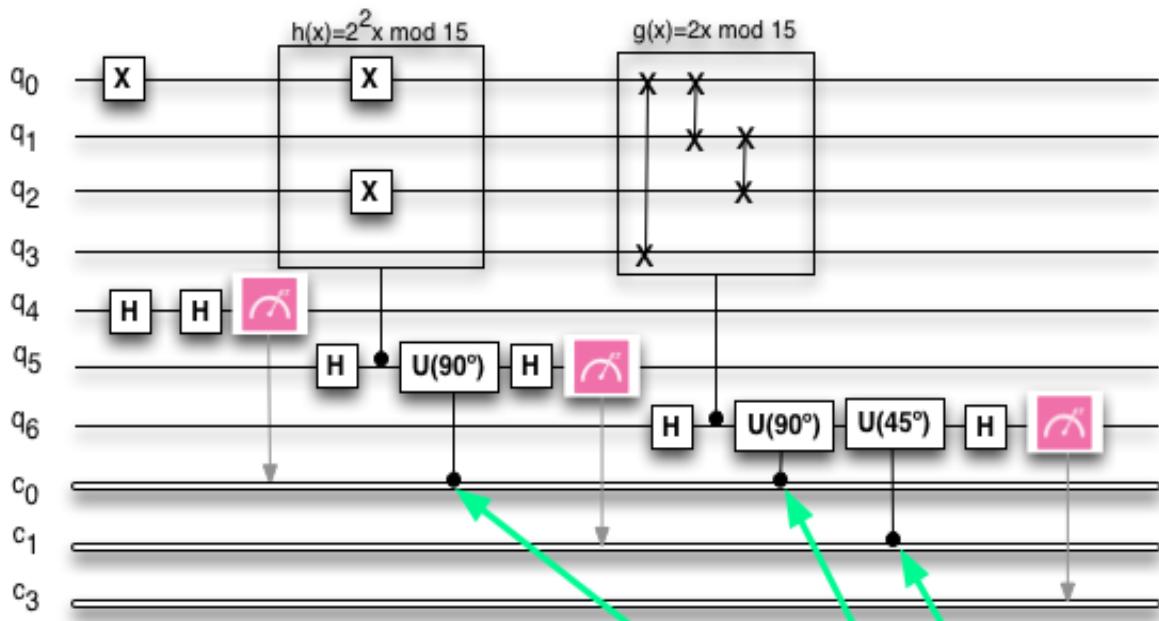
# Finding the period of $f(x) = ax \pmod{N} = 2x \pmod{15}$ using a quantum computer

We want to find the period of  $f(x) = a^x \pmod{N} = 2^x \pmod{15}$  using a quantum computer. The bad news is the most straightforward, naive algorithm (convert bits to qubits and classical gates to their quantum equivalent) works in theory, but if we tried to implement it on a quantum computer, it would need far too many qubits and gates to be practical. Each additional gate in a computation introduces errors, and each additional qubit requires the quantum computer to be of a larger size, which is more difficult to physically engineer.

To implement the quantum component of Shor's algorithm on a practical quantum computer today or in the near future, we will need to optimize the naive implementation of  $f(x) = a^x \pmod{N}$  to use fewer gates and qubits. Such optimizations rely on mathematical details and academic intricacies that are beyond the scope of this book. Instead, this book will prepare you to read a description of an implementation of such an optimization and from this description create an implementation and test the implementation you have created. Designing and fully understanding such optimizations is possible, and this book will prepare you to do so, but other sources and texts, such as *Quantum Computation and Quantum Information by Nielsen and Chuang*, will be very helpful.

In [Chapter 11](#), *Quantum Fourier Transform*, we skipped the detailed mathematics that give rise to why the algorithm (in this case, the optimizations  $f(x) = a^x \pmod{N}$ ) works, and

instead focus on translating a circuit depicting the optimized algorithm to the IBM QX. In that spirit, we will do some research to find if an optimization for creating the circuit for  $f(x) = a^x \pmod{N}$  exists; in the research, we will find that there is an optimization for  $f(x) = a^x \pmod{15}$ , which will be perfect for us. It is depicted in the paper *Realization of a scalable Shor algorithm* (Monz et al. 2015) <https://arxiv.org/pdf/1507.08852.pdf>. Within this paper, circuit c in figure 1 depicts the case of  $a = 2$ ,  $a = 7$ ,  $a = 8$ , or  $a = 13$ , and circuit b of figure 1 depicts the case of  $a = 11$ . In this paper, several qubits are reused via resetting, and instead of doing that, I've added additional qubits for clarity. Furthermore, I've redrawn the circuit diagram, thus switching around the order of the qubits:



Key:

$$\text{SWAP} \quad \begin{array}{c} X \\ \diagdown \\ X \end{array} = \begin{array}{c} \oplus \quad \bullet \quad \oplus \\ | \quad | \quad | \\ \bullet \quad \ominus \quad \bullet \end{array}$$

= every gate in this box is controlled by the qubit/bit under the black circle

Note: This circuit contains elements where a classical bit controls a quantum gate. Thus it is a semi-classical quantum circuit

Note that because this circuit contains classical bits controlling quantum gates, it is a semi-classical quantum computing circuit.

The rest of this chapter is dedicated to implementing this circuit diagram on IBM QX. It's a wonderful culmination to all the material in the book, that is, to be able to reference a research paper and implement its results on IBM QX. Reading quantum circuit diagrams, reorganizing and redrawing them for clarity and purpose, and implementing them is a skill to take away from the book and will be broadly applicable to any future quantum programming work.

# Subroutine to find $g(x) = 2x \pmod{15}$

In the preceding optimized circuit for computing  $f(x) = a^x \pmod{N}$ , we see that we will need to implement a circuit for computing  $g(x) = 2x \pmod{15}$ . Although we won't go into detail as to the full mathematics behind the optimized circuit, for the  $g(x)$  subcircuit in this section, we will trace why it works mathematically so that the generalization to other values might make more sense and seem less random.

Remember that the easiest quantum circuit to write conceptually is to simply take a classical program to compute  $g(x) = 2x \pmod{15}$ :

```
| def g(x):  
|     return 2*x % 15
```

Now, from this circuit, we need to figure out which series of universal classical gates (for example, ANDs and NOTs) correspond to this program. We can imagine seeing the assembly language this Python corresponds to, and then rewriting it in terms of just the gates AND and NOT. The final step is to switch the bits to qubits, and the AND gates to quantum AND gates and the NOT gates to quantum NOT gates, adding in extra qubits where necessary as the reversibility of the quantum AND gates and quantum NOT gates requires ancillary qubits. In the end, we will end up with a gigantic number of qubits, which isn't good for quantum computing as the more qubits, the more difficult it is to engineer a quantum computer that functions well.

Instead, we will use some mathematical tricks to minimize the computation.

The key thing to realize is that since we are always taking  $2x \pmod{15}$ , the result  $g(x)$  will always be between 0 and 14. The second thing to realize is that there is the potential for cycles in the results. Here is one example:

$$\begin{array}{l} 2*1 \pmod{15} = 2 \\ 2*2 \pmod{15} = 4 \\ 2*4 \pmod{15} = 8 \\ 2*8 \pmod{15} = 1 \end{array}$$

The result of  $g(x=1)$  is 2. The result of  $g(x=2)$  is 4, the result of  $g(x=4)$  is 8, and the result of  $g(x=8)$  gets us back to 1. This can be written as a map: 1 / 2 / 4 / 8 / 1.

Here is another example:

$$\begin{array}{l} 2*3 \pmod{15} = 6 \\ 2*6 \pmod{15} = 12 \\ 2*12 \pmod{15} = 9 \\ 2*9 \pmod{15} = 3 \end{array}$$

This can be written as the map: 3 | 6 | 12 | 9 | 3. For  $2x \pmod{35}$ , all the possible cycles in the values of  $x$  are as follows:

$$\begin{array}{c|c|c|c|c|c} 11 & 2 & 4 & 8 & 1 \\ 3 & 6 & 12 & 9 & 3 \\ 5 & 10 & 5 \\ 7 & 14 & 13 & 11 & 7 \end{array}$$

This is called a modular multiplication map. You read it as follows  $g(1) = 2*1 \pmod{15}$ , and you can get the result by going to the preceding chart, finding 1, and seeing what the arrow points to. In this case, the answer is 2! Now, what about  $g(14)=2*14 \pmod{15}$ ? We need find 14, in the last example, and see that the arrow points to 13. Sure enough, that's the answer. How does that help us? Well, it means that if we know what  $x$  is (say, 6), we can easily compute

the result for  $g(x=6)$  using the chart. Since 15 in binary is 1111 it is four bits long, and that means that each input to  $g(x)$  (say  $x = 8$  or in binary: 1000) will simply map the binary representation of the input to an output according to the map above it. In this case,  $f(8)$  is equivalent to  $f(1000)$  in binary and should result in 1 in decimal, which is 0001 in binary.

Our quantum circuit to implement this would need to do the same. If the input is  $|1000\rangle$ , it should output  $|0001\rangle$ , and so on for each of the series of numbers in the example. So to create the correct circuit for  $f(x) = 2x \pmod{15}$ , we would need to do the following:

1. Find the groups of repetitions
2. Convert decimal to binary
3. Create an algorithm that changes one binary to the next, as in the example, consistently

Let's write out the map in binary and see if we can notice a pattern in the modular multiplication map:

- 1 | 2 | 4 | 8 | 1 is written in binary as 0001 | 0010 | 0100 | 1000 | 0001
- 3 | 6 | 12 | 9 | 3 is written in binary as 0011 | 0110 | 1100 | 1001 | 0011
- 5 | 10 | 5 is written in binary as 0101 | 1010 | 0101
- 7 | 14 | 13 | 11 | 7 is in binary 0111 | 1110 | 1101 | 1011 | 0111

We are lucky with this because a couple of patterns jump out at first glance. First, for each of these lines, the binary value has the same number of 1s and 0s for the whole line: each of 1 | 2 | 4 | 8 | 1 in binary have one 1 in binary and three 0s; each of 3 | 6 | 12 | 9 | 3 have two 1s in binary and two 0s; each of 5 | 10 | 5 have two 1s in binary and two 0s;

and each of  $7 | 14 | 13 | 11 | 7$  have three 1s in binary and one 0. This gives us a clue as to a possible pattern. We might not have to use negation to switch any 0s to 1s, or 1s to 0s (which would change the number). We might be able to get away with just swapping the position of 1s and 0s.

The following quantum algorithm works to get from any starting input on the modular multiplication map to a result:

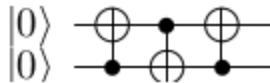
1. Convert decimal to binary
2. Translate each bit into a qubit
3. Swap the 0<sup>th</sup> qubit and the third
4. Swap the 0<sup>th</sup> qubit and the first
5. Swap the first and the second
6. Measure qubits 0, 1, 2, and 3 and place the result in a classical bit register
7. Convert the resulting bits to decimal to read off the result in decimal

Let's see an example of  $g(6)$ :

1.  $6$  is  $0110$  in binary
2. Translate  $0110$  into  $|"0110">$
3. Swap the 0<sup>th</sup> qubit and the third to get  $|"0110">$
4. Swap the 0<sup>th</sup> qubit and the first to get  $|"1010">$
5. Swap the first and the second to get  $|"1100">$
6. Measure  $|"1100">$  to get  $1100$
7. Convert  $1100$  to decimal to get the result:  $12$

Sure enough, if we look up  $6$  on the preceding modular multiplication map, we get  $12$ , the right result. We can verify this for any input.

The last thing we want to do is have a quantum SWAP gate, which exchanges two qubits. This can be implemented with three CNOT gates. The circuit diagram is as follows:



In other terms, to swap the 0th qubit and the first qubit, we'd use the following code in Qiskit, assuming the `qc` variable referred to a quantum circuit and the `qr` variable to a quantum register containing at least two qubits:

```
| qc.cx(qr[1],qr[0])
| qc.cx(qr[0],qr[1])
| qc.cx(qr[1],qr[0])
```

Thus, to implement steps 3, 4, and 5, involving the qubit swaps, we would code the following in Qiskit:

```
def mult_2mod15_quantum(qr,qc):
    # Swap 0th qubit and 3rd qubit
    qc.cx(qr[0],qr[3])
    qc.cx(qr[3],qr[0])
    qc.cx(qr[0],qr[3])

    # Swap 0th qubit and 1st qubit
    qc.cx(qr[1],qr[0])
    qc.cx(qr[0],qr[1])
    qc.cx(qr[1],qr[0])

    # Swap 1st qubit and 2nd qubit
    qc.cx(qr[1],qr[2])
    qc.cx(qr[2],qr[1])
    qc.cx(qr[1],qr[2])
```

In the notebook provided with this chapter, there is additional code to check this quantum algorithm as run on IBM QX (where prior to running, an input is initialized by setting the qubits in binary equal to the value of  $x$  as in step 2) against the classical implementation in Python, which is  $2*x \% 15$ .

# Full algorithm implementation

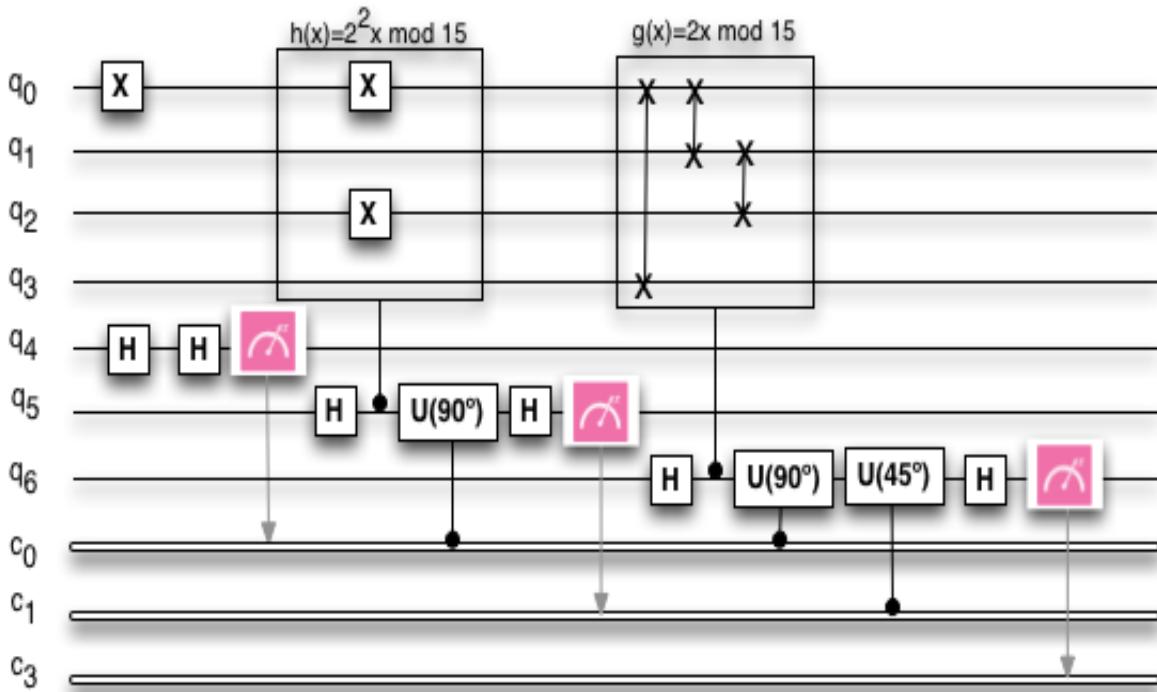
We will first need to make our `mult_2mod15_quantum` algorithm controlled by an additional qubit to fit the circuit. That function performs qubit swaps between three different qubits. Luckily, there is already a controlled swap gate called `cswap` in Qiskit we can rely on to control the computation, so all we have to do is rewrite each of the three swaps using `cswap`. Porting the `mult_2mod15_quantum` function to `controlled_mult_2mod15_quantum` then goes as follows:

```
def controlled_mult_2mod15_quantum(qr,qc,control_qubit):
    """
    Controlled quantum circuit for multiplication by 2 mod 15.
    Note: control qubit should an index greater than 3,
          and qubits 0,1,2,3 are reserved for circuit operations
    """
    # Swap 0th qubit and 3rd qubit
    qc.cswap(control_qubit,qr[0],qr[3])

    # Swap 0th qubit and 1st qubit
    qc.cswap(control_qubit,qr[1],qr[0])

    # Swap 1st qubit and 2nd qubit
    qc.cswap(control_qubit,qr[1],qr[2])
```

Next, we will need to refer back to the circuit to do the rest of the implementation which has been, repeated here for convenience:



The Python implementation is as follows:

```
def shors_subroutine_period_2mod15(qr,qc,cr):
    qc.x(qr[0])
    qc.h(qr[4])
    qc.h(qr[4])
    qc.measure(qr[4],cr[0])

    qc.h(qr[5])
    qc.cx(qr[5],qr[0])
    qc.cx(qr[5],qr[2])
    if cr[0]:
        qc.u1(math.pi/2,qr[4]) #pi/2 is 90 degrees in radians
    qc.h(qr[5])
    qc.measure(qr[5],cr[1])

    qc.h(qr[6])
    controlled_mult_2mod15_quantum(qr,qc,qr[6])
    if cr[1]:
        qc.u1(math.pi/2,qr[6]) # pi/2 is 90 degrees in radians
    if cr[0]:
        qc.u1(math.pi/4,qr[6]) #pi/4 is 45 degrees in radians
    qc.h(qr[6])
    qc.measure(qr[6],cr[2])
```

# Reading out the results

The quantum subroutine to Shor's algorithm can generically produce a variety of different measurement outcomes. Note that none of these measurements give us the period  $r$ . Instead, a measurement, rewritten in the right manner, gives us an integer multiple of the period  $r$  with high probability.

The quantum subroutine of Shor's algorithm exploits constructive and destructive interference, giving us information about multiples of the number of possible measurements divided by the period  $r$ .

If the period  $r$  happens to be a value that divides the total number of possible measurements ( $2^{\text{number of qubit registers}}$ ), then we get as an output a perfectly constructive or perfectly destructive interference as a measurement. If it doesn't, the measurement returns a rounding off of the actual answer. To retrieve the actual period then, we need to determine which integer multiple of total possible measurements divided by the period a measurement is. To do this, we use an algorithm called the **continued fraction expansion**.

To find the period  $r$  from a measurement, we will need the additional mathematical trick of the *continued fraction expansion*. This discussion follows that in *A Lecture On Shor's Quantum Factoring Algorithm*, Lomanaco, 2000 (<https://arxiv.org/pdf/quant-ph/0010034.pdf>). Any positive rational number can be rewritten as a sum as follows:

$$\xi = a_0 + \cfrac{1}{a_1 + \cfrac{1}{a_2 + \cfrac{1}{a_3 + \cfrac{1}{\ddots + \cfrac{1}{a_N}}}}},$$

The continued fraction expansion, given  $\xi$ , finds the coefficients  $a_0, a_1, \dots, a_N$ . The algorithm is as follows (here, we have a base case initialized, and each subsequent  $a$  is computed from the previous one):

$$a_0 = \lfloor \xi \rfloor, \xi_0 = \xi - a_0; \text{ if } \xi_n \neq 0, \text{ then } a_{n+1} = \lfloor 1/\xi_n \rfloor, \xi_{n+1} = \frac{1}{\xi_n} - a_{n+1}$$

We can also rewrite each  $\xi_n = p_n/q_n$  where these can be computed from the following (again we have a base case initialized and each subsequent  $p$  and  $q$  is then computed from the previous one):

$$p_0 = a_0, p_1 = a_1 a_0 + 1; p_n = a_n, p_{n-1} + p_{n-2}$$

$$q_0 = 1, q_1 = a_1; q_n = a_n, q_{n-1} + q_{n-2}$$

Here is some Python code to implement this:

```
import math
def continued_fraction(xi,max_steps=100): # stop_after is cutoff for
    algorithm, for debugging
    """
    This function computes the continued fraction expansion of input xi
    per the recurrence relations on page 11 of
    https://arxiv.org/pdf/quant-ph/0010034.pdf
    """

```

```

#a and xi initial
all_as=[]
all_xis=[]
a_0=math.floor(xi)
xi_0=xi-a_0
all_as+=[a_0]
all_xis+=[xi_0]
# p and q initial
all_ps=[]
all_qs=[]
p_0=all_as[0]
q_0=1
all_ps+=[p_0]
all_qs+=[q_0]

xi_n=xi_0
while not numpy.isclose(xi_n,0,atol=1e-7):
    if len(all_as)>=max_steps:
        print("Warning: algorithm did not converge within %d steps,
try increasing max_steps"%max_steps)
        break
    # computing a and xi
    a_nplus1=math.floor(1/xi_n)
    xi_nplus1=1/xi_n-a_nplus1
    all_as+=[a_nplus1]
    all_xis+=[xi_nplus1]
    xi_n=xi_nplus1
    # computing p and q
    n=len(all_as)-1
    if n==1:
        p_1=all_as[1]*all_as[0]+1
        q_1=all_as[1]
        all_ps+=[p_1]
        all_qs+=[q_1]
    else:
        p_n=all_as[n]*all_ps[n-1]+all_ps[n-2]
        q_n=all_as[n]*all_qs[n-1]+all_qs[n-2]
        all_ps+=[p_n]
        all_qs+=[q_n]
return all_ps,all_qs,all_as,all_xis

```

Try it out on some numbers. What do you get for  $\xi = 51.54639175257732$ ? Try dividing the final  $p$  by the final  $q$  it should be close to  $\xi$ . What happens if  $\xi = \pi$ ?

We aren't quite done yet. To extract the period from our measurement on the quantum computer, we need to perform a step on a classical computer: specifically, we compute the continued fraction expansion of `measurement_result/total_measurement_possibilities`; that is, we compute the continued fraction expansion of *(measurement*

$result)/2^k$ , where  $k$  is the number of qubits that the QFT is computed on.

Here is an example where we can compute the elements of the continued fraction expansion one by one and stop when we reach an element  $q_n$  for which  $a^{q_n} = 1 \pmod{N}$ . Recall that  $a$  is the random number chosen as part of the first step of Shor's algorithm, and  $N$  is the number we wish to factor.



*The previous check is easy to build into code, but there is also a more intuitive formulation. Here is an example where we take the entire continued fraction expansion, and then figure out the period  $r$  without having to do any modular arithmetic in our head. Say that one outcome of our experiment is the value 6 in a 3-qubit register. A 3-qubit register has  $2^3 = 8$  possible values. We then take 6/8 and rewrite it using the continued fraction expansion:  $p_0/q_0 = 0/1$ ,  $p_1/q_1 = 1/1$ ,  $p_2/q_2 = 3/4$ . Going in order, the last denominator ( $q$ ), that is, less than the number we are trying to factor (imagine it is 15) is our answer. In this case it is  $q_2 = 4$ , meaning our period is  $r = 4$ .*

Here is then the final algorithm to extract the period from the quantum measurement:

```
import math
def period_from_quantum_measurement(quantum_measurement,
                                      number_qubits,
                                      a_shor,
                                      N_shor,
                                      max_steps=100):
    """
    This function computes the continued fraction expansion of input xi
    per the recurrence relations on page 11 of
    https://arxiv.org/pdf/quant-ph/0010034.pdf
    a_shor is the random number chosen as part of Shor's algorithm
    N_shor is the number Shor's algorithm is trying to factor
    """
    xi=quantum_measurement/2**number_qubits

    #a and xi initial
    all_as=[]
    all_xis=[]
    a_0=math.floor(xi)
    xi_0=xi-a_0
    all_as+=[a_0]
    all_xis+=[xi_0]
    # p and q initial
    all_ps=[]
    all_qs=[]
    p_0=all_as[0]
    q_0=1
    all_ps+=[p_0]
```

```

all_qs+=[q_0]

xi_n=xi_0
while not numpy.isclose(xi_n,0 atol=1e-7):
    if len(all_as)>=max_steps:
        print("Warning: algorithm did not converge within max_steps
              %d steps, try increasing max_steps"%max_steps)
        break
    # computing a and xi
    a_nplus1=math.floor(1/xi_n)
    xi_nplus1=1/xi_n-a_nplus1
    all_as+=[a_nplus1]
    all_xis+=[xi_nplus1]
    xi_n=xi_nplus1
    # computing p and q
    n=len(all_as)-1
    if n==1:
        p_1=all_as[1]*all_as[0]+1
        q_1=all_as[1]
        all_ps+=[p_1]
        all_qs+=[q_1]
    else:
        p_n=all_as[n]*all_ps[n-1]+all_ps[n-2]
        q_n=all_as[n]*all_qs[n-1]+all_qs[n-2]
        all_ps+=[p_n]
        all_qs+=[q_n]
    # check the q to see if it is our answer
    # (note with this we skip the first q, as a trivial case)
    if a_shor**all_qs[-1]%N_shor == 1 % N_shor:
        return all_qs[-1]

```

# Running Shor's algorithm on a quantum computer

Let's actually run this on a quantum computer. Note that because of its semi-classical nature, it is more convenient to run it in the simulator, which fully supports the `if` statements that are dependent on the preceding classical bits. Quantum hardware, including IBM QX, will be able to support such semi-classical algorithms in the near future. The following code will run Shor's subroutine in the local simulator for multiplication by 2 (*mod* 15):

```
from qiskit import QuantumCircuit, ClassicalRegister, QuantumRegister
from qiskit import Aer
from qiskit import IBMQ
# Authenticate an account and add for use during this session. Replace
string
# argument with your private token.
IBMQ.enable_account("INSERT_YOUR_API_TOKEN_HERE")

def binary_string_to_decimal(s):
    dec=0
    for i in s[::-1]:
        if int(i):
            dec+=2**int(i)
    return dec

def run_shors_subroutine_period2_mod15():
    qr = QuantumRegister(7)
    cr = ClassicalRegister(3)
    qc = QuantumCircuit(qr,cr)
    # run circuit (which includes measurement steps)
    shors_subroutine_period_2mod15(qr,qc,cr)

    import time
    from qiskit.tools.visualization import plot_histogram
    backend=Aer.get_backend('qasm_simulator')
    job_exp = qiskit.execute(qc, backend=backend,shots=1)
    result = job_exp.result()
    final=result.get_counts(qc)
    # convert final result to decimal
    measurement=binary_string_to_decimal(list(final.keys())[0])
    period_r=period_from_quantum_measurement(measurement,3,2,15)
    return period_r
```

Then, we need to reformulate our `shors_algorithm_classical` code to use a quantum period finder instead of a classical period finder, and in the event the quantum period finder doesn't return a period or returns the incorrect period, to keep going in the algorithm until we do:

```

def period_finding_quantum(a,N):
    # for the sake of example we will not implement this algorithm in
    full generality
    # rather, we will create an example with one specific a and one
    specific N
    # extension work could be done to impl
    if a==2 and N==15:
        return run_shors_subroutine_period2_mod15()
    else:
        raise Exception("Not implemented for N=%d, a=%d" % (N,a))

def shors_algorithm_quantum(N):
    assert(N>0)
    assert(int(N)==N)
    while True:
        a=random.randint(0,N-1)
        g=math.gcd(a,N)
        if g!=1 or N==1:
            first_factor=g
            second_factor=int(N/g)
            return first_factor,second_factor
        else:
            r=period_finding_quantum(a,N)
            if not r:
                continue
            if r % 2 != 0:
                continue
            elif a**(int(r/2)) % N == -1 % N:
                continue
            else:
                first_factor=math.gcd(a**int(r/2)+1,N)
                second_factor=math.gcd(a**int(r/2)-1,N)
                if first_factor==N or second_factor==N:
                    continue
                if first_factor*second_factor!=N:
                    # checking our work
                    continue
                return first_factor,second_factor

```

Note that this code will crash unless we "happen" to pick  $a = 2$  in the `a=random.randint(0,N-1)` line. A task in the questions portion of this chapter is to finish the code so that it will work with any  $a$  for an  $N = 15$ . An advanced task in the questions portion of this chapter is to implement the code

so that it works with another  $N$ . For now we can control which  $a$  the algorithm picks by changing the signature to read as follows:

```
| def shors_algorithm_quantum(N,fixed_a=None):
```

Next, change the line that chooses the random number to consider if the user has decided to fix the random number that's been chosen:

```
|     if not fixed_a:  
|         a=random.randint(0,N-1)  
|     else:  
|         a=fixed_a
```

Then, to test our algorithm for  $a = 2$ , we can run the following:

```
| shors_algorithm_quantum(15,fixed_a=2)
```

This prints the following:

```
| (5, 3)
```

To see how the algorithm functions with any possible value of  $a$ , run:

```
| # Now trying it out to see how the algorithm would function if we let it  
| choose a given random a:  
| for a in range(15):  
|     # Here's the result for a given a:  
|     try:  
|         print("randomly chosen a=%d would result in %s"%  
|                 (a,shors_algorithm_quantum(15,fixed_a=a)))  
|     except:  
|         print("FINISH IMPLEMENTING algorithm doesn't work with a  
|             randomly chosen a=%d at this stage"%a)
```

This will print the following:

```
| FINISH IMPLEMENTING algorithm doesn't work with a randomly chosen a=0 at  
| this stage  
| FINISH IMPLEMENTING algorithm doesn't work with a randomly chosen a=1 at  
| this stage  
| randomly chosen a=2 would result in (5, 3)
```

```
randomly chosen a=3 would result in (3, 5)
FINISH IMPLEMENTING algorithm doesn't work with a randomly chosen a=4 at
this stage
randomly chosen a=5 would result in (5, 3)
randomly chosen a=6 would result in (3, 5)
FINISH IMPLEMENTING algorithm doesn't work with a randomly chosen a=7 at
this stage
FINISH IMPLEMENTING algorithm doesn't work with a randomly chosen a=8 at
this stage
randomly chosen a=9 would result in (3, 5)
randomly chosen a=10 would result in (5, 3)
FINISH IMPLEMENTING algorithm doesn't work with a randomly chosen a=11 at
this stage
randomly chosen a=12 would result in (3, 5)
FINISH IMPLEMENTING algorithm doesn't work with a randomly chosen a=13 at
this stage
FINISH IMPLEMENTING algorithm doesn't work with a randomly chosen a=14 at
this stage
```

# Tracing after the period is found on the quantum computer

For values of  $a$  that end up calling the quantum period finding subroutine, here is a trace of what happens after the period  $r = 4$  is found:

5.  $r = 4$  is even, so we keep going!
6.  $a^{r/2} = 2^{4/2} \pmod{15} = 4$  and  $4 \neq -1 \pmod{35} = 14$ , so we keep going!
7. We've found two factors of 15. They are  $\gcd(a^{r/2} + 1, N) = \gcd(2^{4/2} + 1, 15) = 5$  and  $\gcd(a^{r/2} - 1, N) = \gcd(2^{4/2} - 1, 15) = 3$ .

Let's check our work:  $3 * 5 = 15$ . We got it right! We can stop.

# Example implementation on a quantum computer, $N = 35$ , $a = 8$

In this section, we will sketch the process for implementing Shor's algorithm, including the quantum computing sections, to factor  $N = 35$ . The step designing the quantum circuit  $f(x) = a^x \pmod{N}$  involves designing a special quantum circuit for each  $a < N$  that the classical algorithm might randomly choose. To simplify our example, we will assume the classical portion of the algorithm randomly chooses  $a = 8$ .



*Here, we fix the random number that's been chosen to simplify the example. In the questions section, you will get the chance to finish implementing Shor's algorithm with quantum period finding to factor  $N = 15$ , and create the circuitry that's necessary to allow any random number to be chosen. The goal in this section is to partially work an example with another  $N$ .*

Let's say  $N = 35$ . Let's trace through another following. Following the same numbering convention of Shor's algorithm as the previous descriptions in words, math, and as in the previous examples, we do the following on a classical computer:

1. Pick a random integer  $a$ , so that  $a < 35$ . Our code picked 8 in this example.
2. Compute  $g = \gcd(a, N) = \gcd(8, 35) = 1$ :
  3. Since  $g = 1$ , we have to keep going!
  4. Dispatch the problem of finding the period of  $f(x) = a^x \pmod{N} = 8^x \pmod{35}$  to a quantum computer.

# **Finding the period of $f(x) = ax \pmod{N} = 8x \pmod{35}$ using a quantum computer**

The first thing we will do is implement multiplication by 8 mod 35 on a quantum computer (that is, we will implement the  $f(x) = 8x \pmod{35}$  function). The brute-force way to do this would be to take a classical program to compute the function:

```
| def 8xmod35(x):  
|     return 8*x % 35
```

Then, from the assembly language this Python corresponds to, we need to rewrite that assembly language in terms of just the AND and NOT gates. The final step would be to switch the bits to qubits, the AND gates to quantum AND gates, and the NOT gates to quantum NOT gates, adding in extra qubits where necessary as the reversibility of the quantum AND gate and quantum NOT gates require ancillary qubits. In this brute-force translation approach from a classical computer to a quantum computer, we would end up with a gigantic number of qubits. This number of qubits would not be practical as the more qubits an algorithm uses, the more difficult it is to engineer a quantum algorithm that is reliable and can run on real-world hardware. Therefore, we cannot practically take the brute-force translation approach.

Instead, we can use some mathematical tricks to minimize the number of qubits required for the computation. The key

thing to realize is that since we are always taking  $8x \pmod{35}$ , the  $f(x)$  result will always be between 0 and 34. The second thing to realize is that there is the potential for cycles in the results. The following is one example:

```
8*1 mod 35 = 8
8*8 mod 35 = 29
8*29 mod 35 = 22
8*22 mod 35 = 1
```

Here is another example:

```
8*2 mod 35 = 16
8*16 mod 35 = 23
8*23 mod 35 = 9
8*9 mod 35 = 2
```

Finding the modular multiplication map for  $8x \pmod{35}$  is a bit more involved. Here is a bit of code that can compute the modular multiplication map for any value of  $a$  (here  $a = 8$ ) and any value of  $N$  (here,  $N = 35$ ):

```
def U_a_modN(a,N,binary=False):
    """
    a and N are decimal
    This algorithm returns U_a where:
        U_a is a modular multiplication operator map from |x> to |ax mod
    N>
        If binary is set to True, the mapping is given in binary instead of
    in decimal notation.
    """
    res={}
    l=[]
    for i in range(1,N):
        l+=[a*i%N]
    res=set()

    for i in range(1,N):
        mp=[i]
        end=i
        nxt=i-1
        while l[nxt]!=end:
            mp+=[l[nxt]]
            nxt=l[nxt]-1
        res.add(tuple(mp))
    final_res=[]
    for item in res:
        dup=False
        for final_item in final_res:
            if set(item) == set(final_item):
```

```

        dup=True
    if not dup:
        final_res+=[item]
    if not binary:
        return final_res
    else:
        final_res_bin=[]
        for mapping in final_res:
            final_res_bin+=[tuple(['{:06b}'.format(decimal)
                                  for decimal in mapping])]
    return final_res_bin

```

The results for  $a = 8$  and  $N = 35$  in decimal and binary are then computed with the following commands:

```

print(U_a_modN(8,35))
print(U_a_modN(8,35,binary=True))

```

This prints the following:

```

[(7, 21, 28, 14), (34, 27, 6, 13), (2, 16, 23, 9), (26, 33, 19, 12), (18,
4, 32, 11), (24, 17, 31, 3), (15,), (30,), (5,), (8, 29, 22, 1), (20,), (10,), (25,)]

[('000111', '010101', '011100', '001110'), ('100010', '011011', '000110',
'001101'), ('000010', '010000', '010111', '001001'), ('011010', '100001',
'010011', '001100'), ('010010', '000100', '100000', '001011'), ('011000',
'010001', '011111', '000011'), ('001111',), ('011110',), ('000101',),
('001000', '011101', '010110', '000001'), ('010100',), ('001010',), ('011001',)]

```

For  $8x \pmod{35}$ , all the possible cycles in the values of  $x$  are as follows (since each of the tuples are a cycle, we can choose any entry point to the cycle; here, I adopt the convention of having the entry points to the cycle be the lowest value in the tuple, and then ordering the cycles by that first entry point):

1	8	29	22	1
2	16	23	9	2
3	24	17	31	3
4	32	11	18	4
5	5			
6	13	34	27	6
10	10			
12	26	33	19	12
15	15			
20	20			
25	25			
30	30			

What does this mean? Well, it means that if we know what  $x$  is (say, 3), we can easily compute the result for  $x = 24$ ,  $x = 17$ , and  $x = 31$  using the preceding map. Since 35 in binary is  $100011$ , it is six bits long, and that means that each input to  $f(x)$  (say,  $x = 8$  or in binary:  $001000$ ) will simply map the binary representation of the input to an output according to preceding the map. In this case,  $f(8)$  is equivalent to  $f(001000)$  in binary and the result should be 29 in decimal or  $011101$  in binary.

Our quantum circuit to implement this has to do the same. If the input is  $|001000\rangle$ , it should output  $|011101\rangle$ , and so on for each of the series of numbers. So, to create the correct circuit for  $f(x) = 8x \pmod{35}$ , we need to do the following:

1. Find the groups of repetitions
2. Convert decimal to binary
3. Create an algorithm that changes one binary to the next consistently

Can you spot the pattern to help create the circuit in the binary? I can't. This example is much more involved than factoring  $N = 15$ . These patterns are not apparent in terms of moving from one item in the modular multiplication map to the next. In fact, rather than trying to reinvent the wheel and spot the patterns, we can consult the paper *Constant-Optimized Quantum Circuits for Modular Multiplication and Exponentiation* (Markov and Saeedi 2015) <https://arxiv.org/pdf/1202.6614.pdf>. This details how to make an algorithm that creates modular multiplication circuits of a small size compared to the brute-force technique of just taking the classical computation and translating quantum gates into classical gates. For those who like a challenge, doing this for  $8x \pmod{35}$  is a chapter exercise.

# After the period is found

Once the period is found on the quantum computer, continue with rest of the algorithm on a classical computer:

5.  $r=4$  is even, so we keep going!
6.  $a^{r/2} = 8^{4/2} \pmod{35} = 29$  and  $29 \neq -1 \pmod{35}$ , so we keep going!
7. We've found two factors of 35. They are  $\gcd(a^{r/2} + 1, N) = \gcd(8^{4/2} + 1, 35) = 7$ .

Let's check our work:  $5 * 7 = 35$ . We got it right!

# Summary

This chapter explored Shor's algorithm, a quantum algorithm that is used to find the prime factorization of a number. It goes over what prime factorization is, which is the classical implementation of an algorithm to perform prime factorization compared to the quantum implementation as given by Shor's algorithm. Shor's algorithm has a large classical component, with just one portion, a period finding subroutine that can be run on a quantum computer. The chapter first showed an implementation of this subroutine on a classical computer. We then traced through Shor's algorithm with a variety of different factorizations, step by step, to develop an understanding of how Shor's algorithm functions. Finally, for a specific subcase, this chapter provided a full implementation of the quantum period-finding subroutine, along with its integration into a higher-level function that, given  $N$ , factors  $N$  on a quantum computer. We looked at specific  $N$  examples as well as the skills to extend the algorithm to work for more general cases.

In the next chapter, we will go over ways to avoid errors in our quantum algorithms via quantum error correction algorithms.

# Questions

1. Trace through Shor's algorithm step by step for the case that  $N = 20$  and the first random number chosen is  $a = 9$  in step 1. For each time step 1 must be repeated (if any), execute `random.randint(0,N)` in Python to generate a new random number.
2. Trace through Shor's algorithm step by step for the case that  $N = 7387$  and the first random number chosen is  $a = 4108$  in step 1. For each time step 1 must be repeated (if any), execute `random.randint(0,N)` in Python to generate a new random number.
3. In this longer question, we will finish the implementation of the quantum computing implementation of Shor's algorithm. Hint: figure 3 of the paper *Constant-Optimized Quantum Circuits for Modular Multiplication and Exponentiation* (Markov and Saeedi 2015) <https://arxiv.org/pdf/1202.6614.pdf> contains some circuit diagrams that will help!
  - Create the quantum circuit for  $7x \pmod{15}$ .
  - Create the quantum circuit for  $8x \pmod{15}$ .
  - Create the quantum circuit for  $11x \pmod{15}$ .
  - Create the quantum circuit for  $13x \pmod{15}$ .
  - Create the quantum circuit for  $14x \pmod{15}$ .
  - Use the circuits for  $2 \pmod{15}$  (created earlier in this chapter),  $7 \pmod{15}$ ,  $8 \pmod{15}$ ,  $11 \pmod{15}$ ,  $13 \pmod{15}$ , and  $14 \pmod{15}$  that you created to complete the implementation of Shor's algorithm for factoring 15 using quantum period finding. Hint: there are some resources by IBM and others to help you with this, including the Jupyter notebook *Shor's Algorithm for Integer Factorization* by Dr. Anna Phan <https://github.com/Qiskit/qiskit-tutorials/blob/master/community/>

[algorithms/shor\\_algorithm.ipynb](#), which refers to figure 1 of the paper *Realization of a Scalable Shor Algorithm* (Monz et al. 2015).

4. For those who like a challenge: following the methods outlined in the paper *Constant-Optimized Quantum Circuits for Modular Multiplication and Exponentiation* (Markov and Saeedi 2015) <https://arxiv.org/pdf/1202.6614.pdf>, finish the sketch of the implementation of  $8x \text{ (mod } 35)$  in a quantum circuit that was given earlier in this chapter.

# Quantum Error Correction

This chapter describes the problem of quantum error propagation and illustrates the need for quantum error correction. Quantum error propagation is easily illustrated by applying the identity gate multiple times and noting that eventually the measurement is not as expected. The reader is given examples of how to do this. Next, an overview of **quantum error correction (QEC)** is given, and finally, the chapter provides an implementation of a simple QEC algorithm.

In this chapter, we will cover the following topics:

- Quantum errors
- Quantum error correction

# Quantum errors

Quantum computers interact with the environment through decoherence, resulting in information from a computation degrading over time. This introduces errors into a computation. In [Chapter 3, Quantum States, Quantum Registers, and Measurement](#), we discussed ways to quantify this information loss;  $T_1$  helps quantify how quickly the qubits on a given hardware experience energy loss due to environment interaction (energy loss results in a change in frequency, causing decoherence), which can cause a bit flip, and  $T_2$  helps quantify how quickly the qubits experience a phase difference due to interaction with the environment, again a cause of decoherence. The bigger  $T_1$  and  $T_2$ , the more robust a quantum computation will be to errors.

IBM QX hardware and other quantum computing companies try to raise  $T_1$  and,  $T_2$  but at the present moment, and for the foreseeable future,  $T_1$  and,  $T_2$  are so low that every practical computation is likely to contain errors. These errors can come in the form of a phase difference to that expected from the ideal computation, or of a bit flip to that expected from the ideal computation. Quantum algorithms cannot run effectively without quantum error correction, to compensate for these errors. Quantum error correction functions by spreading out information from a single qubit to multiple qubits, meaning that an algorithm' without quantum error correction' will use many fewer qubits than one with quantum error correction. Since QEC is needed for practical quantum computing in the gate model of quantum computing, this means that the number of qubits needed to implement practical algorithms is much higher than the

minimum number of qubits the algorithm theoretically requires.

# Errors on hardware, demonstrating a bit flip error

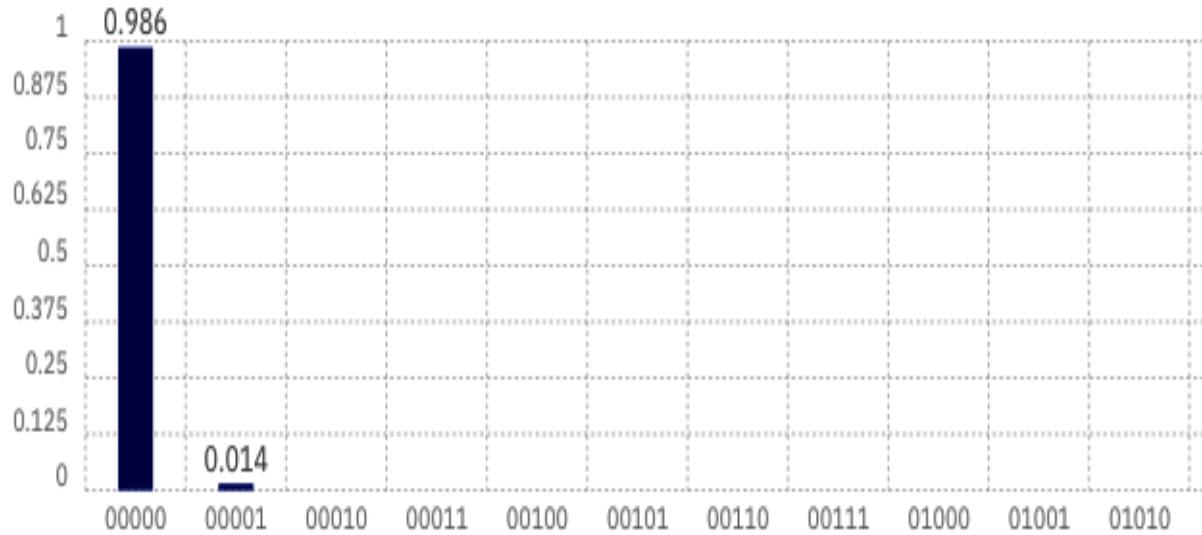
Let's run on the IBM QX. The following is the OpenQASM, as well as the quantum score for running the identity gate 10 times in a row. In this simulator, this would result in exactly the same output as input ( $|0\rangle$ ), but when we run on a device, for a qubit initialized to  $|0\rangle$  and then we run the identity gate, which in the absence of noise should leave the qubit unchanged:

```
include "qelib1.inc";
qreg q[5];
creg c[5];
id q[0];
measure q[0] -> c[0];
```

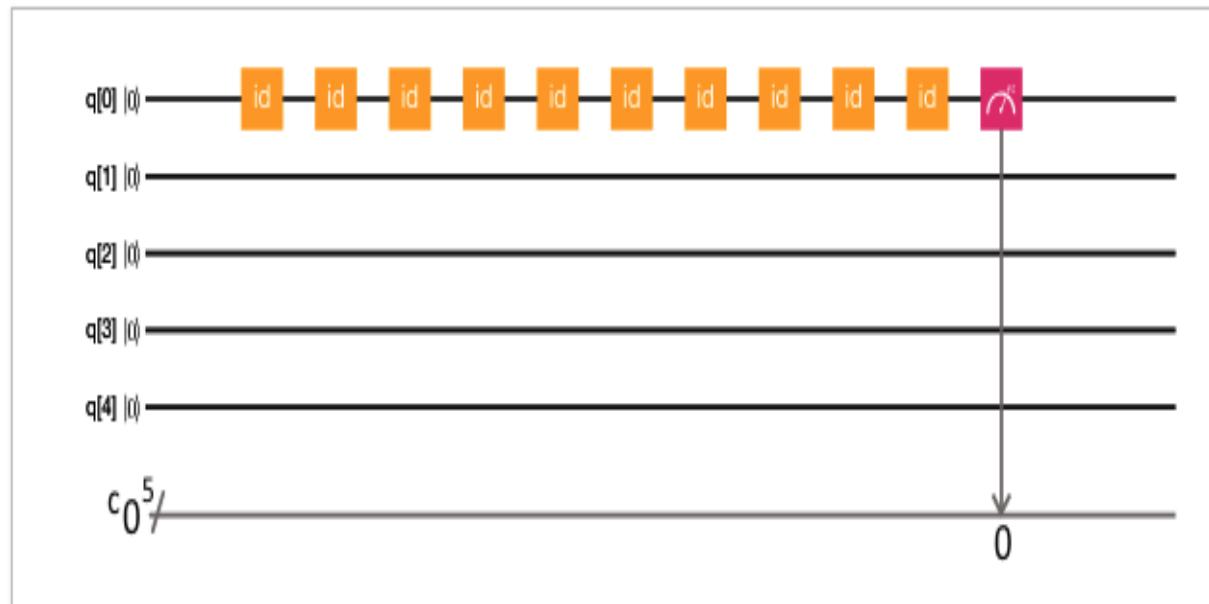
This gives the following on IBM QX:

Device: ibmqx4

## Quantum State: Computation Basis



## Quantum Circuit

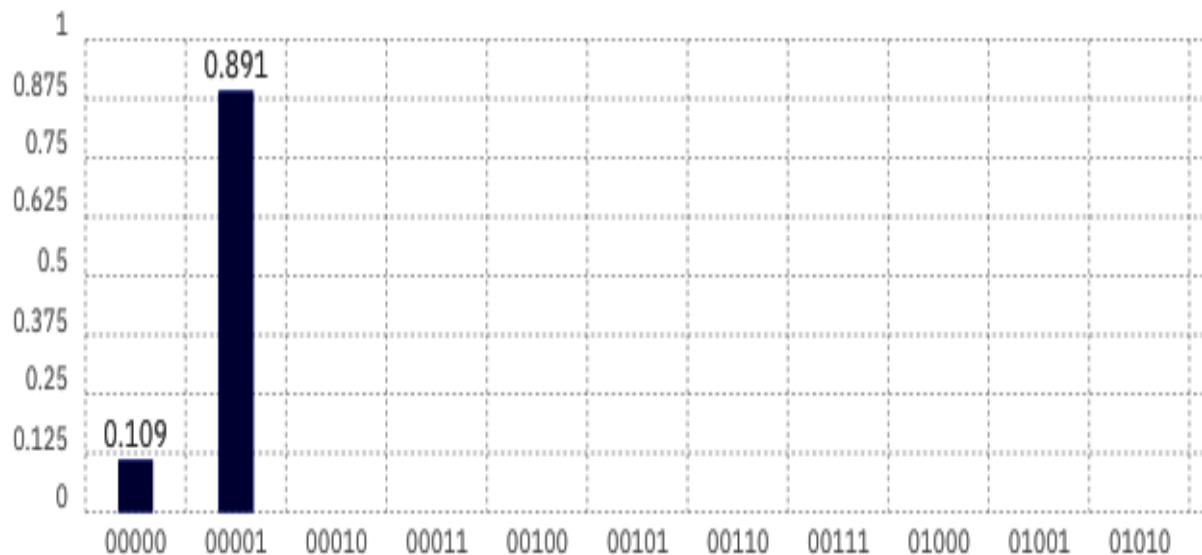


We can see we get  $|0\rangle$  98.6% out of 1024 runs. Then, if we change the first qubit to  $|1\rangle$  by adding  $\times_{q[0]}$ , prior to the

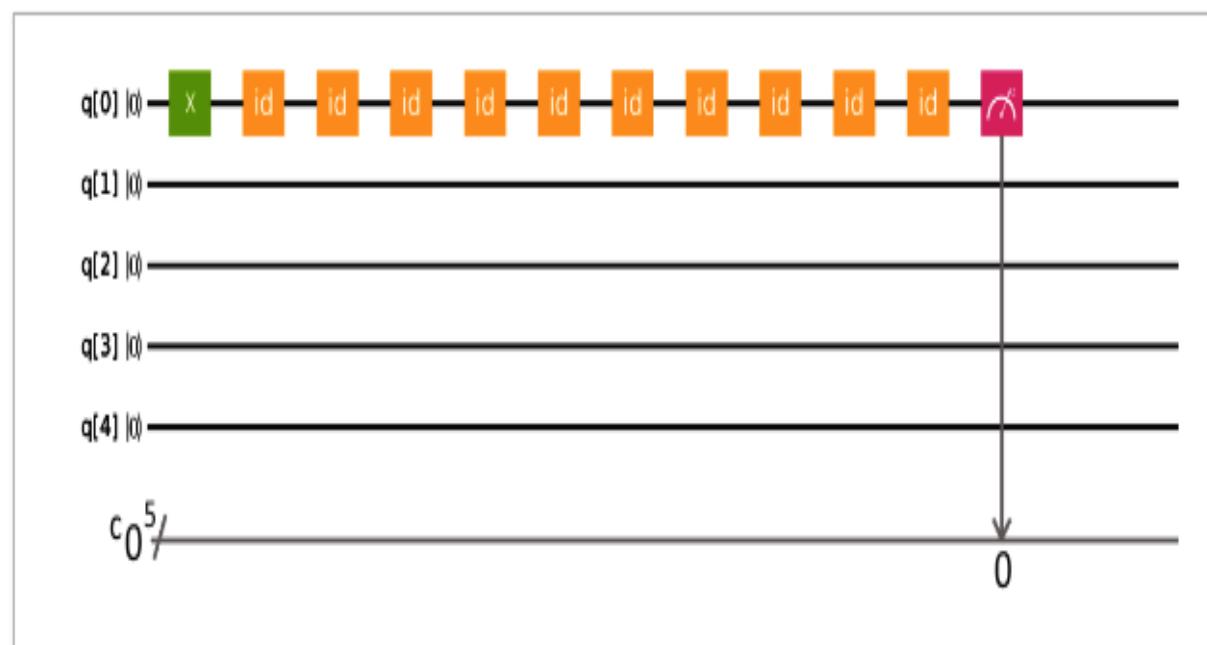
first identity gate, we can see we again get many errors:

Device: ibmqx4

## Quantum State: Computation Basis



## Quantum Circuit



Here, we get the correct result  $|1\rangle$  only 89.1% of the time on this run. Starting out at  $|1\rangle$  has more errors, since  $|1\rangle$  is physically implemented as an excited state, and is more prone to decay to  $|0\rangle$  the longer the computation goes on, than  $|0\rangle$  is to turn into  $|1\rangle$ . The more identity gates that operate, the longer the period of time the computation runs, and the more likely an error will occur.

# Modeling errors in a simulator

In an ideal quantum computing simulation, an  $X$  gate flips the sign of a qubit and a  $Z$  gate flips the phase. So, one way to model errors—qubit flips and phase flips—in an ideal quantum computing simulation, which would otherwise not have them, is to have the error modeling process randomly introduce  $X$  and/or  $Z$  gate(s) to the simulated circuit, that is, to randomly introduce bit flips and/or phase flips. This also is an instructive way to think about error correction: if we could tell just where the bit flip was introduced via an  $X$  gate, we could undo the flip by introducing another  $X$  gate, because the  $X$  gate is its own inverse. Two applications of the  $X$  gate would get us back to the original bit. Likewise with the  $Z$  gate. To undo a phase-flip introduced via a  $Z$  gate, we can simply apply another  $Z$  gate.

Qiskit Aer offers the possibility of simulating noise as well. We can choose a real-world IBM QX device, extract its properties, generate a noise model, and then run the simulation. Let's try it out with our repeated identity gate code from earlier. The following code will allow us to run it in the local simulator or on remote hardware, starting from either `"/0">` or `"/1">` and choosing how many identity gates to apply. If we run it on the local simulator, we can then choose whether that simulator should be noisy or not:

```
import time
from qiskit import Aer
from qiskit.tools.visualization import plot_histogram, plot_state_city
from qiskit import compile, execute
from qiskit.tools.monitor import job_monitor

def apply_identity_gate(qubit_val,apply_times=10,noisy=True,simulator=True):
```

```

shots=1000
qr = qiskit.QuantumRegister(1)
cr = qiskit.ClassicalRegister(1)
qc = qiskit.QuantumCircuit(qr, cr)
if qubit_val not in [0,1]:
    raise Exception("initial qubit must be either 0 or 1")
if qubit_val==1:
    # Setting q0=|"1">
    qc.x(qr[0])
# Applying the identity gate 10 times.
for i in range(apply_times):
    qc.iden(qr[0])
# Measuring the result. If our hardware was perfect,
# it should always yield the same value as qubit_val
qc.measure(qr[0],cr[0])

if simulator:
    backend = Aer.get_backend('qasm_simulator') # Local simulator
    if noisy:
        device = IBMQ.get_backend('ibmqx4')
        properties = device.properties()
        coupling_map = device.configuration().coupling_map
        noise_model =
            noise.device.basic_device_noise_model(properties)
        basis_gates = noise_model.basis_gates
        exp_job = execute(qc, backend,
                           coupling_map=coupling_map,
                           noise_model=noise_model,
                           basis_gates=basis_gates)
    else:
        exp_job = execute(qc,backend,shots=shots)
else:
    if noisy:
        # Preparing to run
        backend = IBMQ.backends(name='ibmqx4')[0] # remote hardware
        exp_job = execute(qc,backend,shots=shots)
    else:
        raise Exception("Hardware is always noisy,
                        to use hardware keep noise=True")
job_monitor(exp_job)
exp_result = exp_job.result()
final=exp_result.get_counts(qc)
print(final)
plot_histogram(final)

```

# Quantum error correction

Since quantum computers have errors, and  $T_1$  and,  $T_2$  are not likely to get large enough to prevent these errors from being introduced in the course of a computation, we will have to find some way to correct these errors if we are to use the quantum computers of today to produce accurate results. The good news is that we can add qubits to a quantum circuit and use these qubits to correct the errors introduced by decoherence running on the original circuit.

*"There are of course many peripheral results in experimental physics, which have enabled progress in building devices with several qubits. However, those are also of interest outside of quantum computing. Within quantum computing proper, the most important result is the fault tolerant threshold theorem. Loosely speaking, this is the proof that quantum computing is even possible!"*

- Aaron VanDevender, PhD; Chief Scientist, Founders Fund

If we use qubits to correct errors in other qubits, the error correcting qubits are themselves subject to errors. We might be worried that the errors might accumulate, resulting in there being no way to effectively error correct a quantum computation in general, but luckily this is not the case. An important concept in quantum computing is the **threshold theorem**, which shows that with enough error correcting qubits, we can always create an accurate computation, so long as the error in the gates is below a particular threshold. This threshold varies depending on how the error in the gates is measured as well as the choice of universal gates, but the important thing is that the theorem shows that it is possible to correct errors faster than creating them under the right conditions. The threshold theorem shows that, although the error correcting qubits are themselves subject to quantum error, they too can be error corrected with other qubits, and their error correcting qubits can themselves be error corrected, and so on (in logarithmic levels). Since each

level of error correction requires fewer error correcting qubits than the original number of qubits in the computation, eventually we reach a point where all the levels of qubits work together to sufficiently error correct the original qubits.

Classical error correction involves duplication of information so that if part of it is corrupted by noise, the true value can be recovered. Quantum error correction can't rely on that strategy as qubits can't be copied. One strategy for quantum error correction, which we will outline in this chapter, involves spreading out the quantum information from one qubit to multiple qubits, and then performing measurements that can diagnose a potential error without affecting the quantum state. Apart from a few simple cases, a quantum error correction code will not restore exactly the original state. Instead, the quantum error correction code increases the probability of obtaining the correct result.

# **Single bit flip error correction**

This subsection examines how to error correct bit flips.

# **Classic error correction bit flip**

Imagine we want to error correct a bit. We can simply copy a bit several times, and then, as long as not too many bits in the copied group experience noise causing a bit flip, the true value of the bit can be just a majority vote. For example, to error correct the bit 1, we could copy it three times, then, as long as only one bit experiences a bit flip, for example the middle bit, the bits would read 101 and the result would be the majority vote, 1, which is just what we want. In this way, by copying the bit, we become less sensitive to environmental noise because as long as the noise isn't too great (in this example, large enough to flip more than one bit in the group of three with high probability), we can recover the correct result in a noisy environment.

# Quantum error correction single qubit flip

As outlined in [Chapter 3](#), *Quantum States, Quantum Registers, and Measurement*, there is no way to clone quantum information; that means we can't simply copy a qubit. So, the copy-the-qubit-multiple-times strategy won't work.

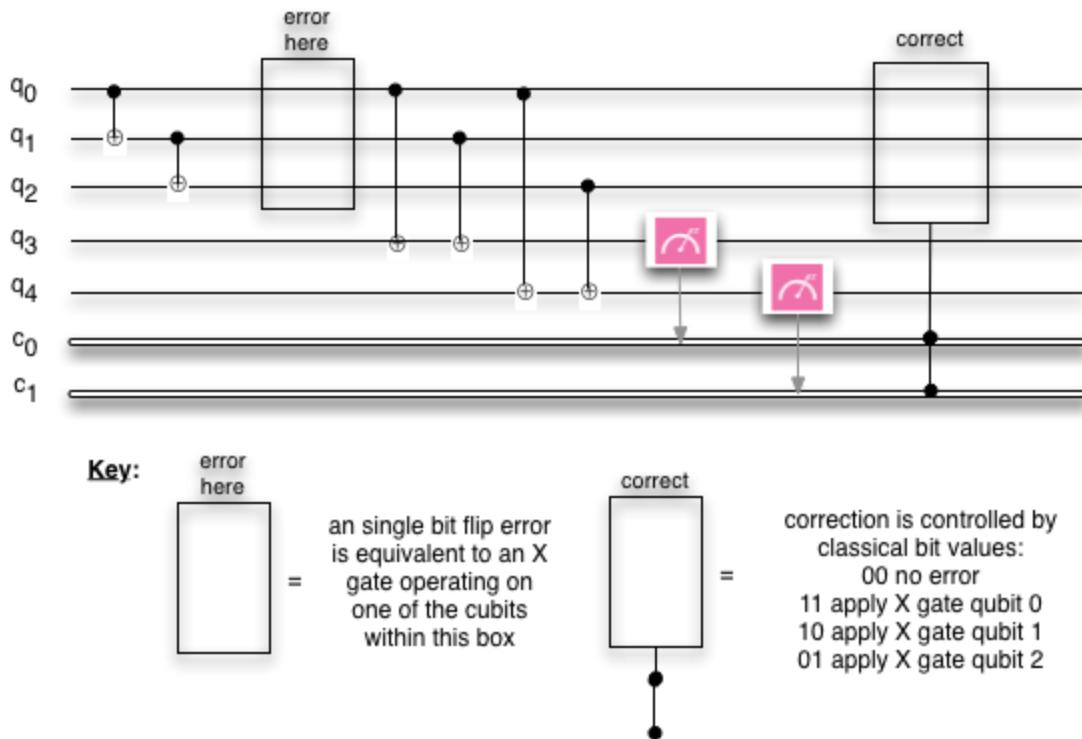
However, if we start out with a  $|0\rangle$  qubit, we can spread out quantum information associated with that qubit to other qubits, via, for example, a set of CNOT gates with other qubits entangling the system. The idea is to create, instead of a single-qubit state, which can be measured to be either  $|0\rangle$  or  $|1\rangle$ , a three-qubit state that can be measured to either  $|000\rangle$  or  $|111\rangle$ .

If a state that is either  $|000\rangle$  or  $|111\rangle$  is affected by noise, a bit can be flipped. How can we detect this? Again, we have to take into account the unique nature of quantum information as opposed to classic information. In general, our error correcting code should work not only if the qubits are 100%  $|000\rangle$  or 100%  $|111\rangle$ , but also if they are some superposition between the two. We can't simply measure the state, as we will then destroy information about it before we can correct it; we won't be able to tell whether a mismatch in the three qubits is because of the probabilistic measurement over a superposition state, or whether it is because of an error if we simply measure the state. So, we have to detect and then correct the error before we measure it. How do we do this?

The key to detecting and correcting the error before we measure the state is to spread out the information over additional helper qubits in a way that measuring those

additional qubits will reveal whether there is bit flip error, and, if so, which qubit the error occurred on. We can do this in such a way that it doesn't disturb the original state because the helper qubits reveal information about qubits that are different from another given qubit, without giving information about what the actual states of those qubits are.

The circuit diagram for one such bit flip code is as follows, as detailed in *Quantum Error Correction for Beginners*, Figure 3 (Devitt et al., 2013):



In this circuit, we expand the initial qubit **q<sub>0</sub>** into three qubits. This expansion is done through two CNOT gates and it will help us correct any errors that occur later. Then, in the box labelled **error here**, an error can occur. Here we assume the error is a single bit flip error. This single bit flip error can come from hardware, on a real machine, or be simulated by adding in an X gate on one of the qubits in the

box: **q<sub>0</sub>**, **q<sub>1</sub>**, or **q<sub>2</sub>**. Next there are a series of four CNOT gates, followed by two measurements. These measurements are what we will use to diagnose whether an error occurred in the **error here** section of the circuit, and if so, on which qubit. Finally the **correct** box is conditioned on the two classical bits we just measured, and based upon those values applies a quantum X gate to the correct qubit to undo the bit flip error. With this circuit, we could put an X gate on any individual qubit in the **error here** box and the rest of the circuit would automatically put an X gate on that same qubit in the **correct** box, meaning, because two X gates in a row flip a qubit then flip it back, there would be no effect of the error on the original qubits.

The following OpenQASM implements that circuit:

1. First, we initialize the circuit:

```
|     include "qelib1.inc";
|     qreg qr[5];
|     creg result[3];
```

2. The following will serve as the bits that will help us figure out whether we have errors:

```
|     creg er[2];
```

3. The first qubit is `/"0"/`, unless you uncomment the following line to initialize to `/"1"/`:

```
|     //x qr[0];
```

4. If want if the first qubit was `/"0"/`, we want now to have `/"000"/`, and if the first qubit was `/"1"/` to now have `/"111"/`. To do this, we entangle the original qubit across three qubits:

```
|     cx qr[0], qr[1];
|     cx qr[1], qr[2];
```

Here, we model an error by uncommenting just one of the following suggestions (as, if you uncomment more than **one**, the error correction is ambiguous):

- Uncomment the following to model a single bit flip error by flipping the zero position qubit:

```
| // x qr[0];
```

Under option 1, if our initial state was  $|0\rangle$  after the error, we now have  $|100\rangle$  because of the error on the zero position qubit, while, if our initial state was  $|1\rangle$ , we now have  $|011\rangle$  because of the error on the zero position qubit

- Uncomment the following to model a single bit flip error by flipping the first qubit:

```
| //x qr[1];
```

Under the second option, if our initial state was  $|0\rangle$  after the error, we now have  $|010\rangle$  because of the error on the first qubit. If our initial state was  $|1\rangle$ , we now have  $|101\rangle$  because of the error on the first qubit.

- Uncomment the following to model a single bit flip error by flipping the second qubit:

```
| //x qr[2];
```

Under the third option, if our initial state was  $|0\rangle$  after the error, we now have  $|001\rangle$  because of the error on the second qubit. If our initial state was  $|1\rangle$ , we now have  $|110\rangle$  because of the error on the second qubit.

Now, our goal is to recover the original state, prior to the error we just introduced. To do this, we entangle with a few extra qubits (third and fourth qubits):

```
| cx qr[0],qr[3];
| cx qr[1],qr[3];
| cx qr[0],qr[4];
| cx qr[2],qr[4];
```

Now, we can make measurements of the third and fourth qubits to get information about our three qubits without disturbing them:

```
| measure qr[3]->er[1];
| measure qr[4]->er[0];
```

We can use these measurements, as detailed in table 1 of *Quantum Error Correction for Beginners* (Devitt et al., 2013), to detect which qubit the bit flip error occurred on, and to apply an X gate on that qubit to undo the bit flip error.

If the bits are 11 ( $_3$  in decimal), the zero position qubit has the error, so flip it back:

```
| if(er==3)
| x qr[0];
```

If the bits are 10 ( $_2$  in decimal), the first qubit has the error, so flip it back:

```
| if(er==2)
| x qr[1];
```

If the bits are 01 ( $_1$  in decimal), the second qubit has the error, so flip it back:

```
| if(er==1)
| x qr[2];
```

If the bits are 00, do nothing, as there were no errors.

Since we have detected and corrected any single qubit errors, we can measure our results, which should either be 111 in the case where our original qubit was in state  $|1\rangle$ , or 000 in the case where the original qubit was in the state  $|0\rangle$ :

```
|measure qr[0]->result[0];
|measure qr[1]->result[1];
|measure qr[2]->result[2];
```

If we place that QASM in a string named `qasm_string`, it can then be run with the following code:

```
from qiskit import Aer
from qiskit.tools.visualization import plot_histogram
backend = Aer.get_backend('qasm_simulator')
qc = QuantumCircuit.from_qasm_str(qasm_string)
exp_job = execute(qc,backend,shots=1000)
job_monitor(exp_job)
exp_result = exp_job.result()
final=exp_result.get_counts(qc)
print(final)
|plot_histogram(final)
```

# Quantum error correction single phase flip

With classic computers, all we have to worry about error correcting is the bit flip, but with quantum computers there is another type of flip that is possible: the phase flip. This flip is the equivalent of the Z gate acting upon the state. So the states  $|0\rangle$  and  $|1\rangle$  would be unchanged by a phase flip, while a state  $|+\rangle$  or a state  $|-\rangle$  would flip to the opposite state. The good news is that to correct this error, we can use a very similar circuit to the bit flip code. Recall that the H gate changes the  $|0\rangle$  state into the  $|+\rangle$  state. Any phase error on a single qubit would change the  $|+\rangle$  state into the  $|-\rangle$  state. Finally, if we apply another H gate, we would get the  $|1\rangle$  state after this error. So, if we surround the portion in which a phase error is potentially introduced by H gates, we can transform on either end the phase error into a bit flip error and use the same bit flip code to detect the error. Then, to correct for the error, we apply the Z gate. Now that we can address both single bit flip errors and single phase flip errors, we are ready to address any single qubit error.

# **The Shor code - single bit and/or phase flip**

We can address any error process that introduces an error on a single qubit that can introduce a bit flip, a phase flip, or both a bit flip and a phase flip on that qubit. The error correction algorithms we have gone over so far can correct for one or the other, but not both at once. To correct a single qubit error that is potentially both a phase and a qubit flip, we will need to spread out the quantum information over an even larger number of qubits. The Shor code is one algorithm that can correct bit and/or phase flips on a single qubit by first spreading out the quantum information over nine qubits, and then performing measurements that reveal information about whether those qubits match or differ in phase and bit, allowing the algorithm to diagnose and to correct both types of errors.

# Summary

Quantum error correction is critical for practical quantum computing applications in the gate model of quantum computing, as errors from decoherence will always interfere with a quantum computation. Quantum error correction works by spreading out information on a single qubit over multiple qubits, and performing measurements that can diagnose the type and location of an error and correct it, without disturbing the information on the original qubit necessary to obtain an accurate result. The threshold theorem shows that, as long as the probability of an error on a qubit is below a certain threshold, it is possible to error correct effectively. Since quantum error correction works by using more qubits to help detect and correct errors than a given original algorithm, quantum error correction will always require more qubits than the minimum necessary for an algorithm. This means, effectively, that for a quantum computing system to become practical, we likewise need hardware that supports many more qubits than we would if error correction were not necessary.

# Questions

1. Draw the circuit for the phase flip error correction code.
2. Implement the phase flip error correction code.
3. What happens in the bit flip error correction code if we had, instead of a single qubit error, an error in two qubits?
4. For the bit flip error correction code, instead of simulating noise ourselves by introducing an X gate, run on a quantum simulator with noise included. Does the error correction code function well in this environment?
5. At the time of writing this book, the IBM QX devices did not support if statements conditioned on classical bits. However, we can still perform measurements to diagnose the types and locations of errors on the actual hardware, even if we can't implement the if statements that will allow us to correct those errors. Take out the if statements in the bit flip error correction code and run it on hardware. How often did a bit flip error on the second qubit occur during that run?

# Conclusion - The Future of Quantum Computing

The final chapter reviews what the reader has learned and places it in the context of areas where quantum computing is poised to disrupt, and those understanding quantum computation and quantum programming are likely to be in high demand. These areas include, among others, computer security, materials science, machine learning and artificial intelligence, and chemistry. These changes will have an impact on medicine, alternative energy, global finance, and the global economy. A review of the current state of quantum computing, as well as the projected state over the next few decades, is provided alongside some predictions as to when the industries examined might experience disruption. In addition, the book discusses why an executive, a programmer, or a technically minded person who doesn't happen to be a quantum physicist might care about quantum computing.

*"With quantum computing, we don't know exactly how far away it is to do a commercially useful thing. To me, quantum computing right now is just a strategic asset. It's uncertain when there's going to be like meaningful revenue that comes from solving real pain points of industry. I would not invest in quantum computing if I was focused on the near term revenue of my investment. I think you kind of need to be investing from a strategic lens believing that this is going to be a massive tectonic shift in one of the biggest industries in the world, the computing industry and that the companies around right now, are going to accrue a lot of the strategic value in that going forward.*

- Dr. Shaun Maguire, Partner, Google Ventures

Imagine if you could be there at the dawn of the computing revolution. When would that be? Would it be during the development of the first computing devices, in prehistory? Would it be during Ada Lovelace's day in the 1840s, when she wrote the first computer program for Charles Babbage's theoretical analytical engine? Or would it be during the days

of the first analog computers? How about during the development of the transistor? Or the mainframe or the personal computer? With what enthusiasm would you greet a language like assembly after programming directly with registers? Or C and Fortran after programming with assembly? With what enthusiasm would you greet a high-level language such as Fortran after programming in assembly? Would you wait until the development of higher-level programming languages such as Java or Python? Or wait all the way until the development of the internet to make your mark?

The history of computing is long and varied. The C language has been around since 1972, and the earliest version of Fortran goes back to 1957. An engineer in the days of their first development might be able to predict the advances that the computing revolution would bring to society, or they might be lost in the weeds, unable to predict the ultimate benefit of computation. The development of quantum computing that we are in the midst of renders us unable to see clearly our place on the progress curve.

*"I think quantum computing is important because it represents a fundamentally new type of computation. When we first built transistors, a whole new world and scale of computation was opened. Building quantum computers provides not only a new hardware resource, but a new computing model itself, which is both extremely exciting and important for humanity."*

*- Dr. Sarah Kaiser, Research Engineer at Pensar Development.*

During the course of this book, I referred to the computers we are used to-based upon silicon and transistors-as classic computers. The "classic" is necessary to contrast these computers with the quantum computers that are the subject of this book.

Quantum computing is a new paradigm of computation that holds the promise to be equally as revolutionary as classic transistor-based computing. Using the unique properties of quantum mechanics, a quantum computer can perform

certain computations in a different manner to classic computers, and offer the possibility of being able to perform those computations with such efficiency that while current machines wouldn't be able to compute a quantity in our lifetimes, a quantum computer could do so in a matter of days. Quantum mechanics applies to all things, but during our everyday experience, the effects of quantum mechanics are mostly so subtle as not to be noticeable. To see the effects of quantum mechanics and to harness them for computation, we need to go to temperature ranges (very cold) and scales (very small) outside of our everyday experience. To harness the properties of quantum mechanics, quantum computers must carefully engineer the physics of devices that are very cold and very small, and it is this engineering challenge that holds up the progress of practical quantum computers.

The classic computing revolution, whether you decide it started thousands of years ago, more than a hundred years ago, or decades ago, didn't happen overnight. Neither will the quantum computing revolution.

*"The only way to progress technology in a groundbreaking way is to develop hardware that is based on new physics phenomena - this is what brought us the transistor. Quantum computing is a way to harness the power of quantum mechanics in a way that can speed up computing, one of the most powerful and influential technologies of our time."*

*- Dr. Guen Prawiroatmodjo, software engineer and quantum physicist*

At this stage, it is unclear where we lie on the maturity curve of the technology. There are practical quantum computers, but each is more of a prototype than a mature technology. The more mature devices are special purpose, built to solve only a certain set of problems such as those related to optimization. The more general devices are less mature, and will require many engineering advances to become practical. Advances need to be made in creating qubits with longer coherence times and in scaling up the

number of qubits, for example. How far away is the day of general purpose quantum computing? We don't know.

In the introduction to the book, I gave some perspective on the history of quantum computing, why it could be important, and some perspectives from experts as to what might happen to the field in the next 5 to 10 years, the lifetime of this book. In this final section of the book, I will review what we learned in the book, and then share some final thoughts from experts as to why we might be pessimistic about the field, and counter these with thoughts that inspire optimism as well as an outlook for the field in the next 50 to 100 years.

# Key concepts of quantum computing

Some key concepts under quantum computing are described as follows:

- **Quantum computers:** These exploit the properties of quantum physics, superposition and entanglement, to perform some computations much more efficiently than on a classic computer.
- **Qubit:** This is a quantum piece of information used in quantum computing that represents one possible combination of two values. Manipulating qubits of information is what powers many modern quantum computers, including IBM QX.
- **Quantum state:** A quantum state is a grouping of one or more qubits.
- **Entanglement:** When qubits that compose a quantum state are related, they are entangled. Entanglement is key to the power of quantum computing.
- **Coherence:** Coherence is the property of waves, and if waves are coherent they can in some senses "work together." Qubits interact with the environment and this process causes the correlations they had due to coherence to be lost, which hinders the ability to perform quantum computation. The longer the coherence time of a quantum computer, the better it will function.
- **Quantum gate:** In the gate model of quantum computing, taking an input quantum state and transforming it in the desired way is done with quantum gates. The right set of gates can perform any quantum

computation. In IBM QX, these are called the I, X, Y, Z, H, S,  $S^\dagger$ , T,  $T^\dagger$  and CNOT gates.

- **Quantum circuit:** This is a quantum circuit is a sequence of one or more quantum gates grouped together.
- **Quantum circuit diagram:** A method to visualize quantum circuits. The IBM QX Quantum Composer is a way to program IBM QX using the quantum circuit visualization.
- **IBM QX programming:** Can be programmed via the quantum composer, a way of visually representing quantum circuits. Can be programmed via a Python API interacting with the computer and simulation tools via the Python `qiskit` module. Can be programmed with a general purpose quantum programming language called OpenQASM.
- **Quantum algorithms:** Grover's algorithm works to efficiently find the one solution to a function out of all possible inputs. Shor's algorithm factors numbers. The Quantum Fourier Transform is a useful subroutine of many quantum algorithms, including Shor's algorithm.
- **Quantum supremacy:** This is when a quantum computing algorithm offers significant speedup compared to the best known algorithm on a classic computer, and this speedup is demonstrated on a quantum computer. As of the writing of this book, many algorithms offer the promise of quantum supremacy, but the hardware of quantum computing lags behind the theory and quantum supremacy has yet to be clearly demonstrated.

# **Fields where quantum computing will be useful**

In the near term, experts disagree as to which field will have the first "killer app" of quantum computing. Quantum computing will be useful to fuel breakthroughs in a variety of fields. These include, among others:

- Chemistry
- Physics
- Material science
- Artificial intelligence
- Optimization problems
- Cryptography

# Pessimism about the quantum computing field

This book has focused on quantum computing in the gate model, and it has assumed that this knowledge will be useful. However, this isn't guaranteed to be the case. Experts worry about the following:

- The quantum computing field is over-hyped
- The lack of human diversity in quantum computing experts will harm progress
- Our progress will be hindered by a lack of fundamental understanding of quantum computing
- The pace of progress thus far in quantum computing will continue, meaning important developments are a long way off

# **Optimism about the quantum computing field**

Experts agree that, while there are reasons to think that making progress in quantum computing will not necessarily be easy, there are reasons to be optimistic. Some of these reasons include the following:

- There are many smart people working on quantum computing
- Quantum computing has a push to be transparent, which will allow researchers to easily collaborate
- Recent progress has been good, leading us to hope progress will continue
- It's an exciting field
- Prototype applications for quantum computers exist, on IBM QX and other quantum computers

Although it is a general concern that the field is overhyped, some experts are optimistic that the field in general is so close to being useful that the hype is warranted.

# **Concluding thoughts on quantum computing**

Over the course of the book, we began to implement our own quantum programs, understand important quantum algorithms, and develop enough of a fluency with quantum computation that, should these computers become practical, the readers of this book will be well poised to take their place in the action: the practical development of a brand new field of computing that will touch us all.

# Appendix

This section is a mathematical appendix for those wanting greater detail, or who consider greater detail to be necessary, in order to understand the book material. This section is optional reading in the context of the book, but a deeper understanding of the mathematics powering quantum computation is necessary for effective quantum algorithm development. The chapter gives a brief overview of the subset of linear algebra needed to mathematically follow ideal quantum computation. This includes matrices and matrix multiplication, as well as the Kronecker product. The chapter is necessarily brief as a detailed mathematical and physical introduction to quantum computation is outside the scope of the work. The reader is instructed to consult the canonical textbook *Quantum Computation and Quantum Information* (Nielsen and Chuang, 2000) for in-depth details, particularly regarding the more precise mathematical specifications as well as the physics described by the mathematics sketched in this chapter.

# **Useful mathematics**

This section mentions just a few bits of notation and items that might be useful to follow the mathematics behind the Python provided in the text. It is not meant to be an exhaustive list or a comprehensive description, rather a springboard to give terminology that might inspire further study in mathematics.

# Summation

The sigma symbol indicates a summation over a function of a variable initialized to the value on the lower portion of the

sigma to the value on the upper . So, for example,  $\sum_{i=0}^{10} 2i$  is equivalent to  $2*0+2*1+2*2+2*3+2*4+2*5+2*6+2*7+2*8+2*9+2*10$  or this:

```
sum=0
for i in range(11):
    sum+=2*i
print(sum)
```

The answer prints  $110$ . If the value the variable is initialized to isn't specified, you can assume it is  $0$ .

# Complex numbers

The math of complex numbers is integral to quantum mechanics and quantum computing.

Think of a complex number as providing coordinates on a plane: on the  $x$  axis (referred to as the real axis) we have positive and negative real numbers and on the  $y$  axis (referred to as the imaginary axis) we have positive and negative multiples of the imaginary number  $i$  where  $i$  is defined to be the square root of negative 1 (since no real number has this property, we refer to  $i$  as imaginary). Since  $i$  is the square root of -1,  $i^2 = -1$ . Any point on the plane can then be specified in the form  $a + bi$ , where  $a$  and  $b$  are both real numbers.  $a$  is referred to as the real part and  $b$  is referred to as the imaginary part.

# Complex conjugation

The complex conjugate of an imaginary number  $a + bi$  is a number with the same real part, but the imaginary part is opposite in sign. So, for example, the complex conjugate of  $6 + 27i$  is  $6 - 27i$ .

# **Linear algebra**

This section mentions just a few bits of the field of mathematics called linear algebra; the primary branch of mathematics behind the Python provided in the text. It is not meant to be an exhaustive list or a comprehensive description, rather a springboard to give terminology that might inspire further study in linear algebra.

# Matrix

A matrix is a set of numbers organized in a two dimensional array, with some additional properties; namely, numbers organized in the matrix format can be multiplied, manipulated, and combined according to the rules of linear

algebra. For example,  $M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$  is a matrix. A matrix's size is given in rows x columns, so our example matrix M is of size 3 x 2. Entries in the matrix are indexed according to their row and column starting from 1. So, for example,  $M_{21}$  is 3 and  $M_{12}$  is 2. Matrices can also be composed of just a single

column or just a single row, for example, the matrix  $K = \begin{pmatrix} 1 \\ 3 \\ 5 \end{pmatrix}$  or the matrix  $B = (1 \ 3 \ 5)$ . To specify a matrix in Python, we use the `numpy` module. To specify our matrix  $M$ , we can use the following Python code:

```
|import numpy as np  
|print(np.matrix('1 2; 3 4; 5 6'))
```

This outputs the following:

```
|matrix([[1, 2],  
|        [3, 4],  
|        [5, 6]])
```

# Matrix multiplication

In this subsection we'll go over matrix multiplication. First, we'll go over the case of a constant multiplied by a matrix. Then we'll go over how to multiply two matrices together.

# A constant multiplied by a matrix

A constant multiplied by a matrix just means each element of the matrix is multiplied by that constant. So, for example consider the following matrix:

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

If we multiply then it would give us the following:

$$2M = 2 \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} = \begin{pmatrix} 2 & 4 \\ 6 & 8 \\ 10 & 12 \end{pmatrix}.$$

To do this in Python, simply use the `*` symbol:

```
import numpy as np
M=np.matrix('1 2; 3 4; 5 6')
print(2*M)
```

With the preceding code, we get the printed result of  $M$  multiplied by the constant  $2$ , which is the following:

```
matrix([[ 2  4]
       [ 6  8]
       [10 12]])
```

# Two matrices multiplied together

With matrices, unlike with numbers, the order of multiplication matters. Two matrices can be multiplied only if the first (left) matrix's number of columns is equal to the second (right) matrix's number of rows. So, for example, the

matrix  $M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$ , which is  $3 \times 2$ , can only be multiplied with matrices that have two rows, but those matrices can have any number of columns. The final answer of the multiplication will have the same number of rows as the first (left) matrix and the same number of columns as the second (right) matrix. So if we multiply  $M$ , which is  $3$  by  $2$ , by a matrix  $N$ , which is  $2$  by  $21$ , the final answer  $M * N$  will be a matrix which is  $3$  by  $21$ , that is, a matrix that has three rows and  $21$  columns. We can't multiple  $N * M$ , as  $N$  has  $21$  columns and  $M$  has two rows, and these two numbers would need to be equal.

Each entry of the result matrix is composed by taking the product of the corresponding row in the first matrix and corresponding column in the second matrix. To take this product, multiply the first item of the row together with the first item of the column, and add the the second item of the row together with the second item of the column, and keep summing until we run out of items to get the final answer.

So, for example, consider the matrix  $P = \begin{pmatrix} 7 & 8 \\ 9 & 10 \end{pmatrix}$ , which is  $2 \times 2$ . The final result of  $M * P$  will be  $3 \times 2$ , which written out entry by entry is the following:

$$M \cdot P = \begin{pmatrix} (M \text{ row 1}) \cdot (P \text{ column 1}) & (M \text{ row 1}) \cdot (P \text{ column 2}) \\ (M \text{ row 2}) \cdot (P \text{ column 1}) & (M \text{ row 2}) \cdot (P \text{ column 2}) \\ (M \text{ row 3}) \cdot (P \text{ column 1}) & (M \text{ row 3}) \cdot (P \text{ column 2}) \end{pmatrix}$$

Let's compute each entry:

- $(M \text{ row 1}) \cdot (P \text{ column 1})$ , would be (1 2) times (7 9) or  $1*7+2*9=25$
- $(M \text{ row 1}) \cdot (P \text{ column 2})$ , would be (1 2) times (8 10) or  $1*8+2*10=28$
- $(M \text{ row 2}) \cdot (P \text{ column 1})$ , would be (3 4) times (7 9) or  $3*7+4*9=57$
- $(M \text{ row 2}) \cdot (P \text{ column 2})$ , would be (3 4) times (8 10) or  $3*8+4*10=64$
- $(M \text{ row 3}) \cdot (P \text{ column 1})$ , would be (5 6) times (7 9) or  $4*7+6*9=89$
- $(M \text{ row 3}) \cdot (P \text{ column 2})$ , would be (5 6) times (8 10) or  $5*8+6*10=100$

So, our final matrix is this:

$$M \cdot P = \begin{pmatrix} 25 & 28 \\ 57 & 64 \\ 89 & 100 \end{pmatrix}$$

Matrices in Python, once specified using the `numpy` module, can be multiplied with the `*` symbol.

Thus, we have this:

```
import numpy as np
M=np.matrix('1 2; 3 4; 5 6')
P=np.matrix('7 8; 9 10')
print(M*P)
```

Which prints the following:

```
matrix([[ 25,  28],
       [ 57,  64],
       [ 89, 100]])
```

# Conjugate matrix

The conjugate of a matrix is the matrix we get from taking the complex conjugate of each element of the original matrix. The symbol for a conjugate matrix is a bar over top of the name of the original matrix.

So, for example, if we have a matrix as follows:

$$Q = \begin{pmatrix} 1+i & 2+6i \\ 3+9i & 4-3i \\ 5-2i & 6 \end{pmatrix}$$

Its conjugate matrix is the following:

$$\bar{Q} = \begin{pmatrix} 1+i & 2+6i \\ 3+9i & 4-3i \\ 5-2i & 6 \end{pmatrix}$$

To take the conjugate matrix in Python, we can do the following (note that Python calls the complex number *i* instead *j* as is customary in some engineering texts, and in addition the *j* can never be used alone to differentiate it from a user defined variable; it must always have a float or integer prefix):

```
import numpy as np
Q=numpy.matrix([[1+1j,2+6j],[3+9j,4-3j],[5-2j,6]])
print(Q.conjugate())
```

# Matrix transpose

The transpose of a matrix is formed by turning rows into columns and columns into rows.

The symbol for the transpose of a matrix is a superscript T on top of the original matrix. So, for example, recalling our example matrix M:

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix}$$

Its transpose would be:

$$M^T = \begin{pmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{pmatrix}$$

To take the transpose of a matrix in Python, we can do the following:

```
|import numpy as np  
|M=np.matrix('1 2; 3 4; 5 6')  
|print(M.transpose())
```

# Matrix conjugate transpose

The conjugate transpose of a matrix is when we take the matrix conjugation and the transpose.

The symbol for the conjugate transpose is a superscript \* on top of the original matrix. Recalling our example matrix  $Q$ :

$$Q = \begin{pmatrix} 1+i & 2+6i \\ 3+9i & 4-3i \\ 5-2i & 6 \end{pmatrix}$$

Its transpose is the following:

$$Q^T = \begin{pmatrix} 1-i & 3-9i & 5+2i \\ 2-6i & 4+3i & 6 \end{pmatrix}$$

```
import numpy as np
Q=numpy.matrix([[1+1j,2+6j],[3+9j,4-3j],[5-2j,6]])
print(Q.getH())
```

Which prints the following:

```
matrix([[1.-1.j, 3.-9.j, 5.+2.j],
       [2.-6.j, 4.+3.j, 6.-0.j]])
```

# Matrix Kronecker product

There is another type of matrix product besides multiplication. This is known as the Kronecker product. Matrix multiplication is written with a \* or a dot ., whereas the Kronecker product is written with a  $\otimes$ . There is no constraints as to the sizes and shapes of the matrices in a Kronecker product. Taking the Kronecker product of an  $n$  by  $m$  matrix with a  $p$  by  $q$  matrix results in a much larger matrix: an  $np$  by  $mq$  matrix.

The way I think of the Kronecker product is that it splits the second (right) matrix onto the first (left) matrix. Specifically, every entry in the left matrix is "expanded" by the product of it and the entire right matrix. Let's take an example; how about taking the Kronecker product between our example matrices  $M$  and  $P$ :

$$M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \quad \text{and} \quad P = \begin{pmatrix} 7 & 8 \\ 9 & 10 \end{pmatrix}.$$

$M$  is a  $3 \times 2$  matrix and  $P$  is a  $2 \times 2$  matrix, so we expect  $M \otimes P$  to be a  $3 * 2$  by  $2 * 2 = 6$  by  $4$  matrix. So, let's go through the math:

$$M \otimes P = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix} \otimes \begin{pmatrix} 7 & 8 \\ 9 & 10 \end{pmatrix} = \begin{pmatrix} 1 \begin{pmatrix} 7 & 8 \\ 9 & 10 \end{pmatrix} & 2 \begin{pmatrix} 7 & 8 \\ 9 & 10 \end{pmatrix} \\ 3 \begin{pmatrix} 7 & 8 \\ 9 & 10 \end{pmatrix} & 4 \begin{pmatrix} 7 & 8 \\ 9 & 10 \end{pmatrix} \\ 5 \begin{pmatrix} 7 & 8 \\ 9 & 10 \end{pmatrix} & 6 \begin{pmatrix} 7 & 8 \\ 9 & 10 \end{pmatrix} \end{pmatrix} =$$

$$\begin{pmatrix} \begin{pmatrix} 7 & 8 \\ 9 & 10 \end{pmatrix} & \begin{pmatrix} 14 & 16 \\ 18 & 20 \end{pmatrix} \\ \begin{pmatrix} 21 & 24 \\ 27 & 30 \end{pmatrix} & \begin{pmatrix} 28 & 32 \\ 36 & 40 \end{pmatrix} \\ \begin{pmatrix} 35 & 40 \\ 45 & 50 \end{pmatrix} & \begin{pmatrix} 42 & 48 \\ 54 & 60 \end{pmatrix} \end{pmatrix} = \begin{pmatrix} 7 & 8 & 14 & 16 \\ 9 & 10 & 18 & 20 \\ 21 & 24 & 28 & 32 \\ 27 & 30 & 36 & 40 \\ 35 & 40 & 42 & 48 \\ 45 & 50 & 54 & 60 \end{pmatrix}$$

Sure enough, the answer is a 6 by 4 matrix. Let's go through this in Python:

```
import numpy as np
M=np.matrix('1 2; 3 4; 5 6')
P=np.matrix('1 2; 3 4')
print(np.kron(M,P))
```

Which prints the following:

```
matrix([[ 7,  8, 14, 16],
       [ 9, 10, 18, 20],
       [21, 24, 28, 32],
       [27, 30, 36, 40],
       [35, 40, 42, 48],
       [45, 50, 54, 60]])
```

# Bra-ket notation

In quantum mechanics and quantum computing, bra-ket notation is often used to denote a quantum state. For example the following are *kets*:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \text{ and } |0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix},$$

Whereas the following are *bras*:

$$\langle 0| = (1 \ 0) \text{ and } \langle 0| = (1 \ 0).$$

A ket is simply a matrix in a single column format; whereas a bra is a matrix in a single row format. For the same symbol within a bra or a ket, one is the conjugate transpose of the other. So,  $\langle 0/* = /0\rangle$  and  $/0\rangle^* = \langle 0/$ , and  $\langle 1/* = /1\rangle$  and  $/1\rangle^* = \langle 1/$ . Since the bra and the ket are different shapes (the bra is a 1 by  $n$  matrix, whereas the ket is an  $n$  by 1 matrix), in general, just because you can multiply a third matrix by one does not mean you can multiply it by the other.

# **Qubits, states, and gates in terms of matrices**

This section repeats part of earlier chapters, using more mathematical language and notation instead of Python code. This section of the Appendix is based upon my paper Moran CC (2018) *Quintuple: A Tool for Introducing Quantum Computing Into the Classroom* (<https://www.frontiersin.org/articles/10.3389/fphy.2018.00069/full>).

# Qubits

A qubit is the quantum generalization of a classic bit. Unlike a classic bit, it can take any value corresponding to a linear superposition of its constituents: formally, two orthonormal eigenstates. In the course of the book, to prevent confusion as to when a number, for example 1, was used as a number, and when it was used as merely a label of a state, we always put "1" in quotes when it was used as a label of a state, for example,  $|"1">\rangle$ . However, in the general mathematical literature it is recognized that whatever lies between | and > or < and | is a label of a state, so here we will drop the quotation marks consistent with the literature.

Our default choice of basis throughout this appendix is the following:

$$|0\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, |1\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

This multi-purpose notation (</ or />), used throughout this manuscript to represent a quantum state, is called bra-ket or Dirac notation and is standard in quantum mechanics. Without getting into a detailed discussion of the mathematics, one can, simplistically, think of the symbol lying between the /> notation as being a label for the state. Whether the notation is /> versus </ indicates whether it is represented as a column or a row vector, respectively, where  $\langle |$  is the conjugate transpose of /> and vice versa. Thus, a generic one-qubit state  $|\psi\rangle$  is  $|\psi\rangle = a|0\rangle + b|1\rangle$ .

The coefficients  $a$  and  $b$  are complex numbers and these complex coefficients provide the representation of  $\psi$  in the

$\{|0\rangle, |1\rangle\}$  basis. The probability of finding  $|\psi\rangle$  in state  $|0\rangle$  is  $|a|^2 = aa^*$ , where  $a^*$  is the complex conjugate of  $a$ ; similarly, the probability of finding  $|\psi\rangle$  in state  $|1\rangle$  is  $|b|^2 = bb^*$ . These two probabilities normalize to one:  $|a|^2 + |b|^2 = 1$ . A single qubit state  $|\psi\rangle$  can be physically realized by a variety of mechanisms that correspond to a quantum mechanical two-state systems, for example a two spin system, or a two level system, among many others. The Bloch sphere is a useful way to visualize the state of a single qubit on a unit sphere. Formally, in the Bloch sphere representation the qubit state is written as follows:

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

Where  $\theta$  and  $\Phi$  are the polar coordinates to describe a vector on the unit sphere.

To make use of the power of quantum computation, we will in general want more than one qubit. In a classic n-bit register, we can initialize each bit to 0 or 1. For example, to represent the base 10 number 19 in a classic five-bit register, we can set its elements to  $_{10011}$ . For  $n$  qubits, to create an analogous state, a so-called quantum register, we prepare the state  $|10011\rangle = |1\rangle \otimes |0\rangle \otimes |0\rangle \otimes |1\rangle \otimes |1\rangle$ . Here,  $\otimes$  corresponds to the tensor product (also known as the direct or Kronecker product). Generically, an  $n$ -bit quantum register can hold any superposition of  $n$ -qubit states.

For an  $n$ -qubit state, there are  $2^n$  possible values the  $n$ -qubit state can, in general, be a superposition of. For example, for a 2-qubit state we have  $2^2 = 4$  possible states,  $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ . For a 3-qubit state, we have  $2^3 = 8$  possible

states or  $\{|000\rangle, |001\rangle, |010\rangle, |011\rangle, |100\rangle, |101\rangle, |110\rangle, |111\rangle\}$ .

Numbering the states from 0 to  $2^n - 1$ , the canonical ordering

used throughout this section is  $\sum_{m=0}^1 \dots \sum_{j=0}^1 \sum_{i=0}^1 |ij\dots m\rangle$ , where the number of summations corresponds to the total number of qubits. Thus, if we incorporate the amplitudes, the complex coefficients of these states, we can compute the probability

of finding  $|\psi\rangle = \sum_{m=0}^1 \dots \sum_{j=0}^1 \sum_{i=0}^1 c_{ij\dots m} |ij\dots m\rangle$  in state  $|ij\dots m\rangle$  as the squared absolute value of  $c_{ij\dots m}$ ,  $|c_{ij\dots m}|^2 = c_{ij\dots m} c_{ij\dots m}^*$ . If we can represent an  $n$ -qubit state as the tensor product of the states of individual qubits  $|q_0 q_1 \dots q_n\rangle = |q_0\rangle \otimes |q_1\rangle \otimes \dots \otimes |q_n\rangle$ , the state is called separable. However, due to the nature of superposition, it may be that a multi-qubit state is non-separable and individual qubit states are not well defined independent of other qubits. This non-local correlation phenomenon known as entanglement is a necessary resource to achieve the exponential speedup of quantum compared to classic computation. As such, the concept of quantum registers, necessary to store multi-qubit non-separable states, will play a primary role in quantum computation simulation.

# Gates

We have outlined the analogy to the classic  $n$ -bit register, the  $n$ -qubit quantum register, for keeping track of quantum data. Here, we will do the same with a classic gate and a quantum gate, which evolve classic and quantum states respectively. In classic computation, a classic gate operates on a classic register to evolve its state. In quantum computation, a quantum gate operates on a quantum register to evolve its state. Quantum states can be represented by matrices; the mathematics of the evolution of quantum states can unsurprisingly be represented by matrices as well. To represent quantum gates, these matrices must conform to the postulates of quantum mechanics as they multiply a state to produce an evolved state. Specifically, we know that the evolution of states must conserve probability (preserve norms); we cannot produce a state that is a superposition of states with a probability greater than one.

Matrices that ensure the conservation of probability when they multiply states are called *unitary*. Formally, this corresponds to any matrix  $U$  that satisfies the property that its conjugate transpose  $U^\dagger$  is also its inverse, that is,  $U^\dagger U = UU = I$ , where  $I$  is the identity matrix. In quantum computation, a quantum gate corresponds to a unitary matrix, and any unitary matrix corresponds to a valid quantum gate. Since unitary matrices are always invertible, quantum gates and thus computation is reversible; any operation we can do, we can undo. As a qubit state can be realized physically by a variety of quantum mechanical systems, so can quantum gates be physically realized by a variety of quantum mechanical mechanisms, which must

necessarily depend on the system's representation of the qubit. For example, in a system where qubits are represented by ions in a quantum trap, a laser tuned to a particular frequency can induce a unitary transformation, effectively acting as a quantum gate.

Gates acting on a single qubit can be applied to a quantum register of an arbitrary qubit number. For example, for a gate  $X$ , if the desired qubit to act on is the 3rd qubit in a 4-qubit quantum register,  $X$  is a gate which flips the qubit it acts on from  $|0\rangle$  to  $|1\rangle$  or from  $|1\rangle$  to  $|0\rangle$ . The appropriate gate is formed via  $X_{3\text{of}4} = I \otimes I \otimes X \otimes I$ , where  $I$  is the  $2 \times 2$  identity matrix. In general, to create a gate  $G_{m \text{ of } n}$  to operate on the  $m^{\text{th}}$  qubit of a register of  $n$  qubits from a gate  $G$  that operates on a single qubit, one may use the following:

$$G_{i \text{ of } n} = \bigotimes_{i=1}^n \begin{cases} I & \text{if } i \neq m \\ G & \text{if } i = m. \end{cases}$$

Here,  $\bigotimes_{i=1}^n$  is the analog of  $\sum_{i=0}^n$  corresponding to the tensor product, instead of the summation operation. We can see that the application of a gate on a single qubit in this fashion doesn't generate entanglement, as it never results in the expansion of the size of the quantum register it is acting on.

The gates we will work with are defined as follows.

# Gate definitions in terms of matrices

Specific sets of classic gates, for example the NOT and AND gates, can be used to construct all other classic logic gates and thus form a set of universal classic gates. Other such sets exist; in fact, the NAND, *negative and*, gate alone is a universal classic gate. In quantum computation, to obtain a universal gate set we will need a multi-qubit gate, which is applied on 2-qubits of an n-qubit register. The CNOT gate is one such gate. CNOT is the *2-qubit controlled not gate*. Its first input is known as the control qubit, the second as the target qubit, and the state of the target qubit is flipped on output if and only if the control qubit is  $|1\rangle$ . The application of CNOT can under many scenarios generate entanglement. CNOT combined with single qubit gates can approximate, arbitrarily well, any (unitary) operation on a quantum computer. Quantum gates can be combined to form quantum circuits, the analog to classic circuits composed of logic gates connected by wires. The full set of gates that the IBM QX supports forms a (non-minimal) universal quantum gate set, such that we can combine the gates in a quantum circuit to create any multi-qubit logic gate we desire.

The H gate is also known as a **Hadamard** gate and is defined as follows:

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

The X, Y, and Z gates are also known as **Pauli** gates and are defined as follows:

$$P = \begin{pmatrix} 7 & 8 \\ 9 & 10 \end{pmatrix}, Y = \begin{pmatrix} 0 & -i \\ i & 0 \end{pmatrix}, Z = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}.$$

The I gate is also known as an **Identity** gate and is defined as follows:

$$I = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

The S gate is also known as a **Phase** gate and is defined as follows:

$$S = \begin{pmatrix} 1 & 0 \\ 0 & i \end{pmatrix}, S^\dagger = \begin{pmatrix} 1 & 0 \\ 0 & -i \end{pmatrix}.$$

The T gate is also known as a  **$\pi/8$**  gate and is defined as follows:

$$T = \begin{pmatrix} 1 & 0 \\ 0 & e^{\frac{i\pi}{4}} \end{pmatrix}, T^\dagger = \begin{pmatrix} 1 & 0 \\ 0 & e^{-\frac{i\pi}{4}} \end{pmatrix}.$$

The CNOT gate is also known as a **quantum NOT** gate and is defined as follows:

$$CNOT = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

# Additional gates

The quantum experience defines three additional gates, which correspond to being able to more easily navigate to different areas of the Bloch sphere. These gates aren't needed for the universal quantum gate set, but can help to minimize the total number of gates, and thus decrease the errors of a computation and/or make the computation easier to follow. These gates are as follows:

$$U_1(\lambda) = \begin{pmatrix} 1 & 0 \\ 0 & e^{i\lambda} \end{pmatrix},$$
$$U_2(\lambda, \phi) = \begin{pmatrix} \frac{1}{\sqrt{2}} & -\exp(i\lambda)\frac{1}{\sqrt{2}} \\ \exp(i\phi)\frac{1}{\sqrt{2}} & \exp(i\lambda + i\phi)\frac{1}{\sqrt{2}} \end{pmatrix},$$
$$U_3(\lambda, \phi, \theta) = \begin{pmatrix} \cos(\frac{\theta}{2}) & -\exp(i\lambda)\sin(\frac{\theta}{2}) \\ \exp(i\phi)\sin(\frac{\theta}{2}) & \exp(i\lambda + i\phi)\cos(\frac{\theta}{2}) \end{pmatrix}.$$

# Quantum measurement

We'll need to understand how measurement functions in quantum mechanics to understand the constraints of extracting information from a quantum register.

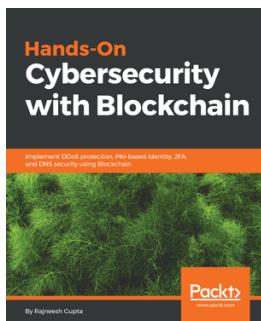
Measurement in quantum mechanics is something that engenders a lot of discussion, but its properties are straightforward to state in mathematics, if not in philosophy. It is possible to perform a measurement of a single qubit with respect to any basis  $\{|a\rangle, |b\rangle\}$  (not just the default  $\{|0\rangle, |1\rangle\}$  basis), so long as this basis is orthonormal, that is, that the total probability is 1. It is likewise possible to measure a multi-qubit system with respect to any orthonormal basis. Earlier, we stated that the probability of finding the following:

$$|\psi\rangle = \sum_{m=0}^1 \dots \sum_{j=0}^1 \sum_{i=0}^1 c_{ij\dots m} |j\dots m\rangle,$$

In state  $|ij\dots m\rangle$  is the squared absolute value of  $c_{ij\dots m}$ , which is  $|c_{ij\dots m}|^2 = c_{ij\dots m} c_{ij\dots m}^*$ . Here, when we perform a measurement, we actually do find the system in one of these states  $|ij\dots m\rangle$  with the appropriate probability  $|c_{ij\dots m}|^2$ . After the measurement is performed, the state is collapsed and all further measurements return the same result, state  $|ij\dots m\rangle$  with probability 1.

# Other Books You May Enjoy

If you enjoyed this book, you may be interested in these other books by Packt:

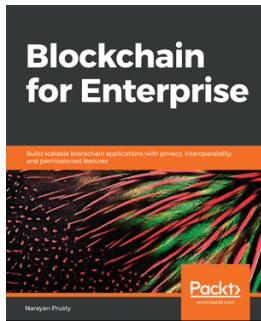


## **Hands-On Cybersecurity with Blockchain**

Rajneesh Gupta

ISBN: 978-1-78899-018-9

- Understand the cyberthreat landscape
- Learn about Ethereum and Hyperledger Blockchain
- Program Blockchain solutions
- Build Blockchain-based apps for 2FA, and DDoS protection
- Develop Blockchain-based PKI solutions and apps for storing DNS entries
- Challenges and the future of cybersecurity and Blockchain



## **Blockchain for Enterprise**

Narayan Prusty

ISBN: 978-1-78847-974-5

- Learn how to set up Raft/IBFT Quorum networks
- Implement Quorum's privacy and security features
- Write, compile, and deploy smart contracts
- Learn to interact with Quorum using the web3.js JavaScript library
- Learn how to execute atomic swaps between different networks
- Build a secured Blockchain-as-a-Service for efficient business processes
- Achieve data privacy in blockchains using proxy re-encryption

# **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!